Bluefish: Composing Diagrams with Declarative Relations

Josh Pollock
MIT CSAIL
Cambridge, MA, USA
jopo@mit.edu

Catherine Mei
MIT CSAIL
Cambridge, MA, USA
meic1212@mit.edu

Grace Huang
MIT CSAIL
Cambridge, MA, USA
gracefh@mit.edu

Elliot Evans
Unaffiliated
Ottawa, Ontario, Canada
vez@duck.com

Daniel Jackson
MIT CSAIL
Cambridge, MA, USA
dnj@mit.edu

Arvind Satyanarayan
MIT CSAIL
Cambridge, MA, USA
arvindsatya@mit.edu

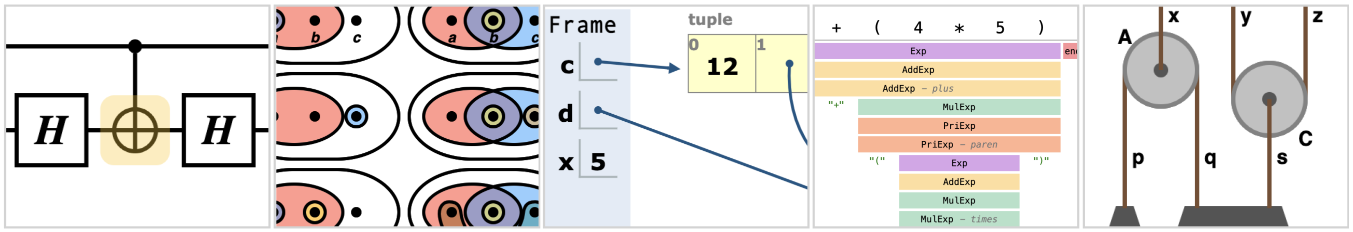


Figure 1: Diagrams built with the Bluefish language. These graphics run the gamut from computer science to physics to math, and are constructed with declarative, composable, extensible relations. From left to right: a quantum circuit equivalence [42], topologies [59], a Python Tutor diagram [34], an Ohm parse tree [23], and a physics pulley diagram [49].

ABSTRACT

Diagrams are essential tools for problem-solving and communication as they externalize conceptual structures using spatial relationships. But when picking a diagramming framework, users are faced with a dilemma. They can either use a highly expressive but low-level toolkit, whose API does not match their domain-specific concepts, or select a high-level typology, which offers a recognizable vocabulary but supports a limited range of diagrams. To address this gap, we introduce Bluefish: a diagramming framework inspired by component-based user interface (UI) libraries. Bluefish lets users create diagrams using *relations*: declarative, composable, and extensible diagram fragments that relax the concept of a UI component. Unlike a component, a relation does not have sole ownership over its children nor does it need to fully specify their layout. To render diagrams, Bluefish extends a traditional tree-based scenegraph to a *compound graph* that captures both hierarchical and adjacent relationships between nodes. To evaluate our system, we construct a diverse example gallery covering many domains including mathematics, physics, computer science, and even cooking. We show that Bluefish's relations are effective declarative primitives for diagrams. Bluefish is open source, and we aim to shape it into both a usable tool and a research platform.

CCS CONCEPTS

• **Human-centered computing** → **Visualization toolkits**; **User interface toolkits**; • **Software and its engineering** → *Constraints*.

KEYWORDS

Diagramming, Domain-Specific Languages, Relations

ACM Reference Format:
Josh Pollock, Catherine Mei, Grace Huang, Elliot Evans, Daniel Jackson, and Arvind Satyanarayan. 2024. Bluefish: Composing Diagrams with Declarative Relations. In *The 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24), October 13–16, 2024, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3654777.3676465

1 INTRODUCTION

Diagrams are essential to problem-solving and communication as they externalize conceptual structures as spatial relationships, thereby aiding recall, inference, and calculation [49, 63, 75]. By representing information in new ways, the best diagrams unlock new ways of thinking about a problem domain — for example, by tracing events as a two-dimensional trajectory through space and time, Feynman diagrams opened "new calculational vistas" and quickly spread to many corners of modern physics [43]. Similarly, citing Dagonet, Latour argued that "no scientific discipline exists without first inventing a visual and written language which allows it to break with its confusing past" [21, 29, 50]. Thus, scientific advances go hand-in-hand with novel diagrammatic notation.

To produce diagrams, authors increasingly turn to programmatic frameworks as these tools enable data-driven diagramming, targeted rendering for different platforms (e.g., as vector graphics for web-based publishing or rasterized images for print-based media), and, more recently, automatic generation via large language models (LLMs). Existing frameworks, however, lie along a spectrum that trades off expressiveness for abstraction. At one end are highly expressive toolkits, such as D3 [15] or p5.js [1], that force authors to grapple with low-level concerns that are often orthogonal to the semantics of their domain-specific diagrams (e.g., manipulating the DOM or issuing Canvas drawing commands). At the other end are high-level typologies, such as Mermaid [76], which offer a recognizable vocabulary of diagrams (flowcharts, sequence diagrams, etc.) but limit authors to only the available diagram types, with only a handful of customization options.

