DESIGN AND IMPLEMENTATION OF A STREAM-BASED VISUALIZATION LANGUAGE

Joseph A. Cottam

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of School of Informatics and Computing
Indiana University
November 2011

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Andrew Lumsdaine, Ph.D.

Katy Börner, Ph.D.

Amr Sabry, Ph.D.

Chris Weaver, Ph.D.

November 2011

This dissertation is dedicated to Clifford and Rebecca Cottam, who inspired to learn while I was young and showed me through their daily lives that learning can continue forever.

Acknowledgements

I thank my parents, Cliff and Becky Cottam. I thank my dad for engaging my curiosity at a young age. The earliest experiments I can remember came from a kit that we worked through together when I was seven. I hope I can retain the sense of exploration and play that he imparted in those early years. I thank my mom, for her love and attention to my pursuits. I sincerely appreciate our weekly conversations and her constant concern for me, despite my wanderings.

I thank my committee: Andrew Lumsdaine, Amr Sabry, Katy Börner, and Chris Weaver. Each contributed a valuable perspective on this work and made it better than I could have alone. I feel honored by my committee's time and attention.

I owe an extra measure of thanks to Andrew Lumsdaine, for the privilege of being his student. He gave me the latitude to follow my curiosity and the guidance to bring those interests to completion. I don't know when I'll have the opportunity to work in as rich and free an environment as he provided. I am deeply grateful for his support throughout this process.

I thank my colleagues at Indiana University, especially the members of the Open Systems Laboratory. Special thanks are due to Josh Hursey, Ben Martin and Russell Duhon for using prototypes of my work in their own, this illuminated early weaknesses and vastly improving my work through their patient experience and honest feedback.

Finally, I would like to thank Neva Cottam. I deeply appreciate the sacrifices of time and convenience in supporting this endeavor. I can't describe the comfort I take in knowing you are there for me, even when I doubt myself. I will always be grateful for you.

Abstract

Information visualization tools are proliferating, with language and library-based visualization tools serving scientific, business and artistic endeavors. While partial formalizations of these tools exist, many details are unspecified. This leaves the tools as *ad hoc* implementations of ephemeral ideas.

This thesis presents Stencil, a domain-specific language for specifying visualizations that has a well-founded basis. Stencil has a formal semantics that establish testable conditions for resource-bounded and deterministic execution. The semantics enable treatment of dynamic data in a principled fashion. Internal consistency of a visualization id defined in terms of these semantics and the implementation is shown to maintain that consistency. Stencil includes a task-parallel execution model that dramatically improves runtimes when working with dynamic data. The improvements are realized by application of the semantics to reason about the runtime, and employing persistent data structures to provide non-interference.

As a domain-specific language, Stencil enables compact description of visualization programs. The key to providing this compactness is the Stencil compiler's ability to infer contextual information. Contextual inference avoids restatement of program constructs and allows guarantees about the inferred information. This contextual inference is used to create axes, legends and other guide structures in a fashion that is guaranteed to reflect both the corresponding analysis and the input data.

The lessons learned developing the Stencil system are applicable to other visualization frameworks. For example, other libraries can employ the data structures implemented for Stencil. More importantly, the formalisms presented provide a basis to define and test properties of a visualization framework implementation. This thesis demonstrates the feasibility of treating visualization programs in a principled fashion and some of the benefits (including greater performance and reduced programming effort) of doing so.

Contents

List of Tables

List of Figures

List of Acronyms

AST: abstract syntax tree	76
DSL: domain-specific language	15
E-FRP: Event-driven Functional Reactive Programming	26
FRP: Functional Reactive Programming	7
GoG: *The Grammar of Graphics*	13
GSI: global state indicator	55
GST: global state tracker	55
JUNG: Java Universal Network/Graph	75
RSL: render-state lock	143
RT-FRP: Real-Time Functional Reactive Programming	44
VTK: The Visualization Toolkit	4

1
Introduction

Data analysis is of growing importance. The ability to acquire, record and share information is increasing as new instrumentation, reporting and data distribution technologies develop. However, when data are plentiful, it is important that data analysis be available. Data analysis tools convert raw data into an interpretable form; for visualization tools, that form is visual. Accordingly, information visualization has become increasingly important in recent years. The principled design and implementation of visualization tools is the focus of this dissertation.

Many effective visualization tools have been developed. Common charts and graphs, analysis applications and programming frameworks are common forms for visualization tools. Each form lends itself to different tasks, users and data types. What makes a tool suited to a particular environment is often determined by fundamental design decisions embodied in the tool itself. Assumptions about the data's quantity, presentation and interrelationships impact a tool's approachable tasks.

This dissertation explores the design of the Stencil visualization system. Three fundamental decisions impact every aspect of Stencil's implementation. These decisions are (1) a focus on dynamic data; (2) a declarative programming style; and (3) a deterministic execution semantics. Though these decisions influence the implementation, they do not fully determine it. The effects of the foundational assumptions and their embodiment in Stencil include lessons applicable to future visualization systems. Comparisons to existing visualization systems, built on different assumptions, illuminate the impacts of the implementation-independent choices in Stencil.

The contributions of this dissertation are (1) a formal semantics of the data-flow model of visualization; (2) a definition of visualization consistency used to evaluate framework features; (3) analytical, visual and computational abstractions based on the semantics that are safe (per the consistency definition); and (4) a minimal set of transformation-operator level metadata to support the given abstractions.

This chapter provides definitions of terms used throughout this dissertation and outlines the rest of the document.

1.1. Dynamic Data

The Stencil system works over dynamic data sources. The salient characteristics of dynamic data are

Sequence: Values appear in an order.

Updates: Incoming information may revise/update earlier information.

Unbounded: The quantity of data is not necessarily known in advance and, in general, there is no bound assumed for the data (e.g., it continues forever).

Knowing that dynamic data implies a sequence of values enables the processing application to take advantage of the order. However, the interpretation of the sequence is not prescribed. The values may be sequenced by a relation to clock time, but this is not guaranteed. Furthermore, the time between two values appearing in a dynamic data source does not necessarily correspond to any time measure either. From the viewpoint of a data-processing application, element ordering must be considered as analysis operators may be order-sensitive. Additionally, the analysis application must be ready to respond to data streams with no current value, but that may later have a new value.

Given that data points are arriving in an ordered sequence, there is an inherent temporal distribution. This provides a natural means to represent changing values over time. Acknowledging that a data point represents an update in a computation process depends on the details of the process. Some processes ignore the fact that a new value is a revision of a prior value (e.g., those that plot value changes through time). Others may treat updates as successive approximations (e.g., coordinate positions in an iterative spring-force embedding).

1.1.1. Examples. Whenever values fluctuate temporally, dynamic data are found. Common examples include time itself, temperature and the position of a moving entity. More rich examples include the internal state of a program, tweets or tagging events (and other incremental social operations) as well as the aggregate value of social processes (e.g., the current price of a stock and cell-phone call densities).

1.1.2. Bounded Data. In the general case, dynamic data sources are unbounded in the number of tuples they contain. However, not all data sources are, in fact, unbounded. Reference information and execution-time parameterizations are analysis components that are often bounded. The key point for dynamic data is that the source is *potentially* unbounded. Two effective ways to represent bounded data sources are (1) as a repeating loop, or (2) as an exhaustible source. A repeating source represents constant values effectively, but an exhaustible source can represent any finite set of values without requiring special handling. The number of values does not need to be represented as part of the source, neither does the end. After the last value is reported, a bounded data source may simply present no additional values. However, explicitly representing the end of the data enables special handling based on this additional information. A sentinel data value can effectively signal that a stream has terminated.

1.1.3. Real-Time Data. Real-time data is a common type of dynamic data. However, real-time data sources include an additional property over dynamic data: timeliness. Timeliness is a guarantee that there is a correspondence between the occurrence of an event and the appearance of that event on the data stream (e.g., a deadline). A real-time data stream is also dynamic, but not all dynamic streams are real-time. Dynamic streams only preserve the *order* of events, not necessarily the timing.

Real-time data sources are not given special consideration because the data processing in Stencil does not support the timeliness contracts required by real-time processing. Therefore, any real-time properties held by the stream are not propagated past the first processing operation. Since adding a visual element to a display involves at least one data-processing step (i.e., adding an element to a display is itself a data-processing operation), all nonempty visualizations in Stencil lack real-time timeliness guarantees.

1.2. Data-Flow Model of Visualization

The data-flow model of visualization is common in scientific visualization. The Visualization Toolkit (VTK), is a powerful visualization framework based on a data-flow model. In VTK, individual transformations are implemented as stream-based transformers [**74**]. The core concern of flow-based frameworks is composing and manipulating data transformations. Composition and manipulation requires an encoding of the dependencies between transformations, whereas the computation performed by the transformation itself may be implemented in some other model. For this reason, data-flow frameworks are occasionally referred to as coordination languages [**54**].

Two attributes drove the selection of the data-flow model as the basis for Stencil. First, the data-flow model naturally corresponds with dynamic data; updates occur incrementally in both models. Second, operator granularity is unspecified. An operator may take in many or few data points before producing a new result. This freedom is used to achieve significant computational efficiency, discussed in Chapter 9. Chi's proof that the data-flow and the data-state models of visualization are equivalently expressive [**20**] implies that generality is not lost in this decision.

Six components comprise the data-flow model of visualization. These components are datum (tuple), stream, transformer, link, bind and render. General data-flow computational models share the first four. Bind, or a similar concept, appears as a component in some models, but is not universally present. Render is a visualization-specific component.

Datum/Tuple. A datum is a single unit of information and is analogous to the *information packet* found in flow-based programming [**82**]. All data moved through must appear as a datum. There are many equivalent structures for the datum, with varying treatments for compound data elements [**72**,**82**]. Because of its simplicity, Stencil uses a *tuple* representation. A tuple is a finite compound datum where the order of the constituent data points is significant. Classic examples are coordinates ((X,Y,Z), notice X is always first, Y second and Z third) and database record-set rows.

Stream. Streams are ordered groups of tuples. Streams are potentially infinite sequences of tuples with no guarantee on the timing of the delivery of the next tuple. Therefore, a stream satisfies the definition for a dynamic data source provided in Section 1.1. Furthermore, streams are forward-only reading; unless explicitly stored, item recall is not possible.

Transformer. A transformer consumes a stream and produces a new stream. It is a node in the data-flow graph. In the majority of cases, a transformer produces a nonempty stream and is thus represented as a node with an outgoing stream. Transformers may represent primitive (e.g., add, sub, trim) or compound operations (e.g., Layout, Dictionary).

Link. The link relation expresses dependency between operators. Informally, if an operator $O1$ sends its output to another operator $O2$, then a link relation exists from $O1$ to $O2$. The link relation may be realized either directly (behaving like a function call) or indirectly through a coordinating mechanism (such as tuple spaces [**54**]).

Bind. Binding is the process of placing values from analysis into some accessible storage. For visualization, this is the so-called "visual store." Bind appears in a data-flow graph as nodes that would be placed in the visual-abstraction region in Chi's taxonomy. The general description of data-flow computation does not address coordination of computation components. In particular, the ordering of analysis between different branches plays heavily into the predictability of the system. Resolving this omission is an important part of the semantics presented in Chapter 5.

Render. Rendering is the process of converting visual abstractions to views. Rendering may include transformations from the values stored in the visual abstraction store, though these are typically limited to zoom/pan style transformations.

The data-flow model is simple. It provides little information about common computational constructs such as sequence, conditionals or effects. This lack of specification gives flexibility for decisions presented in later chapters.

1.3. Declarative

Declarative programming is often indicated as "stating *what* is to be computed, but not necessarily *how* it is to be computed," [**77**]. This definition is imprecise, as the contents of "what" and "how" change meaning as abstraction levels change. The Stencil language, presented in Chapter 3, includes two distinct means of providing declarative constructs. First, it acts as a coordination language [**101**]. In this capacity, operators are tied together and results are grouped, but the exact operator implementation and communication mechanisms are defined externally. By just declaring dependency, timing and execution order are minimally encumbered and available for manipulation through other abstractions. Working as a coordination language is contrary to another common definition of declarative programming that states that a lack of control flow defines a declarative language [**77**]. However, the Stencil language preserves the fundamental concept of this definition by working with *dependencies* rather than working with a full control flow. This gives the system flexibility, constrained by the semantics, when defining the exact control flow used.

The second way that Stencil is declarative is more straightforward. Representational abstractions, discussed in Chapter 8, provide a means of working with the eventual visual result in a higher-order fashion. Visual abstractions, such as guides and animation, are declared to apply in a region of a program. The visual result is achieved by manipulating the source program to achieve the declared effect. This is in contrast to requiring that such things be explicitly coded (i.e., indicating how to achieve an effect in a program region). This second use of declarative concepts in Stencil conforms to both common definitions of declarative programming.

1.4. Contributions

As indicated earlier, the contributions of this dissertation are (1) a formal semantics of the data-flow model of visualization; (2) a definition of visualization consistency used to evaluate specific framework features and implementations; (3) analytical, visual and computational abstractions that are safe (per the consistency definition); and (4) a minimal set of transformation-operator level metadata to support the given abstractions.

The expressiveness and suitability of the chosen semantics is the first item pursued. The second priority is a definition of *consistency* and its utility for evaluating visualizations and visualization frameworks. The abstractions built on top of the semantics and their implementation details, often in pursuit of consistency preservation, demonstrate the semantics' breadth of influence. Supplemental metadata about the analysis operators is a key component of the abstractions presented in this dissertation. The formal semantics and definition of consistency preservation strongly inform the chosen metadata.

Evidence is examined in the following order:

Chapter 2: A catalog of the core concepts and a survey of similar techniques.

Chapter 3: An informal introduction to the Stencil language.

Chapters 4 & 5: A formal treatment of visualization data-flows. Functional Reactive Programming (FRP) is the basis of this formal treatment. This formalization is then used to establish several useful properties of the Stencil system, including a testable definition for a consistent visualization.

Chapter 6: Important details of the Stencil system. This implementation description includes a description of the metadata system and how the system remains faithful to the presented semantics.

Chapter 7: Abstractions used for analysis. Analysis is the basis of insightful visualization, and providing abstract operators improves the expressiveness of the implementing framework.

Chapter 8: Abstractions for animation and guide creation. Other frameworks provide these tools in an *ad hoc* fashion; building them on top of the language semantics avoids pitfalls by providing testable properties that can be evaluated independently of the implementation.

Chapter 9: Application of the formal semantics to low-level implementation details. A task-based decomposition of the Stencil framework is described and shown to preserve consistency (as defined in Chapter 4). Additionally, data-state and data-flow style operators are mingled to exploit the strengths of each.

Chapter 10: A summary of the contributions of this thesis and a discussion of avenues of research related to the semantic and metadata frameworks not explored in this work.

Appendix A: Details on the implementation of the examples used in other chapters.

Appendix B: A description of the meta-data available to the Stencil compiler and how it is represented in resource files of various types.

2 Literature Review

2.1. Production and Interpretation Graphics Theory

Graphically represented data are interpreted according to psychological principles, rooted in evolution and culture. Bertin provides the principal taxonomy of data-based visual communication [**8**, **9**]. Recent works have also tried to blend visual and psychological concerns with production pragmatics [**24**, **120**, **124**]. In a more production/consumption-focused vein are the works of Tufte [**111–114**] and William Cleveland [**25**, **26**]. The major points of these works are discussed below.

2.1.1. Semiology of Graphics.

Bertin's seminal works [8, 9] take a deconstructive and taxonomic approach to statistical graphics. They ask the questions "What are the properties of data that are represented in graphics?" and "What are the properties of graphics that foster data representation?"

The first of Bertin's questions leads to the categorization of data to be represented. This is significant, as the preparatory work of categorization helps ensure that the graphics produced do not misrepresent the data. By identifying the significant characteristics of data, the significant characteristics of graphics are more easily identified. Bertin identifies three basic characteristics:

Number of components: The number of values per observation. Component number determines the (maximum) number of visual variables required to represent the data, or how many *n*-variable graphics are required to represent all of the data if one graphic is impractical. Bertin assumed that only values of significance to the task at hand will be considered at this stage, so prior analysis may reduce the number of observed characteristics to a significant set.

Length: How many distinct and valid values for a particular component. For a discrete variable, these are the statistical *levels* [36] (not to be confused with levels described below). Common length treatments are for binary, finite values less than a dozen and infinite (continuous values).

Level: Levels indicate the organization of values that a component may take. The levels specifically addressed by Bertin are Qualitative/Nominal, Ordered (including rankings, though Bertin asserts that true ordered data has equidistant values [9]) and Quantitative (including ratios, quantitative components permit variable distance and may include units). Levels form a hierarchy, so representation techniques for qualitative components may be used with quantitative components, but not vice versa. Some representation techniques are level agnostic.

The second core Bertin question, "What are the properties of graphics that foster data representation?" leads to a practical graphic vocabulary for describing how data are represented. Bertin uses this vocabulary to assess techniques. He develops three categories of graphic variables: Imposition (the type of coordinates used), Implantation (the type of marks used) and Visual Variables (the properties of marks recorded by the eye).

Impositions describe how the coordinate space is used:

Arrangement: Elements placed on a plane with no significance to their position or relationship. A random graph layout is an extreme arrangement. More common is a layout where proximity matters, but displacement between groups does not.

Rectilinear: Items are placed along a single linear dimension. A single stacked bar chart is a rectilinear imposition.

Circular: An angular layout instead of a linear layout. Circular layouts typically place significance on the angular dimension (but they may use the distance from center instead). The variation in the nonsignificant dimension remains an arrangement. Pie charts employ a circular imposition with significance in the angular dimension. The displacement from center of a slice is often used only to emphasize particular points.

Orthogonal: Using multiple dimensions placed at right angles. Bertin generally assumes only two dimensions will be used [**8**], but admits that a third is sometimes warranted. This forms the basis for a Cartesian coordinate system if both dimensions are continuous, but generalized systems may employ discrete dimensions (such as corporate departments or yes/no questionnaire responses).

Polar: A two-dimensional imposition based on the one-dimensional circular imposition. The relationship between polar and circular is similar to that between rectilinear and orthogonal.

Bertin's graphic vocabulary continues with the a classification of marks. A mark represents a data point in a space; the mark's *implantation* and *imposition* determine its relationship to that space. Imposition, discussed earlier, is the relationship between position and meaning. Implantation is the relationship between the position and mark's extent. The three implantations are point, line and area (or volume, if a 3D space is used). Point implantation is used to represent a location. The mark must have a physical area to be visible, but the extent of that area is not significant in the imposition. For example, a city mark on a world map does not denote the boundaries of the city. Generally speaking, if the map were zoomed in, point implanted features should not proportionally increase in size. A line implantation describes a connection between two points. The intermediate points between the endpoints may or may not be part of the relationship. Roads on a political map are an example of a line implantation that includes the space between points in the relationship, while flight schedule maps often use a line implantation that does not include the intermediate points. Area implantation is intended to convey an extent in the imposition space. Political boundaries on a map are a geographic example of an area implantation, stacked line charts are a statistical example. When zooming in on an area implantation, the extent of the shape changes proportionally.

Visual variables describe how a mark actually appears. There are two subcategories: location and retinal variables. Location is the physical position of the mark on the represented logical space. Imposition dictates the relationship between position and meaning, so imposition is the logical positioning in a semantic space. Location is the position of the mark on the physical page, regardless of imposition used. Location is significant for two reasons. First, it is the only visual variable that is continuous; all other visual variables have a limited number of values that can be expressed, forcing groupings of data. Second, it is homogeneous, so no values are universally preferred (though some cultural and psychological factors apply [**120**]).

The second category of visual variables is retinal variables, so called because they are recorded by the retina without respect to the physical location. Retinal variables are shape, orientation, color, texture, value and size. Properties associated with retinal variables include associativity, selectivity, ordering and quantization. These properties correspond to data properties and guide proper representation of data based on the task at hand. For example, the size of a mark allows inequalities to be identified quickly (selectivity), or allows *greater than* and *less than* comparisons (ordered), or estimates of the ratios between values (quantization). It does not effectively create groups across variation (associativity), but shape variations do. The implantation may also impact retinal variable properties.

The properties of the data described (number, length and level) guide the graphical properties (imposition, implantation and retinal variables) used in a specific representation. For example, if a continuous, quantitative variable is present, it is preferable to use an axis to represent it. By contrast, a nominal variable of secondary importance would be more likely represented as an associative retinal variable to allow more significant graphical features to dominate. The categorizations presented above are not sufficient to automate graphical representation in all cases, but they do construct a framework for discussion and suggest foundations for objective evaluation of any particular representation.

2.1.2. Leland Wilkinson: *The Grammar of Graphics*. Software systems provide an implicit taxonomy of the problem space they target. The data structures and methods provided embody a problem representations and organizational principles for solutions. Leland Wilkinson extends his work on statistical graphics software into a language for describing the construction of a graphic from base tables. His work is presented in *The Grammar of Graphics* (GoG) [**124**]. Wilkinson's work presents a more pragmatic view than that of Bertin, as the grammar presented is tied directly to a particular object-oriented graphics library. However, the issues addressed in that library are shared by all statistical graphics systems. Understanding the components of the overarching grammar illustrates a distinct view on the problem of understanding graphics that extends the work of Bertin.

The primary contribution of GoG is an analysis of the graphical elements common to all data graphics, with an eye towards generalizing the relationships between them. This produces a set of graphical elements and their respective properties. Wilkinson identifies five elements:

Scales: Expanding the idea of data levels, scales address the pragmatics of internal representation and composition. Wilkinson's grammar-based description addresses such issues as the results of values calculated from variables of differing levels and transformations of variables (and their implications). Scales also present a unified way to handle unit-carrying and unit-less values. The theory of scales introduces two additional Bertin-like levels: Order and Measure.

Coordinate: Coordinate spaces expand on Bertin's impositions. Bertin notes that circular and rectangular graphs are essentially transformations of the same linear graph. Wilkinson extends this relationship to arbitrary transformations. Affine, projective and conformal transformations are all discussed. Their relationships to each other and to the representation of different types of data (abstract, geographic and illustrative) are all examined.

Aesthetic: Functions that move abstract data to the visual sense are the aesthetic functions. These functions account for not only the data characteristics, but also the observers psychological characteristics (such as customary representation or cultural meanings) and physical characteristics (such as color blindness and the unequal intensity of different colors). This type of function is the major focus of the psychology-based visualization research including Ware [**120**], and Piroli and Card [**94**].

Facets: Distinct views of the same data are termed facets in GoG. The central idea is that high-dimensional data may be better served by multiple 2-dimension displays than by an attempt to capture all of the data in a single image. This problem is the origin of brushing and linking [**15**] and the focus of papers in its own right [**76, 84, 85**]. Facets also capture the essence of Tufte's small-multiples [**112**], though Tufte expands the use beyond viewing multiple dimensions.

Guides: A visualization is abstracted from its raw data, guides provide the link between the abstraction and the original. Guides are the marks that contextualize interpretation. Poorly placed guides can be confusing at best and misleading at worst. Overly dense guides become illegible. The interaction between the data, message and the guides is complex and requires a separate, but related, consideration from the data itself.

Since Wilkinson is developing a grammar for graphics construction, he also includes notes about generalized algebras. Though Wilkinson focuses on statistical graphics of static data, he also includes chapters targeting time and uncertainty. These are particular types of data that have special graphical considerations; they represent potential extensions to Bertin's notion of *level*.

The majority of language description found in GoG is targeted at a specific graphics library. However, the concepts of the language have been embodied in domain-specific language (DSL) iterations and libraries. These languages include ViZml, Graphics Processing Language [**125**], nViZn [**126**] and ggplot2 [**123**]. These additional languages are closely related to the core grammar described by Wilkinson and are tied to the their particular packages. In addition to the items described by the core grammar, these grammar-driven systems must precisely describe the incoming data format, provide default values, describe data processing and resolve aesthetic issues. These pragmatic issues are of importance in any data-driven graphics system, and Wilkinson's work lays a foundation for generalization.

2.1.3. Visual Description Languages. Research into languages for describing pictures has been pursued previously. Mackinlay [**79**] developed "A Presentation Tool" (APT) based on a language that closely followed Bertin's data classification system [**8**]. APT is designed to take relational information and an APT description as input. It then prepares a graphic according to Mackinlay's interpretation of Bertin's rules. This required Mackinlay to prioritize different presentation styles over others, beyond Bertin's prescriptions. APT focused on automatic graphic creation, externalizing the core presentation decisions. Mackinlay also required relational data; his work is a precursor to database visualization languages such as VizQL [**64**].

Roth and Mattis [**100**] followed Mackinlay's approach of automatic data mapping, but extended Bertin's data descriptors. In particular, Roth and Mattis introduced special categories for coordinates versus quantities, domain membership, and algebraic dependencies. They also made relational structures explicit. Their work allowed conventional representations to be identified and applied (particularly through domain identifiers, provided the domain vocabulary is known by the user). Their work illustrated that Bertin's classification system can be meaningfully extended, especially with respect to data decomposition. However, Roth and Mattis' system focuses on automatically creating charts based on rules, keeping the transformation process hidden. This work is instantiated in the SAGE and Visage systems [**99**]. In Visage, data classification was extended to explicitly handle certain types of metadata separately. The focus in both systems is on automatically creating visualizations based on data descriptions.

Casner [**17**] departed from Mackinlay's approach of data descriptions and developed languages to prepare graphics based on tasks rather than data characteristics. Casner described the graphical production as a sequence of abstract actions. The key insight is that each graphical operation (zoom, highlight, rotate, etc.) corresponds to a step towards a solution. As such, Casner's descriptive language abstracts graphical operations to goal-seeking tasks. This work has been recently revisited by Yi et al. [**128**]. Yi's and Casner's works illustrate the types of compound operations people perform. As such, they are rich sources of abstractions. Casner focused on interacting with a particular system while Yi presents a taxonomy for discussion purposes; it is not intended as an implementable software system.

2.2. Models of Visualization

There are two major conceptual models for the visualization process. The first is the InfoVis reference model, presented by Card et al [**15**] (see Figure 2.1). The InfoVis reference model divides visualization into four stages and four transitions. Each stage represents data in successively higher levels of abstraction, ranging from raw source data to transformed subsets of visual artifacts. Transitions between the stages are accomplished with

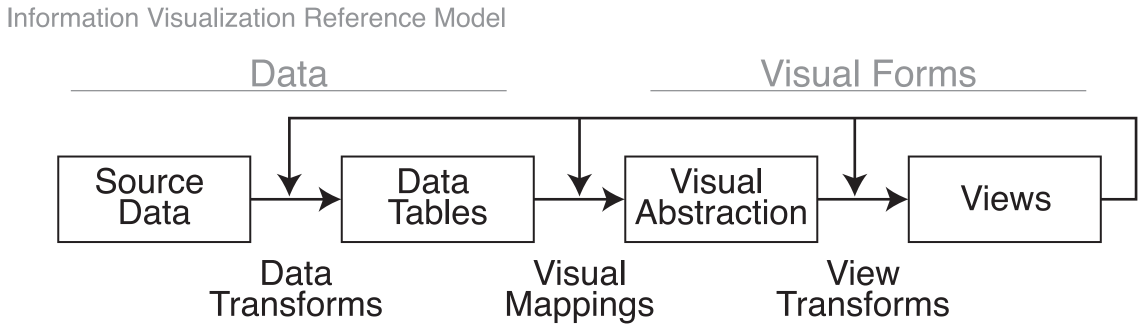


FIGURE 2.1. The Information Visualization reference model as presented by Card et al. [**15**].

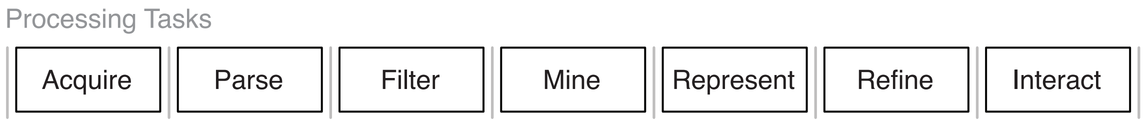


FIGURE 2.2. The Processing model for data analysis and visualization [**51**].

transformations. The first three transformations are straightforward, the final transformation is a feedback loop. The results observed in a view provide the basis for modifications made to earlier transformations (and thus earlier stages). Transformations are categorized according to which stage they operate in.

The second major conceptual model of visualization was proposed by Fry in conjunction with the Processing framework [**51**] (see Figure 2.2). The InfoVis reference model can be seen as a high-level view of the organization of visualization software. In contrast, the Processing model is more activities focused, regardless of the implementing software. For example, the Processing model includes an explicit representation for refinement and interaction. In contrast, the InfoVis reference model combines those two as an implicit part of the feedback loop. The Processing model is useful for evaluating how a data analysis tool (including a visualization framework) fits in with other analysis tools. Any given tool may fulfill multiple (even discontiguous) tasks in the Processing framework. In fact, most tools built around the InfoVis reference model provide facilities for multiple stages (Filter, Mine and Interact are typically well represented, and at least minimal acquire and parse facilities are provided).

From a software standpoint, there are also two major architectural models: the data-state and data-flow models. These two models describe the internal organization of the software itself; both can be mapped onto the conceptual models with little difficulty. The data-flow model is well described in the VTK documentation [**74**]. The data-state model is detailed in Chi's data spreadsheets work [**20**, **24**] and motivated in the discussions of the InfoVis reference model [**21**, **46**, **69**]. In brief, each treats a visualization process as a graph of operations. In the data-flow model, each operation works on a single data element and the entire network is executed once for each element. In the data-state model, operations work on entire collections of data and the network is executed once for each collection.

Despite their differing philosophies for data handling, Chi [**22**, **23**] established the expressive equivalence of the two models. Chi showed that every data-flow visualization network has an equivalent data-state network. Chi provides an abstract process for characterizing the equivalent data-state network given a data-flow network (and vice versa). Chi's proof relies on metadata operators that are not fully specified to exchange individual data-state and data-flow operators.

2.3. Visualization Systems

New visualization frameworks and libraries are constantly being developed. Both conceptual discussions and implementation details provide insight into the practical aspects of visualization construction. Such discussions show how others conceive of the visualization process and how those concepts correspond to realized data structures and control flows.

2.3.1. DEVise. DEVise is a C++ library and accompanying runtime environment [**76**]. The runtime provides support for customizing the visualizations created in the framework. The framework itself is designed to handle streaming data and employs a tuple-mapping metaphor. DEVise divides all data into two tuple types: text or graphic. Data are received from a stream as text tuples and stored in a database. Processing on the text tuples generates graphic tuples that are also stored in the database. DEVise programs specify the mapping between the text and graphic data [**19**]. Graphic data are associated with a DEVise view and rendered to the screen. Multiple views can be linked together via the *cursors* and *links* mechanisms [**76**].

DEVise uses streams and tuples as its fundamental data processing elements. However, this does note extend into handling user input or view coordination; these systems are given special-purpose treatment instead. Another significant issue is that DEVise does not allow views to be composed from smaller parts in a straightforward manner. In DEVise, the mapping between text and graphic tuples is essentially handled as a preprocessing step. Database queries define the views. As such, there are no direct mechanisms for building layer-like abstractions without explicitly including metadata in the graphic data schema.

The *visual template* system [**76**] shows the power of stateful, transportable, interactive objects. This allows a visualization schema to be created on one data set, and then quickly applied to another. This type of collaboration has been little studied, but may provide some benefits [**121**].

2.3.2. Prefuse. Prefuse is a visualization framework with an eye towards generality and runtime configurability. Generality is achieved with general-purpose data structures (principally a cascade table [**67**]). An event framework, expression language and runtime type support provide runtime configurability [**69**].

Two significant issues make Prefuse difficult to integrate with dynamic data and custom anlaysis. First, the threading structure of Prefuse implies the data points are all present when the visualization phase begins. Activities typically involve iterating through the whole data set (and if more data are added, the whole data set is iterated again). This is often unnecessary, and may be impossible with constantly updating data.

Second, Prefuse requires the host application to copy the data into its data structures explicitly. This requires the developer to understand the framework's data structures. More recent frameworks, such as Protovis (see Section 2.3.5), use a simpler iterator-based approach. With iterators, data needs to be presented to the framework sequentially, but the details of storage are handled internally.

2.3.3. Processing. Processing provides a Java library and development environment targeted at visualization tasks. It provides full access to the Java programming language for analysis, but shortens the *write* $\rightarrow$ *compile* $\rightarrow$ *run* $\rightarrow$ *modify* cycle by integrating the compile and execute steps into the programming environment. It provides library-based tools for common visualization tasks [**51**]. Furthermore, Processing presents a simplified version of the Java language, abstracting out tasks not directly related to the visualization process (such as window management, threading and class definition). The net result of these abstractions is that the Processing compiler presents a procedural version of Java, rather than an object-oriented one. Visualization code is still specified in terms of *for* loops and classically-typed variables, but ancillary issues (like the class hierarchy) can be ignored if the programmer so desires.

The Processing library and tool still require the programmer to maintain the general-purpose programming language mindset (e.g., data structure and control flow). Using a DSL simplifies this mindset, making the *write* $\rightarrow$ *compile* $\rightarrow$ *run* $\rightarrow$ *modify* cycle tighter by having the write and execute cycles conceptually more similar.

A second point of interest in Processing is its handling of mouse and keyboard input. There are two methods of handling input devices: events and special-variables [**98**]. The events method is the same as in standard Java. The special-variables method automatically loads globally visible variables with the mouse position, button states and key press information in each animation frame. A similar set of variables holds the information for the prior frame. Using dedicated state variables works for many simple interactions but it makes certain inputs a special case. Novel input devices and multi-stage inputs (such as a selection lasso) must treated in the traditional event model.

2.3.4. The Visualization Toolkit. The Visualization Toolkit (VTK) provides a collection of standard algorithms for data analysis and visualization built into an object-oriented framework. Programs written with VTK build a network of visualization components in a traditional language (C++, Python, Tcl and Java are supported). The start of the network is a data source, its terminal is a display object [**103**].

The standard methodology in VTK is to create a static network in the language of choice. This provides a convenient way to define a visualization system, but it encumbers the definition with the nonvisualization targeted components of the language as well.

Ad hoc processing in VTK is supported, provided the programmer supplies a component that complies with one of VTKs interfaces. This places responsibility on the implementer to understand the semantics of all of the standard methods and implement them accordingly.

2.3.5. Protovis. Protovis is a recent visualization toolkit, constructed as an embedded DSL in JavaScript or Java [**11**, **68**]. Protovis presents a Fluent API [**48**] for a visualization library whose architecture is based on the Prefuse [**69**] and Flare [**66**] toolkits. The specific architecture depends on the hosting language. Because the JavaScript implementation is more mature, it is given prominence, though differences are noted when significant.

The Protovis interface is built in a declarative style, focusing on two main concepts: (1) data iterators and (2) property definitions. Data iterators are zero-argument JavaScript function objects, though these may be omitted if default JavaScript iterators can be created (e.g., over arrays or dictionaries). Similarly, in Java, any object of a class that implements Iterable may be passed to the *data* method. By operating through the iterators interface, a programmer declares what data should be used and Protovis must derive the appropriate data structure for the visualization. Property definitions are also constructed with JavaScript *function* objects. The properties available depend on the mark type [**61**] or the layout prototype [**60**] that is being interacted with. Each property is provided with the relevant data values from the iterator (determined again by the panel or the layout the mark belongs to, which may be a default value). The default property definition may be overridden by providing a new function object of the appropriate number of arguments. In Java, dynamic definitions are achieved through the Java 1.6 tools infrastructure which allows access to a Java compiler to create dynamically defined methods [**68**]. In either case, no explicit link is made between the property and the data store, the link is implicit in the framework. This again leads to the opportunity for a declarative style of visualization construction. Having properties defined in terms of functions of data provides the opportunity to link property elements together as well by simply referring to the related property in the newly created *function* object.

A third way that Protovis enables declarative construction is through its adaptive defaults. The Fluent interface of the Protovis library results in a single visualization object that is the root of the other visualization objects that have been chained together. This root object essentially has a "bird's-eye view" of the visualization program and can provide context-specific default values, called *smart properties.* For example, after detecting that the data store is categorical, a categorical default color scale can be selected [**59**].

From the basis of data iterators and property definitions, along with a collection of built-in data manipulation functions and graphic elements, Protovis provides a rich visualization environment. This includes the ability to define new graphic elements by combining existing elements (leading to scene graphs [**6**] and small multiple visualizations [**112**]), and to construct interactive visualizations with properties dependent on user input. Of particular relevance to this thesis is the handling of reference marks (discussed further in Section 2.4).

An important part of Protovis is its scene-graph facility. This allows the composition of multiple visualizations or complex marks without adding compound graphic objects or complex visualization types (as is done in the InfoVis Toolkit [**45**]) as primitives in the framework. Each graphic element defined by Protovis has the capacity to have elements as its children. Child elements principally interact with parents in two ways. First, children inherit visual properties from their parents. Most scene-graph frameworks (for example, Piccolo [**6**]), have child elements inherit the coordinate space of their parents. Protovis takes this idea further, also inheriting data types, strokes, fills and other graphic design elements. If a child does not have a property defined, it is inherited from the parent. Default values thus propagate from the top of the scene graph to the bottom. Second, data element distribution is the responsibility of the parent. Parent elements act as data distributors in the same way that the root 'Scene' object does. This enables a natural expression of visualizations for recursive and hierarchical data structures [**11**, **68**].

2.4. Guide Creation

Guides provide context for the marks present in a visualization. Axes and legends are the basic guide types, but trend lines and point labels also provide valuable context.

The process of creating visualizations from data has been examined on a number of occasions. The general study of visualization and its relationship to data was discussed in Section 2.1. This theory is made concrete with the study of specific transformation functions, such as those performed by Bertin [**8**], Few [**47**] and Brewer [**13**, **65**]. These works are concerned both with the types of analysis that can be performed on different categories of data (e.g., continuous or categorical), as well the properties of visual space that are amenable to representation (e.g., what can color effectively show, how is size visually compared). Understanding these concerns is essential in both the process of visualization creation and in the creation of reference-marks to contextualize the visualization created.

One major roadblock to the expansion of abstract visualization processes is a lack of formal semantics for extendible visualization frameworks. Chi [**23**] provides some formalization of visualization program semantics. Those semantics are sufficient to discuss program transformations between major control-flow styles (data-state versus data-flow). In fact, Chi's work demonstrates the value of even partial semantics in treating programs as abstract entities. However, those semantics are insufficient to reason about execution, shared memory or the transformations of concern to this dissertation. The VizQL project [**64**] presents a set of visualization abstractions built on top of the semantics of SQL. The power of Tableau [**108**] is predicated on VizQL. However, VizQL has limited support for the development of novel visualization techniques and custom computations outside of its provided function set. It is a codification of best practices rather than a tool for exploring new visualization concepts.

Visualization applications tend to provide predefined visualizations, accompanied by predefined guide sets. The guides are effective for the visualizations provided, but permit limited customization. This style of guide creation is seen in spreadsheets (e.g., Excel [**102**].), statistical packages (e.g., R [**36**]) and more visualization-focused commercial packages (e.g., Tableau [**80**].). The ability to create *dashboards* in Tableau that can be repopulated from different data sources allows customization to be reused in a straightforward manner [**108**]. However, even when providing good support for the required operations, stand-alone software has a major drawback: it is difficult or impossible to integrate as a component into a larger program. While stand-alone software can share its data store, it often cannot directly integrate or be integrated with custom analysis.

The basis for automatic guide generation is found at the heart of analysis-based visualization software. Since the intrinsic guides are directly related to the analysis process, systems that address the analysis problem often partially address the guide problem as well. Prior systems that addressed guide creation, automatically or otherwise, include the InfoVis Toolkit [**45**], Prefuse [**69**] and *The Grammar of Graphics* (GoG) [**124**].

The InfoVis Toolkit provides a set of widgets with predefined visualization behaviors [**45**]. The reference-mark generation process is similar to that of most spreadsheets. If a programmer accepts the default behavior, the system permits control over the supplemental graphic design. However, creating custom guides, displays or postprocessing guide elements requires detailed understanding of the underlying toolkit, perhaps even a custom compilation. Widget-based libraries generally include this type of limitation.

Prefuse enables more customization and control than the InfoVis Toolkit [**69**]. Guides are typically 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 (1) the guide is an axis and (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 general process of creating an abstract descriptor of guide contents is the basis of the process presented in Section 8.1.

The Protovis DSL [**11**] presents an alternative approach to creating visual guides. Guide support methods are built into Protovis graphic objects. For example, pv.scale has a "ticks" method, mark types have "anchors", and the data iterator is pervasively available. Anchors and the data iterator can be combined to simulate some of the basic techniques presented in this dissertation. As an embedded DSL in a language, Protovis allows arbitrary post processing of automatically provided values in its host language. Postprocessing is a powerful technique discussed in Section 8.1.8; however, in Protovis it is often the only way to perform the basic task of pairing inputs to outputs. Despite the advanced concepts in Protovis, it cannot guarantee correctness in its guides and the techniques listed above still treat guides as any other data mark.

In language-based approaches to visualization and analysis, GoG provides the most complete discussion of reference marks [**124**]. The specification style for guides in GoG is declarative, mirroring the one called for in this dissertation. However, the *derivation* 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 [**122**]. ggplot2 provides concise production of standard guide types. However, ggplot2 is limited by the R data model, 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 [**13**]. Proper selection of tick marks on axes, and selection and placement of point labels is a perennial topic (examples include Talbot et al. [**109**] and Luboschik et al. [**78**]). Such work provides details on parts of guide creation, but does not integrate it within a larger guide framework.

2.5. Functional Reactive Programming

Functional Reactive Programming (FRP) is a programming model for constructing reactive programs; i.e., programs that are driven by their external environment. Rather than determining what computations to do internally, reactive programs perform computations in response to signals received from sources external to the program. FRP facilitates the construction of such programs in a functional programming environment. In FRP computations are represented as a collection of mutually recursive *behaviors* and *events*. Behaviors are continuously varying values and events are single values associated with a timestamp. Time is a central component of traditional FRP (though Event-driven Functional Reactive Programming (E-FRP) used in Chapter 4 is a notable exception).

One of the earliest FRP implementations is the Functional Reactive Animation framework (Fran) [**43**], which, along with its successors [**35**, **92**, **118**, **119**], was developed in Haskell. These implementations rely on the higher-order functions, lazy evaluation and functional semantics that Haskell provides. They are also typically expressed as an embedded DSL using Haskell's liberal operator syntax. Java [**34**] and Scheme [**27**] FRP implementations also exist. Notably, the non-Haskell FRP implementations make use of the state mutating operations of their respective environments, indicating that the "functional" aspects of FRP are principally at the conceptual level and need not be carried through to the implementation. The implementation of FRP used in this dissertation permits stateful transformation operators, provided certain operator relationships are maintained. In particular, a subset of E-FRP [**119**] is presented and discussed further in Chapters 4 and 5.

2.6. Persistent Data Structures

Persistent data structures are those that use nondestructive techniques to achieve updates to the contents of the structure. The old version of the data structure "persists" in that it is still available if a reference to the original structure is still retained. A linked list of elements with immutable contents and an immutable *next* pointer forms a persistent data structure. If an update is required, new cells are created to the point of the update (see Figure 2.3). Any reference to the old list does not contain the change. In contrast,

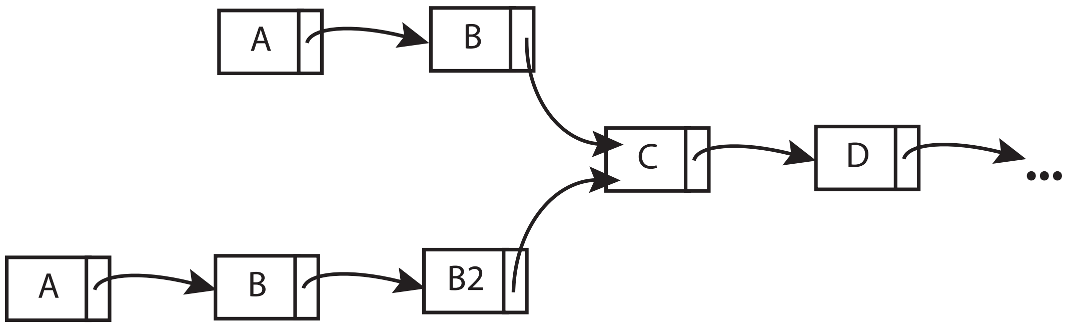


FIGURE 2.3. Linked list as a persistent data structure. After an update is made, a new head node has been created. However, any part of the program retaining a reference to the original head can use the list without concern for updates that may occur while using it.

updates done using mutation constitute "ephemeral" data structures [**129**]. Each version of the data structure effectively disappears when changes are made [**86**]. An alternative way of viewing persistent data structures is in terms of generators. As a generator, when a modification is made to the data structure, it produces a new data structure based on the old one plus the requested changes [**28**]. Persistent data structures are common in functional languages where mutation is not an option (or at least penalized). Implementations of persistent structures in Java include Clojure [**70**] and the pcollections framework [**28**].

Persistent data structures are characterized by the actions that can be performed between versions. If all versions of the structure can be modified, it is called *fully* persistent. However, if old versions are read-only and only the most recent version can be used as a generator, then it is *partially* persistent [**39**]. If modifications made to two versions of a fully persistent data structure can be merged into a new current version, the data structure is *confluent* [**40**].

Persistent structures can be thought of as realizing a form of transactional semantics [**105**]. Each change is made to the structure, resulting in a new structure. This structure can be discarded (e.g., rolled back) or propagated to other locations with references to the original (e.g., committed). True transactions require commits to appear atomic, so additional coordination is required in the propagation phase. However, persistence can be used in concert with other techniques to achieve the required effects.

Persistence is part of the general ability to automatically identify parallelization opportunities in functional programming languages because it provides explicit limits on the possible interference between program regions [**55**]. Interference with respect to concurrently executing processes is defined in terms of preconditions and effects. The process A *interferes* with process B if A executes an operation that violates the precondition of the currently executing portion of B [**3**]. In effect, A has caused B to enter an invalid state after the associated validity checks have been made. If A never performs an assignment that violates B's currently active preconditions, then A and B are *noninterfering*. Interference is a property of an execution of a of programs, not of the program itself. However, guaranteeing noninterference for all executions does depends on the program itself and is, therefore, done by analysis on the program. When using persistent data structures, state changes between processes are only visible through explicit communication. Therefore, changes to a shared data structure made by A can only be seen in B when A explicitly provides the updated data structure to B. Since no implicit exchanges can occur, the program analysis to guarantee noninterference is need only consider the explicit exchanges.

3
Language

Stencil is a system for constructing visualizations based on data-flow transformation of dynamic data streams. The declarative Stencil language is the central component of this system. This chapter describes the syntactic forms and an informal version of the language semantics. The purpose of this chapter is to provide a general view of the structure and interpretation of Stencil programs. Chapter 5 formalizes the semantics presented informally in this chapter, with Chapter 6 providing implementation details.

