Bluefish: A Grammar of Discrete Diagrams

by

Joshua Maxwell Pollock

B.S. Computer Science with College Honors
University of Washington (2020)

Submitted to the the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
the Department of Electrical Engineering and Computer Science
May 13, 2022

Certified by .....
Daniel N. Jackson
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Bluefish: A Grammar of Discrete Diagrams

by

Joshua Maxwell Pollock

Submitted to the the Department of Electrical Engineering and Computer Science
on May 13, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Discrete diagrams show collections of data objects and the discrete relationships between them, which include not just nominal and ordinal relations, but also more general ones such as the parent-child relation in a tree and the friend-friend relation in a social network.

Perceptual grouping principles – such as spatial proximity, nesting, and linking – are central to *reading* a discrete diagram. Yet, despite their importance, these principles are typically only implicit when *creating* one. As a result, a diagram author must either build their own abstractions on top of a grammar-of-graphics visualization toolkit or pre-commit to a structural representation supported by a domain-specific diagramming tool.

The key idea of this paper is that *perceptual groups visualize discrete relations.* We operationalize this insight in Bluefish, a grammar of discrete diagrams, in which diagram authors can use perceptual grouping principles to construct visual encodings of discrete relations.

A prototype of Bluefish has been implemented as a JavaScript library that can be embedded in an Observable notebook. We evaluate Bluefish by comparing it to a direct manipulation editor, a library inspired by the Grammar of Graphics, and a domain-specific diagramming tool.

More broadly, this explicit connection between discrete relations and perceptual groups provides insight into the effectiveness of statistical charts, suggests new interfaces for accessibility, and may prompt new ideas for visualization recommendation, analysis, and synthesis.

Thesis Supervisor: Daniel N. Jackson
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Thanks to my advisor, Daniel Jackson, for allowing me to pursue this project and for his invaluable feedback and guidance; thanks to Arvind Satyanarayan for his input on the paper story and insights about the VIS community; thanks to Geoffrey Litt for iterating with me on API ideas; thanks to Catherine Mei and Tom George for learning Bluefish and building many of the diagrams you see in this paper; thanks to Rob Miller for providing early design critique and sharing his copy of *Languages for Developing User Interfaces*; thanks to Alan Borning for providing early feedback on the Bluefish layout engine; and thanks to the Penrose team for helping me think about diagrams. Thanks to my parents and the UW folks who got me here. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1745302. This research was supported in part by the SaTC Program of the National Science Foundation.

Contents

List of Figures

Chapter 1

Introduction

Discrete diagrams show collections of data objects and the discrete relationships between them, including not just nominal and ordinal relations, but also more general ones such as the parent-child relation in a tree and the friend-friend relation in a social network. They are used in many disciplines and contexts: in management for org charts and business processes; in computer science for data structures and program states; in biology for cellular mechanisms and evolutionary trees.

People infer information from a discrete diagram using perceptual grouping principles [53]. These principles (see Figure 5-1) include attribute similarity (such as color and shape), spatial proximity, spatial alignment, common region (i.e. nested shapes), and element connectedness (i.e. links and arrows) [56]. In Figure 3-1, for example, the lines in the parse tree convey parent-child relationships, and spatial proximity and alignment in the bar chart convey the ordinal relationship between bars.

Direct manipulation editors – such as Adobe Illustrator, Figma, and PowerPoint – allow a user to align and distribute objects as well as construct nested groups. Aside from lines and arrows in tools like Excalidraw and OmniGraffle, most perceptual groups in direct manipulation editors are ephemeral. For small diagrams, this ephemerality allows for rapid creation and customization, because an author can move any object at will. However, a small semantic edit, such as aligning the leaves of a tree, may require a user to manipulate nearly every object in a diagram to preserve its relational structure. The cost of such an edit grows with the size of the diagram.

More seriously, direct manipulation editors (because they are not data-driven) make it hard to generate multiple instances of the same custom diagram. To create a diagram family for a specific problem – say visualizing the stack and heap as a program executes or explaining a parsing algorithm – a user must build each instance by hand. When the input data changes, a user must manually propagate that change through the diagram; and when the perceptual groups change (transposing a list from horizontal to vertical, for example) a user must painstakingly replicate those changes across *all* of their instances.

In contrast, statistical visualization grammars, such as the Grammar of Graphics [58], separate data from the visual encoding, which makes it easy to generate multiple instances of a visualization. They also exploit some perceptual grouping principles. Similarity can be expressed through mark channels, and proximity and alignment indirectly through ordinal scales. Other principles, however, are not so readily expressed. To encode a relationship with containment or connectedness, for example, the user must typically either rely on a specialized layout (such as "tidy tree" [14, 44]), or must construct new marks whose positions are carefully calculated to produce the desired visual grouping. Difficulty in expressing containment and connectedness is particularly troubling for a discrete diagram author, because experimental studies suggest these very grouping principles are the *most* effective [60, 41].

To address the shortcomings of both direct manipulation interfaces and statistical visualization grammars, researchers have developed *domain-specific* diagramming tools and diagram template libraries such as SetCoLa [31], GoTree [37], Graphviz [21], and MermaidJS. These authors have generally converged around support for certain kinds of perceptual groups, such as links and containment, and make it easy for an author to create diagrams visualizing specific structures like trees or node-link graphs. The downside of this domain-specificity is that it requires an author to *pre-commit* to the relational structure of the diagram they want to make, even though an author may not know this structure ahead of time. For example, a diagram author might start with a conventional tree layout (Figure 3-1), but then align and box the leaves to represent the fact that the leaves form a sentence. Similarly, an author may start

with a grouped bar chart and add an annotation that relates a bar to a text label via an arrow mark.

This thesis presents Bluefish, a grammar of discrete diagrams. The key idea in Bluefish is that *perceptual groups visualize data relations*. We operationalize this in two ways. First, we add perceptual groups to the visual encoding language. They form the compositional building blocks of layout in our grammar, allowing diagram authors to construct lists, trees, tables, and graphs from the same core set of spatial relationships. Second, we add discrete relations to the data language. Just as perceptual groups add flexibility to visual encodings, so too discrete relations add flexibility to input structures. Discrete relations may overlap, such as the leaf-list relation and the parent-child relation of the tree in Figure 3-1, and this allows a diagram author to express a wide range of structures as combinations of discrete relations.

In the remainder of this thesis we will identify common features in the layout engines of several domain-specific diagramming tools. We argue these commonalities stem from an implicit, and sometimes explicit, use of perceptual groups. We also discuss tools with similar properties to our grammar, but that have only a nascent notion of perceptual grouping (Chapter 2). Next, to give the reader an intuition for how Bluefish works, we provide three vignettes of representative Bluefish users (Chapter 3) whose diagrams are outlined in Figure 3-1. We then abstract these ideas to identify the essence of Bluefish (Chapter 4).

In Chapter 6 and Chapter 7, we discuss details of our prototype JavaScript implementation of the Bluefish grammar. The JavaScript implementation represents a Bluefish diagram using a *relational scene graph*, which adds *references* to a typical tree hierarchy.

To evaluate Bluefish, we conducted three Cognitive Dimensions of Notations case studies to compare Bluefish against the three categories of systems we introduced above. In our evaluation we find (i) Bluefish supports authoring multiple diagram instances better than Figma, a direct manipulation editor (Section 8.1); (ii) Bluefish introduces abstractions that are latent in diagrams built on D3 [12], an expressive, low-level framework inspired by the Grammar of Graphics (Section 8.2); and (iii)

Bluefish is more expressive than GoTree [37], a domain-specific tool for trees, while providing a similar level of abstraction (Section 8.3).

Finally, we discuss broader implications of making perceptual groups explicit (Chapter 9). A grammar with explicit data relations and perceptual groups captures more information about the semantics of a visualization than one with only data objects and encoding channels. As a result, our work contributes to ongoing discussions of Cleveland and McGill's effectiveness criteria for ranking visualizations (and their limitations) [16, 8]. It also suggests new ideas for visualization accessibility, recommendation, analysis, and synthesis.

Every figure in this thesis was produced by a Bluefish program (unless otherwise noted).

Chapter 2

Related Work

2.1 Constraint-Based Layout in Visualization

Several existing data visualization tools use constraint-based layout to represent perceptual groups.

Scout is a a tool for building user interfaces [51]. It allows a user to specify constraints between UI elements, such as vertical, horizontal, row, and column spacial layout, as well as relative sizes. Scout synthesizes novel layouts and ranks them according to aesthetic criteria like element alignment and overall balance. Bluefish does not include aesthetic criteria, but does encode similar layout constraints explicitly. Unlike Scout, alignment constraints are not a effectiveness heuristics, but rather part of the encoding.

SetCoLa is a tool for laying out graphs [31]. Its constraints include alignment, relative proximity, and ordering. These constraints are also supported by Bluefish. Additionaly, SetCoLa can represent nodes laid out in circles, clusters, and hulls.

Charticulator is a direct manipulation editor for statistical charts [45]. The authors use linear constraints to implement high-level layout primitives should as scaffolds, which evenly space elements in the vertical or horizontal directions. Bluefish uses a similar constraint engine and encodes bar charts in a similar way to Charticulator; however, in contrast to Charticulator, Bluefish exposes this constraint system through a programmatic API.

GoTree is a visualization grammar for trees [37]. GoTree allows users to specify links between parents and children, alignment between neighboring subtrees, and constraints between a parent and its subtrees including nesting and proximity. Bluefish can encode similar constraints as GoTree and supports a more expressive collection of input formats.

The tools we have listed largely focus on constraints for layout, but the ubiquity of constraints across many visual domains inspired us to question their semantic role in diagrams. This led to our realization that constraints represent spatial relations (Chapter 5).

2.2 UI Frameworks

Web Frameworks. React and similar web frameworks allow developers to specify UIs declaratively. Such frameworks generally adhere to versions of model-view-controller architectures that separate data (model) from its visual encoding (view). React components, for example, are written as (internally stateful) functions of type `(data) => DOM` or as objects. This separation of concerns makes components easy to compose. Most web frameworks provide fairly straightforward layout engines that render a DOM tree in a top-down fashion.

App Frameworks. Both SwiftUI [3], Apple's app development framework, and Flutter [27], Google's SDK, operate similarly to web frameworks. However, their layout engines are more complex. This may be due to the existence of CSS on the web, which takes care of complex adaptive layout logic that app frameworks choose to incorporate into their APIs instead.

These app frameworks execute layout in a two-pass fashion, which proceeds in the following way: (i) a parent determines allowable intervals for the widths and heights of its children, (ii) it passes the corresponding information to each child, which computes its own size (possibly after calling *its* children) and returns that size to the parent, (iii) the parent determines the position of its children. This layout algorithm maintains the linear time complexity of a top-down layout engine, but

allows for more flexible layouts. Moreover, since the framework has more knowledge about the sizes of components, it can perform more aggressive optimizations when updating the render tree. For example, if the layout engine determines that a child's size has not (or in fact could never) change, then it will not have to recompute the layout in that child's parent.

Comparison to Bluefish. Bluefish's two key concepts, constraints and references, allow it to express layouts in ways that are more difficult in current UI frameworks. For example, Bluefish allows a user to specify constraints among pairs of neighboring elements in a list. These constraints are then solved jointly to achieve a global layout. In contrast, a user must employ an `HStack` or `Row`, for example, if they want to produce a horizontal layout of a list in SwiftUI or Flutter, respectively. This tightly couples constraints to components, and since components must appear in a tree hierarchy, they become harder to compose than Bluefish constraints.

References are supported by UI frameworks, but they tend to be difficult to use in a data-driven way. React [22] and SwiftUI, for example, both allow users to create references to *components*, and these components may be driven by data; however, it is not as easy to create references to *data*. The complexity of working with references in existing frameworks may result from the fact that they optimize for the developer experience of working with tree hierarchies. Since references break this tree assumption, they introduce additional complexity.

2.3 Diagramming Tools

Penrose. Bluefish was heavily influenced by Penrose, and share its motivation of making diagramming easier and more accessible [59]. Bluefish and Penrose diverge primarily in which diagrams they emphasize support for. Penrose's main domain of interest is mathematical diagrams in Euclidean and differential geometry as well as point-set topology. Exemplar Penrose diagrams include the Pythagorean theorem, meshes, and sets. Bluefish is concerned with discrete diagrams containing data and a sizable collection of discrete relationships.