To better balance between expressiveness and abstraction, we introduce *Bluefish*, a diagramming framework inspired by modern component-based user interface (UI) toolkits such as React. The basic building block of UI toolkits, the *component*, offers authors several advantages: UIs can be specified *declaratively*, in terms of what the interface should look like rather than how it should be laid out and rendered; components can be *composed* together (e.g., by nesting them) to express custom UIs; and authors can *extend* the specification language with custom components (e.g., to capture a recurring design pattern and make it reusable). However, the component model imposes limitations when applied to authoring diagrams (Section 3). Components are assembled in tree-based hierarchies. But unlike UI elements, diagrammatic relationships frequently *overlap* — a single element (e.g., a shape) may participate in many visual relationships simultaneously (Figure 1). These relationships cannot be easily expressed in a structure where an element can only have a single parent. As a result, in UI frameworks, diagram authors are forced to adopt low-level workarounds (e.g., manual bounding box calculations) that undo many of the advantages that components offer.

In response, Bluefish relaxes the definition of a component to a *relation* (Section 4.2). A relation, unlike a component, does not have sole ownership over its children nor does it need to fully specify their layout. Rather, a child element can be shared between multiple relations through *scoped declarative references*, and its layout determined jointly by all parents. With these changes, authors can smoothly trade locality for expressiveness (Section 4.3) — opting for a slightly more diffuse specification as it enables a more nimble prototyping process through the design space — without sacrificing the benefits of declarativity, composability, or extensibility.

Authors construct Bluefish diagrams via the JSX syntax extension, which the language runtime compiles into a *compound scenegraph* (Section 5) — an extension of a traditional tree-based scenegraph that captures both hierarchical and adjacency relationships between nodes. In contrast to traditional scenegraphs, compound scenegraphs introduce two challenges for layout: a node's layout may be specified by too few or too many parents. Thus, to handle underconstrained systems, Bluefish *lazily materializes* coordinate transforms (Section 5.2.3) to ensure references can be properly resolved, even if the referent has not yet been fully positioned; to handle overconstrained systems, Bluefish tracks *bounding box ownership* (Section 5.2.4) and notifies the user when relations conflict.

To evaluate Bluefish, we developed a diverse gallery of example diagrams in collaboration with Elliot Evans, a professional creative coder[1] (Section 6). These diagrams span several domains including computer science, topology, physics, and cooking. We evaluate Bluefish's performance on three examples in this gallery and find that Bluefish's layout time scales linearly with the size of the scenegraph (Section 6.3). Additionally, we compare the relational design of Bluefish to the designs of Penrose [90] and Basalt [6], two diagramming frameworks with different approaches to extending UI composition (Section 7). Bluefish's component-inspired abstraction colocates data and display logic while Penrose, inspired by HTML and CSS, groups data and display logic separately. Bluefish uses declarative references that describe spatial relationships while Basalt uses low-level constraints. Finally, we reflect with Evans on Bluefish's abstraction design (Section 8). We find that relaxing the component model provides a shallow learning curve for UI developers and that Bluefish's relational abstraction pushes specifications to be less hierarchical and more diffuse.

Our long-term goal is to make Bluefish both a usable tool and a research platform for investigating graphic representations from diagrams to documents to notation augmentations the way Vega-Lite has done for statistical graphics [71] and LLVM for compilers [51]. To support this goal, we have released Bluefish as an open source project at bluefishjs.org, and we present several promising directions for future research and tool development (Section 9).

2 RELATED WORK

We first discuss the role of relations in diagrams, then we survey existing diagramming languages and environments, and finally we discuss approaches to layout.

2.1 Relations and Diagrams

Relations are central to diagramming. According to James Clerk Maxwell, a diagram is "a figure drawn in such a manner that the geometrical relations between the parts of the figure illustrate relations between other objects" [57]. One salient kind of geometrical relation are *Gestalt relations* [87], a collection of primitive visual relations that associate elements together. Some examples include distributing items with *uniform density*; *aligning* elements along a particular spatial axis; containing elements in a *common region*; and *connecting* elements with lines or arrows. Bluefish's relational standard library corresponds loosely to these relations: `Distribute` to uniform density, `Align` to alignment, `Background` to common region, and `Arrow` and `Line` to connectedness. `Stack` uses a combination of alignment and uniform density.

Several researchers have formally analyzed diagrams in terms of their relations. Richards extends Bertin's retinal variables (shape, color, size, etc.) [10] with Gestalt principles to describe diagrams [65]. Building on this work, Engelhardt proposes a recursive language for diagram analysis [86]. The two have since collaborated on the VisDNA analysis grammar [27]. Larkin and Simon analyze diagrams by constructing formal relational data structures [49]. These approaches have informed Bluefish's own relational formalism, but whereas these frameworks are designed for analysis, Bluefish allows a user to generate a diagram from a formal relational description.

[1]In recognition of his contribution, we include Evans as a co-author on this paper.

2.2 Diagramming Languages and Environments

Direct manipulation editors like Figma, Omnigraffle, and tldraw allow users to align and distribute objects as well as attach arrows so they move when their attached objects are dragged. StickyLines makes alignment and distribution persistent, first-class objects that can be manipulated [18]. Environments like Sketchpad [74], TRIP [78], GLIDE [67], Juno [62], Dunnart [24], Delaunay [20], Subform [2], and Charticulator [64], have integrated persistent relations in some form. Bluefish complements these approaches, because diagramming environments are typically based on a text representation that captures object state and sometimes constraints or relations between them. Text representations are typically better for defining and using abstractions than are direct manipulation environments. Text thus facilitates authoring data-driven diagrams and creating custom, reusable relation abstractions.