Section 3.1 presents the basic syntactic forms required to understand Stencil examples. Future analysis uses these forms to build complex abstractions. The forms also frame the further discussion of data types (Section 3.2) and transformation operator definitions (Section 3.3). Operator metadata plays a critical role in program transformations (Section 3.3.2).

```
1  stream Data(ID, predictor, percent)
2
3  layer Scatter
4  from Data
5    ID: ID
6    X: XScale(predictor)
7    Y: YScale(percent) -> Limit(_)
8
9  operator XScale : Scale[min:0, max:100]
10 operator YScale : Floor
11
12 operator Limit (in) -> (out)
13   (in > 100) => out: 100
14   (default)  => out: in
```

FIGURE 3.1. A Stencil program that produces a scatter plot, based on a Stencil program used in grade analysis. Input values both projected into the 0 to 100 range.

The basic Stencil syntax implicitly encodes important information; the normal form presented in Section 3.5 makes that encoding explicit in order to ease later transformations.

3.1. General Syntax

The Stencil compiler derives much of its information from the structure of the Stencil grammar. The input grammar includes definitions for all of the core constructs and contextual information used to derive auxiliary constructs. Figure 3.1 is an example program that constructs a scatter plot (line definitions refer to this example). The data model for Stencil is streams of tuples, so streams and tuples have declaration forms in the grammar. Stream declarations start with **`stream`** and provide a name and a tuple declaration (line 1 declares a stream named "Data"). Streams are only composed of tuples and all tuples on the stream conform to the tuple declaration. A tuple declaration is a list of field names; a tuple declaration appears as part of the stream declaration on line 1, and input and output tuple declarations for an operator appear on line 12.

A layer definition starts with **`layer`** and includes a name (see line 3). Stream processing occurs in rule groups that start with **`from`** followed by the stream name (see line 4). These rule groups are referred to as "consumes blocks" because the rules consume input from the named stream. Rules have three parts: an attribute, a binding operator and a transformation chain. Colon (`:`) is the basic binding operator; binding operators indicate that the elements on the left take the results of processing on the right. Chains are lists of operator calls, sequenced using the "link" operator denoted **`->`** (see line 7). An element to the right of a link must only depend on either global values or values to the left of the link (see Section 3.4 for details). The passing operator is a binding operation in the programming-languages sense; passing introduces new names into the environment. Each operator application introduces a tuple, bound to the operator name. To manage name shadowing, a name in square brackets may precede an operator application. In such cases, the result is bound to the provided name instead of the operator name. The member chains in a consumes block collectively define the tuples stored in the layer.

In addition to modifying layer properties, rules on a layer may target other contexts. Contexts change when execution and storage occur. The permissible contexts are (in evaluation order) **`prefilter`**, **`filter`**, **`local`** and normal layer rules.. This order enables the most common transformations to occur without requiring sequencing through more complex mechanisms.

Operators are either declared (see lines 9 and 10) or defined (line 12). Operator declarations start with **`operator`** and include a name and the base operator. These declarations provide named instances of operators for controlling when different transformation chains share memory. Operator definitions also start with **`operator`**, but then include an input and output tuple declaration and a list of operator rules (see lines 13 and 14). Operator rules start with a predicate, followed by the *gate* (**`=>`**) and then a rule. The operator rules list is like an if-then-else: only the rule associated with the first true predicate is evaluated.

Specializers are key/value pairings used to provide compile-time arguments. Specializers may appear next to declarations, like the one on line 9.

The common syntactic forms not used in Figure 3.1 are stream definitions, order declarations, and import declarations. Syntactically, stream definitions are like layers, except they start with **`stream`** and include a tuple declaration. Operationally, consumes blocks in a stream definitions put a new tuple onto a stream that may be an input to other consumes blocks later. Streams declared in the original Stencil program are referred to as *external* streams, while defined streams and stream declarations added by transformations are referred to as *internal* streams.

Order and import declarations mediate the interface between Stencil and the outside environment. Import declarations load modules into the namespace; modules then provide operators for use in the Stencil program. Import declarations start with **`import`** and give a module name. Import resolution occurs in lexical order; if operator names conflict, the last occurring definition is used. To reduce naming conflicts, a prefixing syntax is available for imports. In this syntax, the module name is followed by a colon and a prefix. If prefixing syntax is used, operators must be referred to with the prefix, followed by two colons and then the operator name.

Order declarations are less common than import declarations. The order declaration prioritizes the stream inputs for the dispatcher that handles sequencing. The keyword **`order`** initiates an order declaration, followed by a list of names, groups and relative priorities. (Discussion of predictability and the implemented degrees of control appear in Chapters 5 and 6.) A Stencil program requires exactly one order statement. If omitted, it is automatically generated.

3.2. Data Types

The Stencil language provides two data types: tuple and value. A tuple is a value, but a value is not necessarily a tuple. A tuple is an ordered list of values (and since tuples are values, nested tuples are permitted). All Stencil operations, except for operator construction and invocation, occur in terms of tuples. Tuples are analogous to flow-based programming's 'information packet' [**82**]. Operator invocation and construction unpack tuples to access the values.

Exact values are not the concern of data-flow models; instead, data-flow models focus on the connection between transformers. Therefore, data values only appear as components of tuples; the exact value (or even type) of a value is not a concern of the model. However, string, integer and real numeric values can be specified as literals because of their ubiquity. The precision and representation of numeric values is implementation-dependent. Values in tuples are not limited to those that may be literally specified; for example, an operator may return a java.awt.Color object but the grammar provides no explicit treatment for this data type (Section 6.1 discusses value types in more detail).

The positions of a tuple are often associated with names. For example, the position tuple in Cartesian coordinates has the fields X and Y. The names of fields in a tuple comprise its declaration. Syntactically, tuple declarations are lists of names enclosed in parenthesis. Tuple declarations enable name-based reference to values in tuples. Section 3.4 describes the resolution mechanisms and how name and index-based resolution (from Chapter 1) interact.

Breaking with the traditional tuple definition (e.g., that used in Haskell or Clojure [**63**, **73**]), tuples are not distinguished by their cardinality. Differing cardinality tuples can appear in the same location. In this way, they behave more like lists. However, variable cardinality tuples cannot have declarations, and may thus only have their values retrieved positionally (see Section 3.4). When a declaration is present, a tuple must conform to the provided declaration. If that declaration is part of the Stencil program (e.g., not supplied by an operator library), conformance is statically checked.

3.3. Operators and Modules

Operators perform transformations. Operators are collections of callable entities that potentially share state and must be instantiated. Operators are similar in principle to objects, but there is no subtype relationship (neither through classes nor prototype nor cloning). Each operator instance exists as a logically independent entity from all other operator instances. A *facet* is a callable part of an operator. Only facets that belong to the same instance of the same operator can share state.

Operators come from five sources. First, externally created operators reside in importable modules. Second, layer definitions introduce an operator with multiple facets. The operator takes its name from the layer and includes the facets for search, store and deletion. Three variations on search exist, but all can be concurrently supported. Third, stream definitions also implicitly define an operator that has one facet: *store*. Finally, operator definitions construct operators inside of a Stencil program.

Operator instances enable memory sharing between facets. Modules perform instantiation based on explicit parameters (found in specializers) and usage-context characteristics. Instantiation depends on where the operator definition appears. Each use site of an operator imported from a regular module incurs a separate instantiation. In contrast, operators created in the Stencil program join a special *ad hoc* module. Operators in the *ad hoc* module are singletons. Therefore, all use cites share the same instance by default. Adding the keyword **`template`** to an operator definition makes an *ad hoc* operator behave like other operators (one instance per use-site). Conversely, using an external operator in an *ad hoc* operator will cause multiple use sites to share a single instance.

3.3.1. Facets. Facets perform the actual transformation work, they constitute the nodes in a data-flow network. Modules and operators are support infrastructure for getting a facet (i.e., a transformation operator) into a transformation network. Facets actually perform transformations.

There are three core facets to Stencil operators that fulfill semantic functions: *map*, *query* and *stateID*. Map and query are counterparts to each other that fulfill the conditions listed in Definition 3.3.1. Informally, conditions (Stable) and (NoMutate) imply that the *query* facet must be a pure function. The result produced by the counterpart varies depending on the type of the codomain of the operator. If the operator codomain is continuous, then the result of the counterpart is the same as the original operator (condition (Continuous)). If the operator codomain is categorical, then the result of the counterpart is a peek at the current state (condition (Categorical)). If the input yields an error in the codomain, the result of the counterpart must be a value that the original operator never produces ((Error)). A single operator may have a codomain with any or all three of the above cases present. *Map* is the default transformation facet. *Query* is the default facet in filters, guards and other contexts where state changes are undesirable.

DEFINITION 3.3.1. *O.query is a counterpart to O.map if and only if, for all memory states* $\mathbf{M}_n$ *of O.map and for all x in the range of O, given* $O.map(x,\mathbf{M}_{k-1}) = (y0,\mathbf{M}_k)$, $O.query(x,\mathbf{M}_k) = (y1,\mathbf{M}_a)$, $O.query(x,\mathbf{M}_k) = (y2,\mathbf{M}_b)$ *and* $O.map(x,\mathbf{M}_k) = (y3,\mathbf{M}_{k+1})$ *then*

$$\begin{array}{lll}
\text{(Stable)} & y1 = y2 & \\
\text{(NoMutate)} & \mathbf{M}_k = \mathbf{M}_a = \mathbf{M}_b & \textit{and} \\
\text{(Categorical)} & \quad y1 = y0 & \textit{or} \\
\text{(Continuous)} & \quad y1 = y3 & \textit{or} \\
\text{(Error)} & \quad y1 \neq y3 \iff O.map(x,\mathbf{M}_k) \neq (y1,\mathbf{M}_l) \forall x \forall \mathbf{M}_k &
\end{array}$$

StateID is a means of tracking the occurrence of state changes. Each call to map *may* change the operator state. *StateID* tracks when changes have actually occurred. To fulfill this role, *stateID* must be a 1:1 function from the memory states of an operator to identifiers. If the *stateID* facet is not 1:1, some updates may not occur when needed (discussed more in Section 9.3).

Two optional facets discussed later are *state* and *discont*. *State* is an optimization presented and discussed in Section 9.4. *Discont* covers a special circumstance for guide creation discussed in Section 8.1.

3.3.2. Metadata.

Facets are the focus of most of Stencil's metadata. The most important pieces of metadata are (1) facet aliases, (2) return tuple declarations, and (3) memory behaviors. Additional module, operator and configuration metadata are also used by the implementation. Appendix B discusses metadata storage and loading.

Aliases. Aliases match a facet implementation to its name(s). This decouples the implementation name from the semantic role. For example, pure functions do not need distinct *map* and *query* implementations. Therefore, a single facet may serve both roles. The facet metadata would include both "map" and "query" in the alias list. All operators must supply a facet with the alias 'map' and a facet with the alias 'query'. Other facets are optional, though use of an alias with a semantically significant name will result in use of that facet as if it held the required semantics. (For example, not all operators need a *stateID* facet but if a facet is marked with that alias, it will be treated as if it fulfilled the role described in Section 3.3).

Tuple Declarations. Tuple declarations provide the necessary information about a facet's return values to provide name resolution (discussed in Section 3.4). Operators may supply a declaration of their return tuple. A compile-time tuple declaration is required if the operator's results are referenced by name. Null declarations are permitted (different from declarations with zero fields), but then tuple values can only be referred to positionally.

Memory behaviors. Valid facet memory behaviors are

Function: Relies on no mutable state.

Reader: May rely on mutable state, but will not modify it.

Writer: May rely on or mutate state.

Opaque: Memory behavior is not indicated or the map/query relation of Definition 3.3.1 is not satisfied.

To preserve semantic properties, a *query* or *stateID* facet should not be of the Writer behavior and preferably not of the Opaque behavior. The Opaque memory behavior acts as a barrier through which memory-dependent reasoning cannot be reliably performed. Typically, if the *map* facet is Opaque, the *query* facet is as well. Opaque facets appear in three circumstances. First, the operator is a link to some larger piece of analysis that may perform memory updates outside of the normal Stencil update cycle (e.g., in a separate thread). Second, some random element prevents reliable repetition. Third, some internal utility methods are marked Opaque to deliberately delimit analysis.

3.4. References and Name Resolution

Explicit identifier placement is a major part of the Stencil normal form. Resolving the implicit scoping rules depends on the context the name is used in. The contexts are (1) operator position, (2) left-side value, (3) right-side value, and (4) prefixed-right-side value.

Resolution of names in the operator position occurs in an environment determined by the imports and *ad hoc* operator definitions. When operator name shadowing occurs, *ad hoc* operators have highest precedence, then imports (with later imports of higher precedence) and finally default operators. Left-side values appear on the left side of a bind statement. The operator, layer or stream declaration containing the binding determines which names are valid as left-side values.

Right-side values are arguments to operators and appear on the right side of a binding. Identifiers in the right-side position refer to tuples. These may be multipart references (with parts separated by a dot). The initial identifier is first assumed to indicate a tuple from which values are taken. The majority of tuple names come from operator calls. When an operator call is made, it introduces a tuple into the environment. If the call is prefixed by a name in square braces, then the resulting tuple will be bound to that name. If no name is supplied, then the tuple will be given the same name as the operator that produced it (excluding the facet name). In addition to operator calls, tuples for the view and canvas states are always available, as well as tuples produced by **`local`** and **`prefilter`** statements (if present). The most recently bound tuple of the given name is used to resolve the remainder of the value. Fields may be referred to by index or by name. By default, if there are no subsequent parts in a tuple reference, then it is resolved to the first value of the tuple. Therefore, if a tuple named **`Sum`** were bound, then **`Sum`** and **`Sum.0`** are identical. To get the tuple itself as a value, **`Sum.*`** is used instead. If a reference is made by name, it is the responsibility of the resulting tuple to resolve the name.

For convenience, there are two deviations from the above rules. First, the underscore is a special right-side reference value. It resolves to the first value in the most-recently bound tuple. Second, if a name cannot be identified as a tuple, but is a field in the input of the context (the stream for a consumes block or the input prototype in an operator definition), then the name will be resolved to the input tuple.

Operator names, when used as arguments, are resolved in a different environment than tuple values. Prefixing a name with an **`@`** will change the resolution rules, causing the name to be resolved against the operator definitions instead of the tuple values. This is

Basic Input Grammar

PROGRAM ::= IMPORT* STREAM* LAYER* OPERATOR*
IMPORT ::= import ID (as ID)?
STREAM ::= stream ID TUPLE CONSUMES*
LAYER ::= layer ID TYPE CONSUMES+
OPERATOR ::= operator ID ID OPRULE+
CONSUMES ::= consumes ID ID RULE
OPRULE ::= APPLY **=>** RULE
RULE ::= rule ID APPLY*
APPLY ::= [ID] ID TUPLE
TUPLE ::= List of id's and primitive values

FIGURE 3.2. Basic Stencil grammar. This system is the basis for building complex forms.

```
stream Data(ID, predictor, percent)

layer Scatter[type:"SHAPE"]
consumes Data
  (ID): (#input.ID)
  (X): [#R4] &XScale2(#input.predictor) -> (#R4.0)
  (Y): [#R5] &YScale3(#input.percent) -> [#R7] &Limit6(#R5.0) -> (#R7.0)

operator &XScale2 : &Scale[min:0, max:100]
operator &YScale3 : &Floor[]
operator &GT8     : &GT[]

operator &Limit6(in) -> (out)
  (&GT8(in, 100)) => out: 100
  default         => out: #input.in
```

FIGURE 3.3. Scatter plot from Figure 3.1 in normal form.

similar to Common Lisp, with `@` instead of the "function" operator (or the abbreviated "#" syntax).

3.5. Normal Form

Figure 3.2 shows the core grammatical forms. The rough structure of the normal form is a tree where rules inhabit definitions and declarations provide global context. Rules are the basis for the data transformations described by the Stencil program. Before analysis or semantically significant transformations, input programs are put into this normal form. This normal form includes:

- Specializers in all applicable contexts.
- Naming all anonymous operators.

- Pack statements in all rules.
- Naming all results.
- Exact tuple value references specification (making tuple and field references explicit).
- Replacing constants with values.
- Lifting local and prefilter actions to stream definitions.
- Placing operator and value identifiers into the same namespace (done here with prefixes: # for generated values and & for operator names)
- Sequencing of multiple consumes blocks.

The normal form simplifies the transformations performed by future analysis. Also, by explicitly providing naming for all tuples and operators (in accordance with the rules in Section 3.4), this form enables α-conversion [**49**]. Figure 3.3 includes an example program in input and normal form. Stencil programs are not generally presented in normal form, but the transformations assume they will be operating on the normal form.

A non-obvious component of the normal form is that all operations are placed in unambiguous sequence in this form. The complete name resolution provides sequencing at the rule level. To provide explicit sequencing between consumes, a stream declaration for a unique name is created for each block . A stream definition is added that consumes the original stream and produces a tuple on each of the newly declared streams. These new streams are placed in the **`order`** clause in the lexical order of their corresponding blocks. After this transformation, ordering is no longer tied to lexical order. An example of this transformation is shown in Figure 3.4.

```
import JUNG

stream VertexList (parent, child)

stream Prime ()
from VertexList
  () : Layout.add(parent,child)

layer Nodes
from VertexList
    ID: child
    REGISTRATION : "CENTER"
    FILL_COLOR: Color{BLUE}
    (SHAPE, SIZE) : ("ELLIPSE", 5)
    (X,Y) :* Layout.query(child)

layer Edges["LINE"]
from VertexList
    ID: Concatenate(parent, child)
    (X1, Y1):* Layout.query(parent)
    (X2, Y2):* Layout.query(child)
    PEN_COLOR: Color{GRAY30,80}

layer Labels["TEXT"]
from VertexList
    ID : child
    (X,Y):* Layout.query(child)
    TEXT: child
    FONT: Font{10}
    COLOR: Color{GRAY60}

operator Layout : BalloonLayout
```

(a) Before sequencing

```
import JUNG

stream VertexList (parent, child)
stream #VertexList_1(parent, child)
stream #VertexList_2(parent, child)
stream #VertexList_3(parent, child)
stream #VertexList_4(parent, child)

stream #VertexList_0()
from VertexList
  (): #VertexList_1.Store(VertexList.*)
     -> #VertexList_2.Store(VertexList.*)
     -> #VertexList_3.Store(VertexList.*)
     -> #VertexList_4.Store(VertexList.*)

stream Prime ()
from #VertexList_1
  () : Layout.add(parent,child)

layer Nodes
from #VertexList_2
    ID: child
    REGISTRATION : "CENTER"
    FILL_COLOR: Color{BLUE}
    (SHAPE, SIZE) : ("ELLIPSE", 5)
    (X,Y) :* Layout.query(child)

layer Edges["LINE"]
from #VertexList_3
    ID: Concatenate(parent, child)
    (X1, Y1):* Layout.query(parent)
    (X2, Y2):* Layout.query(child)
    PEN_COLOR: Color{GRAY30,80}

layer Labels["TEXT"]
from #VertexList_4
    ID : child
    (X,Y):* Layout.query(child)
    TEXT: child
    FONT: Font{10}
    COLOR: Color{GRAY60}

operator Layout : BalloonLayout
```

(b) After sequencing

FIGURE 3.4. Explicit consumes block sequencing, combined with exact tuple value specification, remove all dependency on lexical order on execution.

4

Functional Reactive Programming and Visualization

Formal semantics provide a basis for establishing framework properties and evaluating proposed abstractions. The properties of interest are determinism, bounded resource consumption and consistency. Informally, determinism indicates that only the inputs matter in determining the resulting visualization. Bounded resource consumption indicates that the Stencil framework does not preclude a visualization because of framework overheads. Consistency indicates that the components of a visualization produced are derived from the same data. In a consistent visualization, the individual parts are comparable and conclusions drawn from the visualizations are valid for the data represented. Later transformations, by reducing down to basic operations, are shown to preserve these properties. Having a definition for Stencil that demonstrates these properties allows work in Stencil to be extended to frameworks. The formal semantics effectively define Stencil independent of any particular implementation.

The data-flow model of visualization, as described by Chi [**23**], is an expression of the dependencies between data transformations. It is underspecified for the purposes of program transformations because it does not prescribe an order of operations. For this reason, it is unsuitable for establishing useful properties about programs that can guide transformations. This chapter presents a correspondence between a subset of E-FRP and the data-flow model of visualization in Section 4.6. That correspondence is then used to present Stencil's formal semantics in Chapter 5.

4.1. Stencil Requirements

The desired properties for constructing a visualization framework are (1) dynamic updates, (2) deterministic execution, (3) bounded resource consumption, (4) interoperability, and (5) internal visualization consistency.

The full data for a visualization are not necessarily available when a visualization is first produced. Any visualization over "live" data must deal with temporally distributed updates. Any visualization that involves user interaction is inherently over live data. Interactivity is a hallmark of information visualization, so dynamic updates are a significant requirement for any framework.

Deterministic execution is desired because a visualization should be reproducible. The input and the process combine to create an image. Modifying the inputs or the process produces new images, but repetitions on the same input and process should return the same results. Nondeterminism undermines the ability to reproduce results. Some visualizations include randomness or intentionally nondeterministic operations; for example, spring-force embeddings often involve random iteration orders or jitter to escape local minima. These deviations from deterministic execution are useful, but are exceptional cases. Permitting such explicit inclusion of nondeterminism is acceptable, but the default behavior should be deterministic.

Creating a visualization requires resources beyond those of simply storing data and images. The analysis steps that go into the visualization often have their own bookkeeping. For example, calculating a maximum requires storing the current maximum, occupying additional space. Additionally, time is spent calculating contents of a visualization, even if all transformations are pure functions. Bounding the framework's resource consumption allows an implementation-independent evaluation of a visualization's feasibility. Resource consumption linearly bounded with respect to the number of data points loaded was the target for our semantic investigation. As in deterministic execution, some analysis may not conform to the target bounds. This should be permissible, but not inherently imposed by the framework itself.

Visualization algorithms are often released as libraries. Custom visualization designs often require custom processing that modifies existing definitions. An accessible visualization system needs to be able to employ these existing algorithm implementations without requiring a reimplementation. This implies that the semantic model needs to handle foreign functions in some fashion. Particularly, stateful operations need to be permissible because many transformations retain internal state.

Consistency indicates that a visualization represents the data, as it was understood at a certain point in time. If a data point has had any influence on the visualization state, it should have had its full influence. If this does not hold, then the visualization will have subcomponents that cannot be directly compared. Consistency requires that the extent of a data point's influence can be measured or controlled. (A more formal definition of consistency is given in Section 4.7).

These properties led to an investigation of data-flow frameworks. E-FRP, a variant of FRP, can directly support the five desired properties.

4.2. Functional Reactive Programming Overview

FRP is a domain-specific language (DSL) for synchronous data-flow networks [**117**, **119**]. Implementations exist in Haskell [**92**, **117–119**], Scheme [**42**] and Java [**34**]. The core concepts of functional FRP are (1) behaviors, (2) events, and (3) FRP combinators. Behaviors are time-varying values (later FRP nomenclature changed *behavior* to *signal* [**35**]). Behaviors include constants and functions that take time as a parameter. An event is a value at a specific time. Events include mouse/keyboard information or the value of a behavior as sampled at a particular time. The general form of a combinator is behavior $\times$ behavior, though combinators for event/behavior combinations have been defined. Combinators permit behaviors and events to have values that depend on each other. Denotational semantics of FRP have been developed [**43**] and Real-Time Functional Reactive Programming (RT-FRP) describes a deterministic and resource-bounded subset [**118**]. Resource-bounded means that the response to any given event will only require finite time and memory.

FRP makes two broad assumptions. First, all operations performed are pure functions. All state-related information must be computable based upon time, behavior values and event values. This enables a program to respond to its running time in a continuous fashion. Unfortunately, using the continuous time parameter can lead to so-called "time leaks" where all prior times must be considered before the current time can be. Attempting to avoid time leaks using memoization can lead to memoization tables that grow without bound; this situation is referred to as a space leak.

The second assumption of FRP is that the system will be responding to dynamic data, i.e., that it is "reactive" to its environment. A reactive system can be seen as one that responds to external entities that determine which computations will occur. Such systems can be structured around callbacks (as is the case with Frappé [**34**]), but the complexity of dynamically changing callbacks is difficult to manage. Therefore, the callbacks are often hidden under a layer of abstraction or are only part of the conceptual model, not the implementation [**27**].

E-FRP is based on FRP, but E-FRP discretizes time by assuming that behavior values only change when an event occurs. This eliminates space and time leaks and provides a natural way to handle stateful behaviors (state changes conceptually occur *between* events). The result is that E-FRP programs can eliminate space and time leaks in a straightforward fashion. The subset of E-FRP used to formalize Stencil is also a subset of FRP, and, therefore, retains the deterministic semantics of FRP [**119**]. Other discrete reactive models can be found in Signal [**53**], ESTEREL [**7**] and RT-FRP [**118**]. Signal and ESTEREL use clock calculi to determine when updates are permitted. The clock calculi add complexity to the overall system by requiring timing information on all operators and input sources. RT-FRP logically constructs a global clock and requires all event production to be in integer multiples of that clock frequency. Both the clock calculi and global clock techniques require the whole program to have a global notion of time, and all operations occur in response to that global clock. The E-FRP discretization allows statically separable parts of the program to run in independent time, only synchronizing on less frequent epochs [**119**].

The general execution model of E-FRP is shown in Figure 4.1. The results of an E-FRP program are a new E-FRP program and a set of key/value pairs (referred to as a *store*). All execution is in response to events received by the system from the "outside." E-FRP strictly enforces that events occur one at a time. Therefore, some means of selecting the next event to process must be employed. The sequencer and dispatcher work together to provide that ordering. The sequencer is essentially a priority queue that gathers events and presents them individually to the dispatcher. The dispatcher is the gatekeeper to the system, accepting an event from the sequencer whenever the E-FRP program is not currently processing an event. After accepting an event, the dispatcher executes the E-FRP program, accepts the results and distributes them appropriately. The major distinction between this model and other reactive systems is the restriction that events occur exactly one at a time. This restriction can be relaxed if there are regions that do not share memory, using a dispatcher for each disjoint section [**119**]. The E-FRP program execution is deterministic with respect to the events *accepted* by the dispatcher.

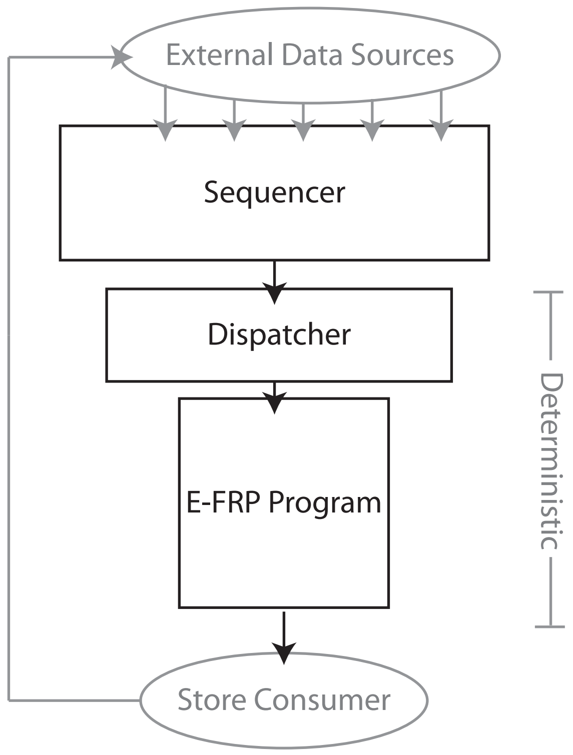


FIGURE 4.1. Event-driven Functional Reactive Programming system block diagram. External streams provide data to the sequencer. The sequencer provides data to the dispatcher. The dispatcher feeds the analysis engine. The analysis engine produces a store full of name/value pairs that may be consumed by external entities to produce new events.

4.3. Event-driven Functional Reactive Programming Syntax

Figure 4.2 defines the E-FRP syntax. An E-FRP program, P, is a set of mutually recursive behavior definitions. Free-variable rules are straight forward, except for reactive behavior definitions. The only binding form in the language is r, the reactive behavior definition. r is also the central rule for performing work in an E-FRP system. A reactive behavior, $r \equiv init\text{id}{=}\text{c}in\{E_i \Rightarrow d_i\}$, initially has value c and changes to the value of d_i when event E_i occurs. The free-variable rules indicate that x can occur free in d_i. In such cases, x is bound to the old value of r, allowing the new behavior definitions to respond to the old value of the definition. Since r is changing values on event E_i, each E_i for a given r must be unique in H, though multiple behaviors can define handlers for the same event. For

Program	P	::=	$\{id_i = b_i\}$
Behavior	b	::=	$r \mid d$
Non-reactive	d	::=	$id \vert c \vert f\langle d_i\rangle$
Reactive	r	::=	init id = c in H
Handler	H	::=	$\{E_i \Rightarrow d_i\}$
Identifiers	id	::=	Variable names
Constants	c	::=	literal values
Functions	f	::=	Primitive functions
Events	E	::=	Event identifiers (predefined, finite set, scoped independently)

$$FV(x) \equiv \{x\} \qquad FV(c) \equiv \emptyset \qquad FV(f\langle d_i\rangle) \equiv \bigcup_i FV(d_i)$$

$$FV(\text{init } x = c \text{ in } \{E_i \Rightarrow d_i\} \equiv \bigcup_i FV(d_i) - \{x\}$$

FIGURE 4.2. Event-driven Functional Reactive Programming grammar without now/later phases and free-variable rules.

simplicity, it is assumed that an event does not carry a value with its occurrence. If a value is required for an event, a primitive function can be provided to give the associated value. A primitive function, f, is any function defined in the surrounding context.

4.4. Event-driven Functional Reactive Programming Semantics

As indicated in Section 4.3, an E-FRP program is a set of mutually recursive behaviors that evaluate in response to events. The evaluation rules are given in Table 4.1 with an informal description of each rule in Table 4.2. The Whole Network rule indicates that when an event occurs, a program evaluates to a pair: a store and a new network. This implies two parts to any evaluation, one that produces values for the store (the Vx rules) and one that produces rules (the Rx rules).

A store is defined as a set of mappings from identifiers to values ($S ::= \{id_i \mapsto c\}$). The set of Vx value-rules indicates that the store will contain exactly one entry for each variable bound in a reactive rule. Rule V5 is the critical value production rule. It provides the mechanisms for both responding to an earlier state and determining the values produced. Response to earlier state is achieved by substituting the initial value of x when evaluating the selected handler response d. When an event occurs, any behavior with a handler

$P \vdash b \overset{I}{\rightharpoonup} c$	Behavior values
	V1 $\dfrac{}{P \vdash c \overset{I}{\rightharpoonup} c}$
	V2 $\dfrac{\{I \Rightarrow d\} \notin H}{P \vdash \text{ init } x = c \text{ in } H \overset{I}{\rightharpoonup} c}$
	V3 $\dfrac{P \vdash b \overset{I}{\rightharpoonup} c}{\{x = b\} \in P \vdash x \overset{I}{\rightharpoonup} c}$
	V4 $\dfrac{\{P \vdash d_i \overset{I}{\rightharpoonup} c_i\} \qquad \text{prim}(f, \langle c_i \rangle) \equiv c}{P \vdash f \langle d_i \rangle \overset{I}{\rightharpoonup} c}$
	V5 $\dfrac{P \vdash d[c/x] \overset{I}{\rightharpoonup} c'}{P \vdash \text{ init } x = c \text{ in } \{I \Rightarrow d\} \uplus H \overset{I}{\rightharpoonup} c'}$
$P \vdash b \overset{I}{\rightharpoondown} b'$	Behavior changes
	U1 $\dfrac{}{P \vdash d \overset{I}{\rightharpoondown} d}$
	U2 $\dfrac{\{I \Rightarrow d\} \notin H}{P \vdash \text{ init } x = c \text{ in } H \overset{I}{\rightharpoondown} \text{ init } x = c \text{ in } H}$
	U3 $\dfrac{\{I \Rightarrow d\} \in H \qquad P \vdash d[c/x] \overset{I}{\rightharpoonup} c'}{P \vdash \text{ init } x = c \text{ in } H \overset{I}{\rightharpoondown} \text{ init } x = c' \text{ in } H}$
$P \vdash b \xrightarrow{I} c; b'$	Single FRP operator
	Single Op $\dfrac{P \vdash b \overset{I}{\rightharpoonup} c \qquad P \vdash b \overset{I}{\rightharpoondown} b'}{P \vdash b \xrightarrow{I} c; b}$
$P \xrightarrow{I} S; P'$	Whole network
	Network $\dfrac{\{\{x_i = b_i\}^{i \in K} \vdash b_j \xrightarrow{I} c_j; b'_j\}^{j \in K}}{\{x_i = b_i\}^{i \in K} \xrightarrow{I} \{x_i \mapsto b'_j\}^{i \in K}}$

TABLE 4.1. Event-driven Functional Reactive Programming (E-FRP) semantics used to formalize Stencil. This is a subset of the full E-FRP semantics, selected from the full E-FRP semantics because it corresponds to a deterministic subset of Functional Reactive Programming semantics. Informal interpretations are given in Table 4.2.

V1	Non-reactive behavior c does not change on event I.
V2	A reactive behavior with no handler for I does not change when I occurs.
V3	If variable x is defined by behavior b, then when b goes to c due to I, x goes to c.
V4	A *lifted* behavior. If primitive f is applied to behavior d, then when d responds to event I producing c_i, the lifted behavior takes the value c if $f(c_i) = c$.
V5	If a reactive behavior is defined in terms of event I and event I occurs, then the value of the reactive behavior is determined by substituting the current value (denoted c) for x (which can occur free in d). The reactive behavior returns value c' if $d[x := c] = c'$.
U1	Non-reactive behaviors do not take on a new set of handlers when event I occurs.
U2	Reactive behaviors without a rule defined on event I do not take on a new set of handlers when event I occurs.
U3	Reactive behavior change defined on I updates its state handlers on I by substituting c for x in d. This affects what may happen when the next event occurs.
Single Op	When an event occurs, the E-FRP operator will return a value and be replaced with a new behavior (even if the new is just a copy of the old).
Network	When an event occurs, the current value of all behaviors is returned in S and a new set of behaviors is created.

TABLE 4.2. Event-driven Functional Reactive Programming rule interpretation. Formal rules are given in Table 4.1.

defined will evaluate a new value. Since only one handler per event is permitted, each behavior can respond at most once to each event. This helps keep E-FRP deterministic by allowing only one change to a value per event. However, event reactions may rely on other behaviors (reactive or non-reactive), having chains of changes possible. The single-response restriction and a lack of additional sequencing indicators implies that such chains must be acyclic. Full E-FRP uses a tag to indicate a phase for updates to be visible, either *now* or *later*. However, *now* updates may not depend on *later* updates and each phase must be acyclic. The *now/later* phasing was motivated by their original application model, is not known to be consistent with FRP semantics and is not sufficiently general. A more general technique for cycle breaking is presented in Section 4.6.1.

With acyclic event-response-chain groups, E-FRP semantics for value production are deterministic. Given an event and a program, any order of event response executions that respects the chain sequences will yield the same result. Independent chains may be arbitrarily interleaved. Rules $V1$–$V3$ indicate when values stay as their defaults.

In addition to producing a store, an E-FRP program also produces a new E-FRP program. This newly produced program will be executed in response to the next event. The rule-update rules are labeled U in Table 4.1. Rule $U3$ is the functional counterpart to rule $V5$, indicating a reactive behavior can define a new behavior. The substitution expression in $U3$ uses the value specified from the prior round (the initial value) to define behaviors for the next round. This is the same substitution used in $V5$. Since both substitution rules use the prior round value, rule updates and value production logically occur at the same time.

4.5. Event-driven Functional Reactive Programming and Visualization

E-FRP has properties for a visualization framework. Recall from Section 4.1, the desired properties are (1) dynamic updates, (2) deterministic execution, (3) bounded resource consumption, (4) interoperability, and (5) internal visualization consistency. Consistency is deferred to Section 4.7, as the definition is closely tied to the visualization domain. Events directly encapsulate dynamic updates, as is trivial to demonstrate.

Deterministic execution and resource consumption directly relate to how interoperability is approached. Interoperability is desired to enable reuse of existing visualization algorithms without requiring reimplementation. E-FRP represents all stateful operations as reactive behaviors that can be redefined in response to events (using rule $U3$ in Table 4.1). This enables stateful operators to be wrapped directly as reactive behaviors based on primitive function calls. In this form, the stateful operator inherits the single-response and acyclic-paths restrictions of reactive behaviors. The only restriction is that the stateful operator must present a consistent memory state for the entire time an event is being handled, only changing state “between” events. This is the essence of *let-snapshot* in RT-FRP [**118**]. This can be achieved with memoization without inducing a space leak because the memoization table is bounded in size by the number of nodes in the network and can be discarded between events. In this way, E-FRP enables interoperability with minimal facades for existing code.

The deterministic and resource-bounded guarantees of E-FRP are conditional on well-behaved primitive functions. If the primitive functions are deterministic and resource-bounded themselves and present a stable state for the duration of event processing, then the proofs for FRP and RT-FRP semantics can be applied to E-FRP. However, if a primitive function is nondeterministic or not resource-bounded, this property will "leak" into the rest of the system. Therefore, E-FRP programs are *conditionally* guaranteed to have these properties. What can be guaranteed is that the E-FRP framework itself is deterministic and that the framework itself has resource consumption linearly bounded in the number of events. When used to compose deterministic and resource-bounded primitive functions, E-FRP programs do not loose those properties.

The principle difference between FRP and E-FRP is the treatment of time. However, the FRP continuous time semantics lost in E-FRP are not important to the visualizations addressed in this dissertation. Moreover, a highly time-dependent scientific visualization framework did not find practical use for the FRP notion of time [**41**].

With dynamic updates, interoperability and a conditional guarantee of deterministic execution and resource-boundedness, E-FRP provides all of the desired properties for the semantic basis for visualization frameworks, including Stencil.

4.6. Data-Flow Model of Visualization

The data-flow model of visualization is a set of principles upon which frameworks can be implemented. Its principles are shared with data-flow programming in general: transformation operators and expression of dependence. All data-flow programming environments, for visualization or otherwise, express a dependency graph between operations. Figure 4.3 provides an example data-flow network that computes a scatter plot. Nodes represent operators and links represent data dependencies with the sink as the arrow target. However, each implementing framework is free to provide its own semantics for the interpretation of that graph. This overt lack of implementation-independent semantics

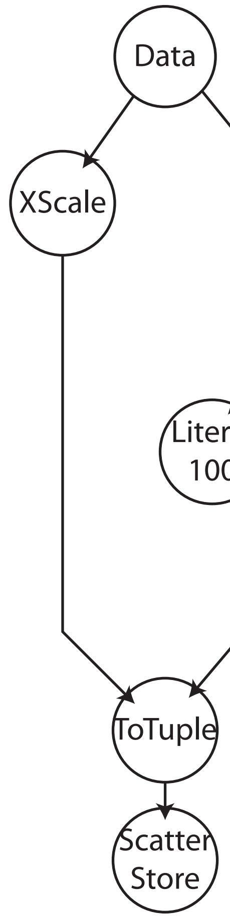


FIGURE 4.3. A simple plot program as a data-flow network. This program places points based on scaling of two variables, one of which is also capped at 100.

was noted by Chi [**23**], but not considered significant given that dependency and transformation were all that was considered in that work.

E-FRP is directly related to data-flow programming. Each node in a data-flow graph is a behavior. Data-source nodes are necessarily reactive behaviors. Sink nodes are not distinguished syntactically or semantically. The data-flow network may branch or merge based on the multiple dependencies in either direction that may be expressed in event handlers. A behavior may participate in multiple chains either through responding to multiple events or by having multiple behaviors depend on it.

The relationship between E-FRP programs and data-flow networks provides a foundation on which visualization frameworks can be compared, independent of implementation artifacts. The ability to compare frameworks extends to data-state–based frameworks as well, given their proven correspondence [**23**]. However, such a comparison relies on a formal expression of the behaviors of the target framework. This expression need not be a complete description of the behavior involved. For example, the Stencil language is defined in terms of a deterministic data-flow framework. Furthermore, it does not currently include any operators that modify the transformation network's topology in reaction to presented data (though such actions are permitted in E-FRP). Therefore, any framework that includes nondeterminism or topology mutation will have higher objective expressiveness. It can be argued that Prefuse [**69**] falls into the "more expressive" category, given its heavy use of reflection and full access to Java language facilities that permit the potential for dynamic reconfiguration of the visualization schema. However, most of those facilities are provided by the Prefuse implementation environment (i.e., Java), and are, therefore, only questionably "part of" the of the framework.

4.6.1. Cycle Breaking. The full E-FRP semantics include a simple sequence indicator to enable two-phase updates. This method is insufficiently general to break nested cycles or resolve cycles in chains with multiple cycles. Breaking cycles more generally involves the dispatcher.

In the standard E-FRP framework, cycle breaking is achieved with a two-phase value update cycle. Operators may either have their values change on an event *now*, thus all dependencies see the new value, or *later* and have all dependencies see the old value until the next event occurs. However, now/later phasing is not known to be consistent with FRP semantics [**119**] and is also not known to be deterministic. Furthermore, now/later phasing only allows simple cycles to be removed. Nested cycles or chains of cycles are still inaccessible. In order to support cyclic transformation chains while remaining in the FRP subset of E-FRP, the dispatcher can be used to create explicit sequences. Simply put, all cycles must include the dispatcher. This creates a *de-facto* now/later division because,

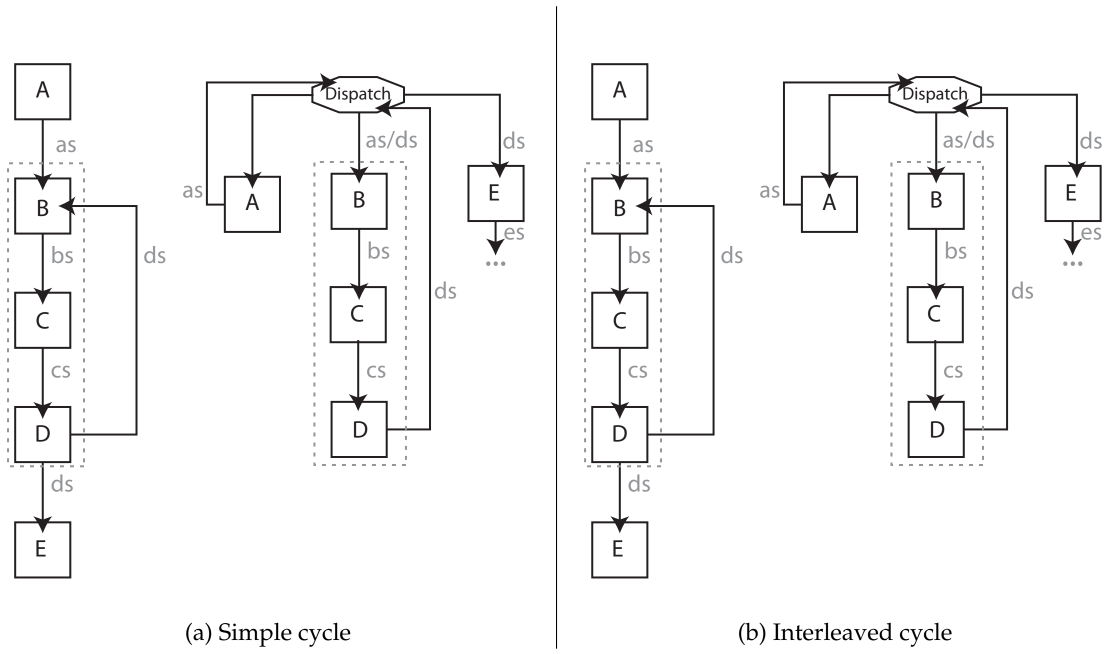


FIGURE 4.4. Cycle breaking cases. Applied recursively, (A) simple and (B) interleaved cycle breaking can reform graphs so that cycles always loop to the dispatcher.

in accordance E-FRP semantics, operator values are updated between event dispatches. Examples of cycle breaking using the dispatcher are given in Figure 4.4. This method is consistent with synchronous data-flow, but only represents the simple case where all operators have a delay of size one [**75**]. All buffering must be handled by the operator itself (an operation supported through the Range operator, discussed in Chapter 7). Certain calculation effects also require appropriate use of the **`order`** keyword to coordinate the streams that constitute subcomponents of the calculations.

By involving the dispatcher, the visibility and timing of memory updates is made explicit. This is analogous to the effects of *let-snapshot* in RT-FRP . let-snapshot defines a memory state against which the contained set of operations is performed. Since recursive RT-FRP behaviors must be tail recursive [**118**], let-snapshot effectively delimits the retention period for memory states. The presented cycle-breaking semantics use the dispatcher to delimit both ends of the memory window instead of let-snapshot at the start and a tail call at the end.

Conducting cycles through the dispatcher resolves the order and visibility of updates; it does not eliminate the possibility of infinite loops. It is possible to loop through the dispatcher in such a way that the dispatcher queues are never empty. This is directly achieved by having a stream definition that echoes any value it receives back onto itself. The possibility of infinite loops does not destroy the resource-boundedness guarantee though because those bounds are defined with respect to event dispatching. Any individual event still requires bounded time and memory to address [**118**] (provided the behavior conditions from Section 4.2 are met), but may introduce new events through the sequencer.

4.7. Consistency

To properly evaluate the claims made in later chapters, it is important to establish a notion of *consistency* for a visualization. For a visualization to be interpreted as a whole, it must be internally consistent. Informally, consistency indicates that (1) all data loaded have completed all of their effects, and (2) that all effects of data loaded are included in the rendering. An inconsistent visualization includes partial effects of some data and cannot be cohesively interpreted. The cycle-breaking techniques of Section 4.6.1 enable an event to have multiple effects, but divides and sequences those effects into smaller internal steps. A visualization is not consistent until all of these smaller steps are complete. Visualization frameworks typically provide for consistency by making analysis and rendering mutually exclusive phases [**11, 22, 45, 68, 103**].

The notion of consistency must be formalized to enable later reasoning. To assist in the definition of consistency, the we use a global state tracker (GST) as an auxiliary function. The GST tracks how many state changes have occurred in the system. The value of the GST is the global state indicator (GSI), indicating how many state changes have occurred thus far during processing. Analysis results are associated with the current GSI value when stored. The exact representation of the GST is not significant. The important properties of the GST are that its sequence (1) is repetition free, and (2) progresses with each state change. Counting the number of events dispatched and wall-clock time both satisfy these conditions but are pessimistic (not all events or milliseconds include state changes). A

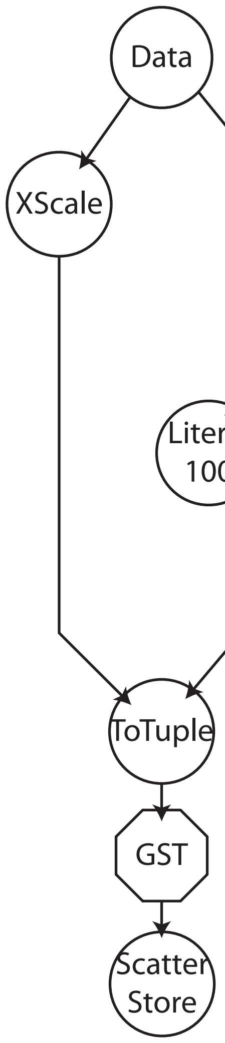


FIGURE 4.5. Scatter plot program from Figure 4.3 with global state tracker call added.

more exact GST is developed in the rest of this section. For the purposes of description, it is assumed that all computations on one tuple complete before the next tuple is introduced. Therefore, there are no partial states in the system. This strict phasing need not be maintained in practice, provided the sequential semantics are still respected.