Bluefish is an embedded language while Penrose stands alone. This difference stems from Bluefish's goal to integrate with existing workflows where data is available in an Observable notebook (e.g.). On the other hand, Penrose targets users familiar with standalone tools like Tikz and Illustrator for making mathematical diagrams.

Delaunay. Delaunay and Bluefish share much in common. Both use references to allow data to participate in multiple relations, and both use constraints to specify layout. Unlike Delaunay, Bluefish makes perceptual grouping an explicit language feature. On the other hand, Delaunay treats constraints as a visual implementation detail, similar to constraint-based layout engines. As a result, Bluefish affords design space exploration more than Delaunay.

Basalt. Basalt [4] is a prototype constraint-based diagramming language implemented in Python. The abstractions of Basalt helped inspire our work. Compared to Basalt, Bluefish more explicitly separates input data from the visual encoding, making edits to data and layout easier and more separable.

2.4 Data Visualization Tools

In this section we cover several existing data visualization tools and their capabilities for growth.

D3. D3 is famous for example-driven development and sharing. More recently, several examples have been ported to stand-alone functions like stacked bar charts. The selection model is imperative, relying on mutating the DOM through JS variables. This hinders compositionality in several ways. Firstly, to convert a snippet of D3 code to a function, it will end up with the signature `(DOM) => void` or possibly `(data, DOM) => void`. Such functions are not readily composable. The DOM argument is modified and not returned. This could be fixed by creating a convention of always returning the input DOM, to yield e.g. `(data, DOM) => DOM`, but this pattern is not enforced or afforded by the framework, and the DOM must still be threaded through a program imperatively, building up one piece at a time, rather than declaratively, specifying several pieces at once.

ggplot2. Ggplot2 extends the Wilkinson's Grammar of Graphics [58] with the concept of a *layer* [57]. Users can layer parts of a statistical chart together. For example, a user can layer marginal histograms on top of a scatterplot. Though restricted to a linear chain of layers, this form of composition has proven immensely powerful and ggplot2 has a vibrant ecosystem of third-party packages for changing charts styles, adding new layouts, and modifying positions of plot elements.

Vega/Vega-Lite. Vega [47] and Vega-Lite [46] are visualization languages inspired by the Grammar of Graphics. Unlike frameworks such as D3 and ggplot2, which are embedded in host languages, Vega and Vega-Lite have been designed as stand-alone languages. Because of this, these languages are able to specify their own runtime semantics based on functional reactive programming, which makes interactive charts much easier to specify than in existing embedded systems. On the other hand, Vega and Vega-Lite have little support for growing the set of language primitives from within the language itself. To properly add a new data transform to Vega, for example, a user must learn Vega's dataflow API, extend the vega-dataflow's `transforms` export, and build Vega from source.

Chapter 3

Demo

We now present a short demo of Bluefish's expressive capabilities via three intertwined vignettes illustrated in Figure 3-1. These scenarios span visual domains: the first is a text layout, the second is a complex parse tree, the third is a grouped bar chart. Though these examples are far-reaching, they are unified by two things: they are all discrete diagrams, and the diagram authors wish to accomplish similar tasks. Bluefish provides declarative, yet granular and expressive, abstractions for discrete diagramming that support these use cases.

3.1 Adjust spatial relations

Jacques designing recipe flash cards for his new cookbook. He is looking at an omelet recipe with the following data:

```
const omeletRecipe = {
  name: 'Omelet',
  ingredients: createListD(['3 large eggs', ...]),
  cookware: createListD(['mixing bowl', 'skillet']),
  steps: createListD([
    `Whisk together eggs and salt & pepper
in a mixing bowl.`,
    ...]),
```

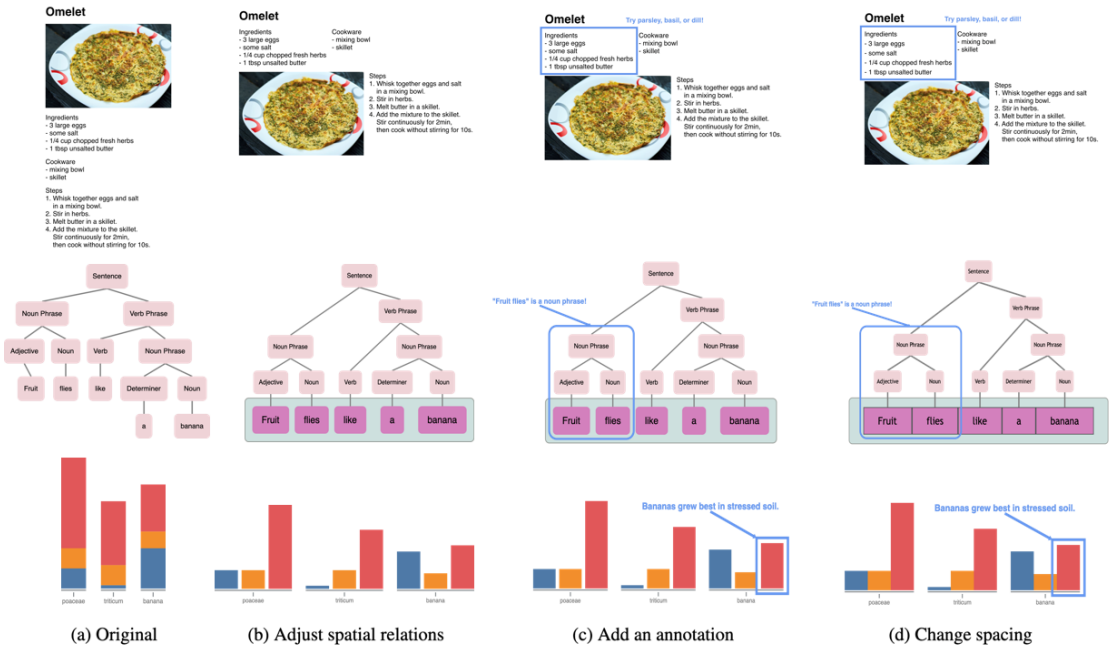


Figure 3-1: Bluefish makes the mapping between data relations and perceptual groups explicit. This allows many relational edits to become easier and more uniform across domains. This figure shows parallel tasks across recipe, parse tree, and bar chart diagrams. All of these are accomplished with the same small collection of Bluefish primitives.

```
}
```

`createListD` is a Bluefish function that creates a list from a JavaScript array of entries. `D` in the function name means that this function creates *data*. He is currently using the following Bluefish visual encoding:

```
const recipeT = createGroup({
  $name$: (contents) =>
    text({ contents, fontSize: '36px' }),
  $ingredients$: textListT({
    title: 'Ingredients',
    style: 'tick',
    fontSize: '18px'
  }),
  $cookware$: textListT({...}),
  $steps$: textListT({...}),
},{
  'name->ingredients': [
    C.vSpace(15), C.alignLeft],
  'ingredients->cookware': [
    C.vSpace(15), C.alignLeft],
  'cookware->steps': [
    C.vSpace(15), C.alignLeft],
})
```

The shape contains four *shape templates* — `$name$`, `$ingredients$`, `$cookware$`, and `$steps$`, which take his input data and convert them to Bluefish shapes using a `text` mark and three `textListTs`. By convention, a shape template variable name ends with `T`. Jacques also specifies *spatial relations* between his three shapes. The ingredients shape is 15 pixels above the cookware shape, and they are left-aligned. Similarly, the cookware shape is 15 pixels above the steps, and they are left-aligned as well. Together, these shapes and the spatial relations between them comprise a Bluefish *perceptual group*. The `createGroup` function reads the shape

template names a user has specified and uses these names to construct a function that consumes a JavaScript object with the same fields and returns a Bluefish group shape. If the user doesn't specify any templates, `createGroup` just constructs a Bluefish group shape. To render the shape, Jacques calls `render(omeletRecipe, recipeT)`.

The omelet recipe is just one of many that will appear Jacques's flash cards. As he traverses through several instances by changing the input recipe data, Jacques decides to align the ingredients and cookware lists, so a reader can tell at a glance if they have everything they need. He modifies `recipeT`'s spatial relations as follows:

```
{
  'name->ingredients': [C.vSpace(15), C.alignLeft],
  'ingredients->cookware': [C.hSpace(15), C.alignTop],
  'ingredients->steps': [C.vSpace(15), C.alignLeft],
}
```

* * *

Barbara is visualizing a natural-language parse tree to understand an NLP riddle. She'd heard the other day that "Time flies like an arrow, but fruit flies like a banana." To understand how fruit can fly, Barbara uses a parse tree template she found online to visualize the parse tree of "Fruit flies like a banana." When she first creates the visualization the words in the sentence, which appear at the leaves of the tree, are hard to visually distinguish. Even after she changes their color, she finds it difficult to understand the parsing structure. She modifies the template to align the leaves and surround them by a border.

* * *

Pat wants to understand how crops grow in different soil types. He has already pre-processed the data from his experiments into the shape he wants in order to build Bluefish visualizations:

```
const cropData = createListD([
  {
```

```
    plant: 'poaceae',
    soils: createListD([
      { name: 'Nitrogen', value: 6 },
      { name: 'normal',   value: 6 },
      { name: 'stress',   value: 27 },
    ]),
  },
  ...
])
```

He first visualizes this data as a stacked bar chart and identifies both Nitrogen-rich and stressed soils seem to produce a lot of bananas. Here's the code for a stacked bar:

```
const stackedBarT = createListT(
  ({ name, value }) => M.rect({
    width: 50,
    height: yScale(+value),
    fill: colorScale(name)
  }),
  { 'next->curr': [C.vSpace(0), C.alignCenter] },
)
```

(The Bluefish code for the axes and list of bars is straightforward and not shown.) This code uses the `createListT` Bluefish function[1], which takes a shape template for each element of the list and a set of spatial relations for neighboring elements of the list (identified by `curr` and `next`). It returns a shape template that takes a list as input. Behind the scenes, `createListD` constructed a *data relation* between neighboring elements in the list. `createListT` gives Bluefish users an easy way to visualize a list and the relation between its neighbors. Neither `createListD` nor `createListT` are built into Bluefish. Any Bluefish user could write or modify these functions. They simply wrap common patterns for list data and list shape templates.

[1] A function that inputs and outputs a template is also called a *template combinator*.

Pat realizes the values for the "Nitrogen" and "stress" conditions are close together, and he'd like to be able to show at a glance which condition yielded more bananas. He therefore decides to switched to a grouped bar chart.

The constraints in `stackedBarT` currently space the bars vertically and center-align them. Pat modifies them so the bars are spaced horizontally and lie on a common baseline:

```
{ 'next->curr': [C.hSpace(3), C.alignBottom] }
```

Bluefish's perceptual groups and their spatial relations provide a *consistent* interface across many diagram domains. Similar semantic tasks, such as modifying spacing and alignment, have similar representations.

3.2 Add an annotation

Jacques likes his current recipe layout. But he wants to add an annotation to the omelet recipe about some of the fresh herbs a reader might want to try. He wants a general way to apply an annotation to any recipe. To add an annotation, Jacques first augments his data with the information for the annotation:

```
const annotatedOmeletRecipe = {
  recipe: omeletRecipe,
  annotation: {
    location: ref(
      '../recipe/ingredients/elements[2]'),
    label: 'e.g., parsley, chives, dill',
  }
}
```

Jacques's data now includes an annotation object, which has a *reference* to the fresh herbs entry in the omelet's ingredients list and a label containing some herbs to try.

Jacques then creates a new shape template for this annotation:

```
const annotationT = createGroup({
  $location$: 'ref',
  highlight: M.ellipse({
    fill: 'none',
    stroke: 'cornflowerblue',
    strokeWidth: '5px',
  }),
  arrow: M.arrow({ fill: 'cornflowerblue', }),
  $label$: (contents) => M.text({contents, ...})
},{
    'highlight->location': C.containsPadding(
      {top: 10, bottom: 10, left: 10, right: 10})
    'highlight->arrow': [
      C.makeEqual('left', 'right'),
      C.makeEqual('top', 'bottom')],
    'arrow->label': [
      C.makeEqual('left', 'centerX'),
      C.makeEqual('top', 'bottom')],
    'label->highlight': [
      C.hSpace(-50), C.vSpace(50)],
    'highlight->arrow': [
      C.makeEqual('right', 'left'),
      C.makeEqual('centerY', 'top')],
    'arrow->label': [
      C.makeEqual('right', 'centerX'),
      C.makeEqual('bottom', 'bottom')],
    'highlight->label': [C.hSpace(0)],
    'label->highlight': [C.vSpace(10)],
})
```

`annotationT` has two shape templates, `location` and `label`, and two shapes, `highlight` and `arrow`. `location` is a *reference template* that corresponds to

a data reference in the input. `highlight` surrounds the `location` with some padding, and the arrow points from `highlight` to `label`. There are also several spatial relations between these shapes.[2]

Jacques then combines his recipe and annotation templates together in a composite `annotatedRecipeT` template:

```
const annotatedRecipeT = createGroup({
  $recipe$: recipeT,
  $annotation$: annotationT,
})
```

Realizing he has made a reusable shape template for annotations, Jacques shares `annotationT` online to share it with others.