Some diagramming languages are restricted to just one or a few diagram types. GoTree supports tree diagrams [53], and SetCoLa [37] and Graphviz [26] support node-link graphs using relations like alignment, spatial proximity, and connectedness. In contrast, Bluefish provides a relational standard library that is applicable across diagram types as well as mechanisms that allow users to customize and extend this set of relations.

General purpose diagramming languages provide various mechanisms for composition. Some UI frameworks, including Garnet [61], Grow [8], and Jetpack Compose [4], extend the component model with an additional concept of a *constraint*. Constraints are declarative equations or inequalities between variables that are solved by a constraint solver. Basalt [6] is a diagramming framework that does this as well. Juno [62] and IDEAL [83] allow users to specify reusable procedures that comprise collections of constraints. More recent diagramming systems have experimented with other abstractions. Haskell diagrams [89] and Diagrammar [33] extend components with *coordinate system modifiers*. For example, Haskell diagrams' `align` function modifies the local origin of a component. Manim [68] is a Python library for making animated diagrams. It extends components with *imperative actions*. For example, Manim provides a `next_to` method for shapes that, while similar to Bluefish's `Stack`, mutates objects. This API is useful for making animations where objects change over time. Penrose [90], a language for mathematical diagrams, uses constraints. But instead of organizing code with components, specifications are split across a `Substance` file and a `Style` file that are inspired by the split between HTML and CSS. Bluefish draws inspiration from these systems' approaches to composition. However, rather than augmenting components with a new constraints concept, Bluefish relaxes the component model to relations. This provides a more consistent representation for relations than previous systems, which enables authors to more smoothly trade locality for expressiveness.

2.3 Diagramming Layout Engines

Laying out a diagram typically involves solving a system of constraints. Some engines use iterative methods that gradually converge on a solution such as Newton-Raphson [62], force-direction [37, 67], gradient descent [24], and L-BFGS [90]. These methods can handle a diverse collection of constraints and can provide best-effort solutions when systems are unsatisfiable. Other systems pick a fixed constraint language and a specialized solver for it. TRIP [78], IDEAL [83], GoTree [53], and Charticulator [64], for example, use linear programming. Basalt [6] lays out diagrams using SMT.

The above solvers are global: they solve an entire constraint system simultaneously. In contrast, local propagation methods flow information incrementally between constraints. UI toolkits such as Garnet [61], Grow [8], and Apogee [36] use this approach. In contrast to global solvers, local solvers are much simpler. Because they solve systems locally, it is easier to debug them when they go wrong. However, they can be less expressive. For example, a global solver can create an equilateral triangle from the constraints that three points are equidistant, but this cycle is not solvable with local propagation. More generally, global solvers tend to be better at continuous, geometric problems while local propagation solvers tend to be better at discrete problems. The UI layout engines in CSS [44], Android's Jetpack Compose [3], and Apple's SwiftUI [52] all employ local propagation strategies that are very similar to each other. Just as we relax components to relations, we similarly generalize modern UI layout architectures.

3 COMPARATIVE USAGE SCENARIOS

To better motivate the need for Bluefish, and the design of its language, we begin with a walkthrough of how an author might construct a simple diagram (Figure 2) with common UI frameworks and compare what the process looks like with Bluefish instead. This diagram depicts a row of the four terrestrial planets, with an annotation on Mercury.

3.1 How UI Components Fail For Diagrams

For this walkthrough, we imagine an *idealized* UI framework. Since React relies on HTML and CSS to perform layout, we borrow component abstractions from SwiftUI and Jetpack Compose and express them with React's syntax for easier comparison with Bluefish.

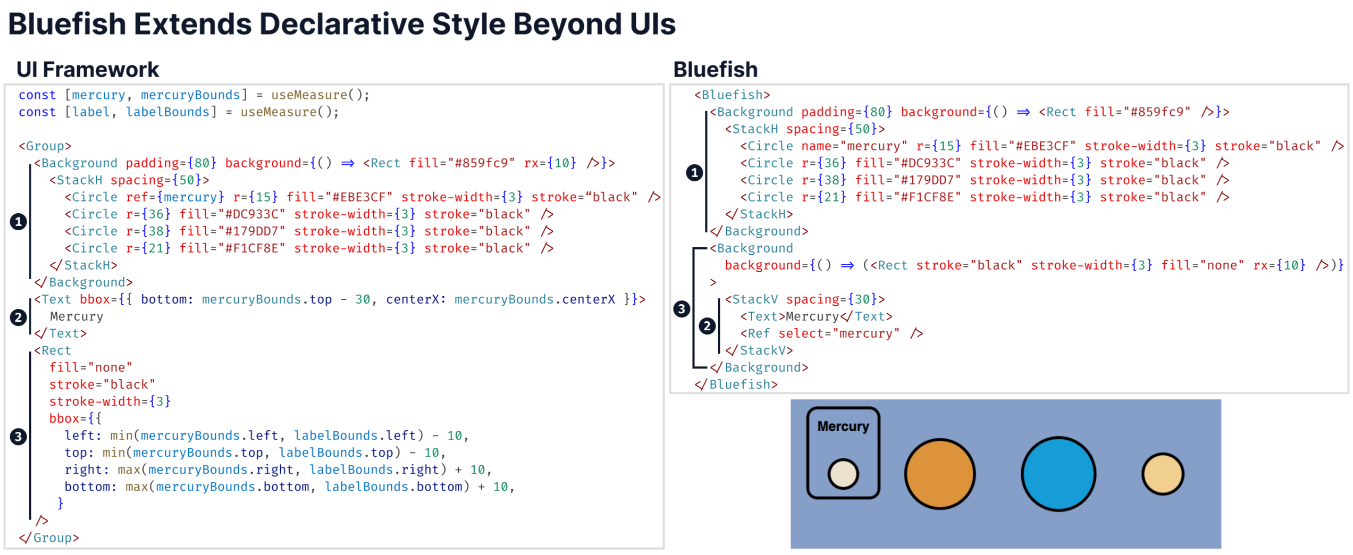
❶ A user might start by creating a `StackH` (a horizontal stack, or row) of `Circle` marks and nest this inside a `Background` component:

```
<Background background={() => ... }>
  <StackH spacing={50}>
    <Circle r={15} fill={"#EBE3CF"} ... />
    <Circle r={36} fill={"#DC933C"} ... />
    <Circle r={38} fill={"#179DD7"} ... />
    <Circle r={21} fill={"#F1CF8E"} ... />
  </StackH>
</Background>
```