Semantically, the GST is treated as a stateful primitive function. Any time a potentially stateful transformer changes state, the GST transformer must also be invoked at least once earlier in the chain. This can be achieved by inserting the GST transformer into the transformation network as a side-branch after each stateful transformation. Using the simple scatter plot network from Figure 4.3, a GST call has been added in Figure 4.5. Identification of stateful transformations and related GST placement are implementation details discussed further in Section 6.6.

Assuming that GST transformers is appropriately placed, consistency is defined in terms of a collection of relations among the visual variables (position, color, shape, etc.) in groups of rendered elements (e.g., points in a scatter plot). A visual variable in a group is consistent if all of the entities in the group belong to one of two relations:

Sequence: Each of values of the variable corresponds to a unique GSI value.

Snapshot: All the values of the variable are associated with the same GSI.

Informally, the consistency relationship for attributes states that if any two attribute values came from the same analysis state, then they must all have come from the same analysis state. The sequence relation is required when the load order is significant. This occurs when the context of a value depends only on values that have come before it (e.g., running maximal value). The snapshot relation is used whenever (1) the extent of the value(s) represented on a visual variable is statically known, or (2) dynamic data are used and global relationships are illustrated (e.g., current value versus global maximum). Constant-valued visual variables are automatically consistent according to either relation, but will be treated as snapshot consistent in each global state for simplicity in comparisons.

A group of rendered elements is consistent if all visual variables in the group are consistent and one of the following conditions holds for all pairs of variables.

Sequence $\times$ **Sequence:** Two sequence-consistent groups are automatically consistent with each other.

Sequence $\times$ **Snapshot:** The greatest GSI in the sequence-consistent variable is less than or equal to the GSI for the snapshot-consistent variable.

Snapshot $\times$ **Snapshot:** The GSIs are the same.

Groups are consistent with each other if all pairs of visual variables between the groups are consistent according to the same rules as intragroup consistency. Finally, a visualization is consistent if all groups are consistent with each other.

4.7.1. Validity of Consistency. In a consistent visualization based on valid analysis, the parts of the visualization can be safely compared to each other because all represented values have had the opportunity to be influenced by the same data. This conforms to the intuitive notion of visualization interpretability while providing a basis to evaluate program transformations. This section presents arguments for the validity of the consistency relationship with respect to visualization interpretability. Snapshot consistency will be evaluated first, followed by sequential consistency.

At the attribute level, snapshot consistency is applicable when a visual element is based on a globally computed value. For example, snapshot consistency applies if fill color is based on a global maximum. In this case, all earlier computed fill colors may need to change in response to a value loaded later. Assume the snapshot consistency property is violated such that a subset of fill colors is based one global maximum, while a disjoint subset is based on a different global maximum. In such a circumstance, the two subsets cannot be visually compared. However, the distinction between the sets may not be clear. Therefore, the visualization cannot be safely interpreted when snapshot consistency is expected but not provided. The same argument extends to the intergroup, intragroup and full visualization consistency for Snapshot $\times$ Snapshot comparisons.

Sequential consistency is applicable when a visual element depends only on computations performed thus far. For example, sequential consistency applies if size depends on a running total. In the simplest case, if size were so defined, an element's size would never change after it is initially set. To conform to concurrent computing notions of consistency, a series of state changes in an operation must be ordered [**110**]. The results of calculations based on those updates can be put into a corresponding order. The GST tracks the order of those updates. If two size definitions are ambiguously ordered (e.g., have the same GST), it is unclear which reflects the state updates of the other. This constitutes a violation of the rules of consistency from concurrent computing. Furthermore, it is a violation of snapshot consistency: it is unclear how to compare such values to each other.

Sequential consistency does not require that a visual value in a group represent each GSI value in a range. Values may be missing for two reasons. First, the GST changes globally for any state change made. Therefore, not all GST updates are in response to inputs that impact any given group. Second, an item may have been created and later deleted. This second case also covers frameworks that permit case-by-case updates to existing values based on new input; element mutation can be achieved by looking up an old value, deleting it and then adding a new element with properties derived from the deleted item.

Consistency between snapshot- and sequence-consistent visual attributes requires that the snapshot be computed at the same time or after the last change in the sequence-consistent group. Assuming again that color is based on a global maximum and size is based on a running total, if the maximum and total are related to the same input field then the color and size can only be interpreted with respect to each other if the stated relationship holds. If the element sizes are based on a later GSI than the colors, then if a subset includes a reaction to a new maximum value, then it cannot be interpreted with respect to the color.

The consistency relationship as presented provides a lower bound on interpretability of a visualization. In some ways, snapshot consistency is concerned with comparisons "in the large," while sequence consistency is concerned with pair-wise comparisons. Intelligent implementation of the GST enables computational savings by helping determine when snapshot-consistent variables need to be updated. (Details on the GST implementation can be found in Section 6.6.)

4.8. Conclusions

E-FRP provides the desired properties for formalizing the Stencil framework. It provides convenient ways to represent and respond to dynamic data while remaining deterministic and resource-bounded. E-FRP also provides a sufficient framework to discuss visualization consistency in a testable fashion. Additionally, E-FRP also provides a means to employ externally defined functions. These external functions may be nondeterministic or many not be resource-bounded, but they do so in an isolated fashion. E-FRP has been used to provide a formalization of the data-flow model of visualization. Chapter 5 will establish Stencil semantics using the tools defined in this chapter.

5
Stencil Semantics

This chapter establishes formal semantics for the Stencil language. These semantics are established by translation to E-FRP, described in Section 5.1. Though E-FRP semantics cover the majority of the visualization semantics in a direct fashion, special consideration is given to rendering in Section 5.2.1.

Through the formal specification, Stencil is established as a resource-bounded, deterministic and consistency-preserving framework. These properties are used through the rest of this dissertation to ground discussion of Stencil's analytical, representational and computational abstractions.

```
stream Data(ID, predictor, percent)

layer Scatter
from Data
  ID: ID
  X: XScale(predictor)
  Y: YScale(percent) -> Limit(_)

operator XScale : Scale[min:0, max:100]
operator YScale : Floor

operator Limit (in) -> (out)
  (in > 100) => out: 100
  (default)  => out: in
```

(a) Input Form

```

    (Data, #R1, &Data.get())
    (Data, #R4, &XScale2(#R1.1))
    (Data, #R5, &YScale3(#R1.2))
    (Data, #R7, &Limti6(#R5.0))
    (Data, #R8, &ToTuple4(#R1.0,#R5.0,#R7.0))
    (Data, ##Scatter, &Scatter.store(("ID","X","Y"),#R8))

operator &Data &Dispatcher[stream:"Data"]
operator &Scatter &Layer[type:"SHAPE"]
operator &ToTuple4 &ToTuple
operator &XScale2 &Scale
operator &YScale3 &Floor
operator &Limit6 &#Limit          *@/
```

(b) Stencil-Triple Form

FIGURE 5.1. Input and Stencil-Triple forms of a scatter plot based on transforming two input variables.

5.1. Stencil to Event-driven Functional Reactive Programming

E-FRP in general, and the specific subset used for Stencil, were introduced in Chapter 4. The three E-FRP elements significant to Stencil are events, reactive behaviors and non-reactive behaviors. Events indicate that something has happened. Reactive behaviors change in response to events. Non-reactive behaviors either do not change or depend only on other behaviors. Roughly speaking, all Stencil stream declarations are converted into (1) an event that signals a change, and (2) a behavior that accesses the values associated with the event through a primitive function. All operators become reactive behaviors that often depend on other behaviors in addition to events.

Transitional to E-FRP

$$\text{P} :: \textsc{Program} \rightarrow \textit{Program}$$
$$P[\![\textsc{Streams}^{*}\ \textsc{Triples}^{*}\ \textsc{Operators}^{*}]\!] = T[\![\text{Triples}]\!]^{*}$$

$$\text{T} :: \textsc{Triples} \rightarrow \textit{Reactive}$$
$$T[\![\text{event, id, op}]\!] = \text{tag} = \textit{init}\ x = \textit{NULL}\ \textit{in}\ \{\ \text{event} \Rightarrow \text{op}\ \}$$

FIGURE 5.2. Translation rules for Stencil-Triple to Event-driven Functional Reactive Programming.

The semantics of Stencil are represented with respect to a transformation on the Stencil normal form called the Stencil-Triple form. In Stencil-Triple form, all operators are translated into a Triple. The first element of the triple is the stream to respond to. The second element of the triple is the results name from the normal form. The final element of the triple is the operator to invoke, including its arguments. Operator declarations are preserved in Stencil-Triple form as they determine the free variables. Stencil-defined operators are generally encapsulated and made to appear as all other operators (though inlining and primitive function substitutions are standard E-FRP treatments when possible [**119**]). Stream declarations are translated into operator declarations, using a special "Dispatch" operator (discussed below). In essence, Stencil-Triple form pushes all contextual information to the operator application, making it explicit in the call rather than implied by the context. The explicit context simplifies the translation to E-FRP. However, Stencil-Triple form makes context-dependent abstractions presented in later chapters more cumbersome than the normal form because it increases the amount of searching required. Stencil-Triple form can be derived from the normal form in a straightforward fashion. An example program in the standard input form and its translation to Stencil-Triple form are found in Figures 5.1a and 5.1b, respectively.

Stencil-Triple form relies on a library function called "Dispatcher" (see Figure 5.1b, line 9). Dispatcher provides the values associated with the event. The dispatcher operator is required because E-FRP events do not carry any values. This operator changes its value between events, based on execution context. When instantiated, the dispatcher operator takes a specializer argument to indicate which streams values should be returned. Used in

```
#input = init x= NULL
   in {Data => x = Dispatch.get("Data")}

#XScale2 = init x = NULL
   in {Data => x = XScale2(#input[1])}

#YScale3 = init x = NULL
   in {Data => x = YScale3(#input[2])}

#Limit6  = init x = NULL
   in {Data => x =
         if #YScale3[0] > 100
            ToTuple(100)
         else
            ToTuple(#YScale3[0])
      }

#ToTuple5 = init x = NULL
   in {Data => x = ToTuple4(#Data[0], #XScale3[0], #Limit6[0])}

#Scatter = init x = NULL
   in {Data => X = Scatter.store(##ToTuple4)}
```

FIGURE 5.3. Scatter plot program from Figure 5.1b translated into Event-driven Functional Reactive Programming syntax.

this way, dispatcher embodies the implementation of the value-carrying events discussed in Section 4.3.

The basic rules for converting a Stencil-Triple program into an E-FRP program are given in Figure 5.2. After the basic translation rules are applied, a single behavior may appear with multiple definitions, one for each event it responds to. These definitions are all prefixed with the same “init x = NULL in”, so the handlers can be concatenated to form a list of handlers. Since each handler responds to a distinct event and E-FRP dictates only one event occurs at a time, the order of the list of handlers does not matter. Operators are not translated, they instead are assumed to be contained in the execution context. Application triples are translated into E-FRP events and event handlers. The event name is given by the first element of the pair. The behavior name is given as the second element of the pair. The event response is given as the final element of the pair. This effectively converts a stream into an event/behavior pair (the event is the event name, the behavior calls the dispatcher to get the values). In this schema, there is one event handler for each consumes block an operator appears in. Figure 5.3 presents the running-scatter plot example in E-FRP form.

```
stream Ticks(i)

layer Layer
defaults:
   FILL_COLOR: Color(Gray60)
consumes Ticks
   ID: i
   FILL_COLOR :* MaybeRed(i)

operator MaybeRed(i) -> (color)
   prefilter(next) : Add1(i) -> Layer.find(_)
   prefilter(now) : Layer.find(_) -> FILL_COLOR
   (next == NULL) => color: Color(RED)
   (ALL) => color: now
```

FIGURE 5.4. Stencil program that "reveals" the scheduling of the renderer by highlighting the results of last tuple before a render in red. If render events are not considered in the semantics, this program is nondeterministic.

5.2. Operations With Special Consideration

5.2.1. Rendering. Due to their far-reaching impacts, rendering and dispatching merit special attention. Improperly handled, these operations can make an E-FRP program nondeterministic with respect to the sequence of inputs (though still deterministic for any given input).

Rendering requires special attention to be properly handled by E-FRP in a deterministic, resource-bounded fashion. The definition of information visualization as "data handling, graphics and interaction" [**67**], indicates that presentation is integrated with the analysis process. The process of presentation is called rendering and must be treated by the semantics. Furthermore, a deterministic system that includes the ability to search a mutable data store (as Stencil does) also requires rendering requests be considered as part of the input data. If they are not considered part of the input data, any program that can reveal the render timing is nondeterministic. The Stencil program given in Figure 5.4 reveals when render events occur relative to the declared data stream.

To reconcile the need for deterministic semantics with the need for rendering and data store search, rendering is treated as a response to a particular event. All Stencil programs logically include a stream definition **`#RENDER`** and a consumer given in Figure 5.5. The **`#RenderBuffer`** operator is triggered only in response to events that occur as part of the

```
from #RENDER
  (): #Update() -> #RenderBuffer()
```

FIGURE 5.5. Basic render stream reaction rule. The call to "#Update" performs prerender calculations, such as those related to dynamic bindings or guides (discussed in Chapter 7 and Chapter 8 respectively). The call to "#RenderUpdate" produces the actual rendered image.

`#RENDER` stream. Rendering in response to a special event is consistent with the approach taken by other FRP animation engines [**71**]. The Render operator is special in two ways. First, it interacts with all layer states to acquire the elements to be rendered. In contrast, all other operators only share memory within their own facets (as defined in Section 3.3). Second, rendering also provides a bitmap to an externally facing canvas. These unusual properties make the **`#RenderBuffer`** operator unsafe in uncontrolled contexts, and so it is not directly accessible to the programmer.

5.2.2. Dispatching. Sequencing of events requires some attention beyond that given in the E-FRP definition. Stencil makes use of internally defined streams to divide up work. A single external event can trigger multiple internal events due to Stencil's preprocessing steps or cycle breaking (see Section 4.6.1 for details). To be deterministic with respect to the data presented, and not the timing of the data presented, the internal events must all complete before a new external event can be responded to. If this were not the case, rendering could reveal, for example, how many loops of a cycle were completed. Complex prioritization rules for the sequencer can achieve the desired effects, but a simpler solution involving the dispatcher is employed instead.

In standard E-FRP, the dispatcher is just a gatekeeper. All queuing and prioritization is handled by the sequencer. To provide the illusion of atomicity in event handling, we augment the dispatcher with a set of priority queues as well. The dispatcher only accepts tuples from the sequencer when (1) the E-FRP program is not executing, and (2) all its internal queues are empty. The dispatcher may also receive tuples through the E-FRP-produced store. These tuples must be presented as lists (so sequence inside of a stream is determined by the E-FRP program). To preserve deterministic execution with respect to sequencer, the dispatcher selection rules must be deterministic and direct-to-dispatcher tuples must only result from the E-FRP program itself. Though any deterministic predicates are permitted, current rules dispatcher rules are based entirely on presence/absence of tuples, not on their internal values. Both sequencer and dispatcher prioritization rules are controlled by the Stencil program's **`order`** statement.

Having rendering and dispatching arranged as described in this section yields a system block diagram that differs slightly from standard FRP systems. The modified diagram is shown in Figure 5.6. The main difference is the possibility of internally generated events bypassing the sequencer and being placed directly into the dispatcher. The rendering subsystem is a significant component, but entirely within the normal realm of E-FRP. The dispatcher/sequencer pair provides more timely handling of high-priority events, like render requests, without the complexity of clock calculi or requiring all event producers to be part of the Stencil system [**7**]. The sequencer provides a clear point where tuple ordering occurs. The dispatcher/sequencer pair delimits where deterministic behavior is guaranteed.

5.2.3. Loops. The dispatcher is also the key to ordering memory modifications, and managing modification visibility. To ensure deterministic semantics, the order of memory modifications must be determined by the program. However, if two disparate parts of a program respond to the same event and modify the same memory, the order of those modifications is not specified by the semantics. In the absence of information about operator associativity and commutativity (which Stencil does not have access to), this lack of ordering constitutes a region of nondeterminism. A compile-time check looks for such occurrences and issues an error if they are detected. A similar technique is used by ESTEREL compilers to detect causality violations [**12**]. Resolution of these ambiguities can be achieved by internally defining a stream that echoes the contents of the original stream and having each co-modifying location consume a separate stream. The internally defined stream creates an explicit ordering to modification events. Stencil allows one exception to this rule: when programmer ordering of modifications is explicitly provided in a single rule chain. In this case, order of events is explicitly made by the dependency relationship

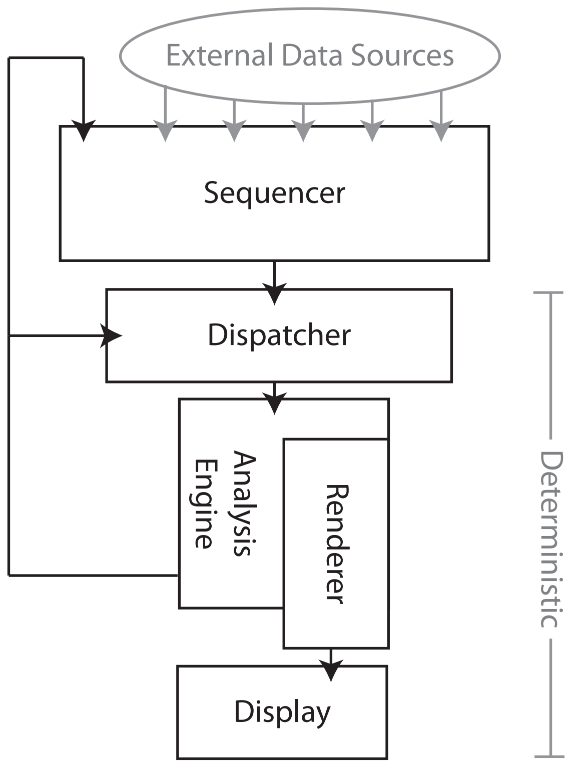


FIGURE 5.6. Stencil system block diagram. In Stencil, the display, dispatcher and sequencer are all store consumers from the Event-driven Functional Reactive Programming model presented in Chapter 4. The renderer is a substantial subcomponent of the analysis engine that also interacts with the display. Event-driven Functional Reactive Programming semantics, with events driven by the dispatcher, ensure deterministic behavior with respect to the sequence of tuples entering the dispatcher.

described by the **->** operator, and the relevant stream definition and order statements can be automatically generated. This limitation on effects in response to a single event carries into stream definitions and is the reason why stream and layer definitions can have at most one consumes block per input source.

5.3. Conclusions

With semantics for the Stencil system established and with the consistency property defined, the manipulation of visualization programs can be approached in a principled fashion. These semantic properties of Stencil are relied on to argue for the validity of abstractions presented in later chapters. These abstractions include higher-order operators, automatic guide creation and efficient scheduling of operations.

6 Implementation

The Stencil language has been implemented as a compiler and runtime in Java that conform with the description given in Chapter 3. An understanding of the current implementation of Stencil is not generally required to understand the theoretical claims of this dissertation. However, this chapter provides some implementation details that are relevant to future implementation discussions and provides details not determined in the semantics.

6.1. Data Types

Tuples and values are the two data types of the Stencil model. Tuples are essentially lists of values. Values are immutable data elements that are passed as arguments to operators; tuples and operators may be used as values. Streams carry tuples into Stencil from the surrounding context and link operators together. All tuples on a stream conform to the tuple declaration for the stream (provided in a stream declaration, stream definition, or operator metadata, depending on the stream's source). These basic data types are discussed in more detail in Chapter 3. The rest of this section describes additional data types used in the implementation and handling of conversions between types.

6.1.1. Sentinel Tuples. To facilitate initialization and response to the end of a stream (if it actually occurs), two sentinel tuples exist. These are **`START`** and **`END`**. It is assumed that each **`END`** pairs with the preceding **`START`** tuple. As noted, not all streams have an end, so some **`START`** tuples do not have a corresponding **`END`**. The **`START`** and **`END`** tuples carry no value and are generally not passed to operators, so they do not generally affect operator implementation. These sentinel tuples are instead handled in the runtime machinery directly. The order of operations (e.g., initialization) performed by the runtime in response to a sentinel tuple follows the order of normal value-tuple processing.

6.1.2. Mark. A mark is a tuple that can be rendered. Any tuple that is part of a layer is also a mark The significant difference between a mark and a regular tuple is that a mark has fields that permit it to be rendered to the screen.

6.1.3. Composite Types. Though the Stencil language only works with values and tuples, values often find a natural representation as a tuple. For example, a color can be a four-tuple of numbers (e.g., HSVa or RGBa). Since tuples can be nested (e.g., subtuples are permitted), arbitrary data can passed as a tuple.

To facilitate interoperability with compound data types in the hosting language, wrapper functions can be registered with a central conversion facility. The conversion facility allows operators to be defined without needing Stencil-specific types.

6.1.4. Value Conversion. The Stencil runtime is weakly typed; it does best-effort automatic type conversion at runtime and type declarations are not syntactically required. However, Stencil is implemented on top of strongly typed [**16**] libraries. To mediate these two systems, reflection is used to retrieve the expected parameter types and a central conversion facility is used to translate between the actual argument types and the expected types. In effect, the runtime must retain the same information as dynamic type systems [**49**], but uses automatic conversions to attempt to avoid type errors. The converter is also exposed as an operator, should the automatic system use an inappropriate method in specific circumstances.

The central conversion mechanism has a two-part interface: Converter.to and Converter.register.

Converter.to($< value >$, $< type >$): Attempts to return a new value of the requested type based on the value passed. The passed value may be of any type. If the value can be converted into the requested type, a new value of the requested type is returned. Otherwise, an error is thrown. The requested type may be any built-in type (including Tuple), or any type registered as the first parameter with register. This function is exposed as an operator with the type specified as a string.

Converter.register($< type1 >$, $< type2 >$, $< function >$): Registers a converter from type1 to type2. This influences the execution of Converter.to, but does not return a value. If the passed type pair is already registered, the new registration will override the old one.

6.2. Error Handling

The InfoVis Reference Model [**15**] is based on the idea of transforming data between representations. Stencil follows the same general concept, where each operator represents a stage in the transformation process. Stencil's error response depends on the stage in which the error occurs. There are three stages when errors can occur: Load, Store and Analysis.

Load errors can occur when a tuple is presented to the sequencer. These errors are usually the result of malformed tuples (i.e., does not have the correct fields for the declared schema). They are typically raised by the sequencer. Load errors are unique because the data are not yet visible to Stencil, and therefore do not have any semantic implications. However, they may have significant implications for the host application. Load errors are propagated out to the host application, but the Stencil-related state remains intact. Processing will continue if more data are supplied.

Analysis errors are generated by transformation operators while processing a value after dispatch. This type of error represents a serious concern for the state of the analysis and the consistency of the resulting visualization because some processes have responded to data while others may not have. Any time a Stencil operator raises an exception, it is treated as analysis error. An operator producing a tuple that does not conform to its output schema (e.g., incorrect number of fields) is also an analysis error. In such cases, the visualization system is halted and the error is propagated out to the host system.

Store errors occur when a tuple is entering a storage context. These errors are similar to Load errors in that they do not represent inconsistency in the analysis process. These errors occur when the analysis produces a tuple that is incompatible with its storage context (e.g., no ID field on a layer tuple or a stream tuple with the wrong number of fields). Store errors are distinct from analysis errors in that they represent a potential inconsistency between the analysis state and the visualization state, but not necessarily an internal inconsistency in the analysis state. However, an inconsistency between analysis and visualization state is visible in the analysis if the analysis includes a reference to layer state or depends on the sequence of results from a stream definition. For this reason, the current implementation treats Store errors in the same way it treats analysis errors. Further work could yield results where the difference between Analysis and Store errors could be used to provide different responses.

6.3. Integration

The Stencil system is designed to be integrated with a larger environment. Integration occurs in three ways: operators, panels and tuples.

Operators are the most flexible means of integrating, as they provide a way for Stencil to use software components developed externally. The tree-map routines provided with the current implementation directly employ the tree-map implementation released by the

University of Maryland [**106**] (see Appendix A (Section A.11)). There are also graph analysis algorithms that employ the Java Universal Network/Graph (JUNG) library [**87**] and coloring routines based on ColorBrewer [**13**].

Operators allow Stencil to *call out* to externally defined code. But, operators have strict behavioral expectations that must be met and they require synchronous communication. The streams interface allows more flexibility in integration. Stream declarations provide a means for Stencil to receive input from arbitrary data sources. This includes computational processes. The results of stream/stream transformations can also be sent outside the Stencil system through Stencil API calls, permitting asynchronous interactions. The streams mechanism also enables Stencil to work with user input [**30**].

Finally, the Stencil compiler actually provides a JPanel object that may be embedded in other applications. This permits Stencil to be used as a library component in a larger application. It is currently used in this way in Epidemics Cyberinfrastructure tool (EpiC) [**44**].

6.4. Phases of Operation

The implementation divides its work into two main phases: analysis and render. (A third, prerender phase is introduced in Section 7.2 for working with global states.) These phases capture the relationships described in Section 5.2.1.

The implementation of rendering exploits the natural division of work afforded by layered tuple groups. Layers can be rendered independently and then composed in the proper order. This enables simple task-based parallelism at the cost of an off-screen bitmap buffer for each layer. The bitmap for each layer is computed, then all bitmaps are composited when all the component bitmaps are complete. This arrangement also affords other optimizations. The potential optimizations include some presented by Piringer et al. [**93**]. Particularly, those optimizations based on semantic and incremental layers can be directly approached.

The execution engine implements the semantics given in Chapter 5. It is based on a topological sort of the E-FRP dependency graph to reduce runtime searching, but otherwise performs no significant optimizations. It is implemented as a tree visitor over a transformed abstract syntax tree (AST) that encodes the topological sort order.

6.5. Dispatcher, Sequencer and Order Statement

The order statement configures the dispatcher and sequencer by prioritizing the constituent streams. Recall that the dispatcher and sequencer are essentially priority queues (and the dispatcher must be deterministic). Three types of priorities are implemented: (1) drain, (2) before, and (3) round-robin chance. Drain relationships (denoted by `>>`) indicate that everything to the left should have reached end-of-stream before anything on the right is started. Before relationships (denoted by `>`) indicate that the left-hand side should always be taken (if available) before the right-hand side. Round-robin chance relationships (denoted by `||`) indicate that the elements on the left and right should be polled in a round-robin fashion (starting with the element on the left). If a stream in a round-robin chance relationship does not have an element ready, it forfeits that round and the next stream is polled. The current implementation does not provide for all reasonable relationships between streams. Omissions that are reasonable but not provided include lock-step consumption and using predicates based on tuple values.

Streams may be grouped into clauses with parentheses. The default order statement groups internal streams into a single round-robin clause, all external streams in a second round-robin clause. These two clauses are separated by the render signal stream, chaining the three in a series of before relationships:

```
(internal1 || internal2...) > #Render > (external1 || external2 ...)
```

The drain relationship enables a common optimization case: priming data structures with significant supplemental information (e.g., Zip Code-to-state mappings). Fully loading the supplemental information before any other data reduces the frequency of existence checks in the Stencil program, effectively reducing the number of cases synthetic operators must cover. If the supplemental information includes stream metadata (e.g., maximum and minimum values for a field), the drain relationship can also be used to eliminate dynamic bindings (dynamic bindings are discussed in Section 7.2).

6.6. Global State Tracker

The GST is used to reason about semantic properties and to ensure that the system respects those properties. In the semantic discussions, a GST operator is discussed as if it were a separate entity. Stencil's implementation uses a distributed scheme where every operator contributes to a collective GST. The individual parts of the distributed GST are accessed through the *stateID* facet, introduced in Section 3.3. The GST is the list of all stateIDs in canonical order. (The only requirement is that the order remain stable for an entire execution; it is currently a breadth-first search of the normal form of the AST.)

LEMMA 6.6.1. *A list of numbers can be converted into a single number with one digit per list element.*

PROOF. Select a base such that the largest element in the list will never exceed the base. Each element of the list contributes a single digit to the resulting number. Since the largest list element does not exceed the base, each element only contributes one digit. □

THEOREM 6.6.2. *The distributed GST satisfies the requirements of the GST.*

PROOF. From Section 4.7, the GST by definition must (1) not repeat, and (2) change each time a stateful transition is made. Recall that properly implemented operators change their stateID each time a state change occurs and that stateIDs do not repeat per Section 3.3 (i.e., they use an acyclic function). It can be shown that the stateIDs of all operators can be composed to satisfy the requirements of the GST.

Construct a number, G, per Lemma 6.6.1, where the order of the list is canonical and the mapping of list to digits is deterministic on the order of the list. The source values to the GST are the stateIDs of the operators in canonical order. Therefore, any change in the state of an arbitrary operator will appear as a digit change in the corresponding digit of G. Thereby, condition 2 is satisfied that G will change its value any time a stateful transition is made.

Given that stateIDs must not repeat, and having selected a base that exceeds the largest list element, each stateID affects exactly one digit. The canonical order of elements and deterministic mapping to digits indicates the stateID of a given operator always exclusively determines the same digit in G. Therefore, no combination of state changes can result in a repeat value of G, satisfying condition 1.

Therefore, stateIDs can be used to satisfy the conditions of the GST. $\square$

The ability to compose stateIDs to form state trackers is used in a similar manner in Section 9.3 to identify opportunities to avoid recompilation of unchanged values. Though each *stateID* is treated as if it were implemented with an infinite range, it is actually implemented with Java ints. The exact tracking mechanism presented in Section 9.3 allows the repetition requirement to be relaxed to "stateIDs must not repeat between consecutive render event triggers." In practice, the range of Java ints is sufficient to meet this relaxed requirement.

7
Analysis abstractions

Analysis is the front end of all visualizations. In the Visualization Reference Model, the first two of the four stages are generally shared by all analysis systems: acquiring data and constructing data tables [**15**]. Analysis abstractions focus on expressing the process of converting raw data to visual elements. The map abstraction is common in functional languages to support analysis idioms in the data-flow model. The split and range abstractions (Section 7.1) could be used in most analysis frameworks, but are particularly useful when working with dynamic data. Finally, dynamic binding (Section 7.2) is specific to visualization, bridging analysis with the visual representation.

7.1. Higher-Order Operator

Higher-order operators provide an opportunity to more compactly represent many visualization schemas. Stencil only supports taking operators as arguments at this time. Using operators as arguments requires compiler support, though the individual operators do not. The Stencil runtime provides access to defined operators for use as arguments. To prevent name-space issues, operators used as arguments are prefixed with an `@` sign, indicating that the name should be resolved as an operator and not a tuple reference. This section describes the semantics for split, range and map higher-order operators, indicating that their behavior can be made consistent with E-FRP semantics.

7.1.1. Split. Split encapsulates the essence of a cross-tab or pivot summarization. In addition to the kernel operator and data arguments, split requires a number of key-fields arguments. The key-field arguments composite to determine an operator instance to invoke. It is similar to a dictionary where the contents of the dictionary are operator chains. Logically, a split is a synthetic operator with one branch for every possible input key combination (see Figure 7.1). Since inputs may have an unbounded or bounded-but-unknown range, this leads to the impractical situation of having an infinite number of branches. Therefore, the split operator dynamically creates branches as needed. Semantically, split takes advantage of E-FRP's ability to modify the program in response to incoming data. Each time split is invoked with a new key-field combination, it modifies its definition by adding a new case.

Split is implemented with the help of the duplicate method of each operator. Split maintains a fresh copy of the operator. When a novel key combination is presented, a duplicate is made and stored in a dictionary under that key. The number of key fields is indicated as the "fields" argument of the specializer. With fields set to *n*, the first *n* arguments to split are used to form the key, and the remaining arguments are passed to the appropriate kernel operator instance. Therefore, **`Split[fields:0](...,@Kernel)`** is the same as just the kernel (since no values are used to create the key). Split can be optimized by providing the flag **`depth`** in its specializer to indicate how many old keys should be

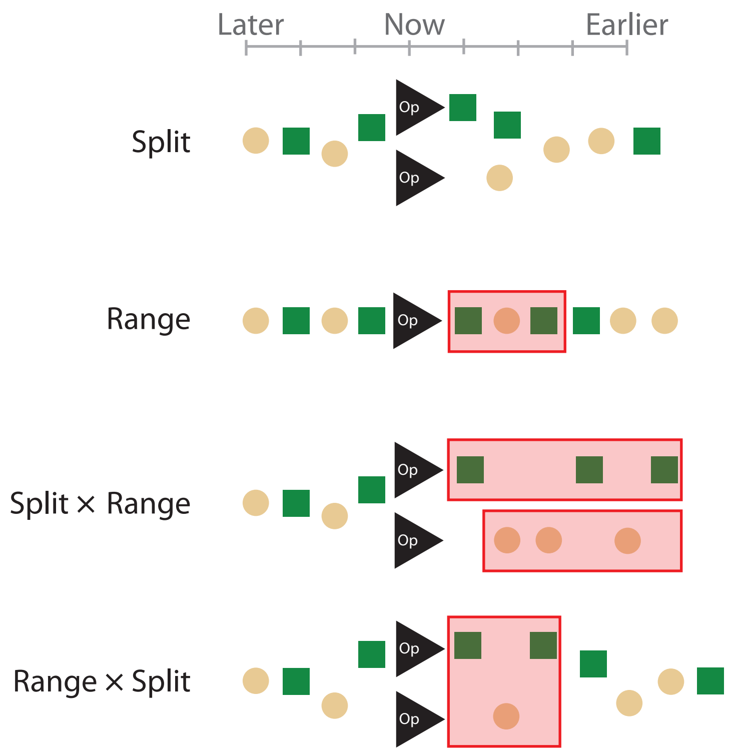


FIGURE 7.1. Split and range with respect to their input streams. The explicit ranges shown are "-3..n", covering the most recent three items.

retained. When depth+1 keys are present, the least recently used operator instance is discarded. Setting depth to a nonpositive number indicates that all operator instances should be retained indefinitely. This optimization builds on the redefinition semantics described above.

7.1.2. Range. Split modifies the operator that arguments are presented to, while range modifies the arguments presented to an operator. The operator that range produces includes a prior-arguments buffer. The actual arguments presented to the kernel operator come from this prior-arguments buffer and are determined by the range indicated in the range specializer. Valid range elements are

ALL: Range includes all values ever seen.

n: The most recently encountered value.

positive-k: An absolute offset from the first tuple.

negative-k: An offset relative to the last tuple.

Elements can be combined: `@Range[range: 5..n](@Sum, value)` adds everything except the first five values seen; `@Range[range: -5..n](@Sum, value)` will sum the most recent five elements.

The provided implementation of range is as a ring-buffer paired with an operator. The range wrapper manages the ring-buffer updates and passes relevant segments to the operator's *query* facet. This effectively moves all of the state maintenance to the range wrapper. From an implementation standpoint, this limits the operators that can have range applied to them automatically. Such operators must have a *query* facet that (1) does not require internal state, and (2) is variable-arity (to accept runtime defined ranges). Operator-specific optimizations can be provided to more efficiently provide implementations through the modules mechanism (see Section 9.2 for details).

Split and range interact in useful, noncommutative ways. The combinations are given in Figure 7.1. The sequence of execution is determined by the which operation is supplied as an argument to the other. The receiving outermost operator executes first. Operator-specific optimizations can be applied by the module to efficiently support split and range. These optimizations are discussed further in Section 9.2.

7.1.3. Map. Map takes an operator and a tuple as an input and produces a new tuple as its output. Unlike primitive operations, map cannot simply be lifted from the surrounding context because the map operation may involve sequential application of a mutative operator. For example, `map(@Count, value.*)` increments count by one for every field in the value tuple. Because of this behavior, uses of map semantically translate into multistream applications (thereby involving the dispatcher and keeping memory changes within the presented semantics). If map is used anywhere in the program, two new stream definitions are created and the original map site is reconstructed. The full reconstruction is represented in Figure 7.2. The first new stream is called the *MapStream*. It includes a consumes block for the original input and performs all the processing not dependent on the map. The call to map is replaced with a call to *MapStreamLoad* which formats the tuple to instruct the dispatcher to treat it in a special way. The tuple produced by MapStream will have each value of the MapStreamLoad argument aligned with the other values and prepended to the output stream in reverse order. This unusual behavior enables nested and mutually recursive calls to map to exit the MapStream in the correct order. The alignment rules are described later in this section. The second stream is *MapStreamProcess*. MapStreamProcess includes a consumes block for each map stream block (filtered on SourceID). The MapStreamProcess invokes the original mapped operator and stores the results in a special *Accumulate* operator, split on the invocation identifier. Accumulate has two rules: (1) On any normal tuple, collect the value and emit nothing, and (2) when a sentinel **END** tuple is received, emit a tuple. The tuple produced when **END** is received is a tuple where each field is one of the value sets received by Accumulate. Therefore, the only tuples emitted by MapStreamProcess are ones whose results are the result of applying a function to each element of a tuple (i.e., the map results; see Figure 7.3). Accumulate also stores the other values computed by MapStream, to be emitted when Accumulate receives the **END** tuple. The original call site is changed to consume the results of MapStreamProcess, filtered by the call site identifier (again). It then unpacks the result value and other passed-through values into the appropriate field; this process may also involve tuple alignment according the same rules used for the MapStream.

The system uses three rules to handle stores with partial results from maps (see Figure 7.4): (1) single-value results are repeated in all result tuples, (2) multivalued results (i.e., the results of maps) contribute one value to each of the resulting tuples (the n^{th} value goes to the n^{th} tuple), and (3) if multivalued results produce a different quantity of results, the layer operator emits an error instead of storing the results.

From an implementation standpoint, stream blending update tracking and invocationID tracking are expensive operations. Therefore, the implementation instead creates a new dispatcher each time map is used. This special-purpose dispatcher only updates the state of the operations involved in the map and removes the stream blending and invocationID tracking. Nested or recursive calls to map generate a stack of dispatchers. Resolving map at compile time means that the operator cannot be set at runtime, differentiating the

```
stream Lines (line, text)

stream RawWords (line, word)
from Lines
    line: line
    word: SplitOn(text, "\\s+") -> Map(@NormalizeWords, *) -> *
```

(a) Input

```
const #SourceID1 : "Lines"

stream Lines (line, text)

stream #MapStream(SourceID, InvokeID, OVS, MV)
from Lines
   filter(SourceID =~ #SourceID1)
   SourceID : #SourceID1
   InvokeID: UniqueID()
   OVS: ToTuple(line)
   MV: SplitOn(text, "\\s+") -> MapStreamTuple(SplitOn.*)

stream #MapStreamProcess(SourceID, OVS, MV)
from MapStream
   filter(SourceID =~ #SourceID1)
   SourceID: SourceID
   T : NormalizeWords(MV) -> @Split[1](@Accumulate, InvokeID, OVS, NormalizeWords)

stream RawWords (line, word)
from #MapStreamProcess
   filter(SourceID =~ #SourceID1)
   word: T
   line: OVS.0
```

(b) Transformed

FIGURE 7.2. Map represented as a series of streams. This example is pulled directly from the TextArc example found in Appendix A (Section A.9).

Stencil map from the traditional functional programming definition of map [**49**]. Using a stack of dispatchers, the more traditional meaning of map is feasible (though not implemented).

With the modified binding operation, the map operator enables convenient expression of multiple results in response to a single input. This is used in the TextArc schema (see Appendix A (Section A.9)). Since map can be implemented on top of normal stream and operator declarations, it does not expand the technical power of the overall Stencil system. However, it is more compact and direct than the equivalent set of stream declarations. Therefore, map represents an increase in the overall expressiveness of the framework. Being able to employ higher-level concepts based on the lower-level framework is the power that abstraction enables.

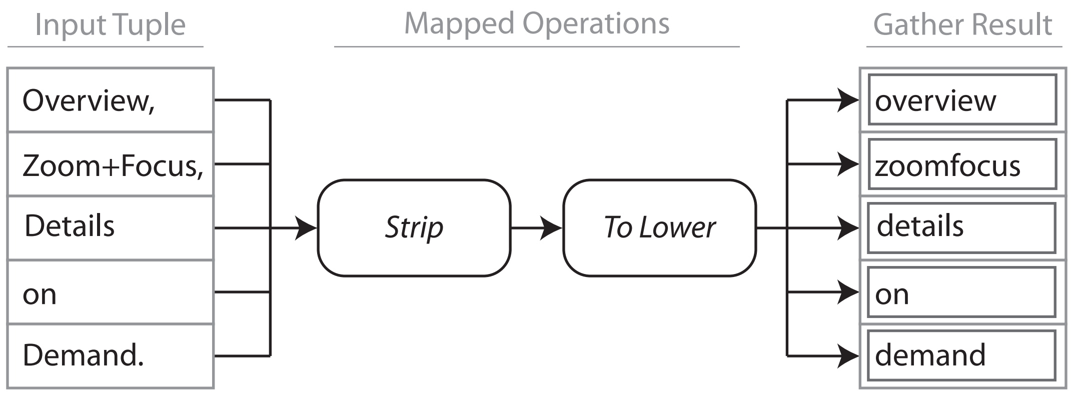


FIGURE 7.3. The map operator working with a sample input tuple. Strip removes all nonletters and ToLower converts everything to lower case. The results are actually nested tuples (thus the double-boxes). Because map always produces a tuple full of tuples, it is common for the next operation to work on tuples of tuples. *Select* is one such operator, pulling the given fields from each of the component tuples and concatenating them into a flat tuple.

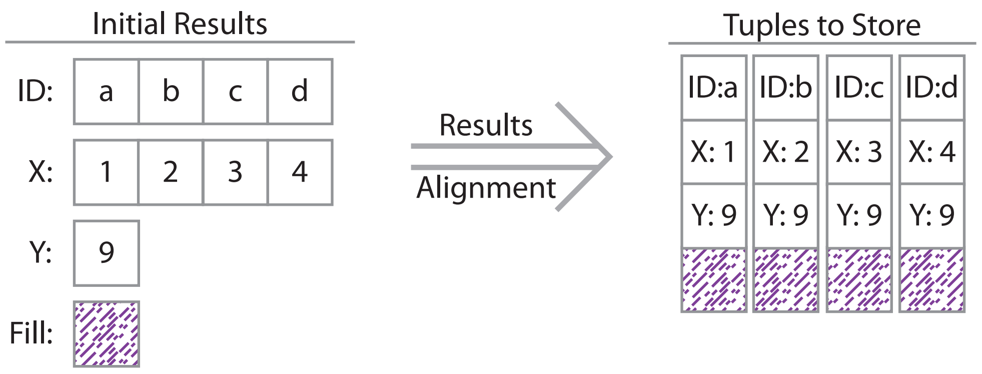


FIGURE 7.4. Tuple alignment in layer or stream store for rules involving multiple map rules. The alignment protocol is handled by the stream store operator before results are passed to the dispatcher. Therefore, exceptions in the alignment process appear as analysis errors.

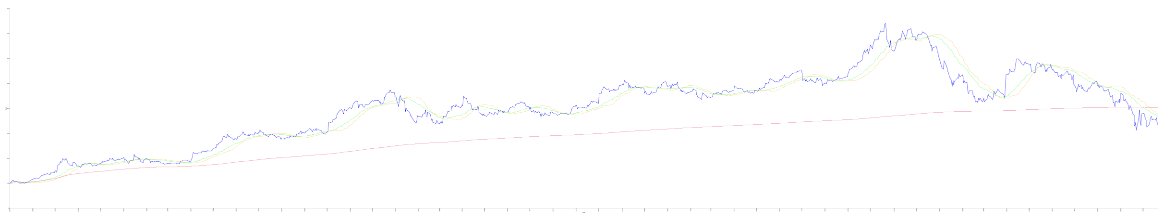


FIGURE 7.5. Stock ticker visualization produced by either Figure 7.6 or Figure 7.7.

```

import Dates

stream Stocks(Date, Open)

stream AlignedStocks(Date, Open)
from Stocks
  Date : Parse[f:"dd-MMM-yy"](Date)
  Open : Open

layer Ticker["POLY_POINT"]                /*Poly-point is a group of groups of line segments.*/
guide
    axis Linear from Y
    axis[unit: "MONTH"] Date from X
      label.TEXT: Parse.format[f:"MMM yy"](Input)

from AlignedStocks
    ID: AutoID(4) -> *
    GROUP: MultiResult("Base", "Avg20", "Avg30", "AvgAll") -> *
    ORDER: Count()
    X: TimeStep(Date)
    Y: MapApply(@Echo, @Avg20, @Avg30, @AvgAll, Open) -#> *
    PEN: Stroke{1}
    PEN_COLOR: ToTuple("BLUE", "LIME,150", "CRIMSON,150", "ORANGE,150")
                 -> Map(@Color, ToTuple.*) -> *

operator Avg20 : Range["-30..-10"](@Mean)
operator Avg30 : Range["-30..n"](@Mean)
operator AvgAll : Range[ALL](@Mean)

operator Ids : Count

operator TimeStep (T) -> (S)
default => (S) : Range[ALL](@DateMin, T)
                   -> DateDiff("DAYS", Range, T)
                   -> Mult(_, 3)

operator DateMin : Min[c:"Date"]
```

FIGURE 7.6. Stock ticker using higher-order operators.