* * *

By looking at her new parse tree visualization, Barbara has a key insight into the fruit fly sentence. The "fruit" isn't flying. "Fruit flies" is a noun phrase! To make her insight clear and share it with others, Barbara decides to add an annotation. She searches for existing annotation templates online, and stumbles upon Jacques's `annotateT`. Because `annotateT` requires only a reference to another part of a Bluefish data structure and text for a label, Barbara doesn't need to change Jacques's template to use it in her program. Like Jacques, she creates a new annotated version of her data and her shape.

* * *

Using the grouped bar chart, Pat is able to see that the stressed condition yields more bananas. "Maybe it's all the fruit flies..." he wonders. To share his insight with his colleagues, he decides to annotate his chart. While searching for stressed bananas, he stumbled across Barbara's use of `annotateT` in her Bluefish diagram, and he decides to use it in his, too. Pat reads Jacques's original `annotateT` code, and finds the `ellipse` mark called `highlight`. He replaces it with a `rect` mark.

[2]Notice that by treating an arrow with the same bounding box abstraction as everything else, the code becomes a bit messier. One solution to this may be to add a separate line abstraction similar to Delaunay's [19].

This change works without any modifications to the other shapes or spatial relations thanks to Bluefish's unified bounding box (bbox) model for representing shapes such as ellipses and rectangles.

Bluefish's *references* allow a user to split a complicated diagram into smaller, compositional shapes and shape templates. This allows a user to develop a Bluefish diagram piece by piece, and these pieces can be shared with other Bluefish users.

3.3 Change spacing

After adding their annotations, Jacques, Barbara, and Pat all decide they want to change the spacing in their diagrams, since it's hard to tell some elements apart. These are easy to change. Pat, for example, changes `hSpace(3)` to `hSpace(10)`. Because their annotations have been defined *relationally*, the highlights, arrows, and labels move along with the data they point to. This means changing spacing only requires modifying the relevant constraint. Bluefish automatically propagates constraint information to connected elements, so the user doesn't have to do this manually.

3.4 Summary

Bluefish's design provides several concrete benefits to these diagramming workflows. First, Bluefish provides a *consistent* interface for specifying spatial relations, whether those appear in a recipe diagram, a tree, or a bar chart. Second, Bluefish allows a user to write *reusable relational components*, such as annotations that take references to other pieces of data as input. These components can be reused across different diagram types. Finally, because positions and sizes in Bluefish can be specified relative to each other, Bluefish makes small semantics edits, such as changing the spacing between groups of bars, small edits to a Bluefish specification. Instead of requiring a user to manually edit the sizes and positions of many parts of their diagram, Bluefish handles constraint propagation automatically.

Chapter 4

The Essence of Bluefish

We have seen some vignettes of Bluefish in action, but only hinted at the full generality of its design. In this chapter, we outline the principal elements of Bluefish, and how diagrams are constructed from them.

An abstract syntax for Bluefish diagrams viewed as a grammar is shown in Figure 4-1. A Bluefish diagram is composed of shapes; each shape is either a mark (a primitive shape, such as a rectangle or a text label), or a perceptual group. A perceptual group is a composite shape – that is, a set of subshapes – along with a set of spatial relations that constrain pairs of these subshapes. To bind relations to their respective shapes, we have introduced shape names, so that a group comprises a mapping from names to shapes and a mapping from name pairs to spatial relations. For example, a group may contain a rectangle named A and an ellipse named B, along with a spatial relation that maps (A, B) to an alignment relation. A shape may belong to multiple perceptual groups, which allows Bluefish to represent overlapping spatial relations.

The language of marks is extensible; currently, marks include: The SVG primitives `rect`, `ellipse`, `circle`, `text`, `line`, `image`. Specially constructed marks: `arrow`, `nil` (has 0 width and height and doesn't render), `html` (embeds an arbitrary `JSX.Element` with statically explicitly provide width and height.)

For the purposes of spatial relations, Bluefish abstracts a shape as an *axis-aligned bounding box*. The dimensions of the bounding box are constrained such that Width =

Right − Left, Height = Bottom − Top, CenterX = 1/2(Left + Right), and CenterY = 1/2(Top+Bottom). By default, a perceptual group's bounding box is the smallest axis-aligned bounding box that contains its subshapes; however, this can be overridden by a user (Chapter 6).

The spatial relation definition corresponds to firstDim⟨op⟩secondDim+spacing(strength), which is a linear constraint with a strength. The strength roughly corresponds to the priority in which relations are satisfied. Required relations must be satisfied to render a diagram, but other relations can be thought of as being executed in order, starting with the largest strength first. This general formulation allows us to specify three common classes of spatial relation: `space`, where `firstDim` and `secondDim` are opposite sides of the bounding box, `align`, where `firstDim === secondDim`, and `contains`. `contains` is a composition of required alignment inequalities, which specify that the container's bounding box lie outside the containee, and strong alignment equalities, which encourage the container's edges to align exactly with those of the containees. Together, these constraints approximate `min` and `max` constraints using only linear constraints with strengths. We elaborate on the use of these constraints to express perceptual grouping principles in Chapter 5.

To render a Bluefish diagram, a layout must be found that respects both the basic properties of the marks (eg, the dimensions of each rectangle) and the constraints of the perceptual groups. A diagram therefore has potentially any number of renderings if the constraints are underspecified. In practice, a diagram designer generally includes sufficient spatial constraints so that there is exactly one rendering, but it is possible for a diagram to have multiple renderings (should the design omit some relations) or even no rendering at all (if the relations specified turn out to be incompatible).

A syntax for the data associated with a diagram is shown in Figure 4-2; it uses the term DataRelation for a mathematical relation over a finite set of atoms (that is, a table with one or more columns). This datatype is similar to JSON, except that an object may belong to multiple data relations. Similar to the rationale for a shape belonging to multiple perceptual groups, this multiplicity allows a user to express overlapping data relations.

```
Diagram ::= Shape
Shape ::= Mark | Group
Group ::= {
  subs: Name => Shape,
  rels: (Name, Name) => SpatialRelation
}
SpatialRelation ::= {
  firstDim: BBoxDimension,
  op: Operator,
  secondDim: BBoxDimension,
  spacing: number,
  strength: Strength,
}
BBoxDimension ::=
| Left | Right | Top | Bottom
| Width | Height | CenterX | CenterY
Operator ::= Le | Eq | Ge
Strength ::= Required | number
```

Figure 4-1: The essence of a Bluefish diagram.

```
Data ::= Object | DataRelation
Object ::= Primitive | Compound
Primitive ::= string | number | boolean | null
Compound ::= Name => Object
DataRelation ::= Object[]
```

Figure 4-2: The essence of Bluefish data.

The designer of a Bluefish diagram does not directly specify the diagram itself, but rather a shape template – a function that maps data instances (described by the data grammar) to diagrams (described by the diagram grammar). Usually a primitive will map to a mark, and a compound object to a group, but this is not always the case. For example, a label represented in the data as a simple string might be drawn as a text mark in a rectangle; conversely, a pair of values might be drawn as a single mark with those values defining distinct dimensions, colors, etc.

4.1 Summary

In this chapter we have identified the essential features of Bluefish. The key features of the Bluefish diagram language are marks, groups, and spatial relations. The key features of the Bluefish data language are primitive and compound objects, and data relations. By distilling the essence of Bluefish, we can identify the key extensions to typical data visualization languages: data relations and spatial relations. While

visualization languages based on the Grammar of Graphics map data objects to data marks, Bluefish extends this notion by (typically) mapping data relations to spatial relations.

Chapter 5

Perceptual Groups and Spatial Relations

With the Bluefish grammar now specified, we can see how it can be used to express a variety of perceptual grouping principles. This section outlines the spatial relations Bluefish supports. These relations are summarized graphically in Figure 5-1. We also explain tradeoffs we made when choosing which relations to include and which not too as well as how to represent them.

Perceptual grouping is "the fact that observers perceive some elements of the visual field as 'going together' more strongly than others" [56]. Grouping principles apply not just to static visual elements, but also to moving elements, sound, and touch [56, 24, 13]. Bluefish only supports static grouping principles: spatial proximity; alignment; similarity of color, size, shape, and other object attributes; common region, i.e., objects surrounded by some border or contained in another object; element connectedness, e.g., an arrow connecting two objects. We now detail our implementations of these principles.

Similarity. Similarity is implemented in Bluefish via shape templates. Two shapes generated by the same shape template with similar inputs are said to be *similar*. This formalism maps well onto the encoding-channel paradigm of most visualization grammars. A more sophisticated implementation might allow a user to specify constraints across channels on different objects, for example to declare an ordering on fonts between headers, subheaders, and body text. Scout, for example, allows a user to specify relative sizes [51].

Proximity. Proximity is a deceptively complex grouping principle. Liu et al. highlight a couple of the complexities that arose when applying proximity grouping to strokes in StrokeAggregator [39]. Proximity can often be interpreted in two ways. First, proximity can be view as *uniform* spacing or density. Objects are related if they are placed in a region of similar density to nearby objects. Second, proximity can be viewed as a *negative* relation, i.e., we may express proximity as "A is closer to B than to C." In fact the TriMap dimensionality reduction algorithm is based on this notion [2].

We formalize proximity in Bluefish by allowing a user to specify relative spacing between two elements. With only a handful of elements, this principle is useful for using proximity as a negative relation. We leave relative distances up to the user, because this property requires non-local reasoning beyond just a pair of objects. When proximity is scaled up to a large number of elements, for example a list represented as a line of equally spaced objects, the notion of proximity as relative density appears to take precedence.

Alignment. Alignment is similar to proximity, except that regions of two objects are equal, rather than spaced apart.

Common Region. Bluefish represents containment using a combination of required and (optionally) strong alignment constraints (Chapter 4).

Element Connectedness. Element connectedness between two objects A and B may be thought of as a unique spatial relation or as a pair of proximity relations, one from object A to the line and one from the line to object B. We take the second approach, because it allows us to cleanly separate spatial relations from shapes. To represent element connectedness in Bluefish within a perceptual group containing shapes A and B, a user creates a new shape and expresses proximity and/or alignment constraints between A, B, and the new shape.

Additional Static Grouping Principles. There are some static perceptual grouping principles we chose not to support. These include global properties like object *symmetry* that are difficult to fit into our local compositional model; emergent visual phenomena such as object *continuity* and *closure*, which seem to rely on perceptual

heuristics significantly more than other principles; and *parallelism* between objects, such as parallel lines and curves, because they rely on more complex shape representations than a bounding box.

These and other principles are likely to be important for other domains. For example, in differential geometry, curvature and smoothness are salient concepts in a visualization. In topology, connectedness and continuous maps are important. We hypothesize that, in the spirit of Kindlmann and Scheidegger [32], these formal properties of mathematical spaces may map to Gestalt principles in visual space.

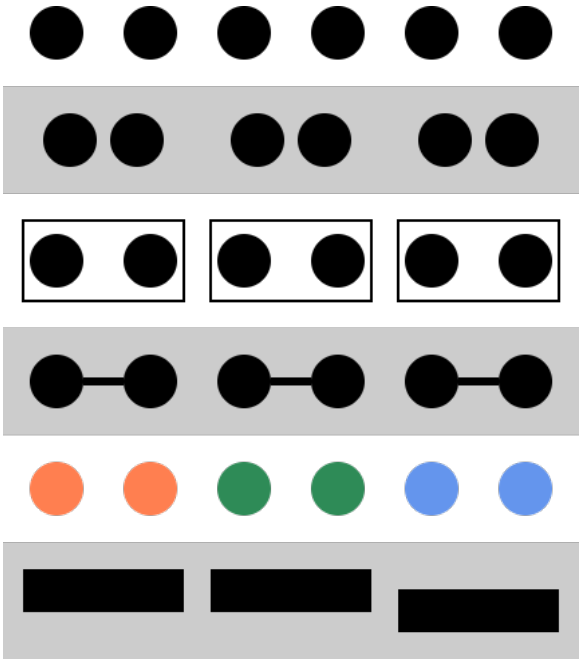


Figure 5-1: Perceptual grouping principles (top-bottom): none, proximity, common region, element connectedness, similar color (similar attribute), alignment. This diagram was made in Bluefish.