❷ But problems quickly arise when they try to annotate Mercury with some text. Ideally, the author should be able to place a `Text` component relative to the planet's position. That way if, for example, the `StackH`'s spacing or layout changed, the `Text` would move with it. However, the planet component is already contained within the `StackH` so it cannot participate directly in any other spatial relationships.[2]

Situations like these can arise when specifying UIs as well (e.g., when placing a tooltip), so UI frameworks provide escape hatches

[2]Note: One might be tempted to group the Mercury text and planet into a new component and use that in the `StackH`. But the Mercury text is longer than the planet, so grouping them together will affect the spacing between the planets.



Bluefish Extends Declarative Style Beyond UIs

UI Framework

```
  const [mercury, mercuryBounds] = useMeasure();
  const [label, labelBounds] = useMeasure();

  <Group>
    <Background padding={80} background={() ⇒ <Rect fill="#859fc9" rx={10} />}>
      <StackH spacing={50}>
        <Circle ref={mercury} r={15} fill="#EBE3CF" stroke-width={3} stroke="black" />
❶       <Circle r={36} fill="#DC933C" stroke-width={3} stroke="black" />
        <Circle r={38} fill="#179DD7" stroke-width={3} stroke="black" />
        <Circle r={21} fill="#F1CF8E" stroke-width={3} stroke="black" />
      </StackH>
    </Background>
    <Text bbox={{ bottom: mercuryBounds.top - 30, centerX: mercuryBounds.centerX }}>
❷     Mercury
    </Text>
    <Rect
      fill="none"
      stroke="black"
      stroke-width={3}
      bbox={{
❸       left: min(mercuryBounds.left, labelBounds.left) - 10,
        top: min(mercuryBounds.top, labelBounds.top) - 10,
        right: max(mercuryBounds.right, labelBounds.right) + 10,
        bottom: max(mercuryBounds.bottom, labelBounds.bottom) + 10,
       }
    />
  </Group>
```

Bluefish

```
  <Bluefish>
    <Background padding={80} background={() ⇒ <Rect fill="#859fc9" />}>
      <StackH spacing={50}>
        <Circle name="mercury" r={15} fill="#EBE3CF" stroke-width={3} stroke="black" />
❶       <Circle r={36} fill="#DC933C" stroke-width={3} stroke="black" />
        <Circle r={38} fill="#179DD7" stroke-width={3} stroke="black" />
        <Circle r={21} fill="#F1CF8E" stroke-width={3} stroke="black" />
      </StackH>
    </Background>
    <Background
      background={() ⇒ (<Rect stroke="black" stroke-width={3} fill="none" rx={10} />)}
    >
❸     <StackV spacing={30}>
        <Text>Mercury</Text>
❷       <Ref select="mercury" />
      </StackV>
    </Background>
  </Bluefish>
```

Figure 2: A comparison of specifying a simple diagram of the four terrestrial planets in a UI framework and in Bluefish. In both cases the user (1) makes a horizontal stack (`StackH`) of `Circle`s contained in a `Background`; (2) places a `Text` component above Mercury; and (3) surrounds the annotation and the planet with a background. In a UI framework, while (1) is declarative, (2) and (3) require error-prone, low-level bounding box computations. In contrast, all three steps are declarative in Bluefish.

to express more complicated layouts. A common escape hatch is a low-level layout or constraint API based on bounding boxes [3, 4, 52]. For the purposes of demonstration, we present this as a hypothetical `useMeasure` hook, akin to those found in React. The `useMeasure` hook is a function that provides a reference that can be assigned to a component and a bounding box object that can be used to read and write dimensions of the referred component. Using this hook, the author could reference the bounding box of the Mercury circle and use it to position the text. They would first introduce a measure for Mercury, consisting of a reference and its bounding box:

```
const [mercury, mercuryBounds] = useMeasure();
```

Then they would assign the `mercury` ref to the `Circle`:

```
<Circle ref={mercury} r={15} ... />
```

Finally, they would use this to compute the position of a new `Text` component:

```
<Text bbox={{ bottom: mercuryBounds.top - 30,
              centerX: mercuryBounds.centerX }}>
  Mercury
</Text>
```

Unlike ❶, this step requires the user to suddenly switch the level of abstraction they are working at: thinking explicitly about bounding boxes. Moreover, the user must remember that the y-axis of the coordinate system points down (they must use −30 not +30) and that they have to offset the `bottom` of the `Text` component from the `top` of the `Circle`, and not vice versa.