7.1.4. Application. Higher-order operators add an additional cognitive burden to the developer since control flows can become more complex. However, they can be used to simplify the implementation of visualizations. For example, the stock ticker visualization shown in Figure 7.5 has had the separate lines encoded with higher-order functions (Figure 7.6) and explicitly (Figure 7.7). The use of range, map and mapApply (which takes a list of operators and applies each to an argument) results in a visualization program that is two thirds the length of the explicit representation. Additionally, the guide system is able to automatically produce more cohesive guides when these operators are used (see Section 8.1 for details on the guide system) because the relevant information is located in a single transformation chain instead of spread across multiple layers.

```

import Dates

stream Stocks(Date, Open)

stream AlignedStocks(Date, Open)
from Stocks
  Date : Parse[f:"dd-MMM-yy"](Date)
  Open : Open

layer Ticker["POLY_POINT"]             /*Poly-point is a group of groups of line segments.*/
guide
   axis Linear from Y
   axis[unit: "MONTH"] Date from X
     label.TEXT: Parse.format[f:"MMM yy"](Input)

from AlignedStocks
   ID: TimeStep(Date)
   GROUP: "Google"
   ORDER: Count()
   X: TimeStep(Date)
   Y: Open
   PEN: Stroke{1}
   PEN_COLOR: Color{BLUE}

layer Averages["POLY_POINT"]
from AlignedStocks
   ID: Ids()
   GROUP: "Google_30"
   ORDER: Count()
   X: TimeStep(Date)
   Y: Range["-30..n"](@Mean, Open)          /*Average the most recent 30 values.*/
   PEN: Stroke{1}
   PEN_COLOR: Color{LIME,150}
from AlignedStocks
   ID: Ids()
   GROUP: "Google_ALL"
   ORDER: Count()
   X: TimeStep(Date)
   Y: Range[ALL](@Mean, Open)               /*Average all values.*/
   PEN: Stroke{1}
   PEN_COLOR: Color{CRIMSON,150}
from AlignedStocks
   ID: Ids()
   GROUP: "Google_20"
   ORDER: Count()
   X: TimeStep(Date)
   Y: Range["-30 .. -10"](@Mean, Open) /*Average that lags by 10 and includes 20 values.*/
   PEN: Stroke{1}
   PEN_COLOR: Color{ORANGE,150}

operator Ids : Count

operator TimeStep (T) -> (S)
default => (S) : Range[ALL](@DateMin, T)
                 -> DateDiff("DAYS", Range, T)
                 -> Mult(_, 3)

operator DateMin : Min[c:"Date"]
```

FIGURE 7.7. Stock ticker individually representing each transformation case.

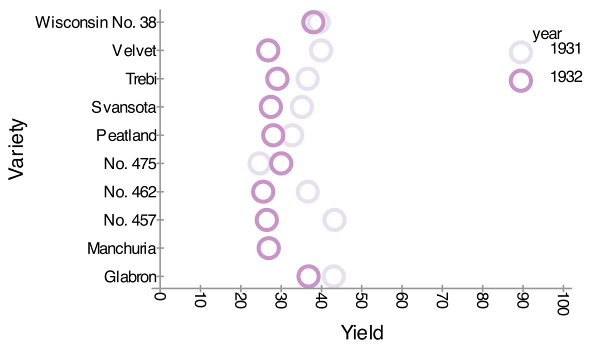


FIGURE 7.8. Static and dynamic binding in the Becker's Barley visualization. The X position is based on a stateless projection of field yield to a pixel location. Once set, it is never reset. However, the Y position is based on the name of the wheat type. It is set and reset based on subsequent values presented to keep types in alphabetical order. Using a dynamic binding for Y allows the visualization schema to (1) be single pass over the input data while (2) not requiring the varieties to be hard coded (keeping the schema general). Full source for this visualization can be found in Appendix A (Section A.6)

7.2. Dynamic Bind

An important implication of working with dynamic data in the data-flow model is that, without additional support, all analysis is single pass. Once a value has been responded to, that response is fixed. However, complex relationships inside of data make visual representations more natural if prior responses can be revised when new data are acquired. Dynamic binding provides the additional support to recompute values based on new data *without* needing the new data to identify where changes might appear. Intuitively, dynamic binding indicates that any update to the given attribute of any data point may affect an arbitrary number of existing data points. For example, the y-axis in Figure 7.8 is kept in alphabetical order without prior knowledge of the values to be presented. From the standpoint of consistency (see Section 4.7), static bindings imply sequential consistency, while dynamic bindings imply snapshot consistency.

Dynamic binding is lexically indicated with `:*`, extending the normal bind operator. Semantically, it indicates that three new actions must be taken with respect to the binding: (1) A value to facilitate recalculation must be stored as part of the bind, (2) A recalculation chain must be established, and (3) Recalculation must occur periodically. Dynamic bind is not permitted in Stream/Stream contexts because the binding is to an ephemeral tuple. The support structures of this system can only be used for mark bindings.

Value storage may logically be any value that permits the values to be recalculated. If all operators in an analysis chain involved in a dynamic binding are reversible (i.e., 1:1 and have an inverse), then the calculated value would satisfy the condition. However, the Stencil metadata system does not currently include 1:1 as a property or inverse as a facet. Instead, the original input value is retained in a data table, associated with the same ID as the ID calculated for the resulting mark. Semantically, this requires no additional machinery. In the AST representation of Stencil, this storage appears as an additional binding statement to a field call **`#data`** on the mark. Because of the "#," **`#data`** is not a general-purpose identifier, so the values cannot be accessed programmatically. The value bound to **`#data`** is a tuple that contains those inputs necessary to perform all dynamic binding recalculations. Retaining the original data ensures that the calculation can be repeated later.

With this implementation of data store, the recalculation chain is a slightly modified copy of the original chain. First, all tuple references are modified to refer to the tuple declaration stored in **`#data`**. Second, all operators in the recalculation chain are tagged and replaced with their counterpart operators (see Section 3.3.1; by convention, this is the *query* facet). This second change is performed because (1) recalculation has no clear order, especially in the presence of mark updates and removals, and (2) changes are only intended to reflect new data, not repeated appearances of old data. The recalculation chain appears as a new consumes clause that receives a stream with a name distinct from all existing stream names. Because ID is stored in **`#data`**, it need not be recalculated and can simply appear as a direct binding in the generated consumes block. (A dynamic binding chain may also be explicitly supplied if the automatically generated one is inappropriate.) Figure 7.9 shows how the Stencil program is modified for dynamic binding.

Recalculation occurs as a precondition to rendering and is mutually exclusive to analysis (see Section 6.4). Always executing dynamic bindings before rendering ensures that

```
ID: Count()
Y:* Rank(variety) -> Mult(7,_) -> Add(-5,_)
```

(a) Input

```
local(ID) : Count()
ID: local.ID
Y: Rank.map(variety) -> Mult(7,_) -> Add(-5,_)
#data: ToTuple(local.ID, variety)

from #Data128
   ID: #Data128.0
   Y:* Rank.query(#Data128.1) -> Mult(7,_) -> Add(-5,_)
```

(b) Transformed

FIGURE 7.9. Dynamic binding substitutions for the y-axis from Figure 7.8. The stream name "Data128" is unique to the y-axis binding and the consumes block is part of the same layer as the original binding.

the visualization remains consistent, as defined in Section 4.7. This introduces a two-phase rendering operation: (1) Prerender recalculates dynamic bindings, and (2) rendering produces bitmap images. Recalculation of all dynamic bindings in the visualization must occur as part of the prerendering phase for snapshot $\times$ snapshot consistency to be preserved between all pairs. Semantically, when recalculation occurs, the stored values are given as a parameter to the dispatcher associated with the layer's dynamic binding consumes block.

The mutual exclusion between analysis and prerendering work helps preserve consistency (see Section 4.7). However, in order to ensure consistency is preserved, binding must be refined as well. In Chapter 4, a GST call was placed in an event chain before all state modifying operations. Preserving consistency requires that the GST not be called during dynamic binding calculations. However, binding itself is a state-modifying operation: It modifies the state of the layer itself. To avoid this issue, we modify the implementation of the binding operator to keep the *observable* state stable during the dynamic binding.

Modifying the state of the layer (i.e., binding the values) cannot be eliminated in dynamic binding, but it can be deferred. A special *global-bind* operator is used instead of the standard bind. Global-bind accepts bindings in the same way as binding *except* it does not actually update the associated layer. Instead, it caches all updates in a non-reactive behavior with an otherwise unused name. When all dynamic bindings on all layers (thus the *global* tag) have completed calculating, the global-bind commits all of the updates. This is triggered through a special event, similar to how render is triggered. Therefore, all calculations occur in the same *accessible* global state, but not the same *actual* global state. This change preserves the conditions that the GST operator tracks.

Dynamic binding introduces a potential problem with the layer operator's *find* facet. In the simple case, find always returns exactly what is currently in the layer. This does not violate any formal notion of consistency, but it can lead to unusual circumstances where the formal notion does not conform to informal expectations. To preserve the intuitive notion of consistency, layers with dynamic bind must return tuples from find that reflect a dynamic binding *if prerendering were to occur at the time of the find*. This calculation is done without modifying the contents of the layer. This preserves both the intuitive notion and the formal notion of consistency. The simpler version of find can be retained as an optional operator if desired; the simple version will be denoted "fastFind" in the remainder of this document because it does not perform calculations that the updated find operation does. (Note: this technique implies that mark identity cannot be dynamically bound, a restriction enforced syntactically.)

THEOREM 7.2.1. *Dynamic binding properly provides snapshot consistency.*

PROOF. Recall that for a variable to be snapshot consistent inside of a group, it must be the case that all the values of the variable come from the same global state (Section 4.7). Since global states are established by composing stateIDs (see Section 6.6), prerendering must occur without any stateful translations.

Per the semantics, the GST changes whenever the *observable* analysis state changes. This typically occurs whenever a non-query facet of a non-function is invoked (including standard binding). This condition is avoided by three tactics. First, prerendering is mutually exclusive with other analysis. Therefore, only analysis specifically identified as part of prerendering can occur while the dynamic bindings are being calculated. Second, all prerendering analysis is, *by definition*, done on counterpart operators (e.g., *query* facets that adhere to the conventional definition). Third, global-binding replaces standard binding to cache individually computed binding updates into an unobservable behavior.

The first tactic ensures that all tuples processed during prerendering are part of prerendering. The second tactic ensures that a GST update is not triggered by any analysis except binding. The third tactic ensures that a GST update is not required for binding.

Therefore, no GST updates occur during dynamic binding calculations. Therefore, dynamically bound attributes satisfy the requirements of snapshot consistency. □

THEOREM 7.2.2. *Dynamic binding provides snapshot $\times$ sequence consistency.*

PROOF. For a sequentially consistent attribute Q and a snapshot consistent attribute S to be consistent, (1) the GST value for S must be greater than or equal to the largest GST value of Q, and (2) the difference must be the minimal possible given the data loaded (as described in Section 4.7).

Both conditions are met by the fact that the analysis state used to compute the dynamic bindings is derived from the analysis state used to create the sequentially consistent attributes. In fact, the analysis state used is the state formed by the processing of the last-loaded data point. The GST of the analysis state used to calculate dynamic bindings is the largest GST yet encountered. Therefore, it is trivially greater than or equal to the largest GST of any sequentially consistent attribute. The GST value used in dynamic bindings is also constant while calculating dynamic bindings (see Theorem 7.2.1).

Condition two is satisfied by the combination of mutual exclusion of analysis and prerendering – with prerendering using the latest state from analysis. Since the GST does not change during dynamic binding and is currently its largest encountered value, it reflects the increase caused by the most recent change in any given sequential attribute. This makes the GST at the time of dynamic binding the smallest one that satisfies the minimum condition for all possible snapshot $\times$ sequence pairs. □

THEOREM 7.2.3. *Dynamic binding ensures the visualization is consistent.*

PROOF. Given that all dynamic bindings are calculated with the analysis in the same state (as indicated by the stable GST), this property follows directly from the earlier proofs. □

```
stream ticker(i)

layer Lines[LINE]
from ticker
   ID: i
   (X2, Y2): Random["X","Y"]() -> (X,Y)
   (X1, Y1): Sub1(i) -> Lines.Find(diff)
            -> (X2,Y2)
   COLOR:* Max[1..n](i) -> Div(i, max)
            -> @color{RED, q}
```

FIGURE 7.10. Stencil program with poor dynamic bind scheduling. The root issue is that a Find is executed, but the dynamically bound COLOR is not used. Therefore, the fast find variant is preferable.

7.2.1. Locals. Dynamic binding can be applied to a local calculation. However, in the normal form, locals are turned into stream definitions and divided from the layer, which makes it difficult to support dynamic binding. Therefore, locals involved in dynamic bindings need additional treatment. In addition to being lifted out, a *query*-facet-only version of the dynamically bound locals is moved to prefix the dynamic binding rule. Corresponding changes to the tuple references are also made.

7.2.2. Optimization. Scheduling dynamic binding greatly impacts its cost. The pitfalls and their amelioration are shared with other constructs, and are discussed in detail in Section 9.3.

Specific to dynamic binding is the implementation of a layer's *find* facet. If a mark is requested from a layer through find, but no dynamically bound values are used, the dynamic calculation is wasted. Figure 7.10 provides an example of this circumstance. Providing slow and fast find implementations, as described in Section 7.2, allows this situation to be avoided. Selection of the appropriate find implementation can be done automatically by discerning if a the program accesses a dynamically bound field returned from a find. Using the whole mark tuple should be treated as using a dynamically bound field. If the calculation using find and referring to a dynamically bound value is itself dynamically bound, the fast variant can be used safely for initial calculations, but the more expensive find must still be used in prerendering. This type of optimization can be achieved with the help of modules, as described in Section 9.2.

7.2.3. Limitations. Though arbitrary dynamic bindings can be expressed syntactically, there are two practical limitations. First, as mentioned earlier, identities cannot be dynamically bound. Having dynamically bound identities makes the Layer's *find* facet impractical. Second, and more importantly, dynamically bound fields based on find cannot form cycles of dynamically bound fields. When a dynamic binding refers to a layer, it implies an ordering to the dynamic bindings (the layer referred to must be able to complete its calculations first). However, if a cycle exists then no such order can be guaranteed. Any given cycle may not be infinite (layers are finite and the subsequent *find* requests may shift the ID every time, eventually a nonexistent ID will be requested). However, termination is impossible to check in the general case. As an implementation choice, Stencil currently disallows such cycles.

8
Visual Abstractions

This chapter explores how the foundational elements (streams, tuples, layers and operators) combine into more abstract concepts in the visualization domain. Automatic guide creation in Section 8.1 presents one technique for deriving guides from a Stencil program. A consistency-preserving animation system is also presented in Section 8.2. In both cases, the metadata system and semantics are used to demonstrate that the transformations are safe (i.e., they preserve consistency and do not modify the results of other calculations).

8.1. Guides

Reference marks provide the context required to interpret a visualization. Some, such as axes, labels and similar guides, have distinct semantics relative to other marks in a visualization. Unfortunately, library-based visualization frameworks do not employ these distinct semantics to assist in the creation of guides. In most circumstances, visualization frameworks allow custom analysis to be specified for a visualization, but 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 chapter describes Stencil's implementation of reference-mark construction.

Simple construction of reference marks is achieved by visualization applications (like Tableau [**108**] or Excel [**102**]) and widget-based libraries (like the InfoVis Toolkit [**45**]) by limiting the 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 set of visualizations is impractically large. Some state-based frameworks enable automatically producing guides by synthesizing guides from the final prerendering data state (e.g., Prefuse [**69**] and ggplot2 [**122**] to varying 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 section describes sufficient language semantics and metadata to achieve guide creation in the general case. Stencil's 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 attain desirable properties, including the potential to guarantee correctness. The system we describe has been implemented in the Stencil visualization system [**30**] to support declarative guide creation.

8.1.1. Declarative Guides.

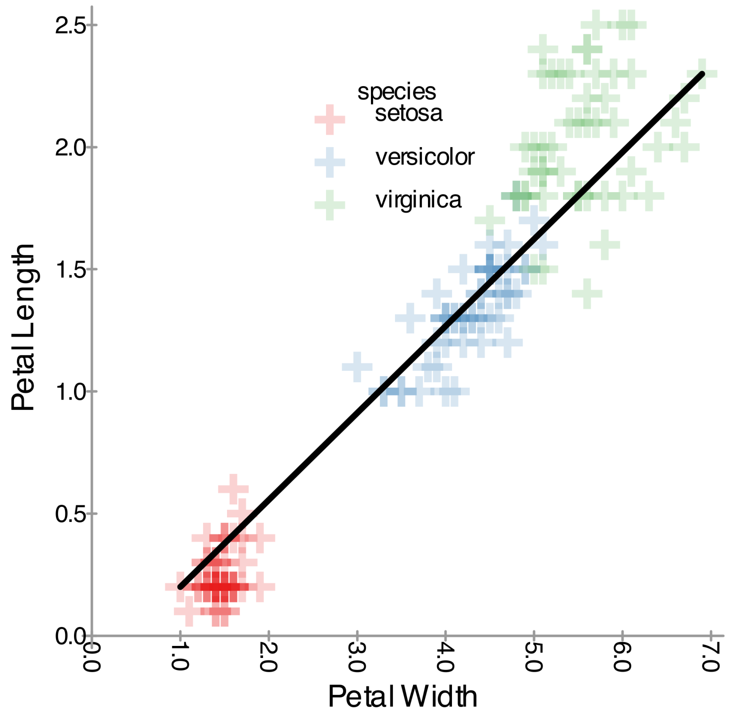
In general, reference marks belong in one of three classes: annotations, direct guides and summarization guides. Annotations include subjective notes and comments; they relate a visualization to external entities through a process orthogonal to the visualization. By their nature, annotations cannot be automatically generated and are not dealt with further. In contrast, guides represent information inherent to the data 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 the summarization style). Guides of both types (illustrated in Figure 8.1a) are treated in this section. The division between direct and summarization guides is similar to division between *scales* and *annotations* in *The Grammar of Graphics* (GoG) [**124**], but GoG did not have a separate category for annotations as defined above.

Declarative programming specifies *what* is to be accomplished, instead of *how* to accomplish it [**77**]. 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 program performing the data analysis. For example, regardless of the analysis or data involved in mapping input data to the -axis in Figure 8.2, the declaration on line 5 provides a suitable guide. The underlying system inspects and modifies the program to determine how to construct guides. This section discusses the framework support required and the analysis and modifications performed to enable declarative guide creation.

8.1.2. 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.

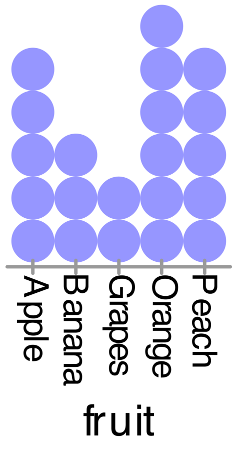


(a) Anderson's Flowers Cell

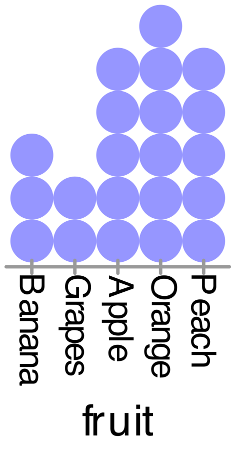
```
 1  import BrewerPalettes
 2
 3  stream flowers(sepalL, petalW, sepalW,
 4                         petalL, species, obs)
 5
 6  layer FlowerPlot
 7  guide
 8    trend from ID
 9    legend[X:20,Y:90] from FILL_COLOR
10    axis[guideLabel: "Petal Length"] Linear from Y
11    axis[guideLabel: "Petal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

(b) Anderson's Flowers Program

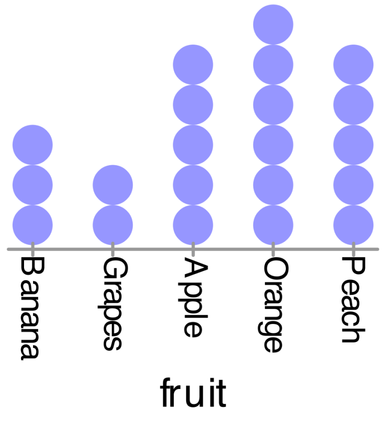
FIGURE 8.1. A single-cell part of the Anderson's flowers trellis visualization [56], augmented with a trend line. Guide declarations appear on lines 8–12. The legend in this program demonstrates some of the design support by adopting the characteristic shape (contrast with the legend shape in Figure 8.7).



(a) Rank



(b) Index



(c) Wider Index

```
1   stream survey(fruit)
2
3   layer plot
4   guide
5     axis from X
6   from survey
7      ID: Count()
8      X:* Rank(fruit) -> Mult(5, Rank)
9   /* X:* Index(fruit) -> Mult(5, index)*/
10  /* X:* Index(fruit) -> Mult(10, index)*/
11     Y: Count(fruit) -> Mult(5, Count)
12     REGISTRATION: "CENTER"
13     FILL_COLOR: Color{150,150,255}
```

(d) Stencil Program

FIGURE 8.2. A series of plots based on a fictitious survey of fruit preferences. The input data are presented as a list of fruit names. The visualization program is found in Figure 8.2d, with Figures 8.2a, 8.2b and 8.2c using lines 8, 9 and 10, respectively. The axis layout in Figure 8.2a is based on the "Rank" function which sorts the values, while Figure 8.2b is based on the "Index" function which tracks the order that elements are first seen. Figure 8.2c spaces the x-axis out more, but otherwise is the same as Figure 8.2b. Each time, the analysis is used to construct the x-axis changes, but the guide declaration in line 5 does not need to be updated.

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 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 uses fewer resources to produce the same result is preferable to one that uses more resources.

Customizable: Permits data reformatting for presentation. Data and graphic aspects of a guide often require reformatting. For example, a guide for an 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 influencing visual values 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 spaced examples.

Separation supporting: Can use separate guides for distinct encodings of values. 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 data to which they refer. 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 all useful guides: They distinguish a guide from an annotation. The other attributes provide a means to reason about the guide system's ability to represent details of the analysis and presentation (e.g., per Wilkinson [**124**]).

The 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 8.1.3. Simplicity, redundancy awareness, attribute crossing, separation supporting and design sensitivity are presentation-focused attributes. Independent of how guide contents are determined, these attributes affect how the guide is displayed. They are discussed in Section 8.1.7.

8.1.3. 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 8.4. This section ties automatic guide creation to the execution semantics and operator metadata. 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 8.3.

$\mapsto$: Operation counterpart (as defined in Section 3.3.1 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 8.3. Description of notational elements.

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 [**1**]. Memory is treated as an input parameter, so all operators are functions, making the composite an applicative transformation chain [**4**, **72**]. As an applicative framework, control flow can be treated separately from the computations involved, similar to the treatment done for coordination languages [**54**]. In effect, the details of the transformation performed can be treated separately from the coordination of the transformation. Using a linear representation makes transformations involving branches, merges and loops less obvious in implementation. We assume that all such nonlinear 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 $(tuple*, \mathbf{M})$ and return two values $(tuple*, \mathbf{M}')$. In practice, this batch-request behavior can be achieved by wrapping a data-flow operator accepting $(tuple, \mathbf{M})$ in a for-each loop (this technique is pursued further in Chapter 9). 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 [**23**] by modifying how memory states are treated between calls and the contents of the input list [**32**]. An analysis chain is given in Equation (1); in general, user-supplied analysis chains start with a read from some source.

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

This set of semantics enables the guide system properties from Section 8.1.1 while remaining applicable to a variety of programming styles.

Transformation operators are expected to have two pieces of associated metadata. First, all operators are expected to be able to provide a *counterpart* operator. Counterpart is formally defined in Section 3.3.1. Informally, an operator, G, is counterpart to A if G calculates the same result as A but does not modify any memory. If G cannot fulfill this contract, it may return a value that A never does to signal that fact. When satisfied, the counterpart relation provides three important properties. First, G may be executed as often as needed without changing the resulting guide, providing flexibility in scheduling its execution. Second, G does not interfere with A [**89**], making the resulting system subordinate (as defined in Section 8.1.1). Finally, 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 $CP(A) = G$ such that G corresponds to A.

The second piece of metadata that all operators must provide is the categorization with respect to memory usage, presented in Chapter 3: function, reader, reader/writer, 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 (time, network, user input, etc.) or randomness are typically opaque. Such operations 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 guide creation, function versus reader versus reader/writer only determines how difficult it is to construct G given A.

8.1.4. 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, and so forth. This leads to two general parts: a transformation and a display system.

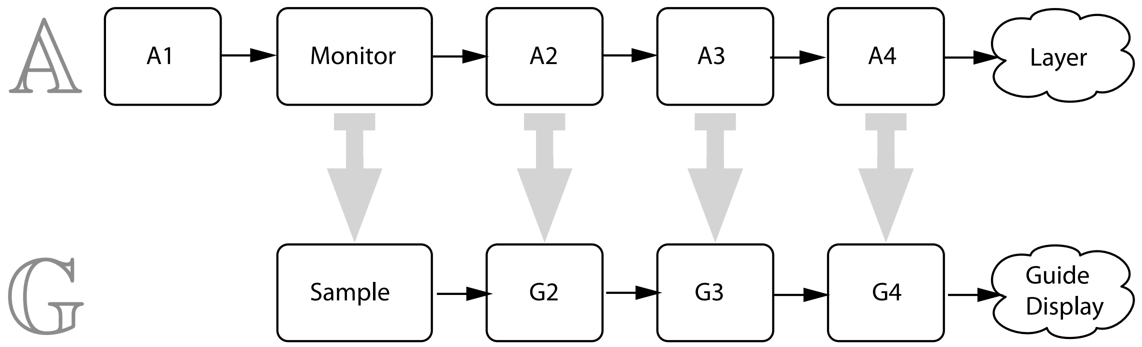


FIGURE 8.4. The guide system constructs $\mathbb{G}$ given a precursor to $\mathbb{A}$. Construction is accomplished by inserting a *Monitor* operation and transforming all operations from *Monitor* forward into counterpart operations. When executed, $\mathbb{G}$ passes its results to the guide display system.

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 Section 8.1.5. Changes required to create summarization guides are given in Section 8.1.6. The shared display system is described in Section 8.1.7.

8.1.5. Direct Guides. A correct guide descriptor for a direct guide, like Prefuse's ValuedRangeModel [**69**], 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 $\mathbb{A}$.

Correspondence is expressed visually in Figure 8.4 and defined in Definition 8.1.1. The counterpart condition indicates that $\mathbb{G}$ may only contain counterparts to operations in $\mathbb{A}$. The contiguous condition ensures that operations in $\mathbb{G}$ appear in the same order and with the same dependencies as their counterparts in $\mathbb{A}$. The tail condition indicates that $\mathbb{G}$ must include the end of $\mathbb{A}$. Conditions contiguous and tail combine to indicate that the guide need not reflect the full analysis process, but it must include the final steps of $\mathbb{A}$. The final two conditions stipulate that the guide system be nonempty and that analysis operators be uniquely represented.

DEFINITION 8.1.1. $\mathbb{G} \mapsto \mathbb{A}$ *if and only if*

(counterpart)	$\forall Gn \in \mathbb{G} : \exists Ax \in \mathbb{A}$ *such that* $Gn \mapsto Ax$
	$\forall Gn, Gm \in \mathbb{G}$ *and* $\forall Ax, Ay \in \mathbb{A}$
	if $Gn \mapsto Ax$ *and* $Gm \mapsto Ay$ *then*
(contiguous)	$Gn \circ Gm \Rightarrow Ax \circ Ay$
	$\exists Gn \in \mathbb{G}$ *where* $Gn \mapsto Ax$
(tail)	*such that* $\nexists Ax+1 \in \mathbb{A}$ *where* $Ax \circ Ax+1$
(nonempty)	$\lvert\mathbb{G}\rvert > 0$
(unique)	$Gn \mapsto Ax$ *and* $Gn \mapsto Ay \Rightarrow Ax = Ay$

Acquiring G corresponding to A is the purpose of the counterpart relationship given earlier. A constructive solution to building a G given an A is presented in Section 8.1.9. 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 a transformation is described, two support operations need to be discussed: *Monitor* and *Sample*. *Monitor* implements the identity relation but also tracks information about what has been observed in its state ($Monitor(x, \mathbf{M1}) = (x, \mathbf{M2})$, with **M1** not always equal to **M2**. The memory state of *Monitor* is used by *Sample* to produce a set of sample tuples. The parameters passed to *Sample*, including the quantity and type information produced by *Monitor*, 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 initialization from Equation (1)).

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

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

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

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

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 in accordance with the tail and contiguous conditions of Definition 8.1.1.

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 Equation 8.1.1.

After $\mathbb{G}$ is constructed, it is used to create a guide descriptor. By construction, the first operator in $\mathbb{G}$ is always *Sample*. Operator *Sample* is defined so it 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 basic guides; however, it must 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 for a discontinuity avoided in the original analysis to be encountered 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 $([],\mathbf{S})$. 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 operator call conventions. However, because such ambiguities are fundamentally related to analysis, ambiguity detection in general logically belongs in the analysis system. The **S** and **V** pair process is therefore extended to include identification of duplicates in **V**. Such ambiguities are stored in the guide descriptor as sets of values, yielding a final guide descriptor of the form $(discont,(\{s+\},v_1),(\{s+\},v_2)\ldots)$.

8.1.6. Summarization Guides. Summarization guides produce a descriptor compatible with that of direct guides. However, summarization guides primarily provide information about the result set. A trend line on a scatter-plot is a common summarization guide.

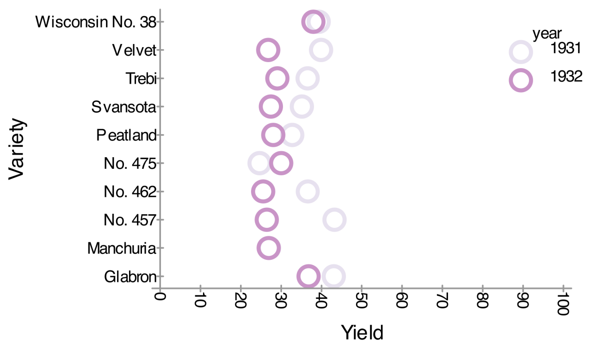
The principle requirement of a summarization guide is complete access to the result space. The simplest form of access is an iterator of identities for all stored items and a function to retrieve those items. Acquiring an iterator of identities takes the place of *Sample*, and collecting entries from the iterator is the whole of $\mathbb{G}$. The resulting descriptor has the identity as the input value and all associated visual values as the results (the discontinuity warning list is guaranteed to be empty).

8.1.7. 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 be 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 8.1.1).

8.1.8. Analysis and Extensions. The presented system provides most of the desirable attributes given in Section 8.1.1. 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 created (each associates multiple inputs with the same visual component of a guide). Separation support, design sensitivity and simplicity depend on the display and associated guide requesting systems. The preceding process does not directly support these properties, but it does not preclude them either.

Formatting data for presentation (i.e., supporting projecting) can be approached in a variety of ways. A general solution is to use a postprocessing 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.). Postprocessing can be used to support the projecting property, but it jeopardizes correctness and consistency. When arbitrary calculations are possible, the correspondence



(a) Becker's Barley Cell

```
import BrewerPalettes
stream Barley(year, site, yield, variety)

const MAX: 100
const MIN: 10

layer BarleySite
guide
  axis[guideLabel: "Variety", X:0] from Y
  axis[guideLabel: "Yield", round: "T",
        seed.min: MIN, seed.max: MAX]
    Linear from X
  legend[X:75, Y: 60] Categorical from PEN_COLOR

from Barley
    filter(site =~ "University Farm")
    ID: Concatenate(variety,year)
    PEN_COLOR: BrewerColors["PuRd","BLACK"](year)
    X:* Scale[min: 0, max:100,
              inMin: MIN, inMax: MAX](yield)
    Y:* Rank(variety) -> Mult(7,_) -> Add(-5,_)
    REGISTRATION: "CENTER"
    FILL_COLOR: Color{CLEAR}
```

(b) Becker's Barley Program

FIGURE 8.5. A single cell of the Becker's Barley trellis and corresponding source code.. This example shows the directness of the automatic guide system and illustrates some weaknesses (discussed further in Section 8.1.11).

between $\mathbb{G}$ and $\mathbb{A}$ is violated if the postprocessor ignores its inputs. The alternative to post-processing is a proliferation of special cases for specific reformations (this can be seen in part in R [**97**] with the multitude of optional formatting arguments to plotting functions). This trade-off needs to be carefully weighed: flexibility and parsimony versus correctness and verbosity.

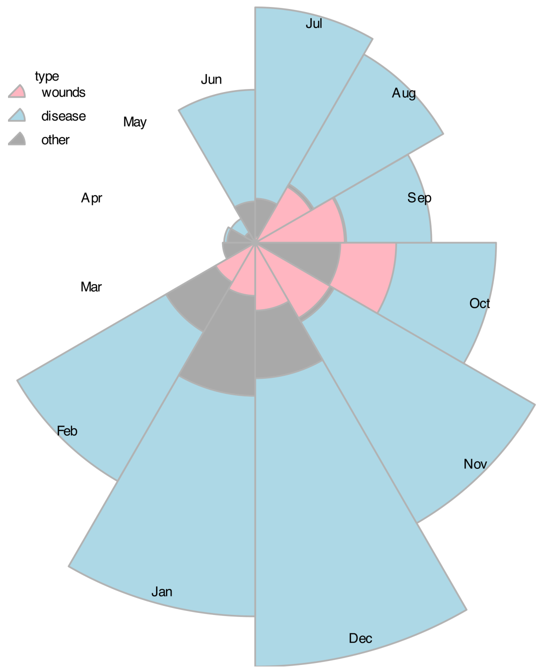
8.1.9. Stencil. The core of the described guide system has been implemented in the Stencil visualization system. Stencil supports basic guide descriptor creation (not including discontinuity or ambiguity reporting) and arbitrary postprocessing.

Figures 8.5a and Figure 8.5b demonstrate direct guides: axis and legend. Lines 8–14 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) postprocessing 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 23).

Part of the Crimean Rose visualization [**62**] and a program listing are given in Figure 8.6. This visualization uses a legend and point labels with postprocessing. The point labels use the mark ID as the text and the default position is at registration point of the associated mark. The postprocessing enables custom formatting and positioning of labels; these effects are defined on lines 11–14 of Figure 8.6b. An example of the trend-line construction and presentation can be seen in Figure 8.7.

By specifying two attributes in a guide definition, redundancy and attribute crosses are supported. If the two attributes are based on the same input data, then a single, redundancy-encoding guide is created (as in Figure 8.7). If the two attributes are based on separate data, then an attribute-crossing guide is created (see Figure 8.8).

The implementation presented is efficient in that it requires only two additional operations (monitor and sample), 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 from (1) maintenance of the input space descriptor (at worst, linear in the data size but often constant) and (2) storage of the visual elements of the guide representation (linear in the sample size produced). By

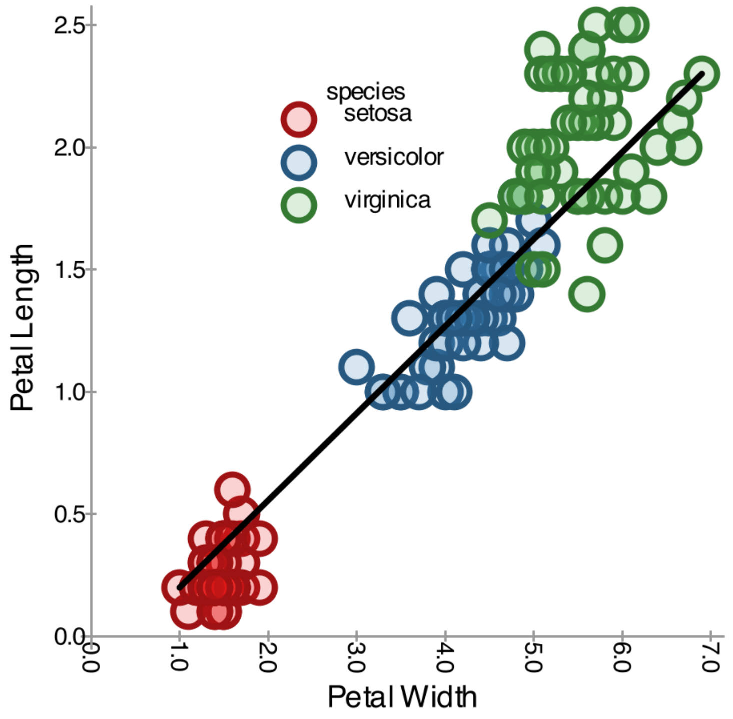


(a) Nightingale's Crimean Rose

```
1   import Dates
2   import Geometry
3
4   stream Deaths(date, type, count)
5
6   layer Rose["SLICE"]
7   guide
8     legend[X: -75, Y: 50] from FILL_COLOR
9       label.REGISTRATION: "BOTTOM_LEFT"
10    pointLabels from ID
11      TEXT: ParseLabel(ID)
12      (X,Y): LabelLayout(ID, OUTER)
13      REGISTRATION: "CENTER"
14      FONT: Font{4}
15
16  from Deaths
17     local(month, year) : Parse[f:"M/yyyy"](date) -> (Parse.month, Parse.year)
18     ID: Concatenate(type, ":", local.month)
19     FILL_COLOR: ColorBy(type)
20     PEN: Stroke{.5}
21     PEN_COLOR: Color{Gray70}
22     SIZE:* MonthMin(local.month, count)
23                                      -> Scale[min:0, max:250](count)
24     Z: Mult(-1, count)
25     (X,Y): (0,0)
26     (START, END): Sub1(local.month) -> Partition(_)
```

(b) Crimean Rose Program (partial)

FIGURE 8.6. Florence Nightingale's visualization of casualties in the Crimean war. Point labels with postprocessing produce the month abbreviations (see lines 11-14). (See Appendix A (Section A.7) for a full listing.).

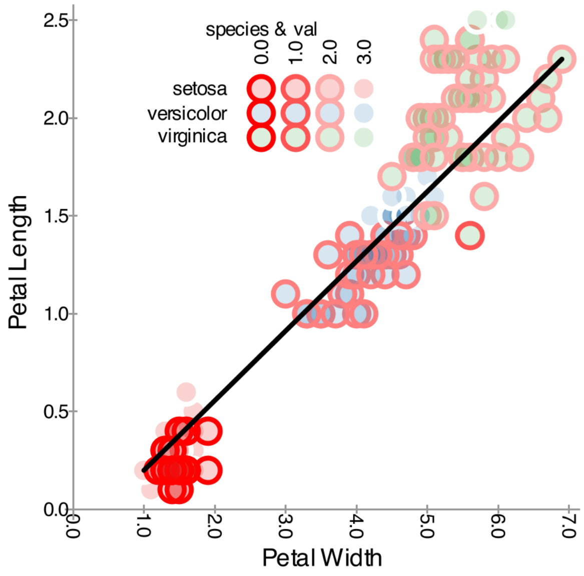


(a) Anderson's Flowers: Redundancy Encoding Guide

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

(b) Stencil Program

FIGURE 8.7. 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 cross-product guide in Figure 8.8, but since both fields are based on the same input, the redundancy encoding style is used.



(a) Anderson's Flowers: Attribute Cross Guide

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

(b) Modified line 18

FIGURE 8.8. Anderson's flowers with a petal width encoded in pen-color. Only the definition of pen color changed from the program show in Figure 8.7b but Stencil determines that a cross product is required since the fill and outline are based on different inputs.

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 system 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 postprocessing statements. The postprocessing statements admit any valid computations, and thus have the power and risks discussed in Section 8.1.8 (the input tokens to postprocessing are the entries of the guide descriptor). The requested guide type and parameters determine the *Sample* and *Monitor* operators used. Default *Sample* placement is overridden with a special link operator, $-\# >$, in a targeted analysis.

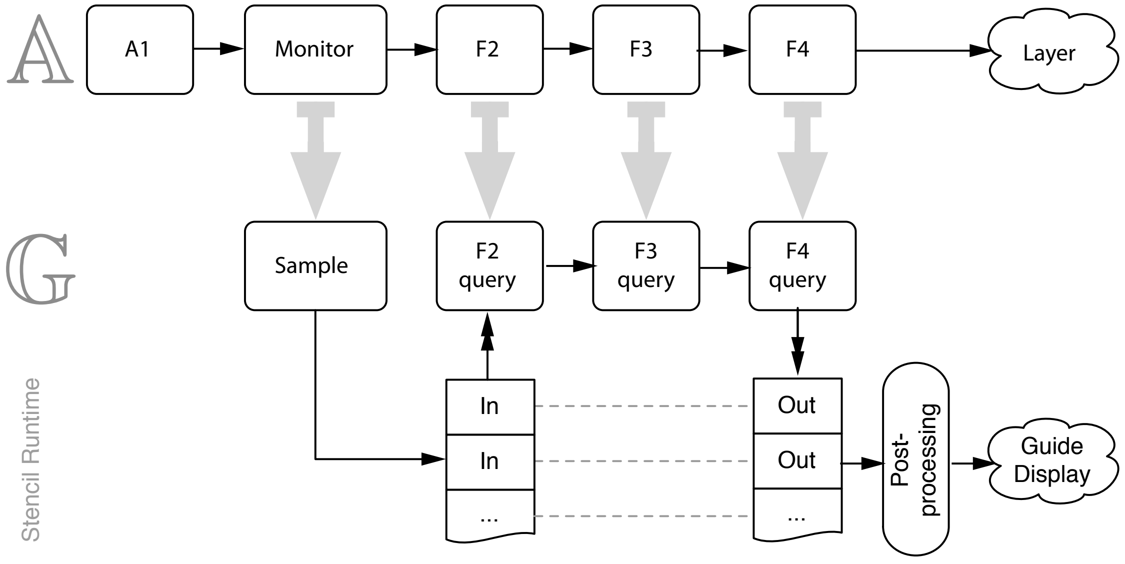


FIGURE 8.9. The guide process implemented by Stencil. The major distinctions are the inclusion of postprocessing and maintenance of the input/output sets separate from analysis.

Stencil's metadata system is rooted in the operator libraries, so metadata are specified by the library creator for use by Stencil subsystems. All of the metadata used for guide generation is used by other subsystems 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* facet that satisfies the counterpart requirements (from Definition 3.3.1). Since transformations are implemented as objects, the counterpart operator appears as a query method in the class definition. If an operator's implementation does not explicitly provide a query method, it can be synthesized using the provided metadata and a deep clone. For function and reader operators, the counterpart is synthesized as an alias for the default transformer. Opaque operators, by definition, cannot provide a reasonable counterpart operator. A reader/writer operator can provide support for *query* in multiple ways. The most direct is to perform a deep clone before each invocation, calling the stateful transformation on the clone, and then discarding it.

Stencil operators do not take the batch list arguments described in Section 8.1.4. Therefore, the control flow is modified (see Figure 8.9). 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 discontinuity reporting is not supported at this time. Full discontinuity reporting is not currently supported, but it could be achieved by adding a *discont* facet that has the call semantics for G described earlier, but built using Stencil's counterpart facets (i.e., *query* facets) or that directly performs discontinuity analysis. Alternatively, discontinuity support could be achieved with a separate facet dedicated to that task. Concurrent support for both data-flow and data-state style operators has been explored in Stencil [**32**] and could also be extended to support discontinuity reporting.

Scheduling of guide-system execution has a significant 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 are loaded. However, in a dynamic data system, such a clear-cut schedule is generally not possible. An optimal recalculation schedule recalculates guides only when this will result in a change in the visual output. An approximation of the optimal recalculation schedule is realized with the help of the *stateID* facet and described in Section 9.3.

The Stencil implementation currently restricts the placement of *Monitor* beyond the restrictions given in Section 8.1.4. 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 8.1.1, though some trade-offs are made. A basic guide can be declared in four tokens: **`guide <type> from <att>`**, with customization options available. This parsimony is possible due to Stencil's succinct statement of dependencies and broad metadata. Redundancy, attribute crossing, separation support and design sensitivity are all represented in the provided examples.

The guides are guaranteed to be complete (reflect the analysis), consistent (reflect the data) and subordinate, provided that postprocessing is not used. Postprocessing support enables customizable guides, but also makes it possible to violate these other properties. Use of postprocessing 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 8.6), a transformation that is consistent with the data. However, an alternative operator on line 11 could change "December" into "June" just as easily, invalidating the consistency. A similar hazard is present with respect to a guide being subordinate if reader/writer facets are used in the guide system. There are no firm rules for when a manipulation invalidates completeness or consistency, or when it makes a guide nonsubordinate. The Stencil design favors the flexibility of postprocessing over the *guarantee* of completeness and consistency. However, all default constructions provide complete and consistent guides.

THEOREM 8.1.1. *Modifying the analysis chain to support guide creation does not modify the result of analysis.*

PROOF. Guide calculation is done in the prerender phase (described in Section 7.2). Therefore, changes to analysis can occur through shared memory mutation. The guide system interacts with mutable memory in two places. The guide system also inserts the seed operator into the analysis chain, which has the potential to influence analysis results.

First, the seed operation could change the inputs to the analysis chain. However, the analysis context seed, by definition, just echoes its inputs.

Second, calculating guides could change the internal operator state. This would cause future analysis to change based on past guide calculations. The guide system works only with *query* facets (e.g., counterpart operators) in sample creation, and *query* facets are the only defined legal operators in postprocessing (though this is not enforced). If *query* facets are properly implemented, they do not modify state and thus cannot modify analysis results.

	Complete	Consistent	Subordinate	Efficient	Custom	Simple	Redundancy	Att. Cross	Separation	Design
Protovis					X		*	*	*	
Prefuse	-	-			X		*	*	*	
ggplot2		*	X	X	*	-			X	X
Stencil	*	*	X	X	X	*	-	-	X	X

TABLE 8.1. Comparison of the guide systems of selected visualization systems. An ideal system would hold all the listed characteristics and capabilities. A cross (X) indicates an attribute held by a system; a star (*) indicates a conditionally present attribute; and a dash (-) indicates a partially present attribute.

Third, guides create marks which could conflict with existing marks. However, guides following the query-only rule in postprocessing can only create marks through their binding mechanism. This binding mechanism is in its own layer and thus cannot interfere with other analysis.

Therefore, provided that no mutative postprocessing is requested, guide creation does not modify the result of analysis. □

Since guides without postprocessing do not modify the analysis state, when guides are created from a consistent state they (1) maintain the consistency of that state, and (2) are consistent with that state. Similarly, if the postprocessing relies only on functions and readers, consistency is preserved.

8.1.10. Comparison. Constructing guides in a framework-supported, disciplined way ultimately saves programmer effort. This section compares the Stencil implementation of automatic guide generation to other frameworks' implementations.

Table 8.1 is a comparison of four visualization systems along the dimensions described in Section 8.1.1. Protovis and Prefuse, lacking well-defined guide support, can still produce common guide types, but require definitions with complexity comparable to the definition of the related analysis. Redundancy-aware, crossing or separate guides are supported to the degree that the programmer is willing to construct them. Furthermore, with the exception of some axes in Prefuse, neither completeness nor consistency are provided by the frameworks. ggplot2 provides a capable automatic guide system. Consistency is guaranteed if default labeling is used, as the values are drawn directly from the stored data table. However, completeness (i.e., representing the analysis) is not provided because labels are produced using the *last* analysis stage results, instead of the original inputs. Other values may be manually substituted for labeling, but this is done at the risk of invalidating consistency. Stencil implements the majority of the system described in this chapter. Section 8.1.9 discusses the conditions, trade-offs and omissions in more detail.

The Protovis visualization framework is based in declarative construction [**11**]. 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 other Protovis objects (e.g., the "ticks" method from pv.scale objects). However, guides themselves are not distinguished as separate entities; combining the relevant components is done using standard analysis operators. Protovis is a natural target for comparison because it has information about both the input and output space and it has a declarative basis.

Eight visualizations were implemented in Stencil to match those provided on the Protovis website. The guide portions were then isolated and compared. A portion of a program was included in the fragment if it is only used in guide construction. An example Protovis guide description is given in Figure 8.10; this Protovis fragment corresponds to guides found in the Stencil program in Figure 8.5b. Table 8.2 presents the guide description token counts for all the visualizations considered. The Stencil guide version was consistently half the length of the of the Protovis one, indicating an advantage for its declarative guide system.

Subjectively, the Stencil definitions were consistently more general than the Protovis ones. Five Protovis definitions explicitly encode information about the input data (e.g., the legend in Figure 8.10 line 13 lists the years found in the data). None of the Stencil definitions included source-data information. 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 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's guide declarations. This source of Stencil verbosity may be ameliorated by having guides inherit graphic attributes from the layers to which they are applied. Protovis provides such inheritance through its scene graph mechanism.

ggplot2 is a popular visualization library for the R statistical environment. ggplot2 automatically constructs axes, legends and trend lines 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 ggplot2 to create consistent guides (reflecting the data). Unfortunately, since only the final analysis state is directly available, it is more difficult to construct guides reflecting earlier analysis states (i.e., complete guides). A token-based comparison of ggplot2 to Stencil is also presented in Table 8.2. 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, nonstandard 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 column that the results being presented actually depend on (thus, consistency is only conditionally supported in Table 8.1).

ggplott2 has zero-token guide creation for axes on certain plots. In certain circumstances, no explicit guide request is required, but the axes will still be properly produced. Building a zero-token solution for Stencil would require improving the default placement

```
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 8.10. Protovis program fragment for producing guides on Becker's Barley [**57**]. 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 is performed in lines 3–8 and the legend for year coloring is defined in lines 11–17

of the monitor operator to respect the need to cleanly divide the data dependencies. However, the default guide definitions themselves are trivial. One disadvantage to a zero-token solution is indicating when guides should *not* be produced. For example, in spring-force based graph visualization (such as the ones in Appendix A (Section A.2) and Appendix A (Section A.12)), the X and Y axis locations do not have semantic meaning; the relative distance between elements is the focus in these visualizations, not the position of individual elements. Determining if an opt-in or opt-out guide-creation system is best is an avenue for future investigation in the Stencil implementation.