5.1 Summary

In this chapter we have identified the perceptual grouping principles, which are a subset of Gestalt relations, that Bluefish can encode. These are summarized in Figure 5-1. We have also detailed the design tradeoffs we have made to operationalize perceptual grouping in a language for generating diagrams. Bluefish makes it easy for

a user to switch between different grouping principles. Future work may explore other points in this design space to use even more visual features in a generative language.

Chapter 6

BluefishJS Grammar

In this chapter we describe the diagram and data types for the BluefishJS implementation of the abstract Bluefish grammar we defined in Figure 4-1 and Figure 4-2. We also cover the user-facing API.

6.1 Shape

Figure 6-1 describes the main BluefishJS `Shape` type. Bluefish gives every shape an axis-aligned bounding box (bbox). By default, this bbox tightly wraps around its children; however, width and height values take precedence if they are specified in the `bbox` field of a shape. Additionally a shape has an associated local coordinate system. All shapes are placed relative to this coordinate system.

We now describe each field of the main shape record. `bbox` contains initial values that may have been specified through the user-facing API. `renderFn` takes solved bbox values of the shape and produces a `JSX.Element`. `shapes` contains subshapes. `rels` contains spatial relationships between two shapes. The bbox of the entire group can also be used in `rels` via the `$canvas` identifier.

With only this record type, all shapes would be tree structures. To support overlapping shape membership, a shape can also be a *reference* to another part of the shape data structure. A reference is specified using a local path where -1 means move up a level and a string means move down into the shape with the same name.

```
type BBoxDims, Constraint, Mark; // see Sect. 4

type BBoxValues = { [key in BBoxDims]: number }

type Shape = {
bbox: Partial<BBoxValues>,
renderFn?: (canvas: BBoxValues) => JSX.Element,
shapes: { [key in string]: Shape },
constraints: {
  [key in `${string}->${string}`]: Constraint[]
}}
// shape reference
| { $ref: true, path: (string|-1)[] }
```

Figure 6-1: The BluefishJS shape grammar. The grammar is expressed in pseudo-TypeScript. `{ [key in K]: T }` defines a map with keys ranging over type `K` and values of type `T`. | is a union type. The `Partial` type makes the inner type's fields optional. `?` after a field name means the field is optional. `` `...${}...` `` is string interpolation at the type level.

Notice that marks and perceptual groups are not syntactically distinguished in the `Shape` type. One can generally think of a mark as a shape with no sub-shapes or spatial constraints; however, this is not true in general. For example, an arrow "mark" may actually comprise multiple sub shapes such as a line and arrowhead. Moreover, the circle "mark" uses a spatial relation to ensure its width and height are equal.

6.2 Data Model

The BluefishJS data model similar to the one given in Chapter 4. It can be viewed as an extension of JSON that includes local references. Primitive objects are primitive JSON elements, and compound objects are JSON objects. A JSON array represents a data relation. To extend this model to represent overlapping relations and compound objects, we add *references*. As with the shape type, references are relative paths through the data structure.

Notice a JSON object can be thought of simultaneously as an opaque object and as a relational instance, where each field of the object can be thought of as a named column.

6.3 The User-Facing API

Bluefish offers a pre-defined collection of marks and constraints as described above. Additionally, it offers a `ref` function to allow a user to easily describe references using familiar UNIX-style syntax.

The central function in Bluefish is `createGroup`. A user can pass arguments to `createGroup` to produce a shape or a *shape template*. To create a visual encoding in BluefishJS, a user defines a shape template, which is simply a JS function inputting data specified by the BluefishJS data model and returning a BluefishJS Shape. There are two ways to specify a shape template. The simplest way is to directly define a JS function, such as `(contents) => M.text(contents)`. However, Bluefish provides additional support for shape templates that produce perceptual groups. A user can create a sub-shape in a perceptual group with the following form: `$fieldName$shapeName`. `createGroup` walks the input parameters, lifts any shape template names to inputs, and returns a host shape function. A user can think about the behavior of this generated host function as follows. When BluefishJS encounters a shape name with dollar signs, it passes the field `fieldName` to the corresponding shape template, and names the shape `shapeName`. If no field name is given, Bluefish passes the entire object to the template. This is useful, for example, if the data is a primitive type without fields. If no shape name is specified, Bluefish defaults to the given field name. A user must supply at least one of the field name or the shape name. If the input data is an array, Bluefish interprets it as a data relation and so maps the shape template over every instance of the relation.

6.4 Summary

BluefishJS is a TypeScript realization of the essence of Bluefish discussed in Chapter 4. BluefishJS combines marks and groups into a single `Shape` type, and renders them as JSX elements. BluefishJS implements data and spatial relations with the help of *references*, which allow a user to link two parts of a data structure or shape structure

together. The user-facing API provides facilities for constructing *shape templates* that specify visual encodings of complex data structures.

Chapter 7

BluefishJS Implementation

BluefishJS is relatively small, comprising just a few thousand lines of TypeScript code.[1] The bulk of the logic is in the conversion from a shape template to a shape function and in the lowering of a shape to a `JSX.Element`. Now that we've described the types and interfaces in BluefishJS, we will explain how this lowering process works.

A user uses the BluefishJS compiler by calling `render`, an overloaded function that either takes a shape or input data and a shape template that has already been converted to a host function by `createGroup`. If the input contains a shape function, Bluefish applies it to the input data. This shape is then compiled to a `JSX.Element`, which can be rendered by frameworks such as React.

From the perspective of the compiler, we refer to a `Shape` as a *relational scene graph (RSG)*. This is because `Shape` is the central representation of the BluefishJS compiler and operates similarly to scene graphs in other tools [47, 11]. Our scene graph differs from most other scene graphs in two respects: it contains spatial relations and references to other parts of the scene graph. BluefishJS compiles the RSG to a `JSX.Element` in several passes, which we now summarize.

Resolve references. Beginning with an explicit RSG, we first resolve the explicit scene graph references to implicit JavaScript references to other parts of the scene graph. To maintain knowledge about which parts of the graph are references or not,

[1] This was calculated using the `cloc` command line utility, and it accounts for dead code that is not in exposed by the public library.

we add a `$ref: true` entry to the resolved version of a shape.

Collect and solve bbox constraints. The main source of complexity in the BluefishJS system is collecting and solving bbox constraints. The types of constraints are:

- *bbox definition*: Maintains relationships between the six variables in the bbox model
- *bbox value*: Encodes concrete values from the RSG's `bbox` field
- *transform*: Maintains the transformation between a shape's local coordinate system and its parent's coordinate system.
- *contain children*: Maintains the containment constraint from a shape to its sub-shapes.
- *spatial relations*: Constraints defined in the `rels` field of a shape.

These constraints are specified using a bbox abstraction. This abstraction is then lowered to a Cassowary [5] constraint problem that is solved by Kiwi.js [18]. Kiwi.js provides solved values of the low-level constraint problem, which are then turned back into bbox variables. These bbox variables are then set in the RSG.

Kiwi.js is a dense linear programming (LP) solver. LP problems can be solved in matrix-multiplication time, which is $O(n^3)$ in practice for dense matrices [17]. In our experience with BluefishJS, this complexity is acceptable for small diagrams. Performance limitations are not intrinsic to our grammar. Libraries such as Charticulator [45] and GoTree [37] have successfully applied *sparse* linear solvers to similar constraint problems to achieve big-O speedups. Other systems including Delauany [19], GARNET [42], and Indigo [10] use variations on *local propagation* solvers to achieve good performance. We strongly suspect such approaches are applicable to BluefishJS as well, and intend to explore this in future work.

Remove references. Once solving is complete, the RSG is prepared for rendering. We strip references from the graph, since references are not rendered and the DOM is effectively a tree. After this step, the RSG has also become a tree.

Render. Finally, this tree is "rendered," i.e. transformed into a `JSX.Element`. The renderer walks the tree, bottom-up. It calls each `renderFn` with the solved bbox values. If a shape has sub-shapes, the renderer takes the rendered sub-shapes as well as the output of the `renderFn` (if it exists) and places them inside a `<g>` element. Local coordinate frmes are reified as SVG transforms, and each element is given a class name corresponding to its shape name. Finally, the result is placed inside an SVG element whose width and height are same as the top-level shape's.

Notice that throughout this entire process, besides removing references, the shape of the RSG is unchanged. This helps a user inspect and debug the resulting DOM, since the DOM's structure corresponds to the non-reference shapes in the original RSG. Similarly, the structure of the shape output by `createGroup` matches the structure of the input, but with shape templates replaced by shapes.

7.1 Summary

BluefishJS is a small codebase, based closely on the essence of Bluefish. It is designed to be read and modified by especially motivated users and researchers. BluefishJS is architected as a compiler that translates user-facing API code into DOM elements. We have constructed the compiler process to preserve the structure of the input user-facing code as much as possible. That is, the structure of a user-defined specification typically mirrors the structure of the DOM generated by BluefishJS. This facilitates easier debugging by allowing a user to reason about how their inputs map to BluefishJS's output.

The two main sources of complexity in the implementation are references and constraints. References must be resolved to bounding boxes for the shapes that they point to. Constraints are handled by Kiwi.js, an implementation of the Cassowary linear constraint solver.

Chapter 8

Evaluation: Three Diagram Replication Studies

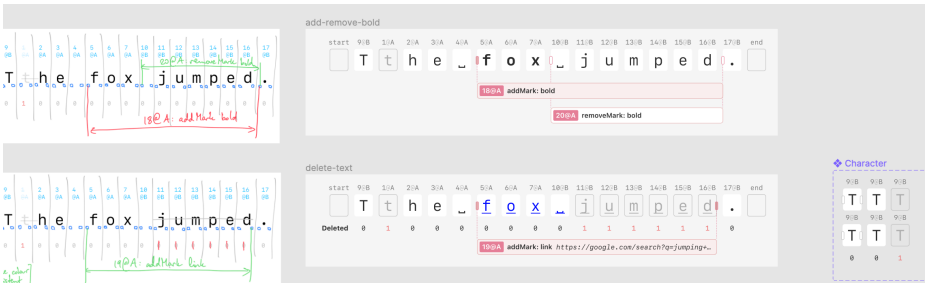
To evaluate Bluefish, we compare our grammar using Cognitive Dimensions of Notations (CDN) [52, 9] to three alternatives: (i) Figma, a direct manipulation editor (Section 8.1), (ii) D3 [12], an expressive, low-level framework inspired by the Grammar of Graphics, and (iii) GoTree [37], a domain-specific diagramming tool. In this section, axes of the CDN will appear in bold, accompanied by footnotes with short explanations as presented by Blackwell and Green [9].

Each diagram in this section renders in less than X seconds. In our experience, this is fast enough that most diagrams can be designed iteratively; however, we leave performance optimizations to future work.

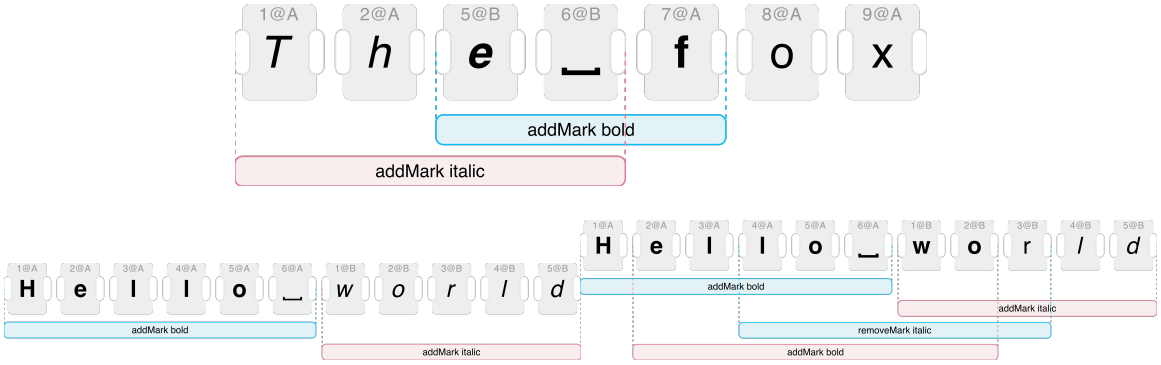
8.1 Bluefish vs. A Direct Manipulation Diagram Editor

A direct manipulation (DM) editor is a common tool for drawing diagrams. To compare Bluefish to a DM editor, we worked closely with a graduate student, Geoffrey Litt, to replicate in Bluefish a diagram family he originally created in Figma.