❸ These low-level escape hatches color the rest of the specification. Suppose the author now wants to place a `Background` behind the planet and the text to further emphasize their relationship. Again, since the `Circle` and the `Text` components have different parents, they cannot also be children of another `Background` component. Instead the author must again use bounding boxes. They first add a new measure:

```
const [label, labelBounds] = useMeasure();
```

Then they assign it to the `Text`:

```
<Text ref={label} ...>
  Mercury
</Text>
```

And finally they must perform another complicated bounding box computation to set the size of the background so that it contains both the planet and the label:

```
<Rect
  fill="none"
  stroke="black"
  stroke-width={3}
  bbox={{
    left: min(mercuryBounds.left, ...),
    top: min(mercuryBounds.top, ...),
    right: max(mercuryBounds.right, ...),
    bottom: max(mercuryBounds.bottom, ...),
  }}
/>
```

To summarize, UI frameworks can prove to be quite brittle when expressing even very simple diagrams. This is because diagrams often contain relationships that break out of a tree-shaped component hierarchy. As a result users must resort to low-level escape hatches that do not closely match the semantics they want to express.

3.2 Bluefish

Now we consider how the same diagram is authored in Bluefish.

❶ In Bluefish the user begins the same way as in the UI framework, except that their code is contained within a `Bluefish` tag:

```
<Bluefish>
  <Background background={() => ...}>
    <StackH spacing={50}>
      <Circle r={15} fill={"#EBE3CF"} ... />
      <Circle r={36} fill={"#DC933C"} ... />
      <Circle r={38} fill={"#179DD7"} ... />
      <Circle r={21} fill={"#F1CF8E"} ... />
    </StackH>
  </Background>
</Bluefish>
```

This tag demarcates the region of their specification that uses Bluefish's *relations* — a relaxed definition of a UI component. While `Background` and `StackH` appear identically to their UI counterparts, in Bluefish we consider them to be relations and they can be used in more scenarios than before.

❷ Rather than resort to bounding box computations to add the label, the user can use a relation instead. To do this, they first name the Mercury `Circle` so it can be referenced:

```
<Circle name="mercury" r={15} ... />
```

Then they write a `StackV` relation (`StackH`'s vertical counterpart) and select the existing planet element using a `Ref` component:

```
  </Background>
  <StackV spacing={30}>
    <Text>Mercury</Text>
    <Ref select="mercury" />
  </StackV>
</Bluefish>
```

Bluefish provides this special `Ref` component to allow relations to overlap — that is, for the same child element to participate in multiple relations simultaneously. Roughly speaking, `Ref` works as a proxy or stand-in for the element it selects. Since the `StackH` already placed the "mercury" `Circle`, the `StackV` treats it as a fixed element and positions the `Text` mark above it.

Compared to the explicit bounding box computations approach, using Bluefish's relations means the user does not have to remember low-level details like whether the label's bottom or top must be offset from the circle's top or bottom. Moreover, this specification is more declarative and has a closer mapping to the resultant diagram: as the `Text` mark is specified before the `Ref`, the label is vertically stacked *above* the Mercury circle — a relationship they would have had to previously decode from low-level calculations.

❸ To place the rectangle behind the planet and the label, the user can wrap their new `StackV` in a `Background` relation:

```
  </Background>
  <Background background={() => ...}>
    <StackV spacing={30}>
      <Text>Mercury</Text>
      <Ref select="mercury" />
    </StackV>
  </Background>
</Bluefish>
```

Compared to the UI framework approach, Bluefish's relations allow the author to specify this diagram much more *consistently*. A `Stack` is a `Stack` and a `Background` is a `Background` regardless of whether it is used in a conventional hierarchy or by referring to existing elements using `Ref`. This consistency extends the declarative nature of UI specifications to diagrams. As a result, compared to a UI framework, an author can create many more graphics using high-level APIs that closely match their intent.

4 THE BLUEFISH LANGUAGE

Bluefish is a domain-specific language (DSL) embedded in TypeScript comprising a standard library of basic marks and relations; scopes and references for overlapping relations; and helper functions and components for composing relations to *create* new composite marks and relations. Figure 3 lists Bluefish's API.

The key innovation in Bluefish is its *relation* abstraction. To address the limitations we describe in Section 3, relations relax the component model by allowing child elements to be shared across multiple parents via *scoped declarative references*. Moreover, a relation, unlike a component, can leave the sizes and positions of its children underspecified. As a result, the Bluefish relation concept allows users to smoothly trade locality for expressiveness (Section 4.3). By gradually making a specification more diffuse, a user can unlock spaces of atomic edits that they can then rapidly explore to prototype alternate diagram designs.

4.1 Design Goals

Motivated by the diagramming literature and by research on notational affordances, we identify three design goals to support expressive and flexible diagram authoring that UI components already exemplify. To these three, we add a fourth goal specific to diagrams that UI components poorly support.