8.1.11. Discussion. Using the guide-creation system described in Stencil demonstrates the practicality of the approach. It also illuminates two shortcomings, both stemming from a need to restate 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 8.5b where the Scale operator on line 20 has its max and min arguments manually copied and represented as 'seed' arguments in the guide declaration on line 11. This manual transfer is dictated by the fact that the provided *Monitor* and *Sample* operators (which together constitute the sample operator of Section 8.1.4) are based entirely on input

Visualization	**Stencil**	**Protovis**	**Ratio**	**ggplot2**	**Ratio**
Bar Chart	44	55	4:5	34	5:4
Becker's Barley	33	64	1:2	34	1:1
Andrson's Lilies	7	26	1:4	9	1:1
Scatter Plot	15	69	1:5	29	1:2
Line Chart	14	60	1:5	20	3:4
Point Labels	5	14	1:3	7	1:1
Crimean	65	49	6:5	104	2:3

TABLE 8.2. Total tokens dedicated to the guide creation in selected examples. Protovis source code was taken from the Protovis website, ggplot2 examples were produced internally. In all cases, punctuation was considered a separator, all nonpunctuation groups were considered tokens. Therefore, numbers, parameters, procedures and mathematical symbols are all tokens but parenthesis, dots, and so on are not. Operators that in math-like notation were considered tokens ("3+4" is three tokens). The Andrson's Lilies line reports just the tokens to produce the legend, not the axes (which were compared using scatter plot instead).

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. (Note: a guide created without the extra parameters is still a correct guide, but it is not as good.) Ideally, some means of communicating that an operator expands the input range would be provided. The earlier iteration of the guide system [**31**] allowed categorical operators to provide a sample instead of relying on dedicated seed/sampler pairs. This was removed from the current system to simplify the operator interface and transformations, but a similar system could be reintroduced to allow operators to determine the sample space. Such a system would require more extensive operator metadata. 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 known.

Implementing this system outside of Stencil appears plausible but will require distinct techniques depending on the specific framework. Generally speaking, the system can be implemented when a visualization library creates its own execution semantics. In Prefuse [**69**], it appears possible to create guides automatically for *calculated fields*, but more difficult with calculations done in *actors.* The reason for this distinction is that calculated fields provide a clear set of evaluation semantics that can be directly inspected, while actors rely on the underlying Java semantics. Protovis [**11**] could likely include declarative guides by attaching a "guide" method to the pv.panel object. A VTK [**74**] implementation might be achieved by wrapping the analysis network creation calls to automatically connect a parallel guide-creation network implementing the described algebra. A tool like OverView [**127**] could also construct the connections network automatically.

The guide-creation process described in Section 8.1.4 relies on noninterference, and is thus simple. A guide process that can modify memory could be used to perform some optimizations discussed in earlier work [**31**]. This requires a more complex set of semantics than was provided in Section 8.1.3, and complicates reasoning about guide/analysis interaction by removing the subordinate property. However, the same issues were encountered when postprocessing was introduced. Investigating the effects of nonsubordinate guides may provide insight into further useful semantics for stateful visualization frameworks.

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 (1) the execution semantics of an analysis process, and (2) the semantic role that guides play in a visualization. This section presented a formalization of execution semantics and used that formalization to develop a declarative guide-creation process. The concepts of this section have been implemented in the Stencil system, demonstrating that the concepts are sufficient and practical.

8.2. Animation

Animation is a simple visual abstraction. Animation presents a series of data points over a period of time. Dynamic bindings can be used to directly construct animation. The common *ease-out/ease-in* case is supported through a syntactic extension, and translated into a dynamic binding. An animated binding is denoted with `<:`, replacing the static binding (a dynamic variant, `<:*`, replaces a dynamic binding). Animated bindings are

```
stream Stream(ID, value)

layer Layer
  from Stream
    ID : ID
    X <:  g(value)
    Y <:*  f(value)
```

(a) Input

```
stream Stream(ID, value)

layer Layer
from Stream
   ID : ID
   local(#xFuture): g(value)
   X: Layer.dFind(ID) -> Time() ->  AnimateX.set(Layer.X, local.#xFuture, RenderCounter)
   X:* Time() -> AnimateX.step(RenderCounter)

   local(#yFuture):* f(value)
   Y: Layer.dFind(ID) -> Time() ->  AnimateY.set(Layer.Y, local.#yFuture, RenderCounter)
   Y:* Time() -> AnimateY.step(RenderCounter)

operator AnimateX : Animate
operator AnimateY : Animate
```

(b) Transformed

FIGURE 8.11. Animated binding substitutions. The *dFind* facet behaves as *find*, but will return default layer values if the ID is not found in the layer.

compiled into an operator definition, a local binding, a static binding and a dynamic binding. An example compilation is shown in Figure 8.11.

The Animate operator has two facets *set* and *step*. *Set* takes a start value, an end value and a current time and records all three (echoing the first one in its return value). *Step* takes a new time and returns an interpolation between the start and end values, based on specializer arguments. By default, the time system used for animation is render counts. Render counts are accessed via the RenderCounter operator. Render counts are used because (1) they are monotonically increasing, (2) they ensure that all intermediate steps are shown, and (3) they are consistent with E-FRP semantics. Recall from Section 4.5 that the semantics presented are conditionally deterministic. If a nondeterministic operator is used, then the semantics are no longer deterministic. Wall-clock time, because its changes are not dependent on the data streams, is not determined by the input data and is thus not safe to use. Using wall-clock time would make the visualization dependent on the data presented *and* the amount of time it takes to process inputs. However, using wall-clock time does not change any of the transformations presented (it just violates the strict definition of determinism used).

The local binding created calculates the desired future value. The static binding initializes the animate operator by calling the *set* facet, and initializes the layer by setting the target attribute to the default value. The dynamic binding reads a new time and calls the animate *step* facet to get the interpolated value. This trio of bindings yields deterministic animations without introducing any new concepts to the Stencil framework.

This arrangement provides basic animation support. Modifying the characteristics of the animation (e.g., pacing, travel paths, etc.) is more difficult. The simple syntactic form, using `<:`, does not provide access to specializers, the standard parameterization mechanism. The full form can be written, but Figure 8.11 demonstrates it takes 17 times as many tokens and is not as direct. Further language extensions for controlling animation properties would philosophically fit with the rest of the Stencil system.

8.3. Conclusions

The foundational concepts of the Stencil system, augmented with the metadata system and supported by the formal semantics, enable reasoning over visualization programs. This further enables employment of visualization-focused abstractions in a consistent and predictable manner.

9 Computational Abstractions

An implementation of the Stencil language is the core of the Stencil visualization system. The semantics from Chapter 5 are the basis of this implementation. This theoretical basis enables construction and use of the visual and analytical abstractions discussed in Chapters 7 and 8. Additionally, the theoretical basis allows Stencil programs to be safely analyzed and manipulated in their own right. Treating the program in a disciplined way enables greater efficiency in the implementation, expanding the contexts in which Stencil can be applied.

This chapter describes treatments of Stencil programs that illustrate the utility of treating visualization programs abstractly. Section 9.1 describes the treatment of constants.

Module-based optimizations are described in Section 9.2. Section 9.3 treats efficient scheduling of dynamic binding and guide creation. Bulk operations also present bulk data; Section 9.4 describes program transformations that enable Stencil programs to exploit the temporal locality (and possible memory locality) of bulk data. Concurrency and parallelism are important components of modern software architecture; Section 9.5 discusses task-based parallelism in Stencil.

9.1. Constants

Compiler-based treatment of constants is a natural place to start optimization. Treatment of constant values is a simple set of compiler-based operations that can have a large cumulative effect. The three optimizations included in Stencil are constant propagation, constant folding and constant attribute lifting. Their cumulative impact is approximately 10% off the running time (9.2% to 11.4%, depending on the Stencil program and the number of data points loaded). Constant attribute lifting also has an impact on the memory consumed by eliminating the per-element allocation for the constant field.

Program-declared constant values are propagated by textual substitution of their reference locations [**104**], a standard compiler optimization. The provided operator metadata enables constant folding [**104**] in conjunction with constant propagation. Each operator facet identifies whether it is a function through metadata (see Section 3.3.2). Functions applied to constant arguments produce constant results, which are then propagated as well. The Stencil system applies constant propagation and constant-function folding as a fixed point during compilation to minimize the amount of work done at runtime. Constant propagation reduces the number of value look ups performed at runtime. Constant folding reduces the more expensive operator applications. Folding is more beneficial, but it requires constant propagation to be effective.

Each layer has a further opportunity for optimization, enabled by constant folding. If a layer includes a constant-valued attribute, then that attribute does not need to be computed at runtime. This compile-time computed attribute can be set once in the layer definition, thereby avoiding costs associated with tuple packing, results merging and storing the duplicate value. A constant-valued attribute can be "lifted" into the defaults block. Packing and merging are simple operations, but their frequency (once for each layer update) makes them effective targets for optimization. On average, constant attribute lifting in the examples found in Appendix A results in tuples with half the fields of their unlifted counterparts.

9.2. Module-Based Optimizations

The Stencil infrastructure provides opportunities for external optimizations related to operators. As described in Section 3.3, modules group operators. Furthermore, modules are factories [52] that produce operators. Arguments to the module to produce the requested operator include the **`specializer`**, enclosing higher-order operators and usage information. This arrangement allows modules to substitute operator implementations based on explicitly declared and usage-derived characteristics.

For example, the Average module uses context arguments to select between implementations of Mean. A state-free implementation is used if the Mean is not part of a Range operator (described in Section 7.1). However, a more space and time efficient implementation is used in the context of a full range operator (i.e., **`Range[ALL](@Mean, value)`**). This is one use of context characteristics to determine operator implementation.

Usage information also enables optimization of the count operator. Count is a variable-arity operator. Most such operators involve a dictionary; however, if there are never arguments, Count is a simply a stateful increment. The Stencil program does not need to explicitly indicate this optimization; the compiler collects usage information that the module uses to identify this optimization.

Module-base operator optimization provides a way to use context-appropriate implementations of operators. It is also a means for operator implementers to provide optimizations independent of the compiler.

9.3. Schedule Optimization

Dynamic binding and guide creation (from Section 7.2 and Section 8.1, respectively) introduce significant work to the prerender stage. This section describes how Stencil decides to schedule that work, with goals of (1) avoiding computing elements that will not be used, and (2) avoiding refreshing values that have not changed. Achieving these goals relies on the *stateID* facet described in Section 3.3 and is closely related to the GST from Section 4.7.

There are four basic rendering schedules: (1) Naive Load refreshes for each new data point loaded, (2) Naive Render refreshes values before each render, (3) Lazy Load refreshes data points after loads that would yield new results, and (4) Lazy Render refreshes data points before a render, but only if that would yield new results. The two naive strategies can be converted into their respective lazy strategies through the same techniques. Lazy Load may perform unused computations that Lazy Render does not. However, Lazy Load leads to lower rendering latencies because work is done in the loading phase instead of the prerendering phase (see Section 7.2). In essence, Lazy Load and Lazy Render exchange render and loading latencies. Stencil conservatively approximates the Lazy Render strategy because Lazy Render is the most inline with the given goals. The approximation used always refreshes values when an exact implementation would, but may still perform some unnecessary calculations. An approximate implementation is used because a full implementation is not decidable [**107**, **115**].

Implementing the Lazy Render strategy adds work to the prerender phase. The fact that prerendering always occurs before rendering implies that any results calculated in prerendering will be immediately used in the rendering. Thus, goal (1) is achieved as elements are only refreshed after the last update opportunity before rendering. Achieving goal (2) requires operator metadata and the *stateID* facet.

Stencil uses the *stateID* facet to calculate an approximation of the GST (see Section 4.7). The *stateID* facets can be used at a more fine-grained level than the GST to determine if particular calculations need to be repeated to satisfy the requirements of consistency. The Stencil runtime caches the return values of stateID for all operators in a prerender process (e.g., dynamic binding operators, guide sample operators, guide transforms, etc.). The grouping of the stateIDs reflects the grouping in the analysis. With grouping preserved, individual guides and dynamic bindings are treated independently. Before scheduling a refresh, the runtime queries for new stateIDs. If no stateID in a group has changed, the refresh may be skipped.

THEOREM 9.3.1. *Omitting recalculation based on no shift in stateID preserves consistency.*

PROOF. The proof that recalculation can be skipped depends on three facts established earlier. First, elements computed during prerendering are, by definition, snapshot consistent (see Section 7.2). Second, snapshot consistency for an attribute states that all values come from a single global analysis state (as indicated by GST, see Section 4.7). Third, prerendering calculations are not permitted to modify state (see Section 7.2).

Since modifying the state in prerendering is not permitted, transformation results are determined entirely by the operator state and its arguments. Properly implemented operators correctly report state changes through the *stateID* facet.

If an operator has not changed its state, then the results of calling a non-state-modifying facet with the same arguments will be the same. Therefore, if the stateIDs of a chain of operators have not changed, the results of calling the chain will not change either. Therefore, the operator chain may be omitted because any results it would produce are already the same as those it would produce in the global state (as indicated by the GST).

Therefore, the results not recomputed are snapshot $\times$ snapshot consistent for all attribute pairs. Similar reasoning applies to snapshot $\times$ sequence consistency. $\square$

Since stateID enables skipping recalculations, stateIDs must track *state transitions* not the state itself. If a stateID simply tracked state then a sequence of transitions that took an operator out of a given state and then back into the original state could mask inconsistencies. Any values processed when out of the original state are not guaranteed to be consistent with the rest of the visualization.

Using an operator's stateID does not guarantee a true implementation of Lazy Refresh scheduling. Changes in stateID only indicate that the operator state changed, which Stencil

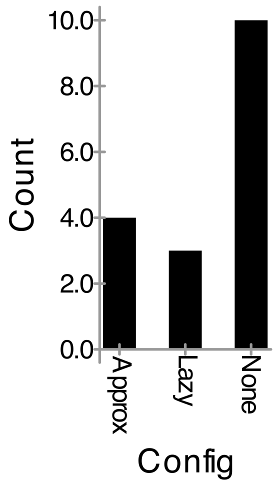


FIGURE 9.1. Impact of schedule optimization on the Becker's Barley data set. Without optimization, dynamic bindings are recalculated three times more often than with it. This impact grows as the data size increases. With the approximate scheduling, they are only recalculated once more than required by the ideal lazy schedule.

interprets as an assertion that for *some input* the operator would return a different result than it did before. The operator may not return different results for data actually used. However, in practice the approximation reduces running time and does not render more in actual runs than a lazy implementation would (see Figure 9.1) .

9.4. Mixing Data-Flow and Data-State Models

The data-state and data-flow models for data visualization are expressively equivalent [**22**]. Early visualization frameworks tended to focus on the data-flow model, VTK being a successful outgrowth of those efforts. More recent work has focused on the data-state model; Prefuse and Tableau being examples in the data-state style. Each model tends to support different types of analysis and circumstances more effectively than the other. This section characterizes the benefits of the data-state model from a computation point of view and proposes a process for automatically integrating data-state execution in a data-flow framework.

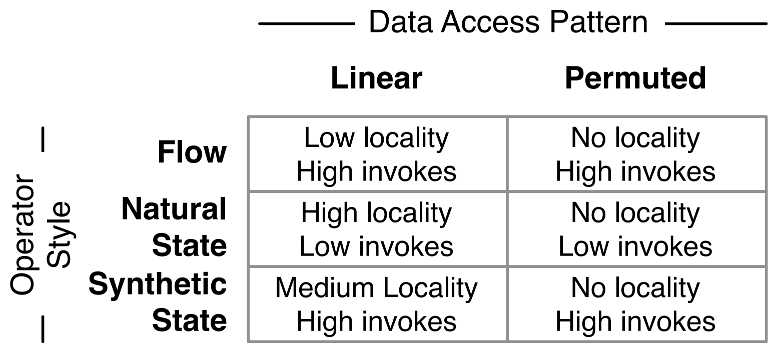
VTK has been the gold standard for scientific visualization frameworks. It excels when working with small updates or data with an *a-priori* organization that can be exposed in data structures. The data-state model has more recently been receiving attention. It is generally most efficient when calculating global attributes (e.g., current value versus max value) and when working with bulk updates that include multiple changes. Each is complementary to the other, but frameworks tend to require explicit transitions between them, when mixed-model support is even offered.

Two practical concerns likely underlie the advantages of the data-state model when working with bulk updates. First, this model incurs fewer method invocations than the data-flow model. Instead of calling analysis operators once for every element in a collection, it calls the operator once per collection. Second, it is more likely to be able to take advantage of data locality as it explicitly works on collections rather than individual elements.

Taking advantage of the strengths of each style requires a principled hybrid approach. Knowing when to apply which style depends partially on the nature of the benefit of using one over the other. This section examines the relative contribution of locality and method call reduction for the data-state model. After establishing the major benefits of the data-state model, a discussion of how and when to translate programs from the data-flow to the data-state model in the Stencil visualization system is pursued.

9.4.1. Comparison. To examine the relative importance of locality and method calls in the data-flow model, we constructed the 3×2 test matrix shown in Figure 9.2. The matrix rows target the impact of method calls through operator styles while the columns target locality. Each matrix cell was evaluated with a sequence of data of increasing size.

The flow and native state operators are straightforward implementations of their respective styles. A *synthetic* state operator is a flow operator wrapped with buffers. The buffers allow the flow operation to be invoked on all data points before proceeding to subsequent operators. The flow operation used in the synthetic data-state style is related to the original flow operation, but is not always identical. A process for constructing synthetic state operations is described in Section 9.4.3. The synthetic and flow operators have



Operator Style	Data Access Pattern: Linear	Data Access Pattern: Permuted
Flow	Low locality High invokes	No locality High invokes
Natural State	High locality Low invokes	No locality Low invokes
Synthetic State	Medium Locality High invokes	No locality High invokes

FIGURE 9.2. Data state vs. data flow treatment matrix. Relative locality and invocation count information is in the cells.

method invocation counts based on the number of data points processed, while the native state operator method invocation count is based on the number of operators.

The columns of the matrix indicate iteration orders. Modifying the iteration order between sequential access and random access modifies the locality characteristic of data accesses. To accommodate differing access patterns, the data-flow driver was modified to take a permutation parameter. The permutation pattern is a list where each element is an index into the data array. In the "linear" case, the index of the permutation was identical to the index of the data array. In the "random" case, the values of the linear case were randomly shuffled. Each data array index was still present, but all correspondence between index and value was eliminated. Data were accessed in the order indicated by the permutation values. The data-state operators were similarly modified, though in these cases each operator received the permutation instead of just the driver. The permutations were computed once per trial.

Each operator style was tested with a sequence of data sets of increasing size. Each trial was repeated 10 times; the garbage collector was executed after each trial. Conclusions are based on the average running time of each configuration.

All tests were conducted on a 2 GHz Intel Core Duo MacBook with 1Gb of 667 MHz DDR2 SDRAM with no other applications running. Tests were executed in a Java 1.5 runtime with a maximum and initial heap size of 768MB. The test network is based on a dynamically bound rule that sizes words in the center of the TextArc visualization (see Appendix A (Section A.9)). The occurrence count of words is taken, then the maximum of that occurrence count is measured. The relative word count is then used to produce an variably sized font object. In Stencil notation, the test conducted was

```
Count(word) -> Max[range: ALL](count) -> Divide(count, max)
        -> Scale(quotient) -> @Font{{scaled}}.
```

To isolate the impact of the flow and state styles from parsing and rendering and the impact of the particular Stencil operator call protocol, tests were not conducted in the Stencil runtime. However, operator invocation was still performed reflectively to preserve the impact of runtime-specified operator call chains.

Results are summarized in Figure 9.3. A general separation between (faster) native state operators and the (slower) data-flow and synthetic state operators is present. Additionally, the eventual impact of access locality can be seen in the inflection point near 2^{15}.

The primary motivation for this study was to identify what practical advantages the data-state model enjoys. Figure 9.4 illustrates a substantial advantage for a native data-state operator unless only one element is being processed. Figure 9.5 illustrates the advantage of removing method invocations in the data-state operator. The substantial difference between the two figures indicates that the majority of the advantage for the data-state operator comes from lower method invocation overhead. Even with the loss of the method-call advantage, Figure 9.5 shows that the synthetic operators retain a consistent advantage over the flow operators once a relatively small number of elements have been processed. This difference is attributed to the data locality advantages of the synthetic state operators.

The importance of method invocation is likely, in part, due to the heavy use of reflection in the test framework. This can prevent some compiler/JIT optimizations that would otherwise be available. However great the impact of method calls, locality still plays a

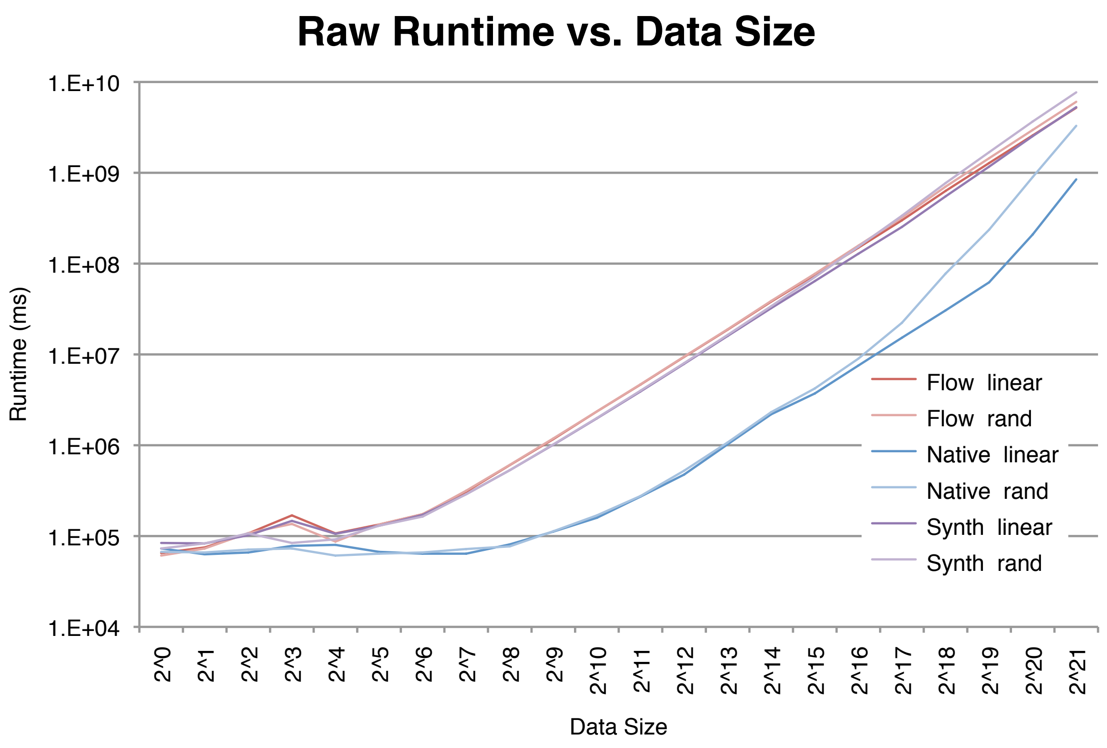


FIGURE 9.3. Direct runtime comparison of all operator styles and access patterns. The flow and synthetic styles overlap for large portions of this chart.

significant role. Figure 9.7 shows the significant impact that data-access locality has, especially on larger data sets, after accounting for method call reduction. For any data set greater than 2^{15} elements, the impact of locality plays a more prominent role. The impact of data locality on the data-flow operators exhibits similar inflection points to that of the data-state operators, as illustrated in Figure 9.6. However, the magnitude of the effects is less. The synthetic data-state operators were comparable to the data-flow operators, but with slightly greater sensitivity to locality. This fact, combined with the slight advantage of the synthetic operators over the data-state operators at larger data sizes (see Figure 9.5), indicates that synthetic operators are a viable option when native operators are not available.

The foregoing analysis focused on circumstances that favor the data-state model. The profiling results indicate that the data-flow model excels at working with small, focused

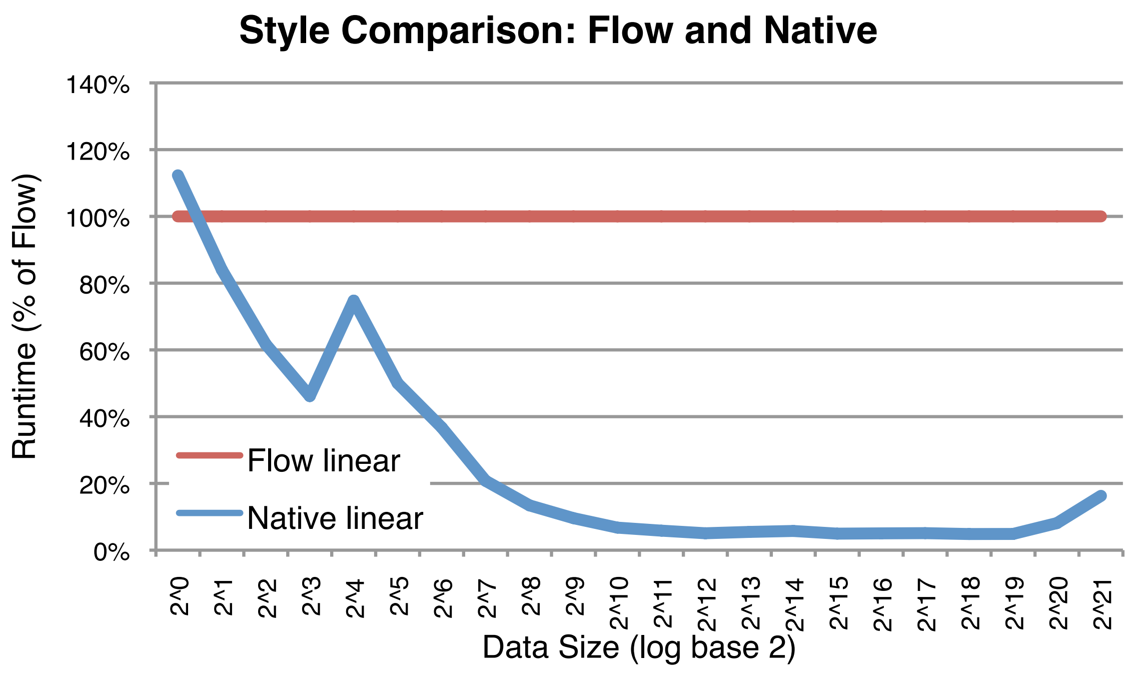


FIGURE 9.4. The data-state operator suffers a disadvantage to the flow operators for doing single updates but is faster for larger updates. Natural and synthetic state operators compare similarly, except synthetic operators have no advantage at small data sizes.

updates (this is especially apparent in Figures 9.4 and 9.5). For single element insertions or modifications that do not involve recomputing global properties, the data-flow model typically behaves as if the data set were of size one. By comparison, data-state networks treat a change of any number of elements as if all elements were updated.

9.4.2. Theory. The analysis in Section 9.4.1 indicates that the data-state model gains significant running time advantages from reduced method calls and data access locality. However, it also shows that this advantage requires that large blocks be treated uniformly.

This section establishes a practical theory for (1) translating between data-flow and data-state operations, and (2) identifying when such a translation should occur. This directly extends the work done by Chi [23], but focuses on automatic translation rather than existential proofs. It includes a complete description of synthetic operators employed

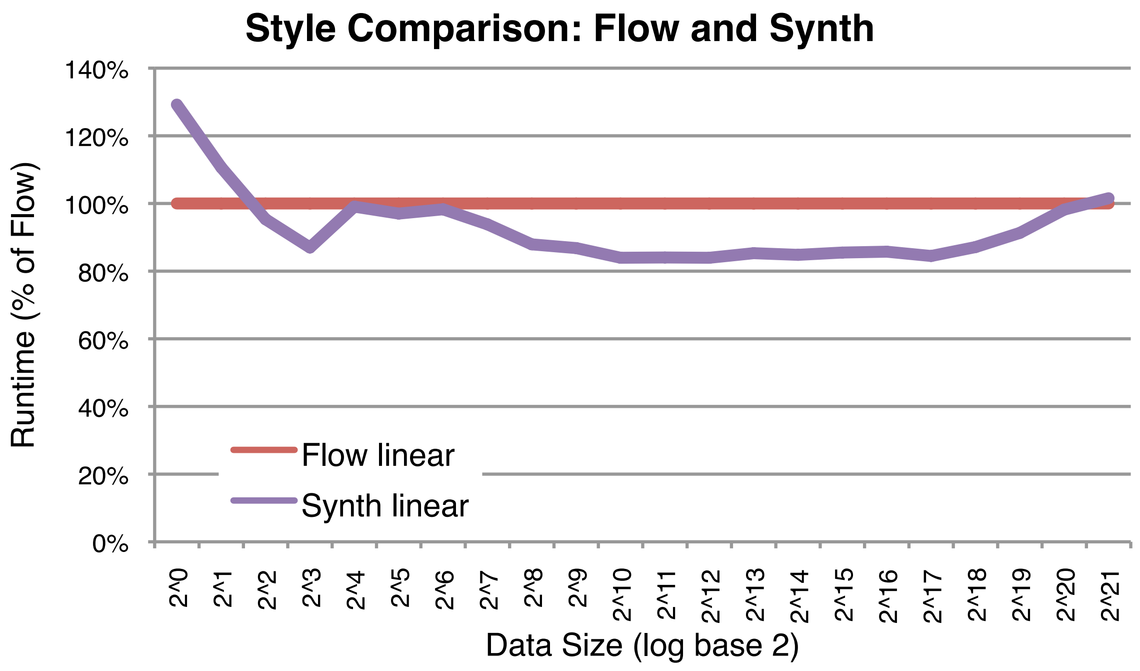


FIGURE 9.5. Comparison of flow and synthetic state operators. The synthetic state operator is at an initial disadvantage, but data locality makes a difference after 2^8 elements.

in Section 9.4.1. The discussion in this section is framework independent; Section 9.4.3 explores a partial implementation in the Stencil visualization system.

9.4.2.1. *Requirements.* For the purposes of this discussion, data-flow operators are represented with F; each flow operator takes a single input and produces a single output. Data-state operators are represented as S; each takes a list of values and produces a new list of values.

To correctly translate from the data-flow to the data-state style, some metadata must be provided to the analysis framework. To access this metadata, we define a relation from flow operators to state operators $FS : F \mapsto S$. To be useful, the operators F and S need to compute corresponding results. The definition of correspondence can be found in Equation 6.

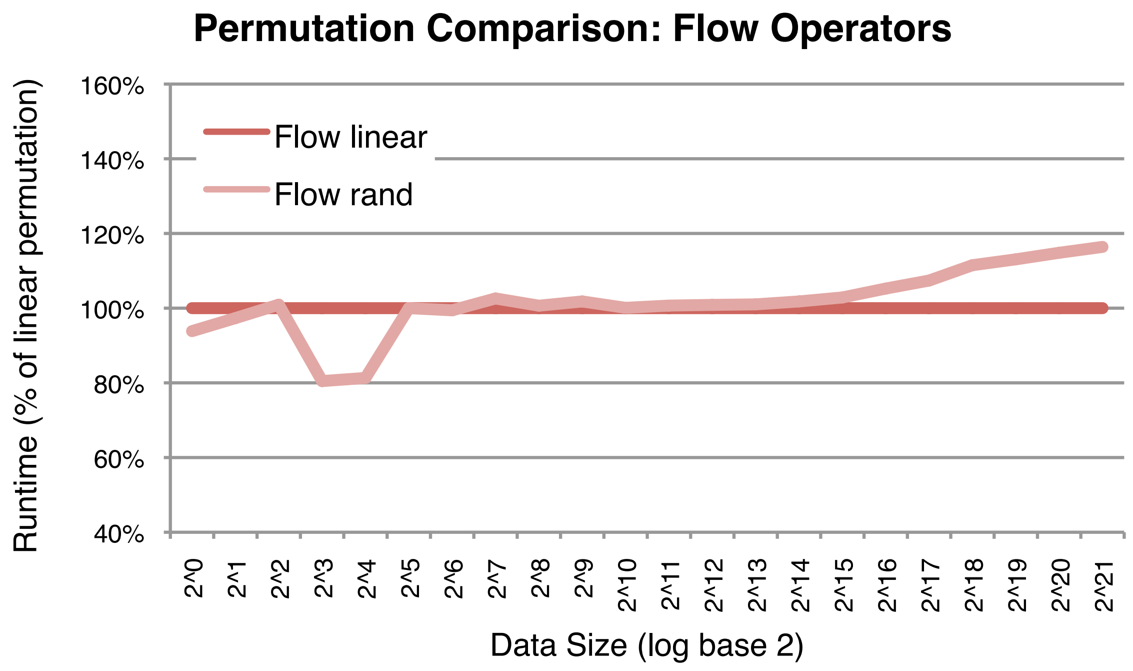


FIGURE 9.6. Impact of locality on flow operators. Data locality provides no consistent advantage for data-flow operators until around 2^{15} data elements. The change near 2^5 elements is unexplained.

$$F \hat{=} S \iff \begin{array}{l} \forall \mathbf{M}[i] \text{ the memory states of } F \text{ after the } i^{\text{th}} \text{ input} \\ \text{and } \mathbf{I} \text{ a list of the } i \text{ inputs} \\ S(\mathbf{I})[i] = F(\mathbf{I}[i], \mathbf{M}[i-1]) \end{array} \tag{6}$$

The cases indicate that operator F corresponds to S when they compute the same values in their respective ordinal spaces. For S, the ordinal space is the stream offset. For F, it is the result offset. In effect, if F is a stateful operator, then S must be able to derive that state entirely from the sequence of values found in $\mathbf{I}$ and any parameterization it received when it was constructed.

Using the *FS* relation and Equation 6, data-flow processes can be converted into a data-state process. The first issue is to identify when the conversion should be performed. This

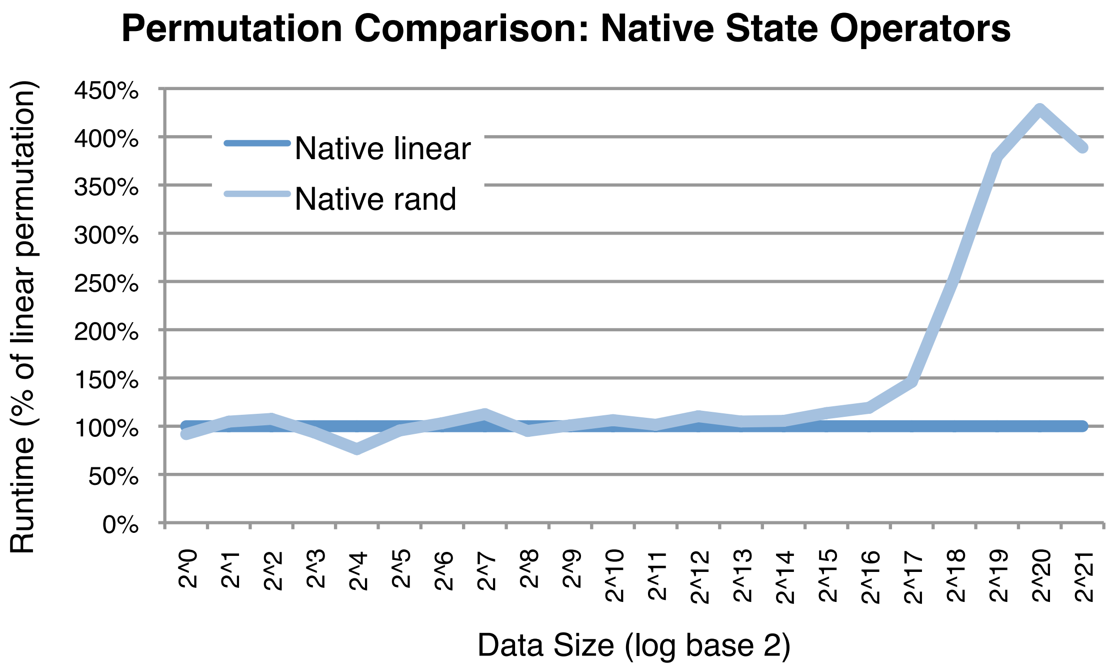


FIGURE 9.7. Impact of locality on native data state operators. Native state operators do not benefit from locality until around 2^{15} elements; thereafter, the lack of locality quickly eliminates the call-count advantage.

identification is framework specific, so we represent it abstractly as a relation between visualization program fragments. The identification process *TryConvert* takes a data-flow program fragment and either returns it unchanged or creates a data-state equivalent. The results of *TryConvert* can always be used to calculate a visualization equivalent to the one specified by the original schema, but it may or may not be in the data-state style. To be effective, *TryConvert* must identify locations where updates will either effect multiple existing values or occur in batches.

If *TryConvert* creates a data-state process, it does so by a two-step process. First, for each operation in a source program fragment, it retrieves the equivalent data-state operator using the *FS* relation. Second, the new data state operators are relinked according to Chi's process [**23**].

9.4.3. Stencil implementation. This section takes the extended theory of Section 9.4.2 and describes an implementation in the Stencil visualization system. The ability to mix natural and synthetic data-state operators allows the Stencil system to take advantage of the data-state efficiencies for dynamic binding calculations.

The *TryConvert* relation is implemented as a two-stage process. The first stage relies on the programmer to indicate where data-state operations may be applicable. This is done by employing the dynamic binding operator, `:*`. Dynamic binding indicates that an update to the given attribute may affect an arbitrary number of existing data points (see Section 7.2 for more details on dynamic binding). These broad updates are an ideal situation for applying data-state operators. The first stage of *TryConvert* identifies the dynamic bindings.

The second stage of Stencil's *TryConvert* establishes that flow operators involved in a dynamic binding have data-state counterparts; in other words, *FS* will succeed for all operators in the dynamically bound chain. This is achieved using Stencil's operator metadata facilities. In the simple case, the Stencil metadata for each operator includes a *state* facet. The *state* facet is guaranteed by the provider to satisfy the requirements of Equation 6.

Most operators do not provide a *state* facet, as it is not required by Stencil. When it is omitted, the *FS* relation does need to fail if a suitable operator can be synthesized. To synthesize the state operation, the counterpart relation is used instead (see Section 3.3 for the definition of counterpart). Recall that the *query* facet is the counter part to *map* by convention. The *query*, as a counterpart, has read-only access to the internal state that is set by the *map* facet. Recall that, when possible, the original and counterpart operators produce the same result for the same initial memory condition. Though counterparts are allowed to deviate to signal errors, this is the exception and not the rule. If the results cannot generally be reproduced, then the operator is labeled *opaque* in the metadata system (see Section 3.3.2). Opaque operators cannot be used for synthetic data-state operators. For operators with other memory behaviors, but no *state* facet, the *query* facet is used for computation and Stencil manages data buffers to simulate a data-state operation. This

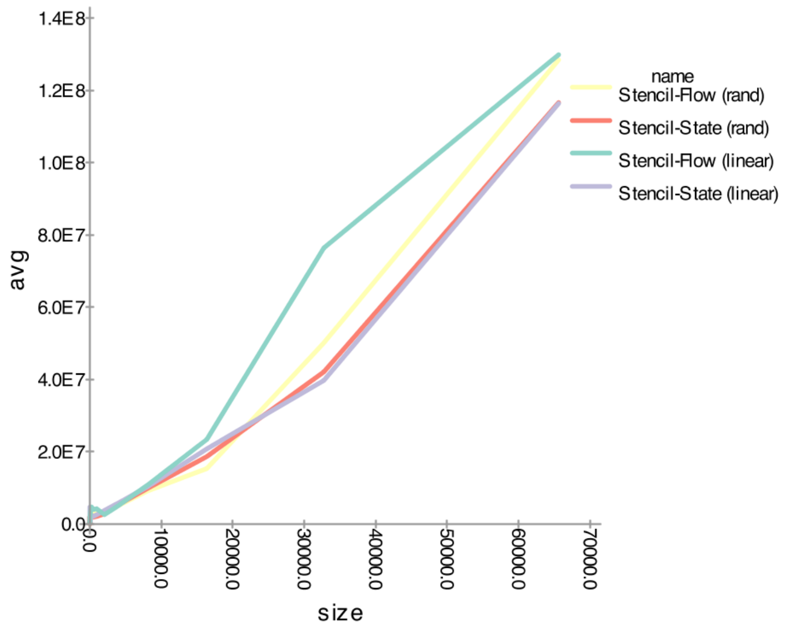


FIGURE 9.8. Running time of the Stencil data-state implementation versus the Stencil data-flow implementation. The Stencil data-state implementation does not the early advantage that the generic data-state implementation showed in Figure 9.4. However, it does eventually enjoy an advantage (around the time 2000 tuples are processed per batch).

process retains the locality benefits of data-state operations and expands the applicability of *FS*.

In summary, the *TryConvert* function in Stencil looks for the dynamic binding operator, `:*`, and verifies that all operators in a dynamically bound process either (1) have a *state* facet, or (2) have a counterpart operator. If these conditions are met, a new analysis process is produced in the data-state style. The new process is used for the dynamic calculations only; the original flow operations are still used for initial processing.

9.4.4. Evaluation. The earlier analysis presents results for best-case scenario for data-state and data-flow operations in an idealized environment. It was deliberately kept separate from the Stencil implementation to investigate the potential impacts without the other overheads the Stencil framework introduces. The analysis was repeated using the design presented in Section 9.4.3 in the Stencil runtime, targeted at dynamic bindings. The same test rule was used as before. Results can be seen in Figure 9.8. Even with the additional overheads of the Stencil runtime, the data-state model retains an advantage for dynamic binding calculation. However, this advantage is not as large as the idealized results and is not present for fewer than 100 data points. This mixed result leads to questions about the overall impact of the data-state model on analysis running time.

To investigate the overall impact, a visualization of a quadrant-filling curve was produced in both Stencil and Prefuse. Prefuse was selected because it is a mature framework with the option to use operators in a data-state style. Furthermore, Prefuse and the Stencil implementation are both Java based and are customizable visualization frameworks. The overall running time of each implementation was tested on up to 524,288 data points, well over the threshold at which the data-state model should exhibit an advantage. Data were loaded into the frameworks from memory to avoid I/O effects. Rendering was deferred until all analysis was completed to eliminate interleaving effects. The timings are from the first tuple loading until the first element is drawn to the canvas. This arrangement captures all loading and analysis activities of the respective frameworks without considering differences in the rendering engines. The test was repeated five times and times were averaged. Timing results are summarized in Figure 9.9. Prefuse was generally faster than Stencil for more than 32 data points. Both implementations scaled linearly, but Prefuse had a lower linear coefficient (presented as a more gentle slope on the plot) and eventually becoming twice as fast as Stencil. This result demonstrates an advantage to using a data-state system when working with large blocks of known data. Therefore, applying the transformation to data-state based computations when performing dynamic bindings is warranted. However, some of Prefuse's advantage is likely due to being compiled to Java Virtual Machine (JVM) byte code and strongly typed (while Stencil is untyped and executes in a runtimes above the JVM). Furthermore, the effects are not as great as those predicted in Section 9.4.1.

9.4.5. Conclusions. Employing the theoretical equivalence between the data-state and data-flow models of visualization requires substantial support in a visualization framework. We have demonstrated (1) that such support is possible with a small amount of

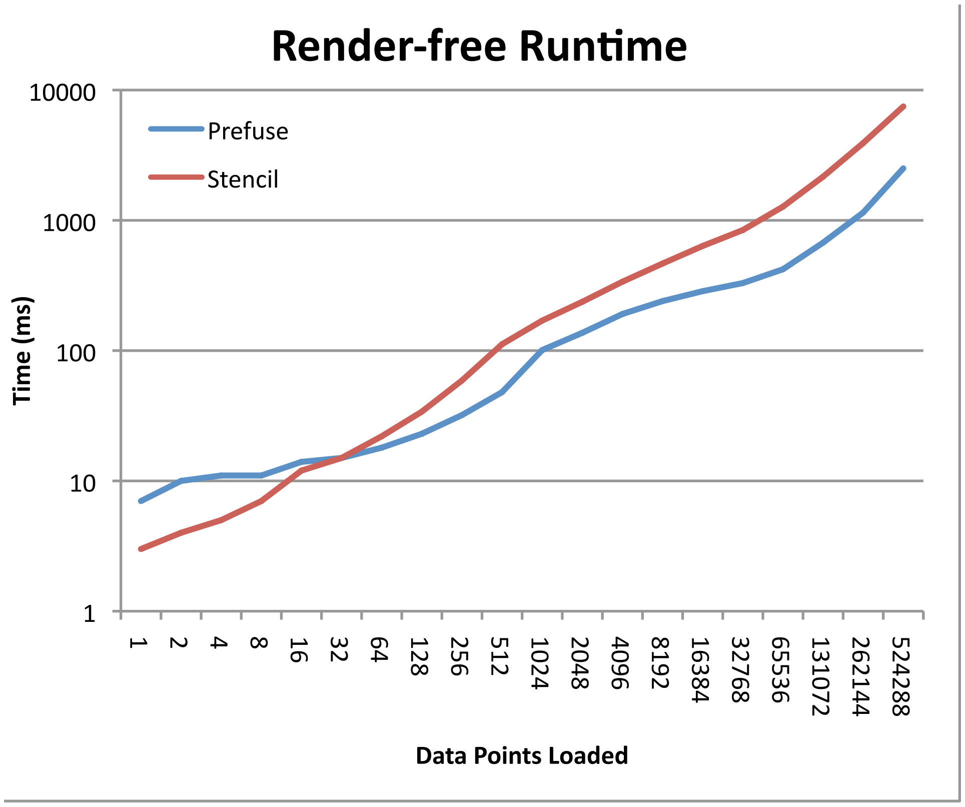


FIGURE 9.9. Prefuse versus Stencil running time to compute a quadrant-filling curve. Rendering was deferred to eliminate scheduling effects, so dynamic bindings are calculated exactly once.

metadata, and (2) that supporting the conversion can yield significant benefits, especially when true data-state operators are provided.