Geoffrey helped created Peritext [38], an algorithm for asynchronous collaboration in rich text editors. With the help of the Peritext algorithm, a writer can edit a document offline, then merge it with their collaborators' edits when they're ready.



(a) Figma Board Screenshot (Not Bluefish)



(b) Changing a Peritext diagram's text and mark spans. (Bluefish)

Figure 8-1: Peritext Example

This algorithm is complex, and the authors of Peritext had to consider many conflict-resolution edge cases when designing their algorithm. For example, what should happen when two users apply a color style to overlapping regions of text?

To help explain this algorithm and the edge-cases involved, Geoffrey created diagrams that represented Peritext's central data structure. Working with his colleagues, Geoffrey first designed hand-drawn prototypes before switching to Figma to finalize the drawings. Figure 8-1a shows some of the prototypes and final diagrams Geoffrey created. Each diagram roughly comprises a list of characters, a list of mark spans that contain styling attributes like bold and italic, and lines connecting a span to start and end characters.

Geoffrey ran into several problems when using Figma to create these diagrams. First, even though he and his collaborators had converged on a hand-drawn prototype, Geoffrey made several changes the layout and visual style of the final design. Figma provides a *component* concept (seen on the right in Figure 8-1a) to help prop-

agate component styles, but layout changes are highly **viscous**[1] in the editor. That is, a small semantic layout change – like "increase the spacing between characters" – requires *many* manual edits to accomplish. To change the spacing, for example, Geoffrey manually move each character, or group them and redistribute. After this change, the vertical arrows connecting the mark spans to characters, as well as the position and width of the mark spans, must be adjusted to match the new positions of the characters. Finally, these edits must be replicated manually for *each* diagram. Geoffrey's Figma board contained about a dozen such diagrams.

Second, changing the input data is also a highly *viscous* operation. For instance, Geoffrey wished to experiment with using "The R.M.S. Titanic" instead of "The fox jumped." in his examples. Similar to layout propagation, making this change would require Geoffrey to manually update and relayout each figure, including updating the positions of mark spans to match the new characters. All told, both semantically atomic layout and input changes in Figma can require Geoffrey to edit most if not *all* the objects in his diagrams.

Finally, Geoffrey wished he could create a live view of the Peritext algorithm by hooking diagrams up to the data structures produced by Peritext. This would both allow a reader of his article to experiment with their own examples, and also provide Geoffrey and his co-authors with a highly effective means of debugging the algorithm. Because Figma does not separate the data of a diagram from its visual encoding, Geoffrey had no clear path to converting his Figma diagrams into live examples.

We worked with Geoffrey to recreate this diagram in Bluefish (Figure 8-1b). The diagram takes as input a Peritext data structure, and outputs a diagram. For example, the first diagram in (Figure 8-1b) was generated from the following data:

```
const chars = [{
  value: 'T', opId: '1@A', marks: ['italic'] }, ...]
const markSpans = [{ action: 'addMark', opId: '18@A',
  start: {opId: '5@B'}, end: {opId: '7@A'}, markType: 'bold'},
```

[1] *resistance to change*

```
  ...
]
```

Geoffrey wrote some light-weight pre-processing on this structure to create Bluefish references that associated markSpans with their characters. Geoffrey created a shape template for Peritext diagrams comprising three sub-shape templates: one for the list of characters, one for the mark spans, and one for the relation between mark spans and start and end characters. The modularity of sub-shapes affords Geoffrey **progressive evaluation**[2]. Geoffrey can work on and visualize a character, a character list, a mark span, and relations between mark spans and characters *independently*, because they are different shape templates. We observed that when Geoffrey realized he could *explicitly* represent the relation between characters and mark spans in his data structures and in his visual layout as perceptual groups, Geoffrey began to see some of the full power of the Bluefish system. This explicit representation allowed him to see the lines connecting characters and mark spans as a relation that could be given its own visual encoding and style, in this case by producing dashed lines connecting the elements.

Compared to Figma, Bluefish is affords less *viscous* layout and data changes. For example, to change the spacing between characters in the list, Geoffrey need only modify an `hSpace` relation in his characters list template akin to the spatial relation change in Chapter 3. Thanks to his templates, Geoffrey can easily modify the data in his diagram without adjusting the visual encoding (Figure 8-1b). And because both Peritext and the Bluefish library exist in JavaScript, Geoffrey is now able to create a live debug view of his algorithm.

Bluefish's abstractions trade **viscous** layout and data editing for **viscous** data schema changes. For example, if Geoffrey wanted to use Bluefish to move one of the characters in his list up slightly to draw attention to it, he would have to change his input structure and shape templates to reflect the changes in the perceptual groups in the diagram. On the other hand, Geoffrey could simply drag one of the characters to make this change in Figma, since the editor does not keep track of per-

[2] *work-to-date can be checked at any time*

ceptual groups. Thus DM editors and tools like Bluefish serve complementary roles. When the structure of one's data is not yet known, such as in the beginning stages of diagram prototyping, a user may prefer a DM editor where they can easily move individual shapes. However, once this structure begins to solidify and the number of diagram instances grows, a user is likely to prefer a tool like Bluefish. In Geoffrey's case, since he already had the data structures produced by Peritext and had created a rapid prototype by hand, Bluefish was a natural next step in his design process. We speculate that a *prodirect manipulation* editor [15] for Bluefish, which combines the affordances of both programmatic and DM abstractions, may help users take full advantage of both systems.

Another tradeoff between Bluefish and DM editors that stems from perceptual group abstractions is that an edit to a perceptual group can have non-local effects. As we've described, this non-local behavior benefits a user when making small semantic edits; however, it is possible that such non-local effects cause unexpected changes to the diagram. We rarely encountered this issue in our experiences working with Geoffrey, because of Bluefish's **progressive evaluation** affordances. Geoffrey was able to build his shape templates one group and one relation at a time. This allowed Geoffrey to frequently check the layout of his diagrams to ensure they matched what he expected.

In this case study, we have compared and contrasted authoring diagram instances in Figma and in Bluefish. Compared to Figma, Bluefish makes atomic semantic edits – like changing the input data or changing the spacing between characters – nearly atomic edits to the data and/or visual encoding. Though we looked at only one DM editor in this case study, many editors provide similar cognitive dimensions tradeoffs. We have noticed global edit propagations for atomic semantic edits when creating diagrams in Adobe Illustrator, PowerPoint, and Keynote via informal observations and interviews with other diagram authors.

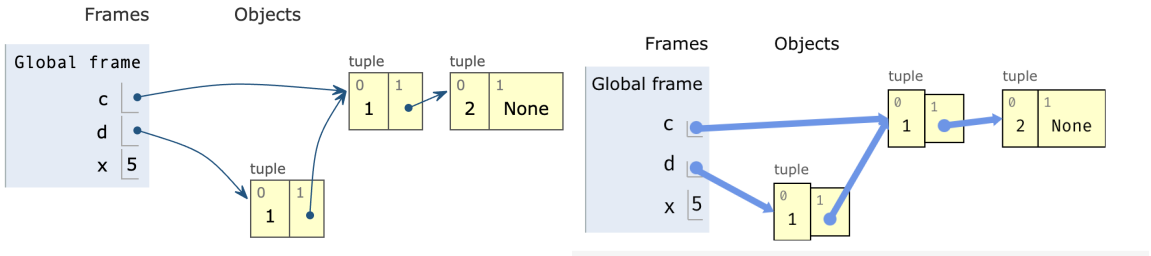


Figure 8-2: A visualization of the stack and heap during the execution of the Python program `c = (1, (2, None)); d = (1, c); x = 5`. Python Tutor (left) and our reproduction (right)

8.2 Bluefish vs. A Grammar of Graphics Toolkit

We now compare Bluefish with D3 [12], an expressive, low-level framework inspired by the Grammar of Graphics. Like the Grammar of Graphics, D3 primarily uses *scales* and object *attributes* construct visual encodings. However, in contrast to the Grammar of Graphics, D3 provides a user more control of the visual structure of a visualization via DOM *selections*.

In this chapter, we compare and contrast the implementation of Philip Guo's Python Tutor [28] in D3 with a partial recreation in Bluefish. Python Tutor – a website that visualizes program state for languages such as Python, Java, and C++ – is one of the most popular diagram visualizations built on D3. According to its website, Python Tutor has reached "[o]ver 10 million people in more than 180 countries," and it has been used "to visualize over 100 million pieces of code." Python Tutor and its visualizations have been amazingly helpful to many students, and one of the original goals for the Bluefish project was to make diagram instances like Python Tutor's easier to build.

Python Tutor is built on a combination of D3, jsPlumb (a toolkit for graph layout), and jQuery. Since both D3 and Bluefish are embedded in JavaScript, they each have access to a Turing-complete language. This can make a comparison harder, because each tool can technically express any visualization with the help of surrounding JavaScript code. Felleisen provides one possible way to determine the relative expressiveness of Turing-complete languages [23]. If a language abstraction cannot be

locally rewritten into existing language constructs, then that abstraction is said to be *expressive* with respect to the base language. For example, though it may improve readability and ease-of-use, a language with both for-loops and while-loops is not any more expressive than one with just while-loops, because a for-loop can always be locally rewritten into a while-loop. On the other hand, a language with exceptions is more expressive than one without, because implementing exceptions requires meticulously threading state through an entire program to model the call stack unwinding that occurs when an error is raised. The non-local abstractions in Bluefish are (i) references, and (ii) spatial constraints. References allow a Bluefish user to relate arbitrary pieces of data or shapes together, and spatial constraints allow non-local propagation of layout information. Python Tutor seemingly requires references to handle pointers between the stack and the heap and between heap objects. It also seems to require spatial constraints to lay out the stack and heap. Since D3 does not provide these abstractions, we hypothesized that the Python Tutor implementation would have to reconstruct or otherwise explicitly manage references and spatial constraints via mechanisms external to D3.

By inspecting the source code, we have confirmed this hypothesis. To implement spatial constraints, Guo uses HTML tables. Tables give a lightweight way to position items based on horizontal and vertical spacing and alignment. This works well for a single diagram family like Python Tutor's, but it limits the expressiveness of a more general tool like Bluefish. Bluefish allows a user to express relations not just using spacing and alignment, but also with containment and linking. Python Tutor diagrams contain these grouping principles as well, but they are managed by table attributes and jsPlumb respectively. Thus Bluefish is more **consistent**[3] than Python Tutor's abstractions.

Additionally, Python Tutor's use of tables means the diagram exists in a `div` element rather than in SVG. This limits a user's abilities to customize the diagram's appearance with SVG elements, since SVG and `div`s are hard to compose.

To approximate shape references, Python Tutor uses jsPlumb to handle pointers

[3]similar semantics are expressed in similar semantic forms

between the stack and the heap and between heap objects. This increases the **viscosity** of the reference interface, because DOM references must be passed back and forth between jsPlumb and jQuery. By contrast, Bluefish treats references as core primitives of the language, and handles lines and arrows **consistently** with other shapes. A user can use the same spatial constraints to control positions that are handled separately by table and jsPlumb abstractions in Python Tutor's implementation. As a result, an author of the equivalent diagram in Bluefish can easily change the data representation and visual encoding to, for example, add arrows between neighboring "cells" in a tuple.