Declarative. Declarative languages are popular across a range of domains (including web design via HTML/CSS, data querying with SQL, and data visualization using libraries such as Vega-Lite [71] or ggplot2 [88]), because they decouple specification (the *what*) from execution (the *how*). As a result, authors are able to focus on their domain-specific concerns — in our case, expressing the semantics of their diagram — rather than contending with low-level computational and rendering considerations. Declarative specification is particularly important for diagramming as authors come from a variety of disciplinary backgrounds, with varying levels of expertise with reasoning about execution considerations.

Composable. In contrast to diagram typologies (e.g., Mermaid [76]) which offer authors monolithic diagram types to pick between, we aim to achieve a greater expressive gamut by identifying a primitive set of building blocks that authors can combine together to achieve their desired output. UI components support composition through *nesting*: a component can be instantiated within another. This nesting structure is possible because components make few assumptions about the styling, layout, or state of their surrounding context. As a result, this compositional approach also allows authors to reason about their specification in a more localized manner — understanding one part at a time. Locality is especially important for authoring diagrams, which are often complex and non-hierarchical structures.

Extensible. While basic graphical shapes and elements — including rectangles, circles, lines, and text — provide the foundations of UIs and diagrams, they can present a greater *articulatory distance* [41] for expressing the semantics of a diagram than a more domain-specific set of primitives (e.g., "pulleys," "weights," and "ropes" for a physics diagram). As Ma'ayan & Ni et al. [55] and the Cognitive Dimensions of Notations framework [11] describe, it is important that authors have a specification language that has a correspondence [55] or close mapping [11] to the vocabulary of their domain. However, it is impossible for language designers to anticipate every possible primitive for every potential domain. And, even if one could produce such a collection, it would impose an enormous maintenance burden on those designers. Thus, following UI components, Bluefish empowers authors to create domain-specific primitives.

Finally, there is one additional design goal that a diagramming language must satisfy that UI components do not:

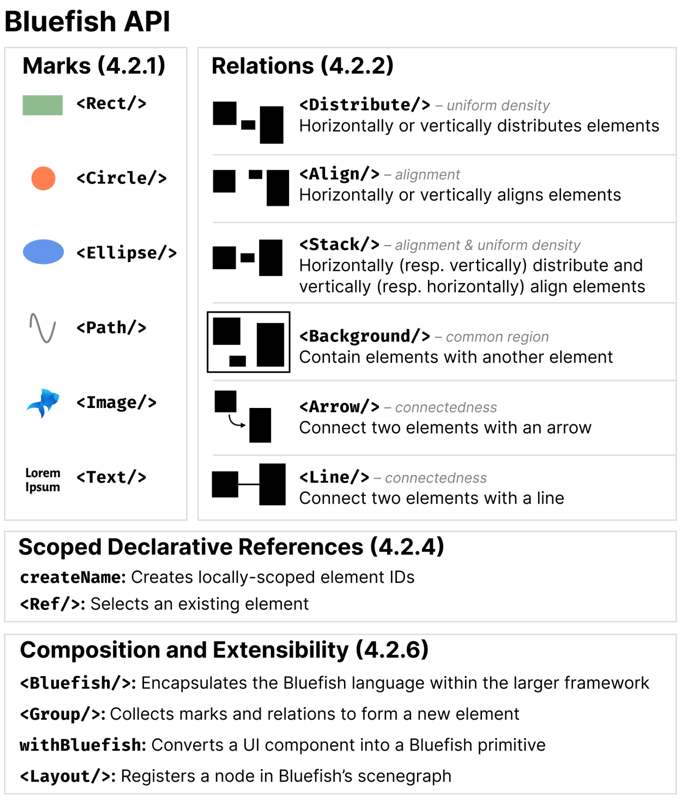
Overlapping. UI components can only relate to one-another via hierarchical nesting. This nesting partitions the visual plane into isolated sections that cannot easily communicate or be visually associated with each other except through a shared ancestor. Partitions help UI components achieve composability, because they can be reasoned about separately. But this trades off the expressiveness we need for diagramming. Diagram elements frequently crosscut a purely hierarchical structure — for example, the Mercury `Circle` in Figure 2 participates in both a horizontal relationship with the other planetary circles and a vertical relationship with its text label. Ideally a diagram author should be able to leverage locality when they can and expressiveness when they must.

4.2 Language Design

4.2.1 Marks. A mark is a basic visual element. Bluefish's mark standard library comprises `Rect`, `Circle`, `Ellipse`, `Path`, `Image`, and `Text`. Marks are thin wrappers around SVG primitives, except for `Text`. `Text` wraps visx's `Text` primitive, which provides better support for text layout than SVG's native `Text`. A mark's position and size arguments are often omitted in a Bluefish specification, because they are determined by relations instead.

4.2.2 Relations. A relation is a visual arrangement of elements that conveys information about an abstract relationship between those elements (e.g., a line connecting two circles represents a chemical bond between two atoms). In addition to marks, relations are the building blocks of diagrammatic representations [49, 57, 66, 79, 86], and Bluefish's design reflects this.

Bluefish reifies the concept of a relation by relaxing UI components, the building blocks of user interfaces. Relations are identical to UI components in many respects. For example, relations can contain zero or more children, be nested arbitrarily, and perform both rendering and layout. But relations relax the component model in two ways. First, whereas components' children are disjoint from other components' children, a relation may share children with other relations. This allows Bluefish elements to relate to other elements via multiple parent relations. Second, while a component must ensure its childrens' sizes and positions are fully determined, a relation can leave some unspecified for other relations to determine. Together, these two relaxations allow Bluefish relations to overlap.