Deriving the maximum benefit from these transformations requires proper scheduling of the computation. This is pursued Section 9.3.

9.5. Task-based Parallelism

Analysis and rendering are the two basic phases of visualization software. Section 7.2 introduced a prerender phase to work with dynamic bindings; guide creation was added to the prerender phase in Section 8.1. Abstractly, analysis provides an initial pass over the data, prerendering performs any recurring operations and rendering presents results at

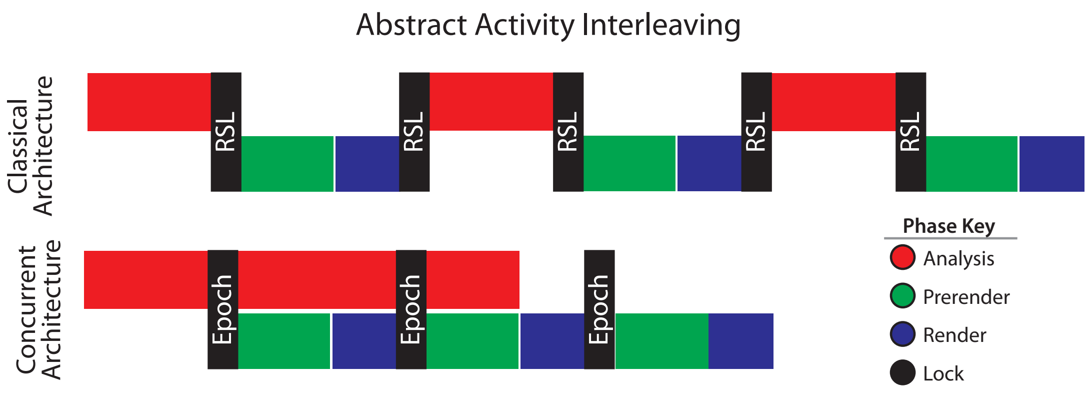


FIGURE 9.10. Abstract concurrency available in visualization programs. When a global lock and ephemeral structures are employed, concurrency is restricted to just that within a single phase. However, if persistent structures are used with an epoch lock instead, concurrency between analysis and the other phases is achievable.

any given time. These phases correspond to the data transforms and visual mappings of the InfoVis reference model [**15**].

To produce a consistent (per Section 4.7) and thus interpretable visualization, all three phases must respect each other when operating. Traditionally, "respecting" other phases is formed as mutual exclusion: only one phase operates at a time. Mutual exclusion manifests itself through the Executor service in VTK [**74**] and synchronizing on the "Visualization" object in Prefuse [**69**]. We refer to such course-grained locking as a render-state lock (RSL). Using a RSL is costly in terms of the responsiveness of the visualization to new data and limits the amount of concurrency that can be introduced into a visualization framework. Persistent data structures, a standard technique in functional programming, enable the removal of the RSL, and thus increase the available concurrency.

In order to preserve the consistency of the visualization (as defined in Section 4.7), the processes involved must not interfere [**3**] with each other. Persistent data structures allow updates to be performed on a data structure, but old references to the data structure still see their original copy (i.e., the old versions *persist* through updates). Strategic use of persistent data structures can permit the three phases of visualization production (i.e., analysis, prerender and render) to proceed concurrently without interference and work

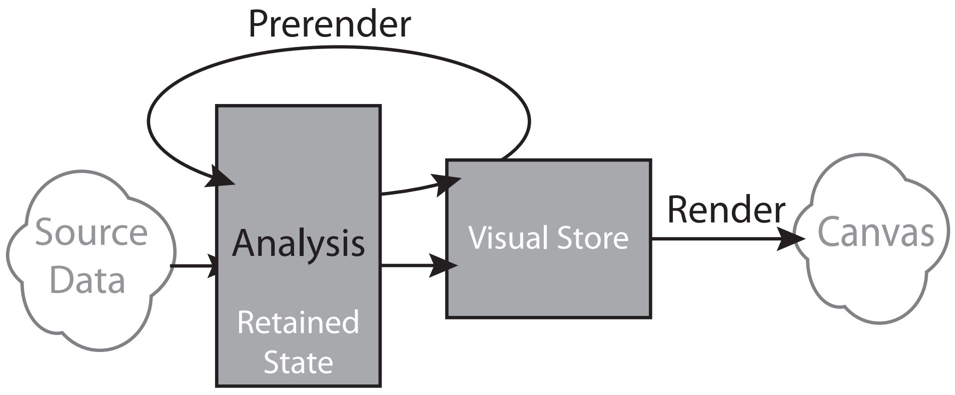


FIGURE 9.11. Phases of a visualization execution and their interactions. The analysis arrow is the traditional analysis of the InfoVis reference model [**15**]. The right-hand arrow is the rendering phase. Prerendering is often treated as indistinguishable from either rendering or analysis (depending on the framework). This conflation is a by-product of the bias towards data-state programming in the reference model. Treating it separately exposes additional opportunities for concurrency and enables optimizations that reduce the number of data points "touched" during recomputation based on global data.

without unnecessary work duplication. The general architecture of a visualization library using persistent data structures remains similar to the traditional structure, except the RSL is replaced with an much less expansive epoch transition lock. A comparison of the abstract concurrency available under these two regimes is shown in Figure 9.10. When using an RSL, phases are mutually exclusive. When using an epoch transition strategy with persistent data structures, phases can be overlapped. This change in architecture yields a substantial increase in visualization responsiveness to incoming data as well as an increased ability to efficiently employ system resources.

9.5.1. Architecture. Data are shared between the three visualization phases in two places. First, analysis and prerendering are often interrelated through shared analysis operator state. This is especially true when efficiency is pursued (e.g., data summaries calculated in the analysis phase can be used to remove iterations over the data in the prerender phase). Second, all three phases interact with the visual store. Properly arranging the visibility of memory changes made in each phase is the key to exposing concurrency in the overall system. Managing the visibility of memory changes is exactly what persistent data structures do.

9.5.1.1. *Principles.* The overarching goal is to provide concurrent rendering analysis without violating the consistency of the visualization. Stencil's approach relies on noninterference [**3**,**89**] as the principal means of preserving consistency. Noninterference must be established between the analysis facilities and each of the prerender and render facilities. If noninterference is established, then the prerender phase can prepare a consistent collection of visual elements for the render phase to present. This leads to a two-part strategy: (1) persistent view support and (2) view collection during epoch switches.

Persistent views are used to reason with both analysis operators and the visual store. Analysis can be viewed as a network of potentially stateful operators. Since operator state may be shared between analysis and prerendering phases, a guarantee of noninterference must be made between the two phases at the level of the individual operators. Treating each operator as a wrapper for a persistent data structure gives the necessary abstractions. Each operator works on its internal state as normal, but upon request, it must provide a persistent view of that internal state. In this abstract system, the visual store is an operator that stores the intermediate and final visual representations. It does not need to differ in concept from other operators. The Stencil operator system includes a 'viewpoint' call that, when properly implemented, returns an operator with a memory state that will not be updated by other operators.

With operators and the visual store able to yield persistent views upon request, the next requirement is that a collection of consistent analysis views be collected to create a consistent visualization. This is the purpose of the collection phase. During the collection phase, no analysis may occur, as interleaving analysis and persistent view collection may yield views that come from different global states. The collection phase, therefore, must be protected by a global lock.

Persistent views for all operators and a protected collection phase are sufficient to provide the required noninterference/consistency guarantee. This system also leads to three observations that influence the implementation. First, since operators only need to supply

```
from #RENDER
   (): #Update() -> #RenderBuffer()
```

(a) Ephemeral

```
from #RENDER
   (): Snapshot() -> SignalRender()
```

(b) Persistent

FIGURE 9.12. Render event response rule given ephemeral and persistent data structures.

the persistent views on demand, they do not *necessarily* need to be implemented to always have a persistent state. However, operators need to be able to supply a persistent state without receiving any additional inputs. Second, in order for a concurrent system based on these principles to be more efficient than serial systems, the collection of global states must be inexpensive because it represents work not done in the sequential system. Third, once a persistent view has been made, any change made to the persistent copy will not be seen in the original operator. This means that any operations performed during prerendering or rendering cannot be seen by analysis. In effect, prerendering is read-only as far as analysis is concerned for operators. There is more nuance surrounding the visual store which is discussed in more detail in Section 9.5.1.3.

9.5.1.2. *Semantics.* From the standpoint of the semantics presented in Chapter 5, using persistent data structures reforms how the implicit render event-response sequence behaves. In the ephemeral system, a render event triggers a recalculation of guides and dynamic bindings, then an actual rendering to a canvas. All of this work was performed as a blocking operation. In an implementation using persistent structures, a render event instead triggers snapshot creation in a blocking manner, but then *signals* that the recalculation and subsequent rendering should occur. The relevant response rules are given in Figure 9.12.

9.5.1.3. *Implementation.* The Stencil visualization system is implemented with support for concurrent analysis, prerendering and rendering following the principles described above. This subsection describes how the Stencil runtime is constructed. There are three main parts to the persistent implementation of the Stencil runtime: (1) epoch switches, (2) operator implementation, and (3) visual-store implementation.

9.5.1.4. *Epoch Switching.* Based on the observation that persistence is not required at arbitrary points in time (see Section 9.5.1.1), the persistence is only guaranteed with respect to finite points in time and for limited periods. The duration over which a persistent view needs to be active is called an epoch. Such limited-time persistent views are referred to as viewpoints and are based on the "stateful view" as presented by Zhu [**129**]. The collection of all analysis operator viewpoints plus a viewpoint on the visual store constitute a viewpoint of the entire analysis state.

To safely switch epochs, a global lock is employed. Acquisition of this lock occurs at similar places to the RSL acquisition (though epoch lock release occurs substantially sooner than RSL release). However, the lock only protects the creation of viewpoints, not entire render and prerender phases. In particular, any analysis will hold the lock for its duration. However, prerendering only holds the lock long enough to create viewpoints of stateful operators related to dynamic calculations and the visual store. This permits analysis to proceed as soon as the viewpoint is created. Epoch creation as part of prerendering also ensures that only one viewpoint exists at a time.

Having explicit epochs creates flexibility in the implementation of operators. A general persistence guarantee does not need to hold at all times. Instead, only the more limited epoch viewpoint behavior is required.

9.5.1.5. *Operators.* Recall from the observations at the end of Section 9.5.1.1, operator viewpoint creation must be efficient to net a consistent benefit from concurrency. With this restriction in mind, there are three types of viewpoint implementations. Which is "best" is determined by the amount of state the operator needs to retain. First are the stateless operators (e.g., arithmetic operators). These require no change for persistence. Second are operators with analysis state comprised of only atomic values. Operators in this category include sum (which keeps a running total) or full-range average (which keeps a running total and a count). Persistence can be achieved efficiently through the a shallow copy (such as Java's clone functionality). The default clone is shallow, so it copies references to any Object type. Therefore, the default clone operation can be used if all fields are (1) primitive types, (2) immutable object instances, or (3) mutable instances where copy on write discipline is observed.

The final transformation operator type occurs when the retained state is a composite of multiple values. Dictionaries, tree layout operators and sliding-range operators fall into this category. Persistent views of such analysis state can be retained by performing a deep clone or observing copy-on-write semantics for all containers. However, this is often inefficient for any nontrivial data set. Persistent implementations of common data structures have been studied and implemented for functional and imperative environments. Stencil uses read-only facets observing the counterpart relation prerender context (see Section 3.3.1). As such, only partial persistence [**39**] is required. The exact means of providing persistence is left up to the operator implementer.

Assuming that only one analysis thread can modify any given operator (a restriction enforced in Stencil), the implementations described above are sufficient to permit prerendering to occur concurrent with analysis (recall from subsection 9.5.1.1 that prerendering is inherently read-only for all analysis state). Each implementation option provides a persistent view of the mutable memory state of the operator.

9.5.1.6. *Visual Store.* Naively implemented, persistent data structures are less efficient at basic operations than their ephemeral counterparts. Adding and removing elements, in the worst case, can incur costs proportional to the size of the data set. Reasonable general-purpose implementations exist for persistent lists, trees and maps. However, using task-tailored data structures provides significant benefits. Observing both the semantic requirements and efficiency desires, a buffered visual store is used in Stencil. The buffering is similar to that used in double-buffered displays, where front and back buffers allow rendering and display to be divided. The architecture of this buffered store is given in Figure 9.13. The basis of the store are *tenured*, *fresh* and *transitional* collections (implemented using standard java collections classes). The tenured collection represents values that are in a consistent state. This collection can be rendered at any time without further processing. The fresh collection represents values that have gone through analysis since the last

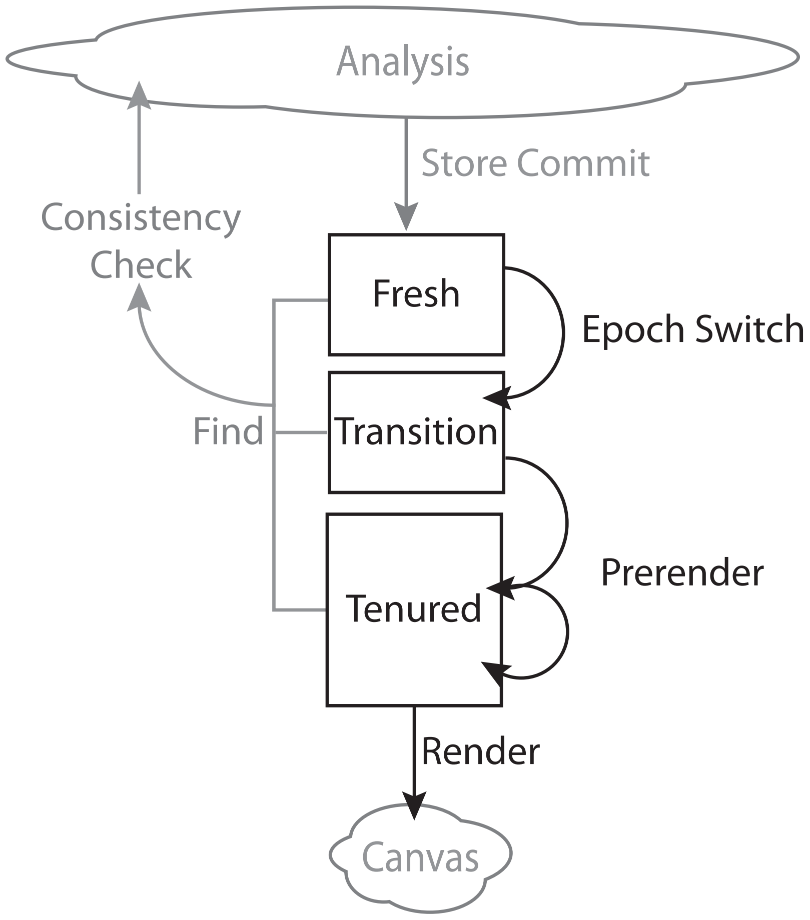


FIGURE 9.13. General architecture of the persistent layer. The fresh collection is moved to the transitional collection in the epoch switch. Prerendering moves values from the transitional to the tenured collection and updates values in the tenured collection Rendering only works with the tenured collection. The Find operation works with all of the collections but does not depend on exact layer state because of a separate consistency check.

prerendering phase. The transitional collection is used to shift values into tenured while still allowing values in fresh to be updated. Prerendering is responsible for merging the fresh and tenured collections, resulting in a new tenured collection. The details of this arrangement, including data flows, are described below.

The data store is the heart of a visualization framework. It retains the required source data and the description of the elements that make up a scene to be rendered. Having such a broad role, the data store is a point of contention for many tasks. Rendering (drawing the elements of the visual store) and analysis (determining elements for the data and visual store) are the two most significant. Fortunately, these two operations have distinct requirements that enable them to be split apart. Rendering requires a stable view of the data set so the rendered results represent the data (and not artifacts of the framework's internal structure). Analysis requires the ability to add and update individual elements so a new state can be presented. Persistent data structures allow both of these requirements to be met concurrently without requiring mutual exclusion. Rendering can work with an 'old' version while analysis produces a new one.

The data store is not a central repository for *all* memory in the visualization, as transformation operators may maintain their own state. Rather, the data store contains a description of visual elements to render (positions, colors, etc.) and selected input data (for labeling, producing tabular reports and updating visual elements when using dynamic binding). To fulfill its roles in a visualization framework, the data store is generally conceptualized as a set of 2D matrices (called tables) and with the ability to (1) add a table schema (column names and optionally types), (2) insert a set of values that conform to the schema, (3) modify values stored in the schema, (4) perform key-based addressing for retrieving items based on a property value, and (5) perform index-based lookup for sequentially reviewing all or a subset of items.

The usage pattern of the data store enables additional flexibility in the implementation details. The principal goal is to support concurrent analysis and render-time activities without sacrificing existing abilities. Analysis produces individual updates, potentially thousands per second. Rendering needs to be based on all updates, but faster than 40 fps is generally pointless as it exceeds perceptual limits. Therefore, merging of old and new information only needs to be finalized at the (relatively) infrequent render events. This infrequency is reminiscent of a transaction, and leads to a solution related to software transactional memory's change-list tracking [**105**]. The Stencil data store is conceptualized as a two-part data structure: one for recent/transient updates and one for long-term committed updates. The transient updates essentially form a change list, but their goal is efficiency instead of rollback. The long-term commit collection is a consistent, persistent reflection of the data at a some point in the past. Combined, these parts describe the store's state at the current time.

Rendering and static-binding updates are simple to support. Dynamic binding, also a render-time activity, is more complex to support. To not repeat dynamic binding work, the data structure must support efficient merging between versions, so called *confluent-persistence* [**14**, **40**]. The general workflow is to (1) capture a persistent view, (2) merge in the static updates, (3) perform dynamic binding, (4) merge updates back, and (5) render. (Strictly speaking 4 and 5 can be done in either order or concurrently; this order was selected for simplicity in implementation but is discussed further in Section 9.5.4.) If dynamic updates are not merged back, then the dynamic binding work must be repeated anytime rendering is performed.

Conceptually, the data store may be thought of as having three parts: (1) a list of updates for the next commit (the *Young* list), (2) a list of updates being committed (the *Transfer* list), and (3) list of committed updates (the *Tenured* list). Data from static bindings are added directly to the Young list. If any lookup is done in analysis to support static binding, the Young values take precedence over all other parts of the data store. Values may be directly overwritten if an update with the same key is added before the values are committed to the column store. The Young list is frequently updated, but rarely read. The Transfer list is a buffer for values currently moving into the committed columns, but not yet there. Values in Transfer take precedence over values in Tenured. Transfer values are not modified by updates (those go the Young list instead) and are not modified by the render tasks either, except to clear them after merging is complete. The Tenured list is read frequently and updated in bulk operations. Our implementation uses a dictionary for the Young and Transfer list to support fine-grained read and updates. The Tenured list is a column store [**67**], supporting bulk updates for both merge and dynamic binding.

Figure 9.14 shows each of these parts, as well as the flow of the data through the system as updates are performed. Data flows in following stages: Young, Transfer, Shadow, and Tenured. Individual static bindings are accumulated in the Young list. When it is time to

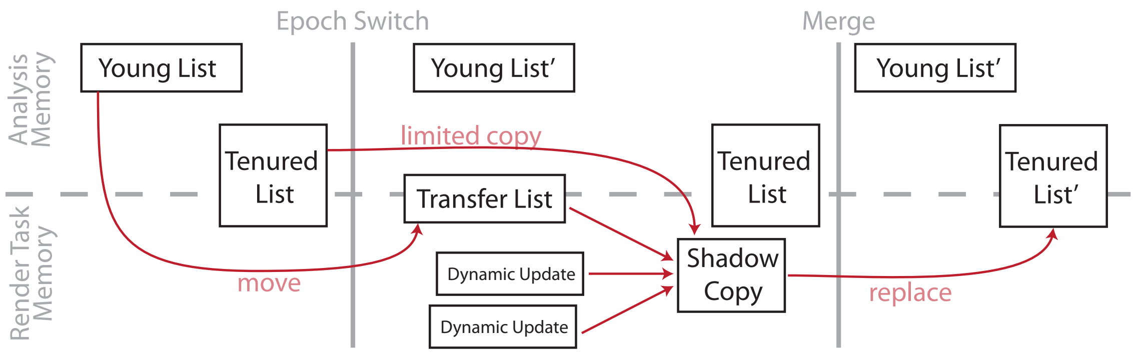


FIGURE 9.14. Activity in the persistent data store. Initially, the Tenured list is empty and the Young list is populated. After an epoch switch, the Young list is empty, but its contents can be found in the transfer list. Transferring and calculating dynamic bindings is done by render-time tasks. After a merge, the Transfer list is emptied, and the Young list may have new updates.

commit values, new updates are blocked while the Young and Transfer lists are swapped, leaving Young empty and Transfer full. A shallow copy of the Tenured list is also made (pointers for each column, not their data; this is called the 'Shadow' copy). The static binding values are sifted into columns and then moved into the Shadow copy. Moving the values into the Shadow copy may involve overwriting or deleting values in existing rows and appending new rows. In any of these operations, the target column values are copied the first time they are modified and directly mutated thereafter. In this way, the original Tenured values remain unchanged while the Shadow is updated. When all static updates have been made, dynamic updates are calculated. Each dynamic update produces a complete column for the Shadow copy. Once dynamic bindings are complete, the Shadow copy replaces the Tenured values in the original data structure and the Transfer list is cleared.

This data structure provides a persistent view by returning its Tenured values. The contents of the Tenured values are never modified, so this view persists after updates are made in Young or committed. This structure is *confluently* persistent because updates made on the Shadow copy are merged into the full data store, even though they may include values never seen in the static binding path (such updates are made through dynamic bindings). To keep the data structure thread safe, two constraints are enforced. First, starting and ending commits must appear atomic. Second, only one commit process may occur at a time (e.g., only one Shadow copy can exist at a time). If these constraints are not enforced, Transfer list values may be lost due to race conditions. In the current implementation, these operations are either constant time (merge back requires three pointer assignments), or linear in the number of columns stored (creating the shadow copy).

9.5.2. Removing the render-state lock. Recall that the RSL is used to ensure a consistent data store for rendering. Persistent data structures provide this guarantee in a different fashion. They instead guarantee that a referenced data set will not be destructively updated (e.g., mutated). Therefore, once a reference to a data store is held, new analysis will not change it. This allows the RSL to be replaced with a Render Epoch Lock (or simply an epoch lock). The epoch lock protects the capture of the globally consistent state from multiple data tables and analysis operators. Once the state is captured, analysis can resume. Each block of uninterrupted analysis is defined as an epoch that will be reflected in the next rendering. Capturing the analysis state is essential to preforming dynamic bindings, and is thus included in the region protected by the epoch lock. This leads to the abstract Concurrent Architecture model shown in Figure 9.10.

When used in the Stencil framework, analysis works with the full data store. Render-time activities always begin by initiating the transfer of Young values to Tenured values. However, the process only needs to protect the *initiation* of the transfer and not the full extent. An epoch lock protects the capture of state for all transformation operators and the transfer initiation. If there are multiple tables in the data store, all of them perform transfer initiation at the same time. Static merge and dynamic binding are calculated while further analysis proceeds. A data-store level lock protects replacing the Tenured list with the newly produced Shadow copy and clearing the Transfer list. Rendering then proceeds on the newly placed Tenured list. The epoch lock ensures that the transfer initiation appears atomic and that the data store captured is consistent with the analysis state captured (this is important to keep dynamic bindings consistent with each other). The data-store lock makes the completion of transfer appear atomic; together these locks satisfy constraint one. If a second render request is made while an transfer is in progress, it is blocked until the current one completes (per constraint two above). To provide more timely responses, a render request delayed in this way proceeds with the newly produced Tenured set instead of initiating a transfer. By structuring the operations in this way, analysis can proceed in one thread while dynamic binding and renderings proceed on others.

From the standpoint of rendering, the strict sequencing of prerender and rendering while analysis updates only modify the fresh collection, is sufficient to ensure a consistent visualization. However, from the standpoint of analysis, any reference to the current state of an entity in the visual store is ambiguous. The "current state" could refer to: (1) a component of the fresh collection, (2) a value in the tenured collection after the last generation change was initiated, or (3) the value that will be in the tenured collection when prerendering is completed. Consistency indicates that it be the value that will be rendered to the screen, option 3. For this reason, any analysis operator that returns the state of an element in the visual store must ensure that an element's values are passed through processing analogous to that done in prerendering. If the requested values are not involved in any prerendering operations, then the operator can return the values in the visual store directly. Similarly, if the tenured collection is in a state consistent with the analysis operators, no additional processing needs to be performed. However, if the requested values are in the fresh collection, or if the requested values are in the tenured collection but prerendering is not yet complete, the referencing operator must do additional processing. If implemented naively, this requirement can lead to circular dependencies. These are broken in our implementation by requiring that all returned values come from the same logical (if not actual) viewpoint of the state. This is consistent with how the prerendering treats such inter-store circularities.

THEOREM 9.5.1. *Using epochs and persistent data structures preserves the noninterference between phases provided by the RSL.*

PROOF. Per Andrews [**3**], The process A *interferes* with process B if A executes an assignment that violates the precondition of the currently executing portion of B.

The three processes are analysis, prerendering and rendering. Since all prerendering must complete before rendering is started, they trivially do not interfere. Interference can arise with respect to analysis state or the visual store as these are the shared memory elements.

The E-FRP semantics require that the analysis state not shift during processing of a single tuple (this is especially true for stateful primitive operations, per the discussion in Section 4.5). Prerendering requires that the visual store not have any elements added or removed during processing. Prerendering further requires that the analysis state not change during the entire phase of prerendering. Rendering requires that all elements to be rendered are consistent.

Elements are moved from fresh to transitional under the same epoch lock capture as viewpoint construction for the analysis state. No analysis state updates or commits to the visual store can occur while this epoch switch is occurring. Furthermore, the first task of prerendering is to merge the transitional values into the tenured collection.

Case 1: Analysis does not interfere with rendering through the visual store. Analysis puts no preconditions on the visual store. Therefore, rendering cannot violate its preconditions.

Rendering requires that the elements of the visual store to be rendered are consistent. The segmented visual store ensures that the elements of the store that rendering interacts with do not change during analysis. Recall that analysis only interacts with the fresh buffer, while rendering only interacts with the tenured buffer. Therefore, no analysis operations can change the consistency of the tenured collection.

Case 2: Analysis does not interfere with rendering through analysis state. Rendering does use the analysis state, so it trivially does not interfere.

Case 3: Analysis does not interfere with prerendering through the visual store. Since analysis puts no preconditions on the visual store, prerendering cannot violate its preconditions.

Recall that analysis only interacts with the fresh buffer, while prerendering interacts with the transitional and tenured buffers. Therefore, no analysis operation can add or remove values from the part of the visual store that prerendering is working with. Therefore, analysis does not interfere with prerendering.

Case 4: Analysis does not interfere with prerendering through the analysis state.

Prerendering works on a viewpoint. Because all operators have persistent representations, no updates made by analysis are seen by prerendering. Therefore, the prerendering analysis state can only be changed by prerendering itself. By construction, the operations of prerendering are not state changing (see Section 7.2 and Section 8.1) except for the global bind. Since global bind is deferred until after all other analysis is performed, no state changes occur. Therefore, prerenderings preconditions with respect to the analysis state hold.

Analysis requires that state not change during the analysis of an input tuple. Prerendering, by construction, does not mutate operators. However, global binding does mutate state in the visual store. However, the redefinition of the visual store's find operator in Section 7.2 makes those changes unobservable: The find operator performs calculations to simulate the results of prerender calculations. Therefore, no observable state change occurs. Therefore, prerendering does not violate the preconditions of rendering.

Since no preconditions are violated by concurrent phases, the definition of noninterference is satisfied. Therefore, operating analysis, prerendering and rendering can be performed concurrently with the help of persistent operators. □

9.5.3. Comparison. We compared a Stencil runtime using an RSL, a Stencil runtime exploiting the concurrency that persistent structures enable (see Section 9.5.2) and a Prefuse (beta) implementation on two tasks. The Stencil RSL implementation and standard runtime differed only in the placement of a single locking primitive (only two lines of code needed to be added). Both employed the persistent data structure from Section 9.5.1.6 but the RSL implementation did not exploit its properties to achieve concurrency. Two tasks

```
 1  public static final Point2D layout(int idx, int of) {
 2          idx = of -idx;
 3          final double x, y;
 4
 5          double shell = Math.floor(Math.sqrt(idx));
 6          double n = shell +1;
 7          double center = Math.pow(n, 2) - n;
 8
 9          if (idx<center) {
10                  x = shell;
11                  y = idx - Math.pow(shell, 2);
12          } else if (idx >center) {
13                  x = shell - (idx - center);
14                  y = shell;
15          } else if (idx==center) {
16                  x = shell;
17                  y = shell;
18          } else {throw new Error("Missed something...");}
19
20          return new Point2D.Double(x,y);
21  }
22  public static final Color colorAt(int idx,  int of) {
23          float perc = (of - idx)/((float) of);
24          return new Color(perc, 0f, 0f);
25  }
```

FIGURE 9.15. Java code for the dynamic quadrant filling curve shown in Figure 9.16. Lines 1 to 21 are for layout and 22-25 for coloring. This code fragment was used directly by the Prefuse implementation.

were used for measurements. The *static* task involved no dynamic bindings, so render-time work involved only rendering proper. The *dynamic* task involved dynamic bindings with significant work on both color and position using stateful operators. Both tasks were based on a quadrant-filling curve, coded in Figure 9.16 and shown in Figure 9.16. In the static task, the oldest element appears in the top left corner, but in the dynamic task the newest element is in the top left. The static task also used constant color, while the dynamic task changed the color to indicate age. (For reference, the Java coding of the layout and coloring code are given in Figure 9.15.)

The test machine used for all measurements is an 8-core Mac Pro with 4 GB of physical RAM. Tests were executed in Java 1.6 configured with 1.5 GB of heap space (both as the maximum and minimum heap size). In all tests, data were loaded into a memory array and then streamed into the visualization. Rendering requests were triggered by a separate thread that initiated a request at programmable intervals (discussed further in Section 9.5.3.1).

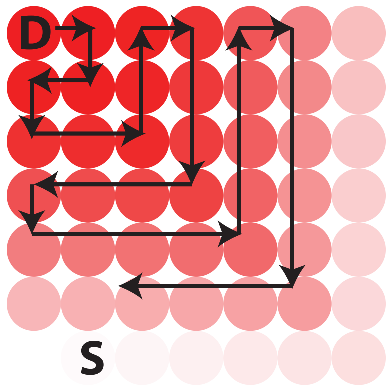


FIGURE 9.16. Quadrant filling curve used for evaluation (colored per the dynamic task). In the dynamic task, new elements are placed at the head of the layout (point D) and other points shifted according to the arrows. For the static task, the new elements are placed at the tail of the layout: the newest element is at S .

9.5.3.1. *Frame-Rate Sensitivity.* Early tests indicated that RSL-based implementations are sensitive to frame rates. This effect was investigated in more detail to ensure that the proposed tasks were feasible. Using the dynamic task described above, each runtime was applied with six different maximum frame rates (between 1 ms and 8000 ms enforced delay between renderings). Each test configuration was run once and results are presented in Figure 9.17. The non-RSL Stencil implementation exhibited no significant change in performance for *any* render delay. Prefuse and the RSL Stencil implementation exhibited longer running times as the render delay was decreased (and frame-rate increased). Furthermore, total time to complete the task grew exponentially with the amount of data loaded. Eventually, the RSL-based implementations would load zero or one data points between renderings, leading to resource starvation in the analysis thread and an expected running time measured in days. This performance difference indicates that the persistent data structure, at a minimum, provides a less delicate system, capable of performing tasks in spite of configurations that are adverse to more traditional implementations.

9.5.3.2. *Bulk Updates.* For a more rigorous comparison of running times, a delay factor of a half a second was selected. This decision was based on the desire to have the RSL-based implementations finish in a reasonable time. Each test condition was repeated five

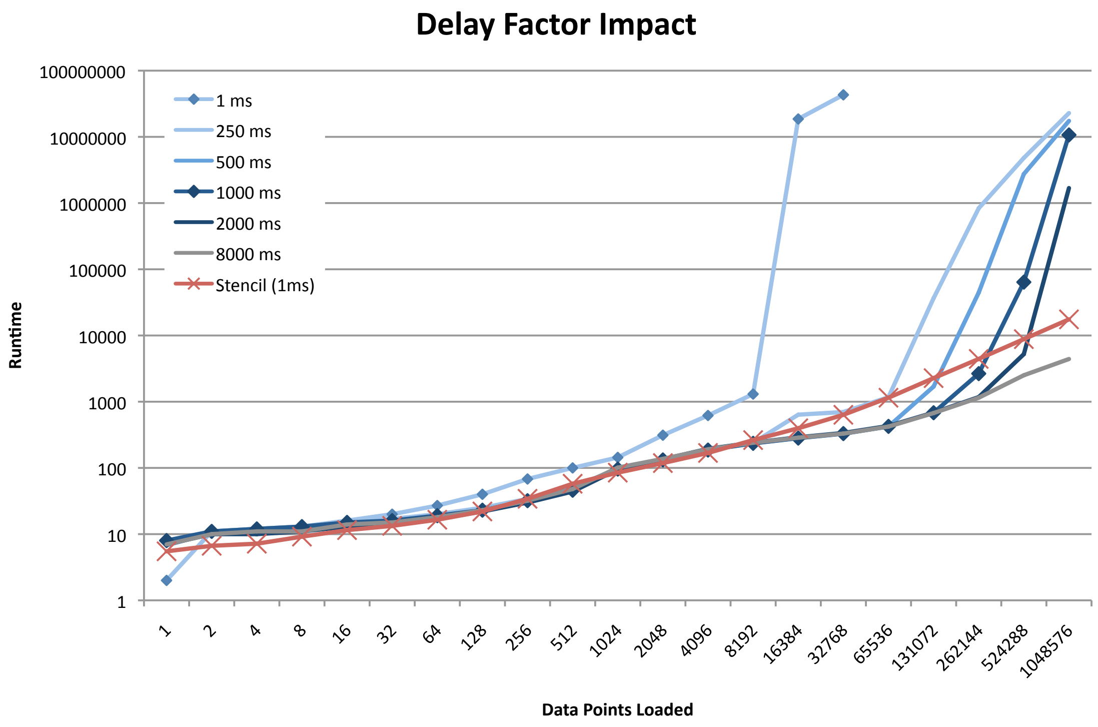
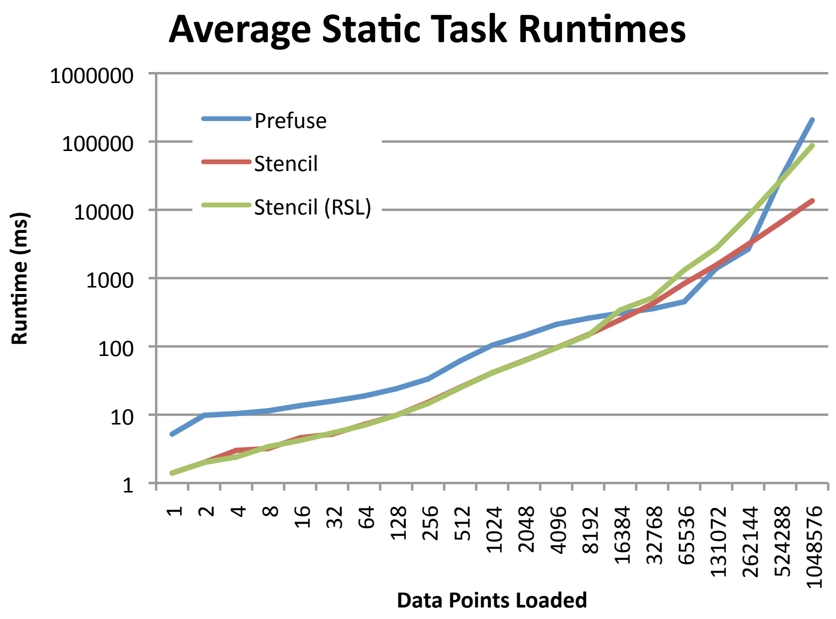


FIGURE 9.17. Dynamic-task running times with modified render request delays. Prefuse is sensitive to the amount of time between render requests. Given an 8 s render delay, the data loaded before the first rendering. However, with a 1 ms render delay, the loading did not finish when left overnight. Stencil exhibited next to no difference in running times as render delay changed. The worst-case, 1 ms delay, is presented here. Other delay settings were comparable. Similar results were observed in the static tasks, though with a more gentle slope.

times in the same JVM instance, with free heap space monitored between iterations to catch memory leaks (none were observed). Results summarized in Figure 9.18 represent the average of all executions. Standard deviation never exceeded 1.5% of total time for data sizes greater than 1. All implementations perform comparably for small data sets. However, as data size grows, the epoch-lock-based implementation builds an orders of magnitude advantage over the RSL-based implementations. In fact, the persistent-data-structure run time grows linearly, while RSL-based implementations running time grow exponentially. The exponential growth is driven by the fact that render-time activity is directly proportional to the number of elements in the data store. Therefore, as more data



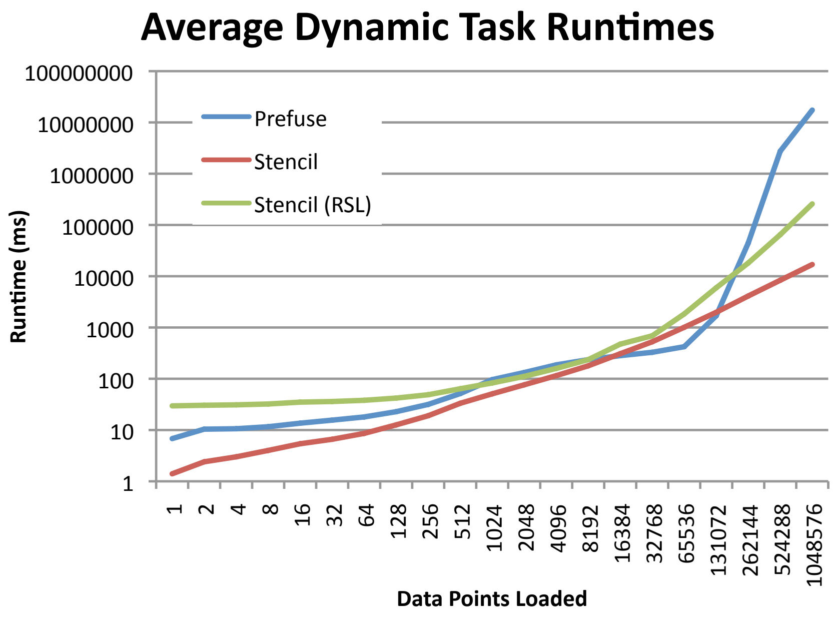


FIGURE 9.18. Comparison of performance using static and dynamic bindings in Stencil and Prefuse. Each point represents the average of five executions with the given data size. The two Stencil implementations show the difference between using an epoch lock and an RSL on otherwise identical runtime environments.

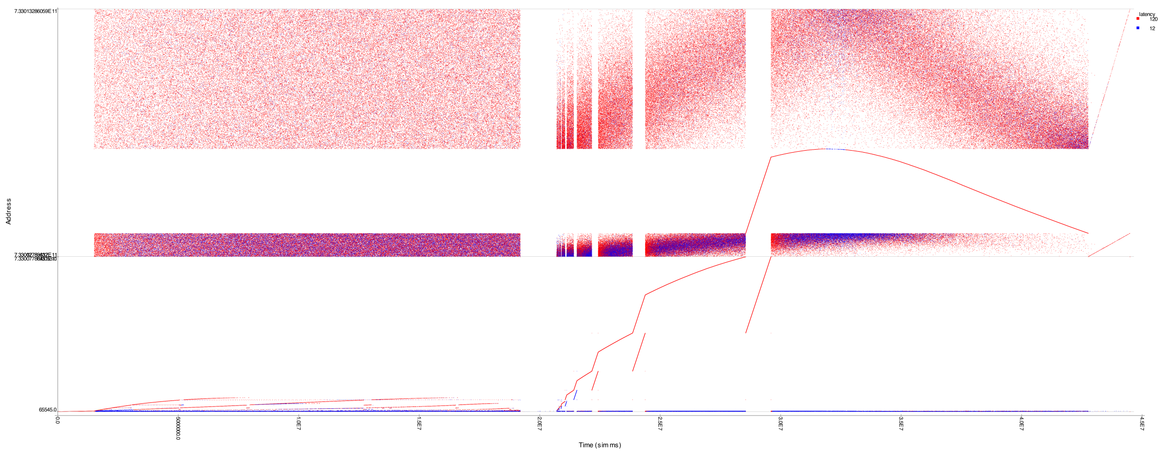


FIGURE 9.19. Memory-access visualization. A more detailed description and full source code are in Appendix A (Section A.3).

are loaded, more time is spent in render-time activities and less time is spent on analyzing new data.

The similarity between the Stencil RSL and non-RSL implementation in the static task up until around 32K indicates that the cost of dynamic binding is greater than the cost of rendering in Stencil for small data sets. However, after 32K the rendering costs are more substantial.

With a 500 ms delay (i.e., a maximum rate of 2fps), Prefuse has an advantage over Stencil for medium-sized data (the Prefuse line dips below the Stencil line between 16K and 131K). However, this advantage is an artifact of the configuration. Stencil performs nearly identically if the render delay is decreased or removed while Prefuse is severely impacted. The inflection point observed occurs shortly after the first time Prefuse renders. This indicates that the advantage Prefuse demonstrates here is more an artifact of our experimental setup than a true advantage. A similar, though less dramatic, inflection point is seen the Stencil-RSL implementation as well. In contrast, the disadvantages seen at large data sizes are reproducible at smaller data sizes times for both Prefuse and the Stencil RSL implementation by adjusting the render delay.

9.5.3.3. *Application Results.* The Stencil implementations were also compared on a visualization of memory accesses during graph analysis algorithms (breadth-first and depth-first search). The clients are trying to understand the impact of cache behavior on algorithms that do not have simple memory access patterns. Memory accesses were tracked during execution and a cache modeler was used to identify probable cache hits and page-table misses. These accesses are plotted as access time versus memory address (with unaccessed memory addresses omitted) and colored by access latency (based on the modeler's output). A sample output is shown in Figure 9.19. Exact Y location depends on a dynamic binding, because previously omitted addresses may be accessed later in the analysis. The presented data set included 23 million data points, loaded directly off disk from a text file. The eventual results include 1.2 million marks. Using the same test hardware, the persistent data store/epoch based runtime completes analysis and rendering in 40 s (average over 10 runs, worst run was 58 s, best was 28 s). No render delay was used for these tests and frame rates were inversely related to the number of data points loaded (initially over 60 fps, but eventually falling to 4 s/frame). The RSL-based implementation did not complete the task in 20 minutes and was abandoned after two attempts. The impact of the more efficient rendering system turned an impractical task into a responsive one, as intermediate results could be examined while analysis proceeded. (The complexity of the schema made an exact copy in Prefuse impractical, and the failure of the Stencil RSL implementation made the performance prospects poor. A simplified schema with no guides and approximate scaling was implemented in Prefuse; similar to the Stencil RSL implementation, this Prefuse implementation did not complete the task in the provided multi-hour time period.)

9.5.3.4. *Observed Concurrency.* An instrumented version of the Stencil runtime was produced using the bTrace code-injection framework. Entry and exit from load, render and update methods were recorded at the nanosecond level. A plot of the representative behavior is shown in Figure 9.20. The concurrency behavior predicted in Figure 9.10 is evident with rendering and (dynamic) updates overlapped with load activity. While calculating the quadrant-filling curve for 10K elements in the persistent Stencil runtime, loading

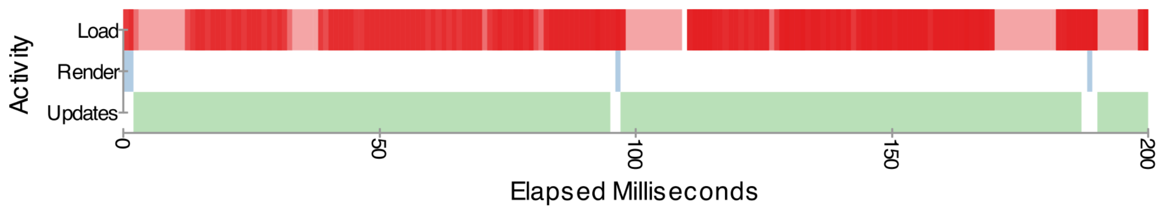


FIGURE 9.20. Activity timing recorded in Stencil. Overlap between analysis and render-time activities (rendering proper and committing static/dynamic updates) is easily observed. The darker regions in the load bar indicate multiple data loads in a single logical time step. The apparent overlap between data loading and updates is an artifact of the interaction between the instrumentation and the synchronization strategy. Instrumentation was based on method entry/exit, but the synchronization always occurred in a synchronized block *inside* a method. Therefore, the large light-colored areas in the load line shortly after the update bar begins indicate the span of the epoch lock.

proceeded 99% of the time and render-time activities occur 95% of the time. The RSL-based Stencil runtime performed loading 30% of the time and render-time activities 65% of the time. Furthermore, the percentages shift from load-time to render-time as more data points are loaded (this shift demonstrates why rendering needs to be throttled in order to complete large data sets: eventually render-time approaches 100% and no more data are loaded).

9.5.3.5. *Additional Frameworks.* The dynamic task was implemented with the Java implementation of Protovis [**68**] and VTK [**103**], and exponential time increases consistent with RSL implementations were observed in each. The Protovis-Java implementation required significant external synchronization in order to execute without concurrency errors (both explicit exceptions and inconsistencies in the rendered results). In the end, a *de-facto* RSL was required, resulting in an exponential increase in time to complete the task. The VTK implementation signaled additional data through the standard "vtkObject.modified()" method. This implementation also exhibited an exponential time increase, but with a gentler slope than the Java RSL-based frameworks. This exponential time increase leads us to conclude that an RSL-like discipline is internally observed, possibly centered on vtkExecutionScheduler.