Python Tutor's DOM tree is deeply nested. An element of a tuple, for example, is roughly 20 nodes deep in the tree. Managing the complexity of hierarchies is therefore important both for reading and writing Python Tutor-like diagramming code. Python Tutor uses sequences of D3 selections to handle hierarchies. For example:

```
var stackDiv = myViz.domRootD3.select('#stack');
var stackFrameDiv = stackDiv.selectAll('div.stackFrame,...')
var stackVarTable = stackFrameDiv.order().select(...)
```

D3's selection model is useful for specifying interactive and animated elements, but Python Tutor, which shows sequences of static visualizations, does not take advantage of this feature. The structure of D3 selections obscures the underlying hierarchy. Idiomatic D3 code keeps every nesting level at the same indentation level in the code, and the structure of the underlying data must be inferred via the callbacks applied to a selection.

Here, in contrast, is similar code in our Python Tutor reproduction:

```
$stackFrame$: createGroup({
    $elements$: createGroup({
      $var$: ...,
      $value$: ...,
    },{ 'var->value': [C.hSpace(9.5), C.alignBottom] }),
    $neighbors$: createGroup({
      $curr$: 'ref',
      $next$: 'ref',
```

```
    },{ 'curr->next': [C.vSpace(10), C.alignRight] }),
  }),
```

Just as JSX affords **closeness of mapping**[4] between React components and the DOM, so too does Bluefish afford a **closeness of mapping**. Unlike JSX, Bluefish must facilitate *two* mappings: one to the data, and one to the DOM. We accomplish this dual mapping through field functions (Chapter 6). Closeness of mapping between data and visualization is especially important, because many visualization researchers posit that visual encodings should be straightforward 1-1 mappings between data and visualizations. For example, Tversky's Congruence Principle states "the structure and content of the external representation should correspond to the desired structure and content of the internal representation" [54].

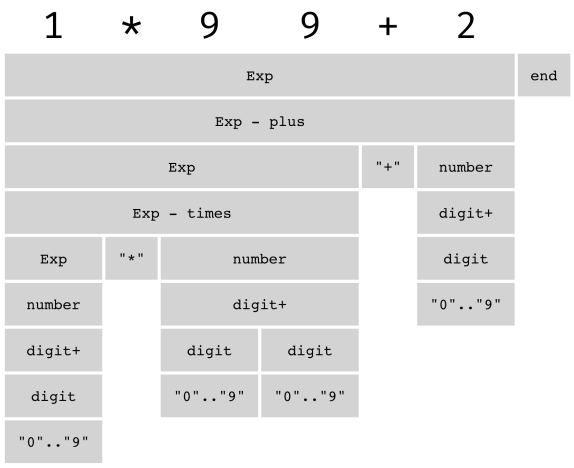
In this case study, we have analyzed the structure of Python Tutor's implementation. We have discovered that Bluefish's key abstractions, references and spatial constraints, were re-implemented by Guo on top of the base D3 language using jsPlumb's node-link model and tables, respectively. We have identified some limitations with this implementations. We also explain how Bluefish's shape template API affords **closeness of mapping** between structures in the data and in the DOM. D3 makes these structures implicit. Unlike D3, Bluefish does not currently support interactive or animated visualizations. Future work may investigate whether this **closeness of mapping** can be retained when adding those abstractions.

8.3 Bluefish vs. A Domain-Specific Diagramming Tool

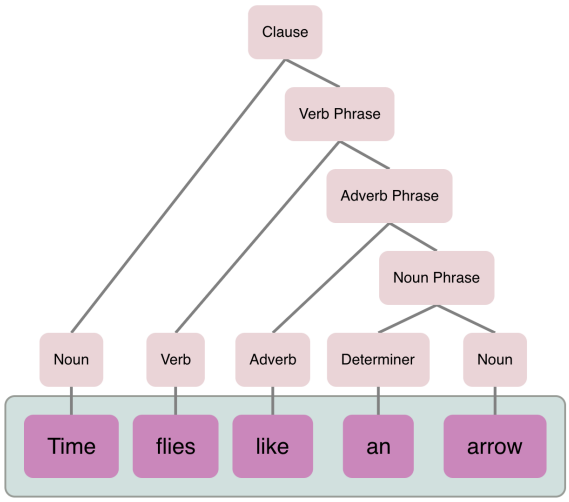
We now compare Bluefish to GoTree [37], a domain-specific diagramming tool for tree visualizations. In our analysis, we focus on similarities and differences that we believe generalize to comparisons with other domain-specific tools like Scout [51] and SetCoLa [31].

Like Bluefish, GoTree uses perceptual groups like links, containment, and proximity for layout. Unlike Bluefish, GoTree accepts only tree structures and limits the

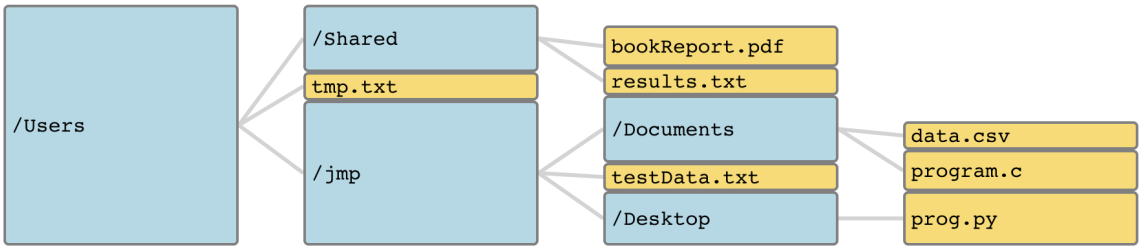
[4] closeness of representation to domain



(a) Ohm Parser Icicle Tree [20]



(b) Natural Language Parse Tree



(c) FlexTree [49]

Figure 8-3: Bluefish Trees

visualizable discrete relations to parent-subtree and subtree-subtree relations. Li et al. define visualizations using only these two relations as *unit-decomposable*. Compared to GoTree, Bluefish can express more relational structures. For instance, Bluefish can represent a FlexTree [49], which is not unit-decomposable, because it relates subtrees with different parents together (Figure 8-3c). Similarly. In this way, a data visualization author does not have to **pre-commit**[5] to a given data structure shape.

By providing high-level constraint abstractions, GoTree arguably provides a **closer mapping** to a user's mental model of perceptual groups in a tree. On the other hand, Bluefish's abstractions are more **consistent**. In GoTree, the root-subtree and subtree-subtree spatial relationships are distinguished. GoTree identifies three root-subtree spatial relations: within, juxtapose, and include. They identify two subtree-subtree

[5]constraints on the order of doing things

spatial relations: flatten and align. These map onto Bluefish's spatial relations as follows. include/within map to `alignSpace`. In GoTree, both include and within are necessary, because one refers to a root contained within a subtree, and the other a subtree contained within a root. In Bluefish a user can express one or the other by swapping the ordering of the shape names in the spatial relations field. Align maps to Bluefish's `align`. Finally, both flatten and juxtapose map onto Bluefish's `space` constraint. The distinction is made in GoTree, because flatten applies to a list of subtrees, where as juxtapose applies to root-subtree pairs. Because Bluefish separates the *structure* of a discrete relation from the *layout* of an individual perceptual group, it reduces five constraint types to just two. This simplified interface provides more **consistency** than GoTree's. Moreover, these constraint can be applied to *any* part of a Bluefish diagram, not just tree structures.

8.4 Summary

In this chapter we have compared Bluefish to three other kinds of systems for diagramming: a direct manipulation editor, a Grammar of Graphics toolkit, and a domain-specific diagramming tool. Compared to direct manipulation editing, Bluefish provides object and relation abstraction mechanisms that make it easy to modify the perceptual grouping principles and input data used in a diagram. On the other hand, direct manipulation editors offer more flexibility for tweaking fine details of a diagram's appearance. Compared to a Grammar of Graphics toolkit, Bluefish's references and constraints offer consistent abstractions, especially for specifying arrows and containers around many shapes. Moreover, Bluefish's hierarchical shape template specifications more closely mirror both the structure of the input data and the structure of the output DOM, making it easier to reason about the behavior of a visual encoding. On the other hand, Bluefish currently lacks abstractions specific to statistical charts such as axes, legends, and scales. Compared to a domain-specific diagramming tool, Bluefish offers more flexibility over input data structures, which means a user does not need to pre-commit to a given data structure like they would

with a more domain-specific tool. Moreover, Bluefish's primitives provide a consistent interface *across* domains. The same spatial relations can be used to specify a tree and a bar chart. On the other hand, because Bluefish was not designed for a specific domain like trees, Bluefish code may be slightly more verbose, as it may require a user to write some aspects of a diagram explicitly that are implicitly handled by a more domain-specific tool. Overall, Bluefish compares favorably to these three classes of alternatives, because it provides consistent, flexible abstractions for diagramming.

Chapter 9

Discussion

9.1 What I Learned About Diagramming

My main goal in this thesis was to find properties of data visualizations that could be useful for authoring diagrams in a wide variety of domains, from text layout to statistical charts and everywhere in between. My search for these properties combines ideas from visualization, programming languages, and perceptual psychology.

In this work I have explored in detail *perceptual groups* as one such domain-agnostic idea. This insight stemmed from two main sources. First, I experimented with different visualizations of the same underlying data in the context of program state visualizations, such as representing a linked list using arrows between boxes, adjacent boxes, or nested boxes. I ultimate recast these transformations as modifying perceptual grouping principles. Second, I observed people creating diagrams and noticed that they often created *edit cascades*, where modifying the position of a node, for example, would require updating all the spatial relations it was transitively involved in. This appears to be a robust pattern in most if not all diagram authoring.

Another idea that surfaced while completing my thesis was the notion of laying out parts of a diagram "one at a time." That is, placing one object or collection of objects, and then fixing their positions and placing more objects in relation to them. For example, one often places the main diagram or chart before adding annotations or labels. This pattern can be found not only in digital editing of a single diagram,

but also when someone makes a presentation on a whiteboard or in slideware. Even if the underlying information is a single structure, a diagram for it is often built up over time to gradually introduce complexity or new information even if the final product is a static diagram. This build-up is semantically distinct from animations or storyboards of diagrams evolving over time, even if they may rely on the same or similar primitives.

I learned that relational structure in diagrams is rich and complex, and many things in our everyday lives are diagrams. The signs we see on the street, the books we read, the charts we look at. All of these and more are diagrams; and we often augment them with our own marks through annotations or even physical gestures.

Through working on this thesis I have come to better understand the terrain of diagrams. Diagramming problems are highly structured, with shared techniques across domains and with small sets of object and layout primitives supporting most diagrams.

Yet even with knowledge of this structure, developing interfaces for diagramming remains a major open challenge. One must balance the freedom of unstructured representations with the efficiency of editing in structured representations. The tradeoffs of these two has plagued diagram authoring tools since their inception [48, 52].

9.2 Broader Impacts of Bluefish

The impact of Bluefish goes beyond just authoring diagrams more easily. Because Bluefish represents perceptual groups explicitly, it could enable more powerful reasoning tools that can leverage relational information. from accessibility tools to diagram recommendation and analysis engines.

Relations for other domains. As we have argued in this paper, spatial relationships in a diagram are inextricably linked to the data relations they represent. The choice of *what* sorts of data and spatial relations to use, however, depends on the information relevant to the problem at hand. For example, to visualize a theorem in Euclidean geometry, one must encode intersection points and collinearity as geo-

metric relationships. In the same way we have formalized perceptual groups for use in discrete diagrams, future work could inventory the data and spatial relationships pertinent to other domains and use them to build new tools. Penrose, for instance, has explored primitives for Euclidean and differential geometry [59].

Perceptual Groups in Statistical Visualizations. Cleveland and McGill [16] identify ten "elementary perceptual tasks" that people use to read information from visualizations. These tasks – such as reading the length, area, or color of an object – pertain only to the properties of *individual* objects; and researchers have argued that, taken to its logical conclusion, this set of tasks suggests that *all* visualizations ought to be scatterplots [8]. Of course, in practice visualization designers find that all sorts of charts are useful in the right contexts. We believe that placing perceptual groups as "secondary" tasks alongside Cleveland and McGill's ten elementary ones could help close the gap between visual effectiveness theory and practice by capturing more details about the information encoded in a visualization.

Accessibility. While many accessibility solutions today are built on tree-, table-, or text-based representations, these structures do not fully capture the complexities of a data visualization. Systems like Bluefish and its relational scene graph (Chapter 7) might help support next-generation accessibility tooling that could, for example, explicitly describe the relationships between diagram annotations and the data they point to. Moreover, since the relational scene graph allows for overlapping relations it can support overlapping views of the same data, like binning points in a scatterplot horizontally, vertically, and in a coarse grid. Overlapping structures are not well-supported by existing accessibility views.