<Distribute/> – *uniform density*
Horizontally or vertically distributes elements

<Align/> – *alignment*
Horizontally or vertically aligns elements

<Stack/> – *alignment & uniform density*
Horizontally (resp. vertically) distribute and vertically (resp. horizontally) align elements

<Background/> – *common region*
Contain elements with another element

<Arrow/> – *connectedness*
Connect two elements with an arrow

<Line/> – *connectedness*
Connect two elements with a line

Scoped Declarative References (4.2.4)

createName: Creates locally-scoped element IDs

<Ref/>: Selects an existing element

Composition and Extensibility (4.2.6)

<Bluefish/>: Encapsulates the Bluefish language within the larger framework

<Group/>: Collects marks and relations to form a new element

withBluefish: Converts a UI component into a Bluefish primitive

<Layout/>: Registers a node in Bluefish's scenegraph

Figure 3: The Bluefish API comprises a standard library of marks and relations as well as a core set of language primitives. Bluefish relations are closely associated with Gestalt relations (listed in gray next to each tag). Scoped declarative references allow users to refer to existing elements. The `Group` relation, `withBluefish` function, and `Layout` component allow users to create new marks and relations.

Bluefish's relations standard library is inspired by Gestalt relations [87]. We provide relations that correspond to uniform density, alignment, common region, and connectedness (Figure 3). We selected Gestalt relations that are commonly found in UI toolkits and design tools. Toolkits like SwiftUI and Jetpack Compose provide components similar to `Stack` and `Background`, but they can only express `Distribute`, `Align`, `Arrow`, and `Line` relations indirectly through modifiers and bounding box calculations, because they do not support overlapping relations. In contrast, Bluefish supports all of these relations using the same abstraction. As a result, Bluefish's API more closely maps to Gestalt theory and allows users to more easily switch between different relations.

4.2.3 Relations Are Expressed with JSX. We surface Bluefish's marks and relations through JSX, an extension to JavaScript popularized by the React library. We map marks and relations to the two kinds of JSX tags: self-closing tags (e.g., `<Circle />`) for marks, and container tags (e.g., `<Arrow>...</Arrow>`) for relations. These tags instantiate elements with zero or non-zero children, respectively. JSX tags take attributes called *props*. For example, `r` and `fill` are two of `<Circle />`'s props.

Our decision to represent relations using JSX instead of as props or vanilla functions has syntactic and semantic consequences. Syntactically, modeling relations as components allows for a *closeness of mapping* [11]: a relation, which groups elements together, is defined by wrapping a container tag around participating elements, generalizing the notion of a grouping in other languages likely to be familiar to diagram authors including HTML (e.g., `<div>` tags) and SVG (i.e., `<g>` tags). Semantically, representing relations as a single, consistent construct means they can be easily swapped for one another. For example, many atomic edits like swapping a `Background` for an `Arrow` or replacing a `StackV` with an `Align` and a `Distribute` take advantage of this consistency (Figure 4).

This representation stands in contrast to other diagramming frameworks. Haskell diagrams, for example, represents a stack using a function, but alignment using a coordinate transform. Similarly, Penrose represents a stack as collection of constraints and a background as a combination of a constraint and a mark. This makes atomic edits much more difficult.

4.2.4 *Scoped Declarative References (`<Ref />`).*

To allow relations to share children, we provide a special `Ref` component that lets a user select an existing element to reuse as the child of another relation. None, some, or all of a relation's children may be `Ref`s.

A `Ref` works like a declarative query selector. A user can reference an element by its *name*. This name is either a globally defined string or scoped locally to the relation using the `createName` function. A user may also specify a *path* of names. To resolve a path selector, Bluefish traverses the path one-by-one, entering a relation each time and searching its local scope for the next named element. Scopes encapsulate names so that changes to names in one relation definition cannot shadow names in another.

We considered using JavaScript's own variable bindings instead of a separate `Ref` component for specifying overlaps. However, we found that this interpretation of bindings competed with users' mental model of JSX: in JSX, using a component bound to a variable in multiple places creates different *copies* of a component rather than *referencing* it, as is needed with Bluefish. Moreover, using explicit `Ref`s simplifies the implementation of the system, because it allows us to construct a relational scenegraph within the confines of a tree-structured hierarchy. Section 5 explains this in more detail. We leave to future work opportunities to expand the expressiveness of how elements may be referenced (e.g., via XML query languages such as XPath [9, 19] and XQuery [13], or by generalizing Cicero's *specifiers* [45] and Atlas's *find* function [54]).

4.2.5 *Relations Are Immutable.*

Because of the hierarchical structure of a UI scenegraph, a component's layout behavior typically depends only on its props, its children, and its parent. As a result, a developer reading a UI codebase usually does not have to look at a component's siblings or cousins to determine the component's behavior. Since Bluefish allows siblings and cousins of a relation to also be its children, this introduces additional dependencies that could break the declarative nature of the component abstraction. To reduce the impact of these non-local dependencies, Bluefish ensures that once some aspect of an elements's size or position, such as its width, has been set by a relation, no other relation may mutate it. As a result, whenever a diagram author sees a relation like `Align`, for example, a user can be confident that `Align`'s children are aligned regardless of other relations in the specification. We discuss how we enforce this property in Section 5.2.