9.5.4. Discussion. The concurrent version of Stencil, made possible by persistent data structures, is able to outperform other frameworks by a wide margin. The advantages observed cannot be attributed to just the Stencil runtime, as the RSL-based implementation used the same runtime but experiences delays similar to those seen in Prefuse. This advantage is held over Prefuse despite the fact that Prefuse is compiled to JVM byte-code and statically typed while Stencil is untyped and has significant abstraction penalties for method calls and type conversions in its runtime. The advantages shown in the preceding sections have been observed informally on machines with fewer resources (a 2 core MacBook and MacBook Air). Selectively disabling processor cores on the formal test setup indicated that Stencil exhibited strong scaling up to 5 cores (e.g., more cores led to greater speed). Five cores roughly corresponds to the number of tasks that Stencil presents in the dynamic schema (static analysis, rendering and three dynamic bindings). Earlier work on Stencil showed a clear means for employing data parallelism in dynamic binding [**32**] that would likely extend the strong scaling result, similar to that seen in Protovis [**68**].

The change-set and committed architecture of the persistent data store is similar to a Map/Reduce system [**38**]. The change-set is the result of the map and the committed columns are the results of the reduce. This correspondence suggests one way that a distributed visualization system might be structured. However, there are significant issues with operator state and overlapping commits that would need to be resolved. A more careful treating of the persistent data structure as a distributed shared memory [**96**] protocol might provide further insight into how to resolve these issues.

The data structure presented does not conform to classical implementations of persistent data structures in a strict language. However, the Young queue essentially enables lazy evaluation, where the final commit is not performed until needed (i.e., until render time). The resulting data structure can produce a persistent view on demand, but only actually produces one intermittently for efficiency. The formal properties of *intermittently* persistent data structures have not been established, but are likely related to Lazy Rebuilding [**86**]. A greater understanding of intermittently persistent data structures may be useful for other task-based environments.

Strictly speaking, the rendering and prerender computations could proceed independently of each other. Rendering relies only a persistent view, while the prerender computations take updates, and a data set, to produce a new persistent view. Section 9.5.1.6 presented the relationships implemented in Stencil, but other options exist. For example, rendering could grab the current tenured collection immediately and render its contents, while an independent process calculates and merges updates. This could lead to additional computations (esp. dynamic bindings) that are never displayed (as two such recalculation/merge actions may occur during a single rendering). However, this could also improve rendering responsiveness as it would always have data immediately available. The balance of this tradeoff has not been characterized. However, with ubiquitous multicore hardware, it may be advantageous to decouple the analysis and rendering tasks further.

A significant portion of Stencil's render-time activity is spent allocating space for data updates and copying values not being changed (e.g., results for columns only involved in static bindings). This shows up as additional garbage collector activity and as time spent at render time not rendering (in the quadrant filling test case, this accounts for approximately 10% of render-time activity when 1 million points are present). One avenue for reducing this cost is to revise the column implementation. The copy-on-first-modification for column updates is rooted in their simple implementation: an array. In a worst-case scenario, all columns require updates and must be copied to update a single row. In a table of n items and c columns, this yields $n \times c$ copies and $2(n \times c)$ memory cells for updating c values. This overhead is similar to that seen in double-buffering to eliminate screen flicker, but with the disadvantage that the buffers generally cannot be reused (they are often of the wrong size). Fortunately, this does not need to be a deep copy and it does not occur in the epoch lock, but it may be sizable and thus increase pressure on the memory system. A more efficient column implementation would reduce this issue. Initial experiments, using 2D arrays so only portions of the array need to be copied, show a significant reduction in memory activity (both directly and in the garbage collector). However, the implementation requires tracking versions on portions of the column and is not yet complete. The impact of such changes is likely workload-dependent as well.

A significant issue with dynamic data visualization in general is view management. When working with static data, the data does not generally change while it is observed. Animated network layouts are a notable exception; tracking elements while the layout is active can be difficult. The general solution is to only use layouts that quiesce or can be halted and to not look at details until after changes have stopped. This is not possible in dynamic data visualization, where new data is assumed to be the norm. How to effectively select the zoom/pan settings for the next render based on what was being observed in the prior one is an unaddressed question.

Using knowledge of how the visual store is used enabled a task-specific persistent data structure. The analysis operators provided with Stencil have not been so carefully treated. Most employ generic persistent data structures (commonly from the pcollections library [**28**]), copy-on-write techniques whenever stateful operations are performed or a deep-clone when the persistent view is requested. These techniques work well enough, but a change-list strategy with merge done on persistent view creation might benefit some of them. When the more complex change-list strategy is better than the others is a valuable question.

9.6. Conclusion

Computational abstractions are an important part of making a software framework practical as well as expressive. Providing such abstractions requires preservation of the defined properties of the framework. Using the semantics of Chapter 5 and the sequencing E-FRP provides in particular, computational abstractions have been provided for the Stencil framework. This includes constant propagation, efficient scheduling of potentially expensive operations and the ability to exploit modern hardware through automatically identified concurrency opportunities (using persistent data structures). A powerful feature of a well-defined framework is to provide such abstractions without requiring additional syntactic elements and without modifying the results produced by the framework.

10
Conclusions

This dissertation has provided a formal framework for exploring visualization-related abstractions. It has included formal semantics for a novel visualization framework, and metadata focused on supporting visualization program correctness and transformations. These semantics and metadata have been used to establish properties important to properly constructed visualizations. With this foundation, abstractions relevant to visualization creation have been provided in a declarative fashion.

10.1. Future Work

The framework described by this dissertation provides a basis for exploring future declarative approaches to visualization. Major future directions for research align with semantic and metadata components that lie at the foundation of this work. Significant, though supplemental, work can also be found in the implementation details.

A fundamental assumption of this work is that deterministic semantics are desirable in a visualization framework. Incorporating nondeterminism in a structured way is a meaningful future direction. Deterministic semantics are useful for evaluating program transformations by establishing testable properties of the program. However, this assumes that all analysis operators are also deterministic. Nondeterministic algorithms used in visualization include some graph layouts sensitive to original positions and iteration order [**50**] or that overtly include probabilistic calculations [**37**] and seeded clustering algorithms which are sensitive to original seed placement [**10**,**81**]. Though some such operators can be represented deterministically by providing explicit seeds to their random processes, this is not always practical (e.g., the operator might represent a long-running, external analysis engine that is only being queried).

The definition of consistency used in Stencil is restrictive. Exploring more relaxed the consistency models would enable greater parallelism and thus more efficient use of current and projected future architectures. The current model is analogous to *linearizable* consistency from distributed systems [**110**] in that there is a strict ordering requirement based on a clock (the clock being the dispatcher in Stencil's case). For example, the consistency model dictates that tuples be processed from the dispatcher in an order predicated only on their arrival order and tuple contents. In this system, the runtime characteristics of the analysis pathway consuming a stream cannot influence dispatch order. Therefore, a slow analysis pathway can cause other pathways to remain idle, even if there are data waiting and the two pathways do not share memory. Processing multiple elements on the same analysis pathway in a pipe-line style is restricted in a similar way.

The operator metadata included in the Stencil framework was driven by the abstractions being approached. It is not intended as an exhaustive catalog of metadata pertinent to visualization. Future work on declarative visualization construction would benefit from further investigation of the presented metadata, as well as additional metadata categories. For example, prior work has shown that associativity and commutativity play important roles in parallel computation (for example, Pottenger [**95**]). Global data properties are often based on associative operators (like *min*). A better understanding of, and structured access to, metadata concerning operator associativity would help identify places where table-based data declarations could be efficiently implemented. In many ways, the metadata system is a simple type system. Extending that type system to encompass more detailed information about operators is one direction for extension. More detailed information about the input data or results of individual operators would also expand the analysis options. Such type extensions will likely yield the highest utility if they remain abstracted from low-level types (like int and double) and focus more on capabilities (like commutative operators or metric data).

The semantics presented in this dissertation are device independent. They guarantee that a proper implementation of Stencil should produce the same visualizations provided with the same data. Recent work on Protovis indicated some device-dependent factors in visualizations [**68**]. For example, a small screen and a large screen afford different characteristic feature sizes and density of reference marks. One of the advantages of the declarative style of programming is the ability to modify a program automatically. Automatic rewrites for multiple devices is a reasonable direction. Representing device characteristics and how these characteristics interact with semantics and operator frameworks remains an open question.

The semantics presented have been implemented in Stencil using Java. However, Stencil does not represent the only possible implementation. Implementations in hardware or software environments with different execution models (such as SQL or GPGPU contexts) could allow the benefits of databases and data-parallel hardware to be applied to visualization in a structured way. VizQL already demonstrates how well-principled integration between databases and visualization can be beneficial both representationally and computationally [**64**]. Taking full advantage of the opportunities of alternative semantic environments may require a different consistency model (as discussed above).

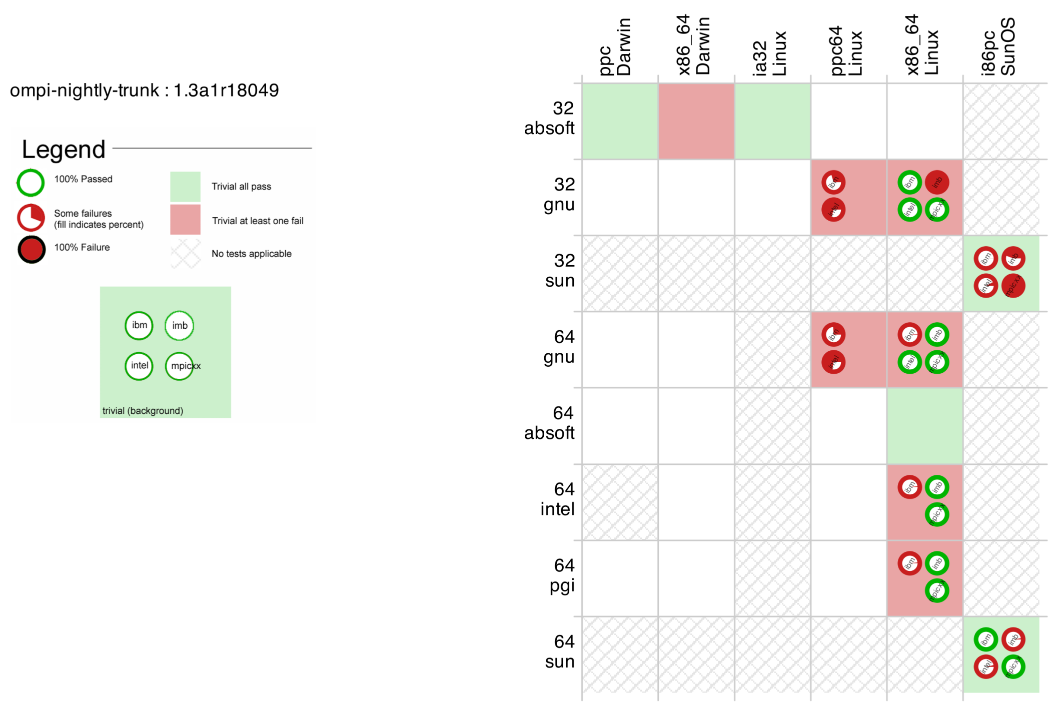
Working with sequences of data points is a natural way to approach visualization problems. This concept is at the core of the data-flow model of visualization and included in the implementation of iterator-based analysis algorithms. However, the data-state model of visualization conceptually leverages the idea of a data table to discuss global properties and local-to-global relationships in a natural way. Though possible in the data-flow model, the concept of a pooled collection of data makes the formulation of some analysis more direct. Stencil provides some access to this model of execution through dynamic binding, but by so doing conflates the visual and analytical data stores. A formal treatment of data-state semantics could illuminate the design space that dynamic bindings touch on. It might also provide alternative ways to move between the two models so the concepts can be intermingled in a flexible but controlled fashion.

Though the framework presented in this dissertation is able to accomplish a variety of visual effects, there are still omissions in its implementation. Some of these omissions indicate directions for future investigation. For example, the order statement only implements some of the potential stream sequences possible (see Section 6.5). Notably, the actual data values (such as a timestamp) cannot be considered in the prioritization algorithm. Some interesting effects can be achieved with internal stream declarations and stateful transformation operators. This suggests that the dispatcher may be reduced to a simple queue with stream declarations collectively creating complex priorities. Another omission is high-level support for data removal. Individual data points can be removed from a layer, but removal from the state-space of a stateful operator relies on the operator itself. This situation is analogous to dynamic binding, where adding information to the system is controlled through high-level abstractions that depend on operator meta-data. With additional operator meta-data, similar support for data removal may be possible.

The Stencil system provides facilities for interactive visualizations of dynamic data. However, the schema used to present the data is specified entirely at compile time. The E-FRP framework underlying the semantics supports higher-order transformations and self-modifying event/behavior networks. This ability is used in the Split operator (Section 7.1), but nowhere else in the Stencil system. These facilities could be used to support interactive visualization creation, where modifications to the schema are displayed directly on preloaded data. They would also enable more complex analysis with interconnections that can be configured based on the data presented. Higher-order functions in the data-analysis framework are implemented in Luster **[72]** and available in Haskell FRP implementations through lifting. How such higher-order analysis schemas apply to visualization in particular has not been explored. However, interactive schema creation often used in GUI tools may give some indications as it represents one avenue where schemas evolve in a controlled fashion.

A
Sample Projects

Several projects use Stencil for visualization. These projects represent real-world problems solved in part using Stencil. This chapter highlights some of those projects. In addition to the project highlights, some common visualizations are also included with the Stencil programs that create them. These visualizations demonstrate the breadth of visualizations that Stencil can approach.


ompi-nightly-trunk : 1.3a1r18049
Legend
100% Passed
Some failures (fill indicates percent)
100% Failure
Trivial all pass
Trivial at least one fail
No tests applicable
ibm
imb
intel
mpicxx
trivial (background)
ppc Darwin
x86_64 Darwin
ia32 Linux
ppc64 Linux
x86_64 Linux
i86pc SunOS
32 absoft
32 gnu
32 sun
64 gnu
64 absoft
64 intel
64 pgi
64 sun

A.1. SeeTest and the MPI Testing Tool

```
import PatternUtils

stream MTTSource(graphLabel, axis1A, axis1B, axis2A, axis2B, suite_name, pass, fail)
stream MTTNotApplicable(axis1A, axis1B, axis2A, axis2B)

order MTTSource > MTTNotApplicable

layer Trivial
from MTTSource
    filter (suite_name =~ "trivial")
    ID : Concatenate(axis1A, axis1B, axis2A, axis2B)
    (X,Y): Layout(axis1A, axis1B, axis2A, axis2B)
    FILL_COLOR : BasicFails(fail)
    SIZE: 50
    SHAPE: "RECTANGLE"
    REGISTRATION: "TOP_LEFT"


layer NotApplicable
from MTTNotApplicable
    ID: Concatenate(axis1A, axis1B, axis2A, axis2B)
    (X,Y) : Layout.query(axis1A, axis1B, axis2A, axis2B)
    SIZE: 50
    SHAPE: "RECTANGLE"
    REGISTRATION: "TOP_LEFT"
     FILL_COLOR: [fore] Palette("N/A") -> PatternFill("hatch", fore, NULL, 10, 1.1)


layer Others["PIE"]
from MTTSource
    filter(suite_name !~"trivial")
    ID : Concatenate(axis1A, axis1B, axis2A, axis2B, suite_name)
    (X,Y): Layout.suite(axis1A, axis1B, axis2A, axis2B, suite_name)
    SIZE: 13
    PEN: Stroke{3}
    PEN_COLOR: RelativeFails(fail, pass)
    SLICE: fail
    FIELD: pass
    SLICE_COLOR: Palette("FAIL")
    FIELD_COLOR: Color{WHITE}


layer SubLabels["TEXT"]
from MTTSource
    filter(suite_name !~"trivial")
    local(ID): Concatenate(axis1A, axis1B, axis2A, axis2B, suite_name)
    ID: local.ID
    (X,Y): Others.find(local.ID) -> (Others.X, Others.Y)
    TEXT: suite_name
    FONT: Font{5}
    ROTATION: -45
    REGISTRATION: "CENTER"
    FONT_STYLE: "BOLD"
    COLOR: Color{0,0,0,166}

layer YLabels["TEXT"]
from MTTSource
    ID: Concatenate(axis1A, axis1B)
    X: Layout(axis1A, axis1B, axis2A, axis2B) -> Add(Layout.X, 12)
    TEXT: Concatenate(axis1A, "\n", axis1B)
    (Y, WIDTH): (5, 50)
    (ROTATION, JUSTIFY): ("VERTICAL", "LEFT")
```

```



layer XLabels["TEXT"]
from MTTSource
    ID: Concatenate(axis2A, axis2B)
    Y: Layout(axis1A, axis1B, axis2A, axis2B) -> Sub(Layout.Y, 12)
    TEXT: Concatenate(axis2A, "\n", axis2B) -> Break(_) -> Break.Label
    X : -5
    (JUSTIFY, REGISTRATION): ("RIGHT", "TOP_RIGHT")



layer CanvasLabel["TEXT"]
from MTTSource
    ID: graphLabel
    Y: NamesIndex(graphLabel)
    TEXT: graphLabel
    X: -375

layer Legend["IMAGE"]
from MTTSource
    ID: "LegendImage"
    FILE: "./SeeTest-Key.png"
    (X,Y,WIDTH,HEIGHT) : (-375,-10,200, 200)


layer XGridLines["LINE"]
from MTTSource
    ID : Concatenate(axis1A, axis1B)
    (X1, X2): Layout(axis1A, axis1B, axis2A, axis2B) -> (Layout.X, Layout.X)
    Y1 : 5
    Y2 :* EqualHeight(axis1A, axis1B, axis2A, axis2B)
    PEN_COLOR : Palette("GRID")

layer YGridLines["LINE"]
from MTTSource
    ID : Concatenate(axis2A, axis2B)
    (Y1,Y2) : Layout(axis1A, axis1B, axis2A, axis2B) -> (Layout.Y, Layout.Y)
    X1 : -5
    X2 :* EqualWidth(axis1A, axis1B, axis2A, axis2B)
    PEN_COLOR: Palette("GRID")

operator EqualWidth (axis1A, axis1B, axis2A, axis2B) -> (X2)
    default => X2 : Layout(axis1A, axis1B, axis2A, axis2B) -> Add(Layout.X, 55) -> Range[ALL](@Max, Add)

operator EqualHeight (axis1A, axis1B, axis2A, axis2B) -> (Y2)
    default => Y2 : Layout(axis1A, axis1B, axis2A, axis2B) -> Sub(Layout.Y, 55) -> Range[ALL](@Min, Sub)


operator NamesIndex(ID) -> (VALUE)
    default => VALUE: Index(ID)

operator BasicFails(fails) -> (C)
    (fails = 0) => C: Palette("PASS") -> SetAlpha(51, Palette.C)
    (fails > 0) => C: Palette("FAIL") -> SetAlpha(102, Palette.C)

operator RelativeFails(fails, passes) -> (C)
    (passes= 0) => C: Color{BLACK}
    (fails = 0) => C: Palette("PASS")
    (fails > 0) => C: Palette("FAIL")

operator Palette(Name) -> (C)
    (Name =~"FAIL")    => C: Color{200,30,30}
    (Name =~"PASS")    => C: Color{0,180,0}
    (Name =~"MISSING") => C: Color{WHITE}
    (Name =~"N/A")        => C: Color{GRAY90}
    (Name =~"GRID")       => C: Color{GRAY80}

java Break
{
   @Facet(memUse="FUNCTION", prototype="(Label)", alias={"map","query"})
   public String query(String original) {
      return original
                .replace("Absoft: Absoft ", "")
                .replace("Absoft: ", "")
                .replace(",", "\n");
   }
}

java Layout
{import java.util.*;}
{
   static final List<String> suites = Arrays.asList("ibm", "imb", "intel", "mpicxx");
   static final double majorOffset = 50;
   static final int border = 15;
   static final int minorOffset = 18;
   static final List<String> XIndex = new ArrayList();
   static final List<String> YIndex = new ArrayList();

   @Facet(memUse="WRITER", prototype="(double X, double Y)")
   public double[] suite(String X1, String X2, String Y1, String Y2, String suite) {
      double[] base = map(X1,X2,Y1,Y2);
      if (suites.indexOf(suite) <0) {throw new IllegalArgumentException("Unknown suite: " + suite);}
      int offset = suites.indexOf(suite);
      double x = base[0] + border + (offset%2 * minorOffset);
      double y = -(-base[1] + border + (offset/2 * minorOffset));
      return new double[]{x,y};
   }

   @Facet(memUse="WRITER", prototype="(double X, double Y)")
   public double[] map(String X1, String X2, String Y1, String Y2) {
        String XBase = X1+X2;
        String YBase = Y1+Y2;
        if (!XIndex.contains(XBase)) {XIndex.add(XBase);}
        if (!YIndex.contains(YBase)) {YIndex.add(YBase);}
        double X = XIndex.indexOf(XBase) * majorOffset;
        double Y = -(YIndex.indexOf(YBase) * majorOffset);
        return new double[]{X,Y};
   }

   @Facet(memUse="READER", prototype="(double X, double Y)")
   public double[] query(String X1, String X2, String Y1, String Y2) {
        String XBase = X1+X2;
        String YBase = Y1+Y2;
        if (!XIndex.contains(XBase)) {return new double[]{-375,300};}
        if (!YIndex.contains(YBase)) {return new double[]{-375,300};}
        double X = XIndex.indexOf(XBase) * majorOffset;
        double Y = -(YIndex.indexOf(YBase) * majorOffset);
        return new double[]{X,Y};
   }
}
```

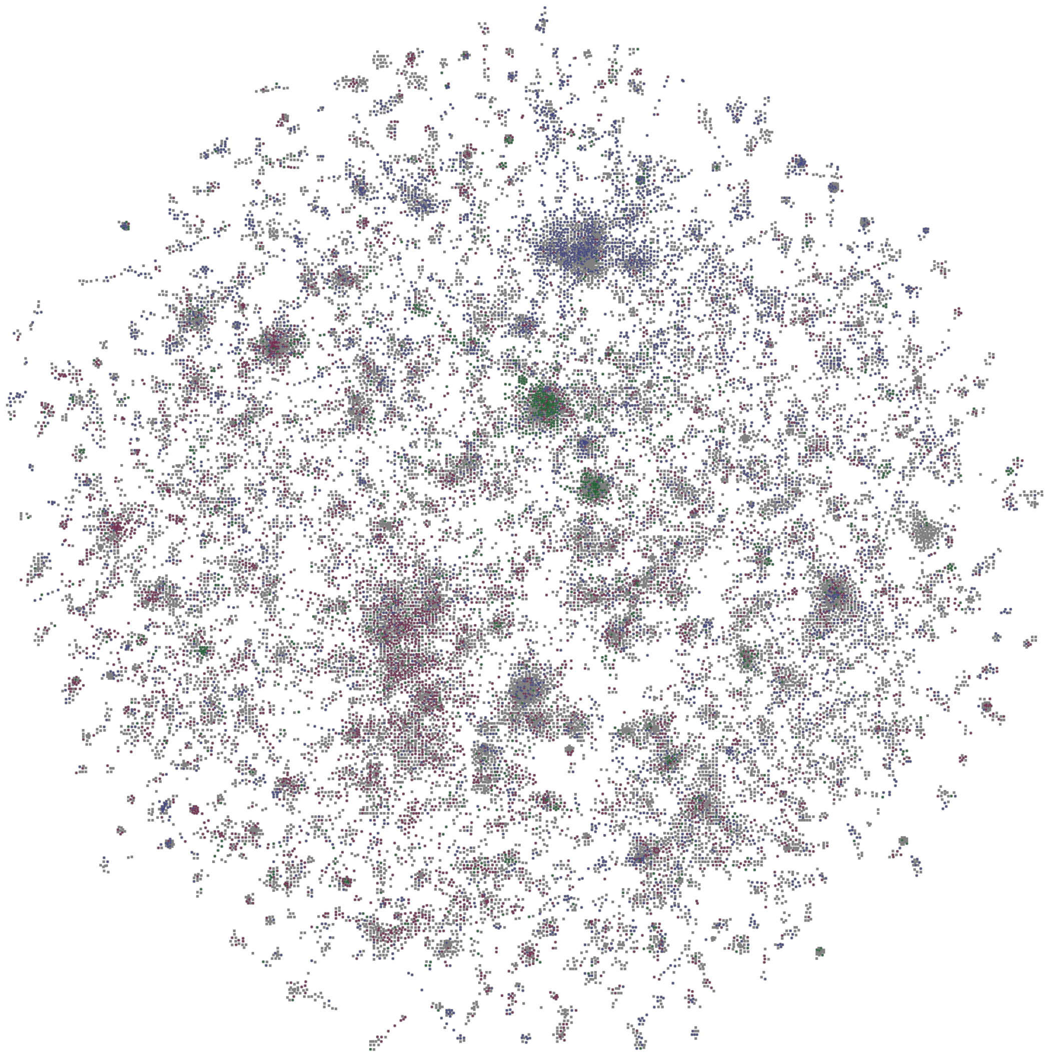
The SeeTest visualization schema represents the unit-test data produced around the Open-MPI project. Building up to a major release, testing data increases rapidly. However, the complex parameter space for the tests, the variety of organization involved (each with its own concerns) and levels of aggregation required made existing test reporting tools unsuitable. A dry-erase board based, hand-updated visualization supported an earlier release of OpenMPI. The SeeTest schema is a refinement and automation of that hand-drawn visualization. It formed the central part of the website reporting nightly test results. Each morning, the MPI Testing Tool collected and aggregated results for each organization and committee. Scheduled visualization production supported these meetings. *Ad hoc* visualization requests (during or in anticipation of a specific meeting discussion) were also served by the MPI Testing Tool.

Two visualization developers produced the basic SeeTest Stencil program, though three other MPI Testing Tool users also contributed Stencil code. The Layer abstraction was one feature noted as enabling experimentation. The clear semantic link between analysis and visual results, and clear separation between layers were the important aspects cited. OpenMPI developers created the"not-applicable" layer and the hinting at near-complete pass/fail on top of earlier schemas without training.

The MPI Testing Tool required efficient handling of both large batches and single requests. This required integration with the MPI Testing database (with its evolving database schema) to acquire data and the ability to generate hundreds of images in batch for nightly updates.

Reports from the OpenMPI development community indicated that the visualization unambiguously indicated that their initial test resource allocation was too concentrated on simple cases (too many tests using too few nodes). The visualization also helped identify testing resources spent on low priority configurations.

A more complete description of the MPI Testing Tool and SeeTest appeared in SoftVis 2008 [**33**].

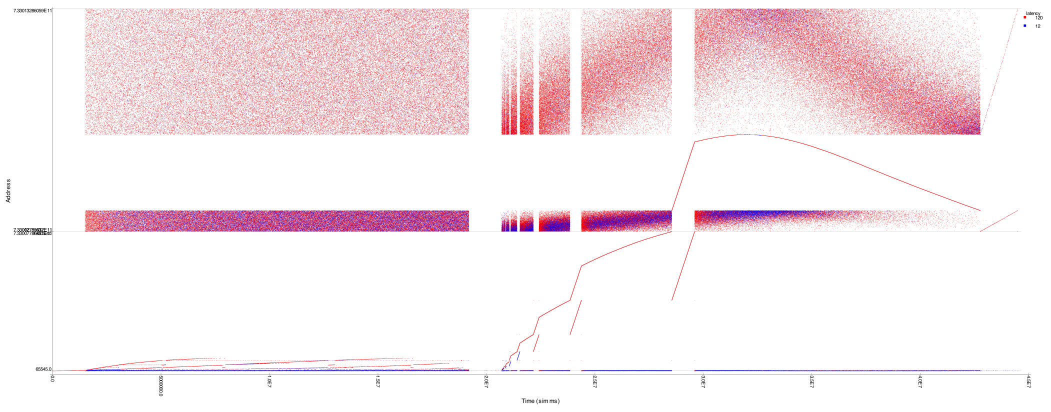


A.2. Sourceforge Social Network Analysis

```

stream NodePositions(ID, X,Y)   /*Precomputed layout*/
stream Attributes(ID, ATT)        /*Additional attributes*/

order NodePositions > Attributes          /*Load the NodePositions first*/

layer Nodes
from NodePositions
     ID: ID
     (X,Y) : (X,Y)
     SIZE: 3
     SHAPE: "RECTANGLE"
     FILL_COLOR: Color{GRAY}

layer Overlay
from Attributes
     filter (ATT =~ "C|Java|Python")              /*Only continue with specific attributes*/
     filter (prefilter.NodesID != NULL)               /*...and with things in the other layer.*/
     prefilter(NodesID) :   Nodes.find(ID) -> Nodes.ID

     ID : ID
     (X,Y) : Nodes.find(ID) -> (Nodes.X,Nodes.Y)
     SHAPE: "CROSS"
     SIZE: 3.1
     FILL_COLOR: Coloring(ATT)

operator Coloring(ATT) -> (C)
     (ATT =~ "Python") => C : Color{30,100,50}
     (ATT =~ "Java")    => C : Color{65,70,135}
     (ATT =~ "C")       => C : Color{120,40,80}
```

This figure presents the relationships between projects in the `Sourceforge.net` repository in 2007. Each node represents a project (links are not shown, but each link was a shared person). Analysis of the largest connected component started with Distributed-Recursive Layout (DRL) [**37**]. The Stencil program overlays information about the programming languages used on the DRL layout. Research conducted in 2007 employed Stencil to produce similar images for hundreds of different attributes. The figure is a composite of the three most visually salient elements. The observation that all three are programming languages led to an investigation of the role of programming languages in open-source software [**29**].

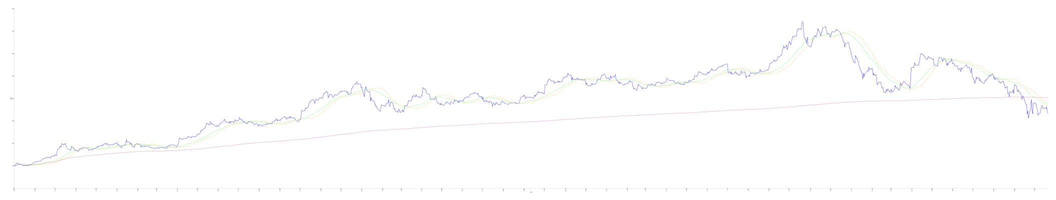
This project relied on the advantages of declarative visualization construction and Stencil's implementation in particular. The ability to rapidly prototype visualizations, combine data from multiple sources and integrate the visualization into a larger software pipeline all played important parts in the research project. Multiple visualization techniques produced an overview of Sourceforge. The network-based visualizations were the most promising and became the focus. Subsequently, the ability to efficiently combine the DRL layout with additional information from the Sourceforge data facilitated the investigation of multiple ideas using a common substrate or base-map. Finally, generating hundreds of images (producing an "image file", in Bertin's vocabulary [**8**]) was efficient because Stencil is easily integrated into batch processes.


7.33013286059E11
Address
65545.0
0.0
5000000.0
1.0E7
1.5E7
2.0E7
2.5E7
3.0E7
3.5E7
4.0E7
4.5E7
Time (sim ms)
latency
120
12

A.3. Memory Access Patterns

```

view
from #Render
  (X,Y,WIDTH,HEIGHT) : (canvas.X, canvas.Y, canvas.WIDTH, canvas.HEIGHT)


layer Memory
guide
   axis[implant: "POINT"] from X
      label.FONT: @Font{20}
   gridlines[implant: "LINE", seed.margin:MARGIN] Gap from Y
   axis[implant: "POINT", seed.margin:MARGIN] Segment from Y
      label.FONT: @Font{20}
      label.REGISTRATION: HighLow(Input.1)
   legend from FILL_COLOR
from Accesses
  filter(latency >3)
  filter(addr < CUTOFF)
  ID: Count()
  X: instr
  Y:* Div(addr, 64) -#> Rank(_)
  FILL_COLOR: Log[base:10](latency) -> Color(_)
  IMPLANT: "POINT"
  SHAPE: "RECTANGLE"
  SIZE: 1


operator Color : HeatScale[hot:"Blue", cold: "Red"]
operator HighLow (end) -> (Reg)
  (end = "start") => Reg: "BOTTOM_RIGHT"
  (end = "end")   => Reg: "TOP_RIGHT"
```

Visualization of the memory access pattern and cache locality behavior of the Boost Graph Library (BGL) for a breadth-first and depth-first search. We recorded the memory accesses of a BGL execution. A simulator provided access latencies for hypothetical hardware. Addresses appear on the Y axis, with time offset along the X. The plot omits memory addresses without accesses to save space; gridlines indicate skipped blocks. Color encodes access latency (blue for long latency, red for short). Y-axis positions are dynamically bound because the maximum and minimum values are recorded directly from the data stream.



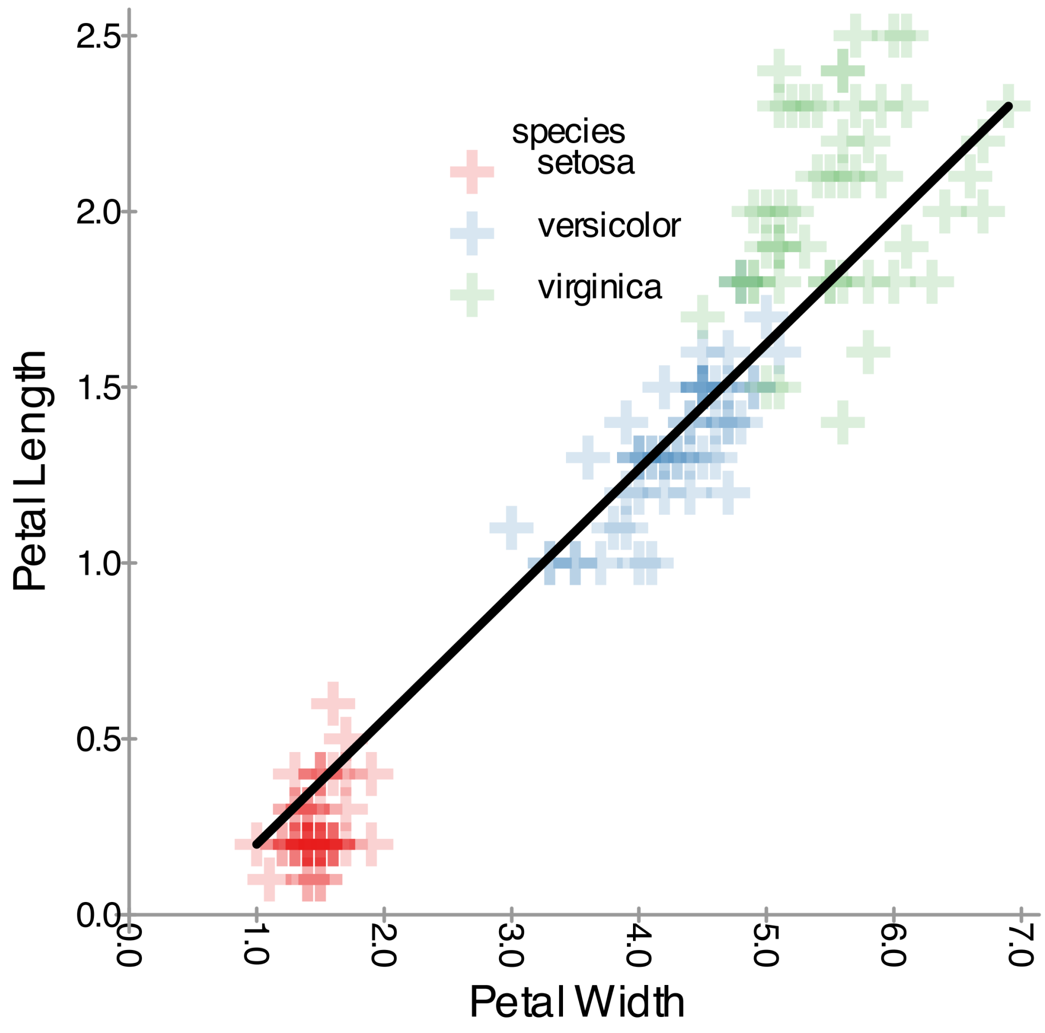
A.4. Stock Ticker

```
NAME: Mouse
FREQUENCY: -2147483648

import Dates

stream Stocks(Date, Open)
stream Mouse(X, Y, DELTA_X, DELTA_Y, BUTTON, SHIFT, CNTRL, COMMAND, CLICK_COUNT)

stream AlignedStocks(Date, Open)
from Stocks
   Date : Parse[f:"dd-MMM-yy"](Date)
   Open : Open

layer Ticker["POLY_POINT"]              /*Poly-point is a group of groups of line segments.*/
guide
     axis Linear from Y
     axis[unit: "MONTH"] Date from X
         label.TEXT: Parse.format["MMM yy"](Input)

from AlignedStocks
     ID: TimeStep(Date)
     GROUP: "Google"
     ORDER: Count()
     X: TimeStep(Date)
     Y: Open
     PEN: @Stroke{1}
     PEN_COLOR: @Color{BLUE}

layer Averages["POLY_POINT"]
from AlignedStocks
     ID: Ids()
     GROUP: "Google_30"
     ORDER: Count()
     X: TimeStep(Date)
     Y: Range[" -30..n"](@Mean, Open)          /*Average the most recent 30 values.*/
     PEN: @Stroke{1}
     PEN_COLOR: @Color{LIME,150}
from AlignedStocks
     ID: Ids()
     GROUP: "Google_ALL"
     ORDER: Count()
     X: TimeStep(Date)
     Y: Range[ALL](@Mean, Open)                /*Average all values.*/
     PEN: @Stroke{1}
     PEN_COLOR: @Color{CRIMSON,150}
from AlignedStocks
     ID: Ids()
     GROUP: "Google_20"
     ORDER: Count()
     X: TimeStep(Date)
     Y: Range["-30 .. -10"](@Mean, Open) /*Average that lags by 10 and includes 20 values.*/
     PEN: @Stroke{1}
     PEN_COLOR: @Color{ORANGE,150}


layer Selected["TEXT"]
from Mouse
   local(point) : Nearest(X,3)
   ID : "Selected"
   TEXT: local.point
   REGISTRATION: "LEFT"
   IMPLANT: "POINT"
   (X, Y) :[FullMax] Add(canvas.X, canvas.WIDTH)
```

```
64              -> Max(canvas.X, local.point) -> Min(FullMax, _)
65              -> Ticker.find(_) -> Add(50, Ticker.Y) -> (Ticker.X, _)
66
67  layer Highlight["LINE"]
68  from Mouse
69    ID : "Highlight"
70    (X1, X2) : Nearest(X, 3) -> (_, _)
71    Y1 : Add(canvas.Y, canvas.HEIGHT) -> Sub(_, 3)
72    Y2 : Add(canvas.Y, 3)
73    PEN : @Stroke{3}
74    PEN_COLOR: @Color{BLUE, 100}
75
76  operator Ids : Count
77
78  operator TimeStep (T) -> (S)
79  default => (S) : Range[ALL](@DateMin, T)
80                    -> DateDiff("DAYS", Range, T)
81                    -> Mult(_, 3)
82
83  operator DateMin : Min[c:"Date"]
```

An example stock ticker that computes the to-date and sliding window averages in addition to displaying the raw data. The guides demonstrate the ability to modify individual graphic elements from the defaults and control sampling rates. Furthermore, this schema includes interactive elements: a "current-value" line displays the current mouse position, and the corresponding price (the image does not include any interactive elements). Interactive elements appear on lines 56–74. Such elements could fit (logically) into the guide system, but would require a more rich module import system. This schema is similar to that used for prototyping in EpiC.

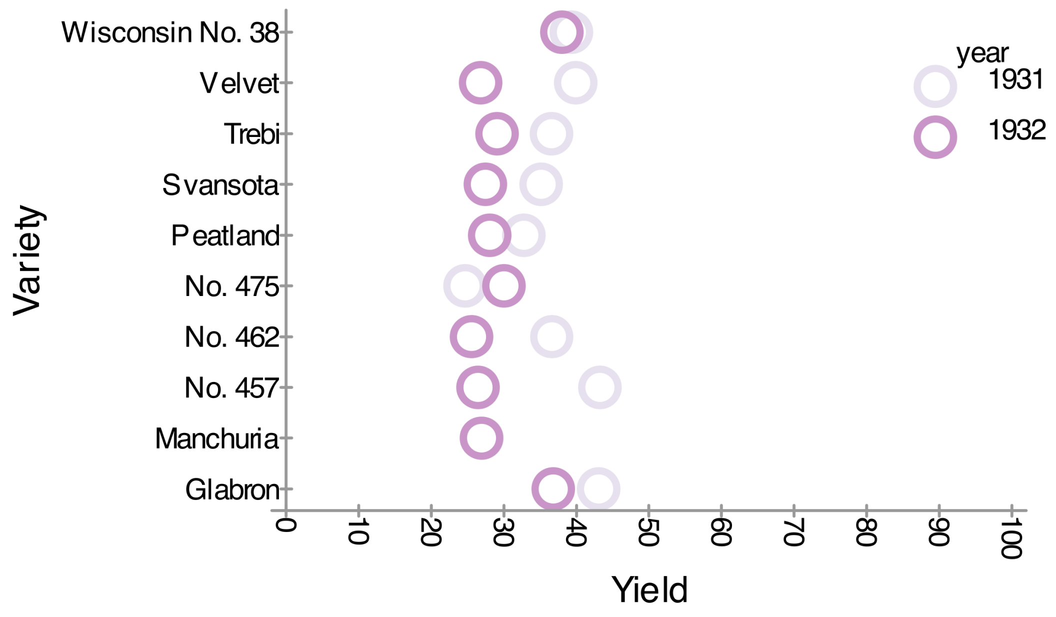
This visualization demonstrates Stencil's date handling. Line 11 parses the incoming date string into a date tuple. Date sampling in the guide and calculating the offset since the minimum date all respect date conventions. For example, the date sampling can be by month (see Line 18). This is in contrast to treating dates as epoch milliseconds, with repeated semantic reinterpretation at each reuse of the date.


species
setosa
versicolor
virginica
Petal Length
Petal Width
0.0
0.5
1.0
1.5
2.0
2.5
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0

A.5. Anderson's Lilly: Single Variable

```
import BrewerPalettes

stream flowers(sepalL, petalW, sepalW,
                         petalL, species, obs)

layer FlowerPlot
guide
  trend from ID
  legend[X:20,Y:90] from FILL_COLOR
  axis[guideLabel: "Petal Length"] Linear from Y
  axis[guideLabel: "Petal Width"] Linear from X

from flowers
   ID: obs
   X:* Scale[0,100](petalL)
   Y:* Scale[0,100](petalW)
   FILL_COLOR: BrewerColors(species) -> SetAlpha(50,_)
   REGISTRATION: "CENTER"
   SHAPE: "CROSS"
```

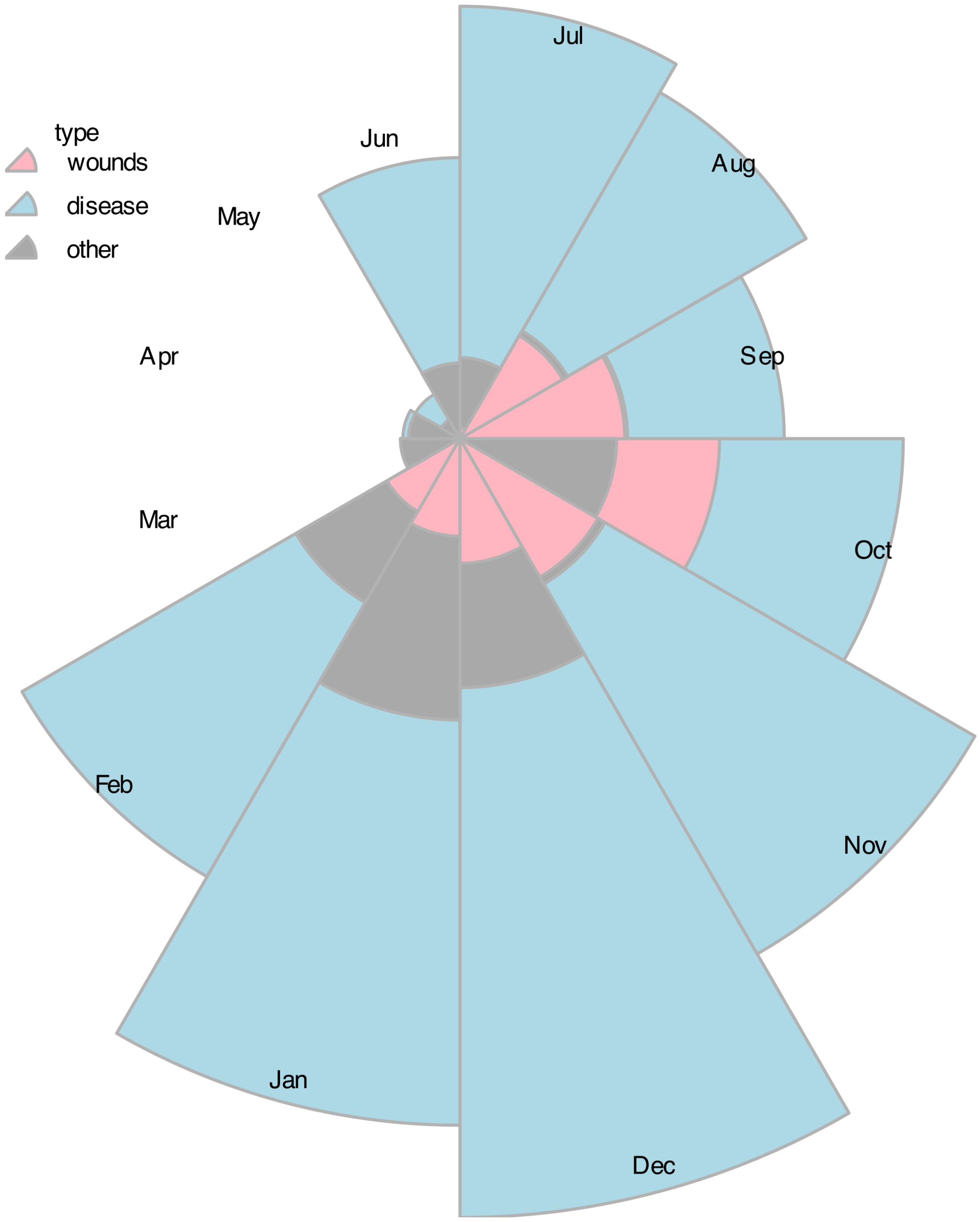
Anderson's Lilly data set [**2**, **56**] demonstrates the simplicity of constructing a basic scatter plot. The *Scale* operator projects input values onto the axes. Guide mark construction, including the trend-line, is entirely declarative and without postprocessing.