Automated Reasoning and Synthesis. Bluefish creates a more structured *design space* of diagram visualizations. As discussed in Chapter 5, Bluefish makes it easy to represent the same underlying data using different relational visual encodings. The concise, yet general, description of diagrams using spatial relations has the potential to facilitate automated reasoning and synthesis tools. For example, one could construct a visualization recommender that systematically searches over a space of possible perceptual grouping principles. Additionally, one could build a system that reasons

about the effectiveness of perceptual grouping principles that operate on top of a Bluefish specification.

While large corpora of statistical charts exist [43], it is much harder to build a corpus of diagrams that captures their rich relational structures. We suspect that by combining emerging ideas for encoding spatial relations in machine learning [40] with a language such as Bluefish, one could begin to build a detailed corpus of diagrams or else train models that use relational information to reason about, produce, and modify diagrams.

9.3 Limitations of Bluefish

In this section we summarize the key limitations of Bluefish.

High-fidelity layouts. Though Bluefish can express many common types of diagrams, there are still many aspects of layout that Bluefish does not handle well. For example, Bluefish's linear programming interface does not help a user specify complex geometric specifications like circle intersections, tangent lines to curves, or more general graphics problems. Because Bluefish is embedded in JavaScript, a user can write arbitrary code to generate expressive custom layouts. For example, the Python Tutor diagram recreation in Section 8.2 relies on output from DagreJS to determine the locations of heap nodes. However, Bluefish's current mechanisms for this kind of extensibility are limited. A user must pass layout information into the input data structure, rather than augmenting the visual encoding directly. As a result, these external algorithms must include their own code to reason about bounding box and position information that Bluefish will recompute later on.

Interfaces for diagram authoring. Bluefish's primary interface is currently text. While this is a useful medium for defining abstractions and as a compilation target for other tools, users often prefer direct manipulation editors to specify complex diagrams. A direct manipulation interface for Bluefish could help a user more easily specify a data or shape reference (e.g., by clicking on the relevant data or shape and constructing a reference rather than having to write out a path). It could also help a

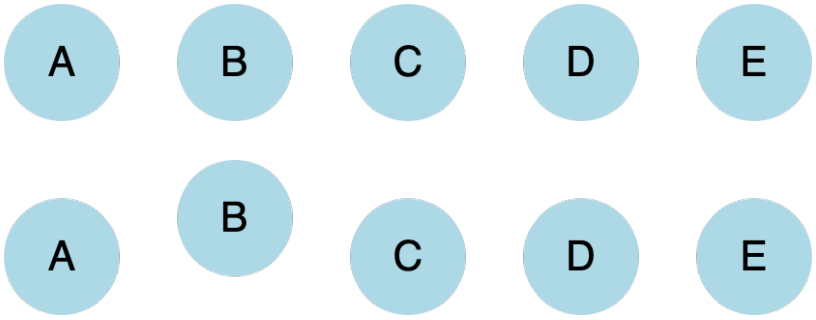


Figure 9-1: (top) A simple list; (bottom) the same list but with "B" shifted up. This seemingly simple modification requires a major restructuring to the relations in the diagram.

user more easily modify the structure of their data or diagram. For example, consider the two diagrams in Figure 9-1. The top diagram is easy to visualize, because there is just one relation between the nodes. The bottom diagram is just a small modification of the top – nudging the "B" node vertically – but it greatly changes the relational structure. In addition to the original list relation, which evenly spaces the nodes horizontally, there is now a relation just for node "B" and a relation for the rest of the nodes. These two relations vertically align the nodes, and the relations are vertically spaced relative to each other. This modification is difficult to do in Bluefish's text format, but a direct manipulation editor could allow a user to drag node "B" and restructure the relations automatically.

Fuzzy Gestalt principles. Bluefish cannot easily represent fuzzier Gestalt principles such as continuation (where two or more objects suggest the existence of another object in negative space) or similarity between bezier curves. This means that some relationships in a diagram cannot currently be captured well by Bluefish. These principles seem to be even more important as one moves from technical diagrams into art, where abstract shapes often combine via fuzzier Gestalt principles to form larger structures.

Interaction and animation. Bluefish does not currently support interactive or animated diagrams. It also does not support building up a single diagram over time well. Though the effectiveness of different kinds of interactions and animations is still up for debate, it is difficult to study these today since few useful representations exist

to test the affordances of different principles.

Layout engine design. Bluefish currently relies on Kiwi.js, a TypeScript implementation of the Cassowary constraint solver [5]. Though Cassowary's API does provide fast incremental updates to layout problems, Bluefish cannot take advantage of this directly, because layout is computed from scratch every time a diagram is rendered. As a result, Bluefish's layout performance has an algorithmic complexity of $O(n^3)$ and does not scale well in practice past dozens of elements. An ideal layout algorithm would run from scratch in linear or quasilinear time. Moreover, even when using a sparse linear solver for improved performance, linear constraint problems are difficult to debug when they are unsatisfiable. Even though the likely culprit is a small collection of constraints, producing such a set and ensuring that the set is interpretable by a user is a monumental task. Finally, as we described earlier, many layout algorithms (such as paragraph breaking and tree layout) cannot be easily expressed as linear programming problems. As a result they must be implemented externally to Bluefish.

9.4 The Data Visualization Archipelago

Now that we have presented Bluefish, gestured at some of its potential broader implications, and identified a few of its key limitations, we turn our attention to what's next for discrete diagram languages.

Today, data visualization domains are fragmented. Statistical graphics, diagrams, text documents, and mathematical notation are treated as separate islands with their own separate tool stacks. User interfaces, which can also be viewed as data-driven diagrams, have their own set of solutions as well. These architectural silos present barriers to mixing different elements together. For example, a visualization author may want to place a diagram in a piece of text, or annotate a mathematical diagram [29]. They may not even know what domain their visualization may fall into. Perhaps they start visualizing a recipe as a piece of text, but then later turn it into a complex flow chart with margin notes and bar graphs. Today's data visualization

tools are monoliths and impose overly strict boundaries that hinder users' ability to make the diagrams they want.

This archipelago not only presents hurdles for data visualization authors, but also for data visualization framework builders. For example, many visualization domains tackle label layout [33], line wrapping [34, 7], and annotation. However, today these algorithms are not easy to port from one domain to another. Though they tend to share underlying abstractions like axis-aligned bounding boxes (Figure 9-2), these algorithms are typically written as opaque boxes, rather than with layout compositionality in mind. As a result, engineers and researchers developing new layout and visualization techniques cannot easily share their work across domains, and today's state of the art techniques are difficult to compose in the same diagram.

What can be done about this problem? We can look to the world of compilers for inspiration. By providing a common language in which to express similar problems, compiler frameworks such as LLVM have helped to consolidate the programming language ecosystem, allowing domain experts in, say, register allocation to focus just on their problem and have the benefits apply to many different languages that target LLVM [35]. LLVM specifically provided new forms of modularity that enhanced its ecosystem compared to its predecessors. Prior to LLVM, compiler stacks like GCC were monolithic applications [36]. The language parser of a compiler, for example, could not easily be repurposed for program analysis or syntax highlighting. LLVM popularized a compiler architecture where optimization passes are separate libraries. This allows a compiler writer to control what passes to run and when, and to easily add their own passes as needed. This modularity made it easier to write frontend languages that targeted LLVM, because it gave those developers more control over low-level optimization.

I believe a similar consolidation is possible in data visualization, and that this consolidation may bring both more *expressive* visualizations by allowing domains to compose easily and richly, as well as more *beautiful* visualizations by concentrating layout effort in domain-semi-agnostic ways. Moreover, I believe Bluefish can become such a platform for consolidation. Realizing this vision is out of scope for a masters

thesis; however, in the rest of this chapter we will sketch some initial thoughts on a future consolidated platform.

By a consolidated visualization language I mean one that can be used to implement higher level languages for specific domains, such as the Grammar of Graphics for statistical charts. A language that can serve as a platform for higher level DSLs is one that is *growable* in the sense of Guy Steele:

> I should not design a small language, and I should not design a large one. I need to design a language that can grow. I need to plan ways in which it might grow—but I need, too, to leave some choices so that other persons can make those choices at a later time [50].

That is, it should be easy for a user to extend a language with their own primitives and have those primitives function like built-in components. Steel argues this is best achieved via in-language extension mechanisms like macros, type-level generics, and even classes, because they allow for the easy and smooth addition of user-defined components. For visualization, a growable language would allow a developer to create a DSL for a new visualization domain by defining primitive shapes and layouts in that language. Since all these user-defined primitives would function like parts of the language, they would easily compose with each other and with primitives from other domains just like the type `Array<T>` in TypeScript (and analogous forms in other languages) can be instantiated using any other type, built-in or user-defined.

9.5 A Growable Diagram Language and Layout Engine

Of course, just because a modular diagram language is desirable does not mean it is obvious what the modularity should be or how it should work. In this section I will outline some initial thoughts on a growable diagram language. Because layout algorithms are integral to visualizations, I will also sketch ideas for a growable layout engine.

To understand how to build a growable diagram language, one can start by looking at similar growable languages that have come before. The closest existing systems are

UI frameworks – such as React [22], Flutter [27], and SwiftUI [3] – and ggplot2 [57]. These systems let users to define new components or layers, respectively, allowing them to create shareable collections of reusable abstractions that effectively extend the frameworks. In this thesis we have argued the centrality of relations for both data representation and visual encoding. Where a growable diagram language differs from related languages, then, is in its ability to grow the sets of relational data and visual primitives.

One major difference between the design of a language with relational growth and these other growable languages is the complexity of the compositional structure. Ggplot2's compositional structure is a linear stacking of layers, and most popular UI frameworks allow users to create tree hierarchies of components. However, a more general relational diagram language must support graph-like compositional structures, such as via Bluefish's data references. But while graph-like structures provide the expressiveness one needs for building diagrams, they are more complex to reason about and use [26]. Is it possible in a data visualization framework to maintain the simplicity of tree structures and hierarchical composition while maintaining the expressiveness of graphs?

In the next stage of Bluefish, I will explore this question. My plan is to study the architectures of common layout algorithms for trees, graphs, text, etc. as well as the structure of Bluefish diagrams to find graph representations that balance structural complexity and layout expressiveness. Already, I have found that many state-of-the-art layout algorithms share an axis-aligned bounding box structure (Figure 9-2). This general structure, which abstracts parts of a graphic into simple width, height, and position properties, appears to be an instance of Herbert A. Simon's "nearly decomposable" observation that complex systems are typically composed of pieces that weakly depend on each other.

I also suspect that ideas regarding "one-at-a-time" layout may prove useful. One may regard layout as a series of "passes" over a diagram data structure, with each stage refining or constraining the positions of elements in the diagram or adding new ones (such as adding annotations or labels to other elements that have already been

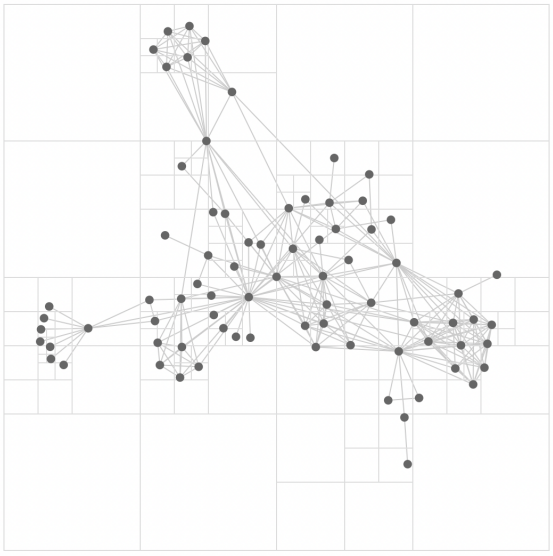
placed). This is similar to the notion of an optimization pass in compilers and a render or shader pass in graphics. For example, in their graph layout algorithm, Gansner et al. (i) place nodes in distinct layers, (ii) order nodes within those layers, (iii) position the nodes in space, and (iv) route edges between nodes [25]. Each such pass relies on information from the previous pass, but does not change it. That is, though a pass may not fully specify the positions of elements, leaving some of those choices up to later stages, once a pass is run it is never revisited. This lack of explicit backtracking helps to ensure the algorithm has good performance. Similarly, in their label layout algorithm, Kittivorawong et al. first rasterize a statistical chart (into a set of pixel-sized axis-aligned boxes), and then place labels to avoid occupied regions without modifying the original chart [33].