Wisconsin No. 38
Velvet
Trebi
Svansota
Peatland
No. 475
No. 462
No. 457
Manchuria
Glabron
Variety
0
10
20
30
40
50
60
70
80
90
100
Yield
year
1931
1932

A.6. Becker's Barley: Single Field

```
import BrewerPalettes
stream Barley(year, site, yield, variety)

const MAX: 100
const MIN: 10

layer BarleySite
guide
  axis[guideLabel: "Variety", X:0] from Y
  axis[guideLabel: "Yield", round: "T",
        seed.min: MIN, seed.max: MAX]
    Linear from X
  legend[X:75, Y: 60] Categorical from PEN_COLOR

from Barley
     filter(site =~ "University Farm")
     ID: Concatenate(variety,year)
     PEN_COLOR: BrewerColors["PuRd","BLACK"](year)
     X:* Scale[min: 0, max:100,
                inMin: MIN, inMax: MAX](yield)
     Y:* Rank(variety) -> Mult(7,_) -> Add(-5,_)
     REGISTRATION: "CENTER"
     FILL_COLOR: Color{CLEAR}
```

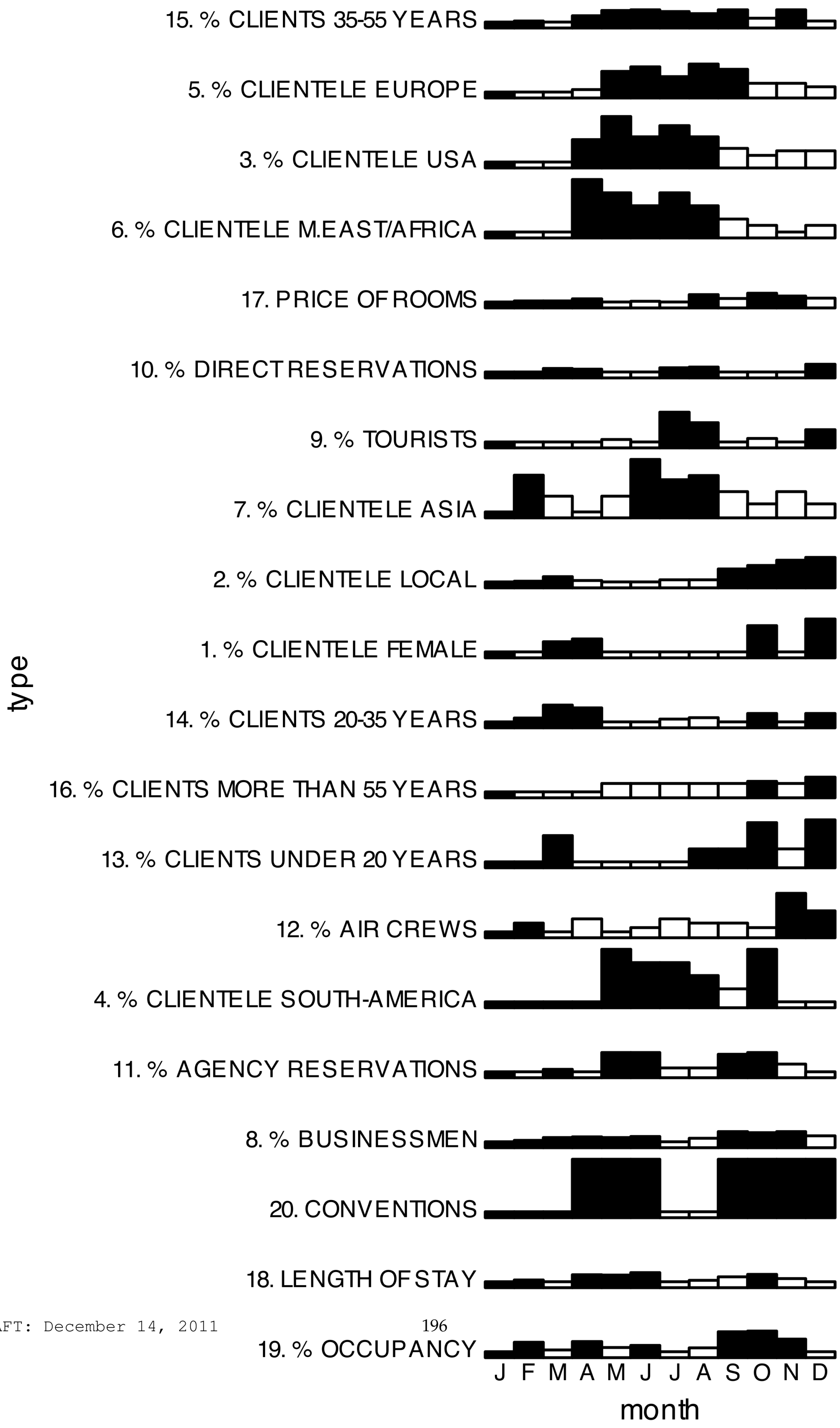
Single cell from the Becker's Barley trellis visualization [5, 57].


type
wounds
disease
other
Jul
Aug
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Jun

A.7. Causes of Mortality

```
import Dates
import Geometry

stream Deaths(date, type, count)

layer Rose["SLICE"]
guide
  legend[X: -75, Y: 50] from FILL_COLOR
    label.REGISTRATION: "BOTTOM_LEFT"
  pointLabels from ID
    TEXT: ParseLabel(ID)
    (X,Y): LabelLayout(ID, OUTER)
    REGISTRATION: "CENTER"
    FONT: Font{4}

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

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

operator MonthMin : Split[1](@FullMin["Integer"])

operator ParseLabel(id) -> (text)
   (id =~ ".*dise.*") => text: IndexOf(id,":") -> Add1(_)
                                   -> Substring(id, _, -1)
                                   -> Parse.reformat(_, "M", 'MMM')
   default => text: ""

operator Partition(n) -> (start, end)
   default => (start, end): Sub(12, n) -> [n] Sub(_,4) ->
                          [end] Mult(30, n) ->
                          Add1(n) -> Mult(30,_) -> (_,end)

operator LabelLayout(id, outer) -> (X,Y)
  (id =~ "") => (X,Y) : (0,0)
  default => (X,Y) :  [origin] Point(0,0)
                                                         -> ParseLabel(id)
                                                         -> MonthMin(_, 0)
                                                         -> Distance(origin.*, outer)
                                                         -> Max(50.0, _)
                                                         -> ProjectAlong(origin.*, outer, _)
```

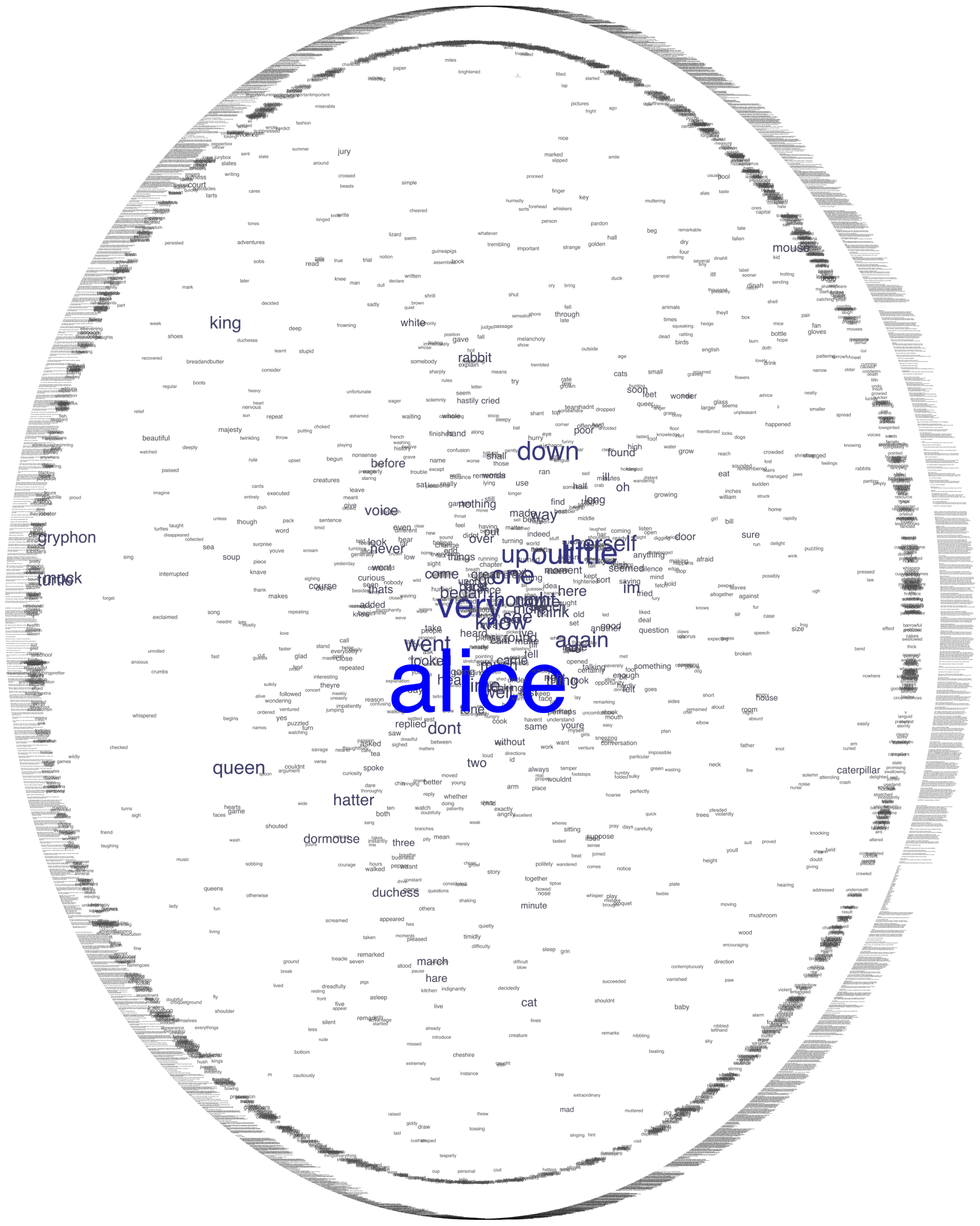
Re-creation of Florence Nightingale's casualty causes in the Crimean war visualization [**62**, **83**]. The Stencil code employs the python bridge and guide postprocessing to place labels according to either the outside edge of the wedges or a minimum radius (whichever is greater).


15. % CLIENTS 35-55 YEARS
5. % CLIENTELE EUROPE
3. % CLIENTELE USA
6. % CLIENTELE M.EAST/AFRICA
17. PRICE OF ROOMS
10. % DIRECT RESERVATIONS
9. % TOURISTS
7. % CLIENTELE ASIA
2. % CLIENTELE LOCAL
1. % CLIENTELE FEMALE
14. % CLIENTS 20-35 YEARS
16. % CLIENTS MORE THAN 55 YEARS
13. % CLIENTS UNDER 20 YEARS
12. % AIR CREWS
4. % CLIENTELE SOUTH-AMERICA
11. % AGENCY RESERVATIONS
8. % BUSINESSMEN
20. CONVENTIONS
18. LENGTH OF STAY
19. % OCCUPANCY
type
J F M A M J J A S O N D
month

A.8. Bertin's Hotel Stays

```
1  import Dates
2
3  stream HotelStays(count, type, month)
4
5  layer Chart
6  guide
7     axis[line.VISIBLE: false]  from X
8        label.TEXT : Substring(Input.0, 0,1)
9                 label.ROTATION: 0
10                label.REGISTRATION: "TOP"
11                tick.PEN_COLOR : Color{CLEAR}
12                tick.VISIBLE : false
13    axis[line.VISIBLE: false]  from Y
14       label.REGISTRATION: "BOTTOM_RIGHT"
15                tick.PEN_COLOR : Color{CLEAR}
16                tick.VISIBLE : false
17 from HotelStays
18    ID: AutoID(2) -> *
19    X: MonthNum(month) -> Mult(MonthNum,5) -> Add(_,60) -> MultiResult(Mult, Add) -> *
20    Y: Permute(type) -> Mult(12,_)
21    HEIGHT: Split[1](@Scale[min: 1, max:10], type, count)
22    FILL_COLOR:* Split[1](@Avg, type, count) -> FillBy(_, count)
23    PEN_COLOR: Color{BLACK}
24    PEN: Stroke{.5}
25    (SHAPE, REGISTRATION): ("RECTANGLE", "BOTTOM")
26
27 operator Avg :Range[ALL](@Mean)
28
29 operator MonthNum (mmm) -> (m)
30    default => m:Parse.reformat(mmm, 'MMM', 'M')
31
32 operator FillBy(mean, count) -> (c)
33    (mean > count) => c: Color{WHITE}
34    default        => c: Color{BLACK}
35
36 operator Permute(label) -> (index)
37    default => index: IndexOf(label, ".") -> Substring(label, 0,_)
38                   -> StaticList[vals: "ERROR,10,11,17,5,18,16,12,3,13,14,4,6,7,9,19,8,15,1,0,2"](_)
```

Jacques Bertin presented the above visualization to illustrate the power of small multiples. This visualization presents hotel occupancy metrics, curated into seasonal trend groups. Presenting each data point twice enables perception of trends that cross year boundaries. Filled bars indicate the above average time-periods, highlighting 'activity periods' (per-category calculations appear on lines 21 and 22). The visualization appears in Bertin's semiology [8]; the Protovis website provided the raw data [58].


alice
very
down
little
herself
know
went
again
queen
king
hatter
gryphon
turtle
mock
dormouse
duchess
rabbit
caterpillar
mouse
march
hare
cat

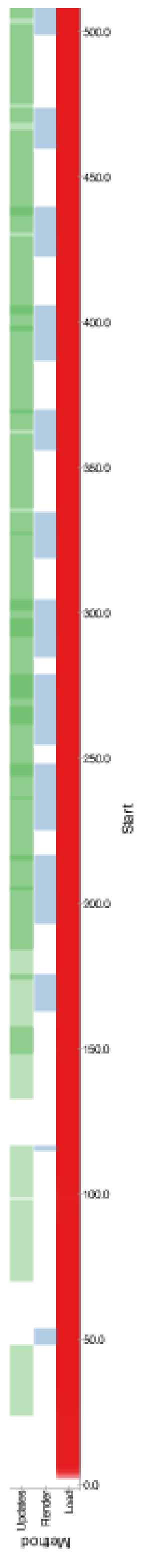
A.9. TextArc

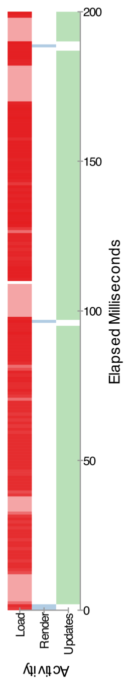
```
import Layouts
import Geometry : Geo

stream RawLines(text)

stream Lines (line, text)
from RawLines
    line: Count()
    text: text

stream RawWords (line, word)
from Lines
    line: line
    word: SplitOn(text, "\\s+") -> Map(@NormalizeWords, *) -> *

operator NormalizeWords(word) -> (word)
  default => word: Strip(word) -> ToLower(_)


stream Words (line, word)
from RawWords
    filter(prefilter.stopWord !~ "true", word !~ "")
    prefilter(stopWord) : StopWords(word)
    line: line
    word: word

stream PrimeCenter ()
from Words
 () : LineIndex.query(word) -> ExtendTuple(LineIndex.0, line) -> LineIndex(word, ExtendTuple.*)
 () : WordCount(word)

layer Border["TEXT"]
from Lines
    ID: line
    TEXT: text
    (X,Y):* Layout(line) -> Mult(-1, Layout.Y) -> (Layout.X,_)


layer Center["TEXT"]
from Words
    ID: word
    TEXT: word
    (X,Y):* Centroid(word) -> Contract(Centroid.X, Centroid.Y)
    REGISTRATION: "CENTER"
    (COLOR, FONT):*
         [count] WordCount.query(word) -> Range[ALL](@Max, _) ->
         [freq]  Divide(count, _) ->
         [color] HeatScale[hot: "BLUE", cold: "GRAY30"](freq) ->
                 Scale[min: 50, max:1000](freq) -> Font{{_}} -> (color, _)

operator WordCount : Count

operator LineIndex : Dict[fields: "lines"]

operator Centroid (word) -> (X,Y)
  default =>
    (X,Y): LineIndex.query(word) -> [Points] Map.query(LineIndex.0 , @Layout)
           -> Select[field: 0](Points.*) -> Values(*) -> [X] Mean(_)
           -> Select[field: 1](Points.*) -> Values(*) -> [Y] Mean(_)
           -> (X,Y)


/*TODO: Do two circular layouts of different radii and a linear interpolation between them as values move around the circ
```

```
operator Layout : CircularLayout[start: 0, size: 10, ratio: .75]
operator Contract : Geo::Scale[by: .94]
```

R. Bradford Paley's Text Arc visualization [**90**, **91**] is a data-based art piece and a forerunner of the word-cloud technique [**116**]. The basic schema places each line of a document in a circular layout and places each word of the text at the centroid of its appearance in the circle. Relative word occurrence count determines size and coloring. Common-word filtering was also applied [**18**].

The Stencil program is not a complete reproduction of the Text Arc schema. The original uses a landscape orientation, includes short lines pointing from high-frequency words to occurrences in the circular layout and graphic design differences. Deviations from the original are to enhance clarity for presentation in this format.

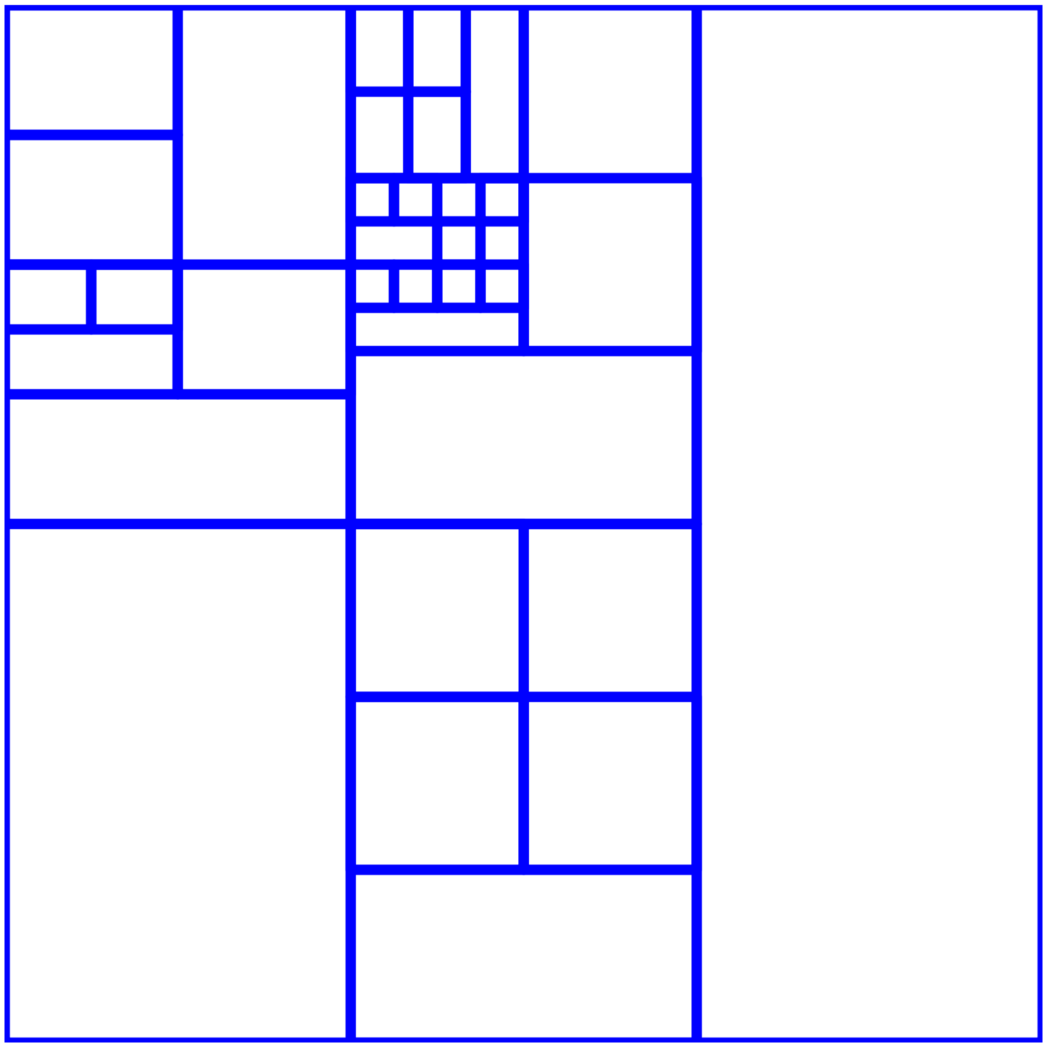

Method
Updates
Render
Load
Start
0.0
50.0
100.0
150.0
200.0
250.0
300.0
350.0
400.0
450.0
500.0


Activity
Load
Render
Updates
Elapsed Milliseconds
0
50
100
150
200

A.10. Thread Interleaving

```
import BrewerPalettes
import LongMath

const width : 8

stream MethodCalls(Method, Dir, Time)

stream Priors ()
from MethodCalls
   filter(Dir =~ "Entry")
   () : Entry.map(Method, Time)

stream MethodSpans (Method, Start, Stop)
from MethodCalls
   filter(Dir =~ "Return")
   local(Start) : Entry.query(Method)
   local(Stop) : Entry.query(Method) -> EnsureTime(_, Time)
   Method : Method
   Start: Range[ALL](@Min[c:"Long"], local.Start) -> Sub(local.Start, _)
   Stop: Range[ALL](@Min[c:"Long"], local.Stop) -> Sub(local.Stop, _)

operator EnsureTime (start, stop) -> (stop)
 prefilter(span) : Sub(stop, start)
 (prefilter.span =0) => stop : Add1(stop)
 default =>      stop : stop

operator Entry : Dict[fields: "start"]

layer Renders["LINE"]
guide
   axis[axisLabel: "Activity"] from Y1
   axis[sample: "linear", stride: 50, axisLabel: "Elapsed Milliseconds"] from X1
from MethodSpans
        ID: Count()
        PEN: Stroke{{width}::BUTT}
        PEN_COLOR: color(Method)
        X1: Start
        X2: Stop
        Y1:* Rank(Method) -> Mult(width,_) -> Mult(-1,_)
        Y2:* Rank(Method) -> Mult(width,_) -> Mult(-1,_)

operator color(m) -> (c)
  (m=~"U.*") => c: Color{GREEN,100}
  (m=~"R.*") => c: Color{BLUE,100}
  (m=~"L.*") => c: Color{RED,20}
```

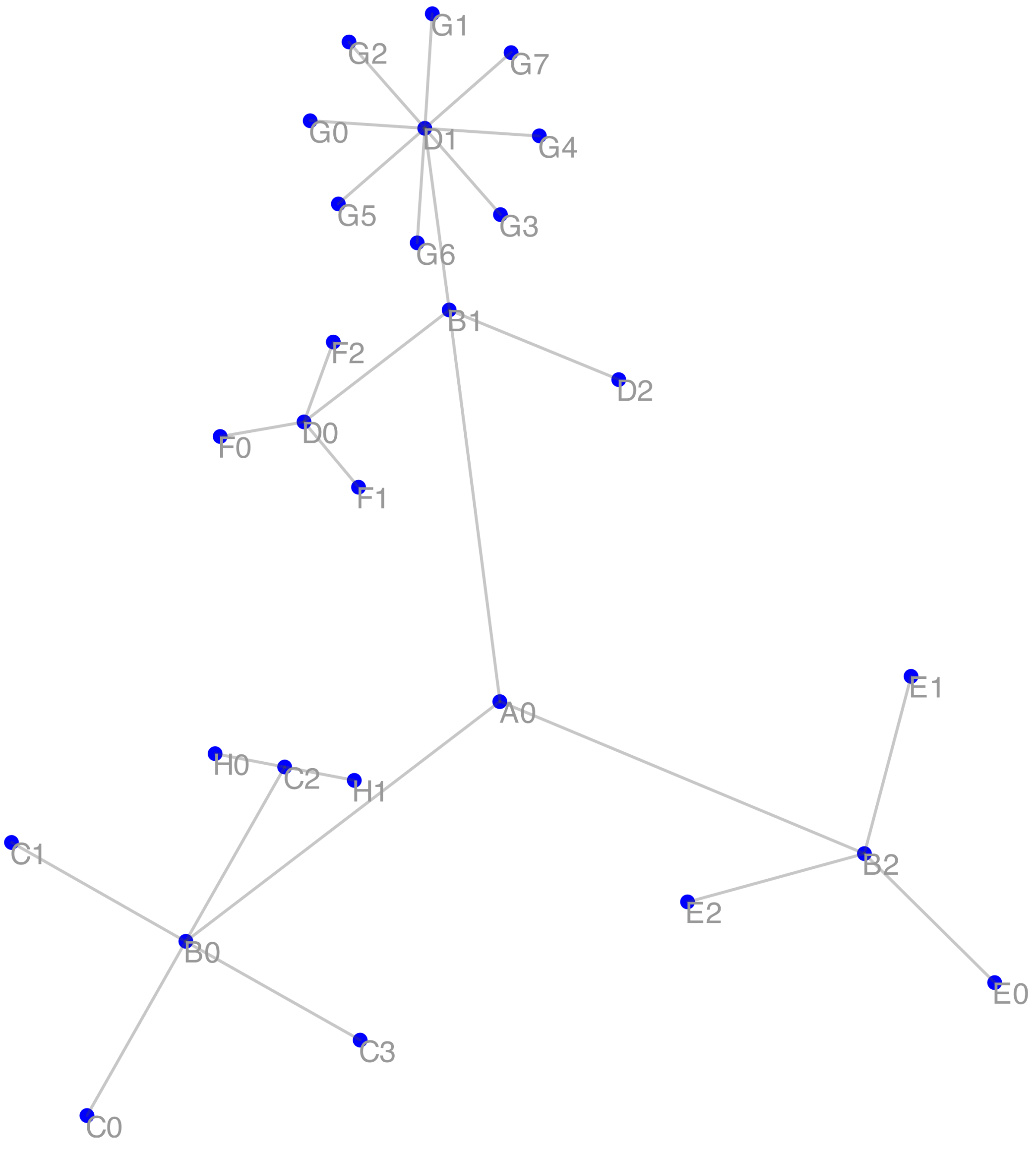
Thread interleaving diagram used in Chapter 9. This diagram indicates when portions of Stencil were concurrently scheduled, and thus potentially executing in parallel. The left data set displayed includes an error in activity scheduling: The prerender phase executes twice per rendering. The first prerender run is the intended one; the second appears as part of the render itself (notice overlap in the two lines). This double execution was due to a call to the prerenderer from the renderer that was required in an earlier version and not properly removed. A series of unexpected log entries originally indicated the presence of an error, but a correct diagnosis and resolution were a direct result of this diagram. After corrections were made, the execution trace presented on the right was produced.



A.11. Tree Map: University of Maryland Algorithms

```
import Layouts

stream VertexList (parent, child)

stream Prime ()
from VertexList
  () : Layout.add(child, parent, 1)

layer Nodes
from VertexList
   ID: child
   (X,Y,WIDTH,HEIGHT):* Layout.query(child)
   FILL_COLOR:  Color{CLEAR}
   PEN_COLOR: Color{BLUE}
   SHAPE : "RECTANGLE"

operator Layout : TreeMap
```

A tree map that employs the University of Maryland tree map implementations [**106**]. The tree-map algorithm itself was not modified from that supplied by the University of Maryland. It had to be wrapped to handle incremental updates to the incoming data. By not requiring algorithms to use particular data structures, it is much simpler to employ independently developed algorithms in analysis.


G1
G2
G7
G0
D1
G4
G5
G3
G6
B1
F2
D2
D0
F0
F1
E1
A0
H0
C2
H1
C1
B2
E2
B0
E0
C3
C0

A.12. Spring Force Embedding

```
import JUNG

stream VertexList (parent, child)

stream Prime ()
from VertexList
  () : Layout.add(parent,child)

layer Nodes
from VertexList
    ID: child
    REGISTRATION : "CENTER"
    FILL_COLOR:  Color{BLUE}
    (SHAPE, SIZE) : ("ELLIPSE", 5)
    (X,Y) :* Layout.query(child)

layer Edges["LINE"]
from VertexList
    ID: Concatenate(parent, child)
    (X1, Y1):* Layout.query(parent)
    (X2, Y2):* Layout.query(child)
    PEN_COLOR: Color{GRAY30,80}

layer Labels["TEXT"]
from VertexList
    ID : child
    (X,Y):* Layout.query(child)
    TEXT: child
    FONT: Font{10}
    COLOR: Color{GRAY60}

operator Layout : BalloonLayout
```

Spring-force embedding using the JUNG framework [**87**]. Eight JUNG layout algorithms are available in Stencil through the a wrapper module. The wrappers are little more than facets to maintain persistent views of the JUNG data structures.

B
Operator Metadata

Metadata are specified either as Java annotations or in the Java Properties XML format [88]. The majority of these pieces were discussed in the context of their use (see Chapter 7 and Chapter 9), but some of the metadata were not relevant to this dissertation. This chapter describes the full metadata available

B.1. Stencil metadata

Stencil has a number of configuration options, exposed in the Stencil.xml configuration file. The primary purpose of this configuration file is to indicate the location of modules, wrapper classes and adapters. Some performance tuning can be done with this metadata as well, though the effects are broad. Stencil system metadata are provided in the Java

Properties XML format [88]. Compound keys are actually pairs indicating both a type and an identifier (see listing below).

adapter:$<$ *name* $>$**:** Graphics adaptors are the underlying rendering system. They are based on the interfaces specified in the stencil.display and stencil.adaptor classes. These properties indicate which adaptors are present on the system; the value of the property indicates the path to the adaptor's implementation of stencil.adaptor.Adaptor.

module:$<$ *name* $>$**:** Modules are named collections of operators. The property value is a path to the class implementing the module. Each name must be unique and match the name given in the module metadata of the class indicated.

defaultModules: The value of this property is a comma-separated list indicating which modules will be imported by default. Modules listed here do not need to be explicitly imported by a Stencil program to have their operators exposed.

wrapper:$<$ *name* $>$**:** Wrappers provide a means to interact with Java types as Tuples, a process described more fully in Section 6.3. The name is used to allow multiple wrappers to be listed. The value of this property is a path to the wrapper class. Wrapper classes must implement stencil.types.TypeWrapper.

threadPoolSize:$<$ *pool* $>$**:** How many threads to create in the specified thread pool? This allows the balance between rendering, analysis and prerendering to be adjusted. If a nonpositive value is specified then the default pool size is used (Runtime.getRuntime().availableProcessors()/2+1).

B.2. Module and Operator metadata

Modules group operators, but provide some independent metadata. Module metadata are specified in Java annotations. The primary purpose of the module metadata is to indicate the name, location and key properties of operators. Module metadata are located by Stencil through the central Stencil metadata *module* tag described above.

name: Name of the module (optional, will default to the simple class name). The name must match the $< name >$ from the Stencil metadata. This is the name used to refer to the module in a Stencil program.

description: A textual overview of the module. This is only used as documentation.

Operator and facet metadata are specified by Java annotations. Operator metadata may be applied to either classes or static methods. Facet metadata may be applied to any method.

Operator metadata:

name: How the operator is referred to; defaults to the class's simple name.

specializer: The default specializer for this operator. During Stencil compilation, specializer values not provided in the Stencil program will be pulled from this default specializer.

Facet metadata:

memUse: This indicates how the facet interacts with memory, as described in Section 3.3.2.

prototype: The names and types of the return values of the facet. This is essentially a default value for the facet, as some operators change their return declaration based upon specializers or argument cardinality.

alias: A list of aliases (optional, defaults to the method's name). Alias is commonly used for operators that implement functions to have a single method serve as both the *map* and *query* facets.

Bibliography

[1] Greg Abram and Lloyd 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.

[2] Edgar Anderson. The irises of the Gaspé Peninsula. *Bulletin of the American Iris Society*, 59:2–5, 1935.

[3] Gregory R. Andrews. *Multithread, Parallel and Distributed Programming*. Addison-Wesley, Reading, MA, 2000.

[4] John Backus. Can programming be liberated from the von Neumann style?: A functional style and its algebra of program. In *ACM Turing award lectures*, pages 1977–. ACM, New York, NY, USA, 2007.

[5] Richard A. Becker, William S. Cleveland, and Ming-Jen Shyu. The visual design and control of trellis display. *Journal of Computational and Statistical Graphics*, 5:123–155, 1996.

[6] Benjamin B. Bederson, Jesse Grosjean, and John Meyer. Toolkit design for interactive structures and graphics. *IEEE Transactions on Software Engineering*, 30(8):535–546, 2004.

[7] Gérard Berry and Laurent Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, 1985. Springer-Verlag.

[8] Jean Bertin. *Semiology of Graphics*. Reprinted by University of Wisconsin Press, 1967.

[9] Jean Bertin. *Graphics and Graphic Information Processing*. Walter De Gruyter Inc., Berlin, 1981.

[10] Katy Börner, Chaomei Chen, and Kevin Boyack. Visualizing knowledge domains. In Blaise Cronin, editor, *Annual Review of Information Science & Technology*, volume 37, pages 179–255, Medford, NJ, 2003. American Society for Information Science and Technology.

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

[12] Jens Brandt and Klaus Schneider. Static data-flow analysis of synchronous programs. In *Proceedings of the 7th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE'09, pages 161–170, Piscataway, NJ, USA, 2009. IEEE Press.

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

[14] Adam L. Buchsbaum and Robert E. Tarjan. Confluently persistent deques via data-structural bootstrapping. *Journal of Algorithms*, 18:18–513, 1993.

[15] Stuart K. Card, Jock Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufman, 1999.

[16] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 278–292, New York, NY, USA, 1991. ACM.

[17] Stephen M. Casner. A task-analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics*, 10(2):111–151, 1991.

[18] Soumen Chakrabarti. *Mining the Web: Discovering Knowledge from HyperText Data*. Science & Technology Books, 2002.

[19] Michael Cheat, Miron Livny, and Raghu Ramakrishnan. Visual analysis of stream data. In *Proceedings of SPIE - The International Society for Optical Engineering*, volume 2410, pages 108–119, San Jose, CA, April 1995. SPIE.

[20] Ed H. Chi. *A Framework for Information Visualization Spreadsheets*. PhD thesis, University of Minnesota, 1999.

[21] Ed H. Chi. A taxonomy of visualization techniques using the data state reference model. *IEEE Symposium on Information Visualization (INFOVIS'07)*, 2000.

[22] Ed H. Chi. Expressiveness of the data flow and data state models in visualization systems. In *Advanced Visual Interfaces*, Trento, Italy, May 2002.

[23] Ed H. Chi. *A Framework for Visualizing Information (Human–Computer Interaction Series)*. Springer-Verlag, Secaucus, NJ, USA, 2002.

[24] Ed H. Chi, Joseph Konstan, Phillip Barry, and John Riedl. A spreadsheet approach to information visualization. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology (UIST'97)*, pages 79–80, New York, NY, USA, 1997. ACM Press.

[25] William S. Cleveland. *The Elements of Graphing Data*. Wadsworth Publishing Company, Belmont, CA, USA, 1985.

[26] William S. Cleveland. *Visualizing Data*. Hobart Press, Summit, NJ, 1993.

[27] Gregory Cooper and Shriram Krishnamurthi. FrTime: Functional reactive programming in PLT Scheme. Technical Report CS-03-20, Brown University, 2004.

[28] Harold Cooper. pcollections: A persistent java collections library. `http://pcollections.org`, July 2010.

[29] Joseph A. Cottam and Andrew Lumsdaine. Extended assortitivity and the structure in the open source development community. In *International Sunbelt Social Network Conference*. International Network for Social Network Analysis, January 2008.

[30] Joseph A. Cottam and Andrew Lumsdaine. Stencil: A conceptual model for representation and interaction. In *Proceedings of the 2008 12th International Conference on Information Visualisation*, pages 51–56, Washington, DC, USA, July 2008. IEEE Computer Society.

[31] Joseph A. Cottam and Andrew Lumsdaine. Algebraic guide generation. In *Proceedings of the 2009 13th International Conference on Information Visualisation*, pages 68–73, Washington, DC, USA, July 2009. IEEE Computer Society.

[32] Joseph A. Cottam and Andrew Lumsdaine. Automatic application of the data-state model in data-flow contexts. In *Proceedings of the 2010 14th International Conference on Information Visualisation*, Washington, DC, USA, July 2010. IEEE Computer Society.

[33] Joseph A. Cottam, Andrew Lumsdaine, and Joshua Hursey. Representing unit test data for large scale software development. *ACM Symposium on Software Visualization (SoftVis 2008)*, pages 57–66, September 2008.

[34] Antony Courtney. Frappé: Functional reactive programming in Java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pages 29–44, London, UK, 2001. Springer-Verlag.

[35] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. *Proceedings of the ACM SIGPLAN Workshop on Haskell - Haskell '03*, pages 7–18, 2003.

[36] Michael J. Crawley. *Statistics: An Introduction using R*. John Wiley & Sons, 2005.

[37] George S. Davidson, Bruce Hendrickson, David K. Johnson, Charles E. Meyers, and Brian N. Wylie. Knowledge mining with VxInsight: Discovery through interaction. *Journal of Intelligent Information Systems*, 11(3):259–285, 1998.

[38] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, January 2008.

[39] James R. Driscoll, N. Sarnak, Daniel D. K. Sleator, and Robert E. Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 109–121, New York, NY, USA, 1986. ACM.

[40] James R. Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41:943–959, September 1994.

[41] D. J. Duke, R. Borgo, M. Wallace, and C. Runciman. Huge data but small programs: Visualization design via multiple embedded DSLs. In *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages*, PADL '09, pages 31–45, Berlin, Heidelberg, 2009. Springer-Verlag.

[42] Conal Elliott. Simply efficient functional reactivity. Technical Report 2008-01, LambdaPix, April 2008.

[43] Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32:263–273, August 1997.

[44] Epic Team. Epic tool. Indiana University, 2009.

[45] Jean-Daniel Fekete. The InfoVis toolkit. In *Proceedings of the 10th IEEE Symposium on Information Visualization*, pages 167–174, Piscataway, NJ, 2004. IEEE Press.

[46] Jean-Daniel Fekete and Catherine Plaisant. Interactive information visualization of a million items. In *INFOVIS '02: Proceedings of the IEEE Symposium on Information Visualization (InfoVis'02)*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.

[47] Stephen Few. *Show Me the Numbers : Designing Tables and Graphs to Enlighten*. Analytics Press, 2004.

[48] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, Reading, MA, 2010.

[49] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Boston, MA, 2nd edition, 2001.

[50] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software–Practice and Experience*, 21:1129–1164, November 1991.

[51] Ben Fry. *Computational Information Design*. PhD thesis, Massachusetts Institute of Technology, 2005.

[52] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[53] Thierry Gautier, Paul Le Guernic, and Löic Besnard. SIGNAL: a declarative language for synchronous programming of real-time systems. In *Proceedings of a Conference on Functional Programming Languages and Computer Architecture*, pages 257–277, London, UK, 1987. Springer-Verlag.

[54] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[55] Alfons Geser and Sergei Gorlatch. Parallelizing functional programs by generalization. *Journal of Functional Programming*, 9:649–673, November 1999.

[56] Stanford Visualization Group. Protovis: Anderson's flowers example. `http://vis.stanford.edu/protovis/ex/flowers.html`, March 2010.

[57] Stanford Visualization Group. Protovis: Becker's barley example. `http://vis.stanford.edu/protovis/ex/barley.html`, March 2010.

[58] Stanford Visualization Group. Protovis: Bertin's hotel. `http://vis.stanford.edu/protovis/ex/hotel.html`, August 2010.

[59] Stanford Visualization Group. Protovis: Documentation. `http://vis.stanford.edu/protovis/docs/`, March 2010.

[60] Stanford Visualization Group. Protovis: Layout documentation. `http://vis.stanford.edu/protovis/docs/layout.html`, March 2010.

[61] Stanford Visualization Group. Protovis: Marks documentation. `http://vis.stanford.edu/protovis/docs/mark.html`, March 2010.

[62] Stanford Visualization Group. Protovis: Nightingale's rose example. `http://vis.stanford.edu/protovis/ex/crimea-rose.html`, March 2010.

[63] Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009.

[64] Pat Hanrahan. VizQL: A language for query, analysis and visualization. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 721–721, New York, NY, USA, 2006. ACM.

[65] M. Harrower and Cynthia A. Brewer. ColorBrewer.org: An online tool for selecting color schemes for maps. *The Cartographic Journal*, 40(1):27–37, 2003.

[66] Jeffrey Heer. Flare: Data visualization for the web. `http://flare.prefuse.org/`, July 2010.

[67] Jeffrey Heer and Maneesh Agrawala. Software design patterns for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):853–860, 2006.

[68] Jeffrey Heer and Michael Bostock. Declarative language design for interactive visualization. *IEEE Transactions on Visualization & Computer Graphics*, 2010.

[69] Jeffrey Heer, Stuart K. Card, and James 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.

[70] R. Hickey. The Clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*. ACM New York, NY, USA, 2008.

[71] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, New York, NY, USA, 2000.

[72] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[73] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[74] Kitware. *The Visualization Toolkit User's guide*. Kitware, 2003.

[75] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), September 1987.

[76] Miron Livny, Raghu Ramakrishnan, Kevin Beyer, Guangshun Chen, Donko Donjerkovic, Shilpa Lawande, Jussi Myllymaki, and Kent Wenger. Devise: Integrated querying and visual exploration of large datasets. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 301–312, New York, NY, USA, 1997. ACM Press.

[77] John Wills Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming (GULP-PROBE'94)*, pages 18–30, 1994.

[78] Martin Luboschik, Heidrun Schumann, and Hilko Cords. Particle-based labeling: Fast point-feature labeling without obscuring other visual features. *IEEE Transactions on Visualization and Computer Graphics (InfoVis'08)*, 14(6):1237–1244, November/December 2008.

[79] Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, 1986.

[80] Jock Mackinlay, Pat Hanrahan, and Chris Stolte. Show me: Automatic presentation for visual analysis. In *IEEE Symposium on Information Visualization (INFOVIS'07)*, 2007.

[81] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkley, CA, 1967. University of California Press.

[82] J. Paul Morrison. *Flow Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, 1994.

[83] Florence Nightingale. *A Contribution to the Sanitary History of the British Army During the Late War with Russia*. John W. Parker and Son, London, 1859.

[84] Chris North. *A User Interface for Coordinating Visualizations Based on Relational Schemata: Snap-Together Visualization*. PhD thesis, University of Maryland, College Park, May 2000. Chair: Ben Shneiderman.

[85] Chris North. Multiple views and tight coupling in visualization: A language, taxonomy and system. In *Workshop of Fundamental Issues in Visualization*, pages 626–632, Las Vegas, NV, USA, June 2001.

[86] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, Carnegie Mellon University, September 1996.

[87] Joshua O'Madadhain, Danyel Fisher, and Tom Nelson. JUNG homepage. `http://jung.sourceforge.net/`, March 2010.

[88] Oracle Corporation. Javatm 2 platform standard ed. 5.0 java 2 platform standard edition 6: Java XML property specification. `http://download.oracle.com/javase/1.5.0/docs/api/java/util/Properties.html`, July 2010.

[89] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976.

[90] W. Bradford Paley. Illuminated diagrams: Using light and print to comparative advantage (poster). In *INFOVIS '02: Proceedings of the IEEE Symposium on Information Visualization (InfoVis'02)*, October 2002.

[91] W. Bradford Paley. Textarc. `textarc.org`, July 2010.

[92] John Peterson, Valery Trifonov, and Andrei Serjantov. Parallel functional reactive programming. In *PADL: Practical Aspects of Declarative Languages (LNCS vol 1753)*, pages 16—-31. Springer, 2000.

[93] Harald Piringer, Christian Tominski, Philipp Muigg, and Wolfgang Berger. A multi-threading architecture to support interactive visual exploration. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1113–1120, 2009.

[94] Peter Piroli and Stuart K. Card. Information foraging. *Psychological Review*, 106(4):643–675, 1999.

[95] William M. Pottenger. The role of associativity and commutativity in the detection and transformation of loop-level parallelism. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 188–195, New York, NY, USA, 1998. ACM.

[96] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: Concepts and systems. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 4(2):63–71, 1996.

[97] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005.

[98] Casey Reas and Ben Fry. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press, September 2007.

[99] Steven F. Roth, Peter Lucas, Jeffrey A. Senn, Christina C. Gomberg, Michael B. Burks, Phillip J. Stroffolino, John A. Kolojechick, and Carolyn Dunmire. Visage: A user interface environment for exploring information. In *Proceedings of the 1996 IEEE Symposium on Information Visualization (INFOVIS '96)*, page 3, Washington, DC, USA, 1996. IEEE Computer Society.

[100] Steven. F. Roth and Joe Mattis. Data characterization for graphics presentation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*, 1990.

[101] Peter Von Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

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

[103] Will Schroder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics (3rd Edition)*. Kitware, USA, 2002.

[104] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, San Francisco, CA, USA, 2000.

[105] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[106] Ben Shneiderman and et al. University of Maryland: TreeMap implementations. `http://www.cs.umd.edu/hcil/treemap-history/`, May 2010.

[107] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1997.

[108] Tableau Software. Business dashboards. http://www.tableausoftware.com/business-dashboards, March 2008.

[109] Justin Talbot, Sharon Lin, and Pat Hanrahan. An extension of Wilkinson's algorithm for positioning tick labels on axes. *IEEE Transactions on Visualization & Computer Graphics (Proc. InfoVis)*, 2010.

[110] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[111] Edward Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, USA, 1983.

[112] Edward Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, USA, 1990.

[113] Edward Tufte. *Visual Explanations*. Graphics Press, Cheshire, CT, USA, 1997.

[114] Edward Tufte. *Beautiful Evidence*. Graphics Press, Cheshire, CT, USA, 2006.

[115] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. In *London Mathematical Society*, pages 230–265, 1936.

[116] Fernanda B. Viegas, Martin Wattenberg, and Jonathan Feinberg. Participatory visualization with Wordle. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1137–1144, 2009.

[117] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. *SIGPLAN Noticies*, 35:242–252, May 2000.

[118] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. *ACM SIGPLAN Notices*, 36(10):146, October 2001.

[119] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, pages 155–172, London, UK, 2002. Springer-Verlag.

[120] Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufman, San Francisco, 2000.

[121] Martin Wattenberg. Communication minded visualization: A call to action. *IBM Systems Journal*, 45(4):801–812, 2006.

[122] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2nd edition, 2009.

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

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

[125] Leland Wilkinson, Daniel J. Rope, Daniel B. Carr, and Mathew A. Rubin. The language of graphics. *Journal of Computational and Graphical Statistics*, 2001.

[126] Leland Wilkinson, Daniel J. Rope, Mathew A. Rubin, and Andrew Norton. nViZn: An algebra-based visualization system. *International Symposium on Smart Graphics*, 2001.

[127] Brian Wylie and Jeffrey Baumes. A unified toolkit for information and scientific visualization. In Katy Börner and Jinah Park, editors, *Visualization and Data Analysis*, volume 7243, page 72430H. SPIE, 2009.

[128] Ji Soo Yi, Youn ah Kang, John Stasko, and Julie Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, 2007.

[129] Dengping Zhu and Hongwei Xi. Safe programming with pointers through stateful views. In *PADL: Practical Aspects of Declarative Programming 2005*, pages 83–97, 2005.