In addition to one-at-a-time layout providing global composition of different diagram fragments, parent-child composition is likely to be an important mechanism. This form of composition is already present in tree-based UI frameworks. Frameworks like Flutter and SwiftUI codify the communication patterns between parent and child components. First, a parent node requests its children fit in specific width and height intervals, then each child performs its own layout and reports its actual size to its parent, then the parent places each of its children. This communication pattern is common in layout algorithms for diagrams, too. For example, van der Ploeg's tree layout algorithm renders subtrees independently and then places them below a parent root node [55]. Similarly, once the bounding boxes for characters in text have been determined, Knuth and Plass's paragraph layout algorithm places them by optimizing various aesthetic qualities of text [34].

9.6 Conclusion

By providing a substrate on which many data visualization tools can be built for many domains, the next version of Bluefish has the potential to bridge the divides between the islands of the archipelago. I imagine a world where mixing visual domains is easy, and new layout techniques developed for one domain can be shared quickly with

many others. I imagine a user crafting their own visual primitives in a week or two, rather than research teams spending months on custom compilers for new problems. When expressive diagrams are much easier to create, and when informal hand-drawn diagrams can be scaled up to beautiful, data-driven programs, visual explanations may become much more common and complex ideas much more accessible.



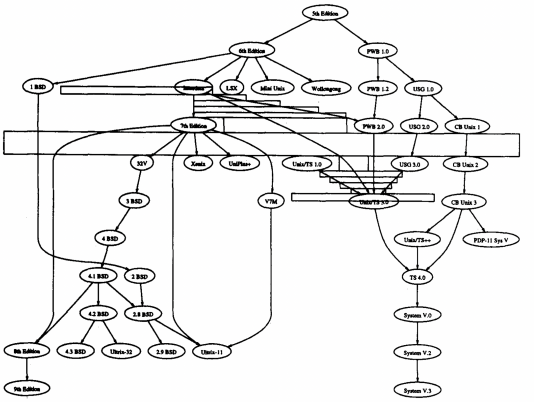


Fig. 16. Region for a spline (0.48 s user time Sun 4/280).

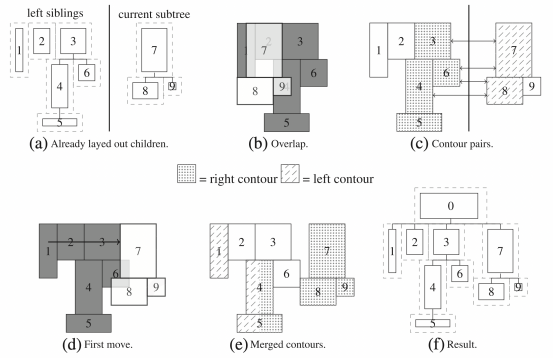
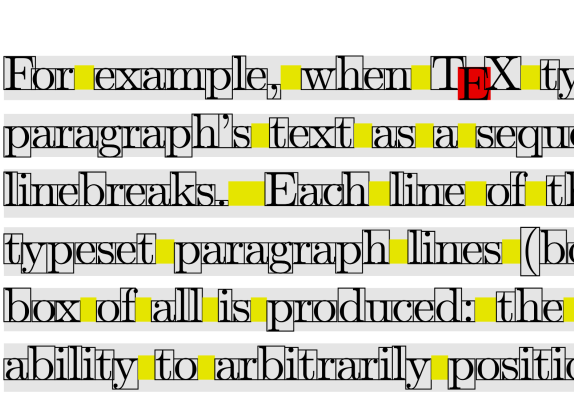
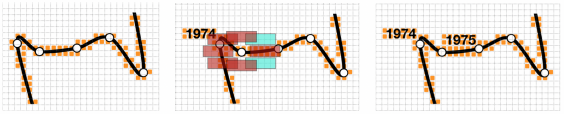


Figure 5. Moving a child subtree.





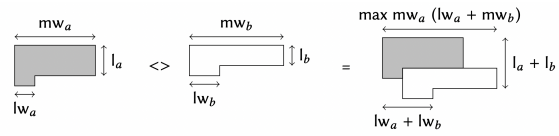


Figure 1: (Left) We rasterize connected scatter plot onto the bitmap to mark occupied pixels, shown in orange. (Middle) We use the 8-position model [5] to generate candidate positions for label placements. The cyan positions are available, while the red ones are not. (Right) After placing the label "1975", the pixels under the label need to be marked as occupied.

Figure 9-2: Bounding box representations for state-of-the-art force-directed graph [6, 30], layered graph edge [25], tree [55], paragraph [34, 1], code [7], and label [33] layout algorithms. These diagrams are from their respective papers or blog posts.

Bibliography

[1] Boxes and glue: A brief, but visual, introduction using luatex.

[2] Ehsan Amid and Manfred K Warmuth. Trimap: Large-scale dimensionality reduction using triplets. *arXiv preprint arXiv:1910.00204*, 2019.

[3] Apple Inc. Swiftui.

[4] Anish Athalye. Experiments in constraint-based graphic design, Dec 2019.

[5] Greg J Badros, Alan Borning, and Peter J Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001.

[6] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[7] Jean-Philippe Bernardy. A pretty but not greedy printer (functional pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–21, 2017.

[8] Enrico Bertini, Michael Correll, and Steven Franconeri. Why shouldn't all charts be scatter plots? beyond precision-driven visualizations. In *2020 IEEE Visualization Conference (VIS)*, pages 206–210. IEEE, 2020.

[9] Alan Blackwell and Thomas Green. Notational systems–the cognitive dimensions of notations framework. *HCI models, theories, and frameworks: toward an interdisciplinary science. Morgan Kaufmann*, 2003.

[10] Alan Borning, Richard Anderson, and Bjorn Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 129–136, 1996.

[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] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011.

[13] Albert S Bregman. *Auditory scene analysis: The perceptual organization of sound*. MIT press, 1994.

[14] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. Improving walker's algorithm to run in linear time. In *International Symposium on Graph Drawing*, pages 344–353. Springer, 2002.

[15] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and direct manipulation, together at last. *ACM SIGPLAN Notices*, 51(6):341–354, 2016.

[16] William S Cleveland and Robert McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American statistical association*, 79(387):531–554, 1984.

[17] Michael B. Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time. *CoRR*, abs/1810.07896, 2018.

[18] S. Chris Colbert and Hein Rutjes. Kiwi.js. `https://github.com/IjzerenHein/kiwi.js`, 2021.

[19] I.F. Cruz and P.S. Leveille. Implementation of a constraint-based visualization system. In *Proceeding 2000 IEEE International Symposium on Visual Languages*, pages 13–20, 2000.

[20] Patrick Dubroy, Saketh Kasibatla, Meixian Li, Marko Röder, and Alex Warth. Language hacking in a live programming environment. In *Proceedings of the LIVE Workshop co-located with ECOOP 2016*, 2016.

[21] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[22] Facebook Open Source. React.

[23] Matthias Felleisen. On the expressive power of programming languages. *Science of computer programming*, 17(1-3):35–75, 1991.

[24] Alberto Gallace and Charles Spence. To what extent do gestalt grouping principles influence tactile perception? *Psychological bulletin*, 137(4):538, 2011.

[25] Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and K-P Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[26] Dimitri Glazkov. Tension between graphs and trees, Feb 2022.

[27] Google LLC. Flutter.

[28] Philip J Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584, 2013.

[29] Andrew Head, Amber Xie, and Marti A Hearst. Math augmentation: How authors enhance the readability of formulas using novel visual design practices. In *CHI Conference on Human Factors in Computing Systems*, pages 1–18, 2022.

[30] Jeffrey Heer. The barnes-hut approximation.

[31] Jane Hoffswell, Alan Borning, and Jeffrey Heer. Setcola: High-level constraints for graph layout. In *Computer Graphics Forum*, volume 37, pages 537–548. Wiley Online Library, 2018.

[32] Gordon Kindlmann and Carlos Scheidegger. An algebraic process for visualization design. *IEEE transactions on visualization and computer graphics*, 20(12):2181–2190, 2014.

[33] Chanwut Kittivorawong, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Fast and flexible overlap detection for chart labeling with occupancy bitmap. In *2020 IEEE Visualization Conference (VIS)*, pages 101–105. IEEE, 2020.

[34] Donald E Knuth and Michael F Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, 1981.

[35] Chris Lattner. Asplos keynote: The golden age of compiler design in an era of hw/sw co-design by dr. chris lattner.

[36] Chris Lattner and Greg Wilson. Llvm.

[37] Guozheng Li, Min Tian, Qinmei Xu, Michael J McGuffin, and Xiaoru Yuan. Gotree: A grammar of tree visualizations. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[38] Geoffrey Litt, Slim Lim, Martin Kleppmann, and Peter van Hardenberg. Peritext: A crdt for rich-text collaboration, 2021.

[39] Chenxi Liu, Enrique Rosales, and Alla Sheffer. Strokeaggregator: Consolidating raw sketches into artist-intended curve drawings. *ACM Transactions on Graphics (TOG)*, 37(4):1–15, 2018.

[40] Nan Liu, Shuang Li, Yilun Du, Josh Tenenbaum, and Antonio Torralba. Learning to compose visual relations. *Advances in Neural Information Processing Systems*, 34, 2021.

[41] Tamara Munzner. *Visualization analysis and design*. CRC press, 2014.

[42] Brad A Myers, Dario A Giuse, Roger B Dannenberg, Brad Vander Zanden, David S Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet comprehensive support for graphical, highly interactive user interfaces. In *Readings in Human–Computer Interaction*, pages 357–371. Elsevier, 1995.

[43] Jorge Poco and Jeffrey Heer. Reverse-engineering visualizations: Recovering visual encodings from chart images. In *Computer Graphics Forum*, volume 36, pages 353–363. Wiley Online Library, 2017.

[44] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Transactions on software Engineering*, (2):223–228, 1981.

[45] Donghao Ren, Bongshin Lee, and Matthew Brehmer. Charticulator: Interactive construction of bespoke chart layouts. *IEEE transactions on visualization and computer graphics*, 25(1):789–799, 2018.

[46] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, 23(1):341–350, 2016.

[47] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics*, 22(1):659–668, 2015.

[48] Frank M Shipman and Catherine C Marshall. Formality considered harmful: Experiences, emerging themes, and directions on the use of formal representations in interactive systems. *Computer Supported Cooperative Work (CSCW)*, 8(4):333–352, 1999.

[49] Hongzhi Song, Edwin P Curran, and Roy Sterritt. Flextree: visualising large quantities of hierarchical information. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 7, pages 6–pp. IEEE, 2002.

[50] Guy L Steele Jr. Growing a language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Citeseer, 1998.

[51] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J Ko. Scout: Rapid exploration of interface layout alternatives through high-level design constraints. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[52] Thomas RG Green. Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay, editors, *People and Computers V*, pages 443–460, Cambridge, UK, 1989. Cambridge University Press.

[53] Barbara Tversky. Spatial schemas in depictions. In *Spatial schemas and abstract thought*, volume 79, page 111, 2001.

[54] Barbara Tversky, Julie Bauer Morrison, and Mireille Betrancourt. Animation: can it facilitate? *International journal of human-computer studies*, 57(4):247–262, 2002.

[55] Atze van der Ploeg. Drawing non-layered tidy trees in linear time. *Software: Practice and Experience*, 44(12):1467–1484, 2014.

[56] J. Wagemans, J. H. Elder, M. Kubovy, S. E. Palmer, M. A. Peterson, M. Singh, and R. von der Heydt. A century of Gestalt psychology in visual perception: I. Perceptual grouping and figure-ground organization. *Psychol Bull*, 138(6):1172–1217, Nov 2012.

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

[58] Leland Wilkinson. The grammar of graphics. In *Handbook of computational statistics*, pages 375–414. Springer, 2012.

[59] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. Penrose: from mathematical notation to beautiful diagrams. *ACM Transactions on Graphics (TOG)*, 39(4):144–1, 2020.

[60] Caroline Ziemkiewicz and Robert Kosara. Laws of attraction: From perceptual forces to conceptual similarity. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1009–1016, 2010.