4.2.6 *Marks and Relations Are User-Extensible.*

In addition to authoring diagrams with Bluefish's standard library, users can define new marks and relations in two ways. Firstly, since relations relax components, Bluefish inherits the compositional affordances of the UI framework model and JSX notation. For example, we can write a custom `Planet` mark like so:

```
const Planet = withBluefish((props) => (
  <Circle r={props.radius} fill={props.color}/>
));
```

The mark may then be used like a native tag:

```
<Planet radius={15} color="#EBE3CF" />
```

Any composition of marks and relations may be used as a custom mark, provided its elements have been completely sized and positioned relative to each other. For example, a user might rewrite `Planet` to place a `Background` around the planet as well:

```
const Planet = withBluefish((props) => (
<Background>
  <Circle r={props.radius} fill={props.color}/>
</Background>
));
```

When compositions of existing marks and relations is not enough, Bluefish allows users to author their own primitives with a low-level API. Inspired by the Jetpack Compose API [3], primitive marks and relations are both described using a special `Layout` component that registers a node in Bluefish's scenegraph. In addition to taking a `name` (which may be provided implicitly by Bluefish), `Layout` requires a `layout` function that determines the bounding box and coordinate system of the element and has an opportunity to modify its children's bounding boxes and coordinate systems as well. `Layout` also requires a `paint` function that describes how the element should render given information about its bounding box and its children, which have already been rendered. Here is the basic structure for authoring a new primitive mark and a new primitive relation:

```
const Rect = withBluefish((props) => {
 const layout = () => { ... }
 const paint = (paintProps) => <rect ... />
 return <Layout layout={layout} paint={paint}/>
})
```

```
const Align = withBluefish((props) => {
  const layout = (childNodes) => { ... }
  const paint = (paintProps) => (<g ... >
    {paintProps.children}
  </g>)
  return (
    <Layout layout={layout} paint={paint}>
      {props.children}
    </Layout>
  );
})
```

Bluefish's standard library is written using this API. As a result, it is fully customizable and extensible from user space. We discuss how `layout` functions work in Section 5.2.

4.3 Design Implication: Smoothly Trading Locality for Expressiveness

In addition to extending the declarative component model to more complex graphics, Bluefish allows a user more flexibility to trade locality for expressiveness. Specifically one can make a specification more diffuse by *denesting* or *breaking up* relations. These processes preserve the output diagram while affording new atomic ways to modify the specification. Consider Figure 4.

❶ Starting with the specification from Figure 2, one can already make some atomic edits to explore alternative designs. For example, by swapping the order of the `StackV`'s children, one can move the label below the planet. ❷ By *denesting* the `Background` and `StackV` relations, one can make the specification a little more diffuse. This can be accomplished by naming the `Text` mark, moving the `Background` so it is adjacent to the `StackV` instead of containing it, and making `Background`'s children `Refs` to `StackV`'s children. ❸ This results in a more verbose specification than in ❶. But the advantage is that now the `Background` can be replaced with another relation like `Arrow` simply by swapping the tag. ❹ Finally, for even more expressiveness, one can split `StackV` into two more primitive relations. `StackV` is a compound relation that horizontally *aligns* its children and vertically *distributes* them. Bluefish provides `Align` and `Distribute` so that a user can specify these relations individually. ❺ Splitting `StackV` allows one to retarget `Distribute` at different children while keeping `Align` fixed. For example one can place the label outside the planets `Background` as follows. First, label the `Background`, "planets", and then change the first child of the `Distribute` to select it. This positions the label so that it is still horizontally aligned with Mercury but vertically spaced relative to the planets.

Notice `Align` and `Distribute` cannot be expressed as components in UI frameworks. This is because they only control their childrens' positions on one axis and so those children must have more than one parent to be fully positioned. In SwiftUI and Jetpack Compose, alignment is available as a *guide* argument to components like `HStack` or as a *modifier* on individual elements. Compose also exposes align and distribute *constraints* in special `ConstraintLayout` components. To summarize, Bluefish's relation model allows one to smoothly trade locality for expressiveness. One can make a specification more diffuse by denesting relations and breaking them apart. By doing so, one gains more opportunities to make atomic edits.

5 THE BLUEFISH RELATIONAL SCENEGRAPH

Bluefish is a implemented in SolidJS, a reactive UI framework. Solid provides a JSX component abstraction, signal library, and renderer for Bluefish. We maintain a separate scenegraph and provide a custom layout engine for this scenegraph. When a user composes Bluefish marks and relations, the language runtime compiles this specification to a *relational scenegraph*: a data structure used to resolve references between elements and compute layout. Critically, in contrast to tree-based scenegraphs that are standard in UI and visualization toolkits, Bluefish's relational scenegraph is an instance of a *compound graph*: a data structure that maintains the hierarchical information of traditional scenegraphs while also encoding adjacency relationships between nodes (Section 4.2.4). To compute layout, we adopt the principle of *conservative extension* [28] such that when a relational scenegraph is purely hierarchical its layout behavior is indistinguishable from the behavior of a tree-based scenegraph. This principle allows us to extend the benefits of UI layout runtimes to Bluefish.

5.1 Adapting a Compound Graph Structure

Compound graphs have been explored in research on graph drawing [73] and hierarchical edge bundling [38]. They encode not only *hierarchical* relationships between nodes (i.e., parent-child) but also allow for non-hierarchical relationships called *adjacencies.* In Bluefish, we instantiate a compound graph as follows:

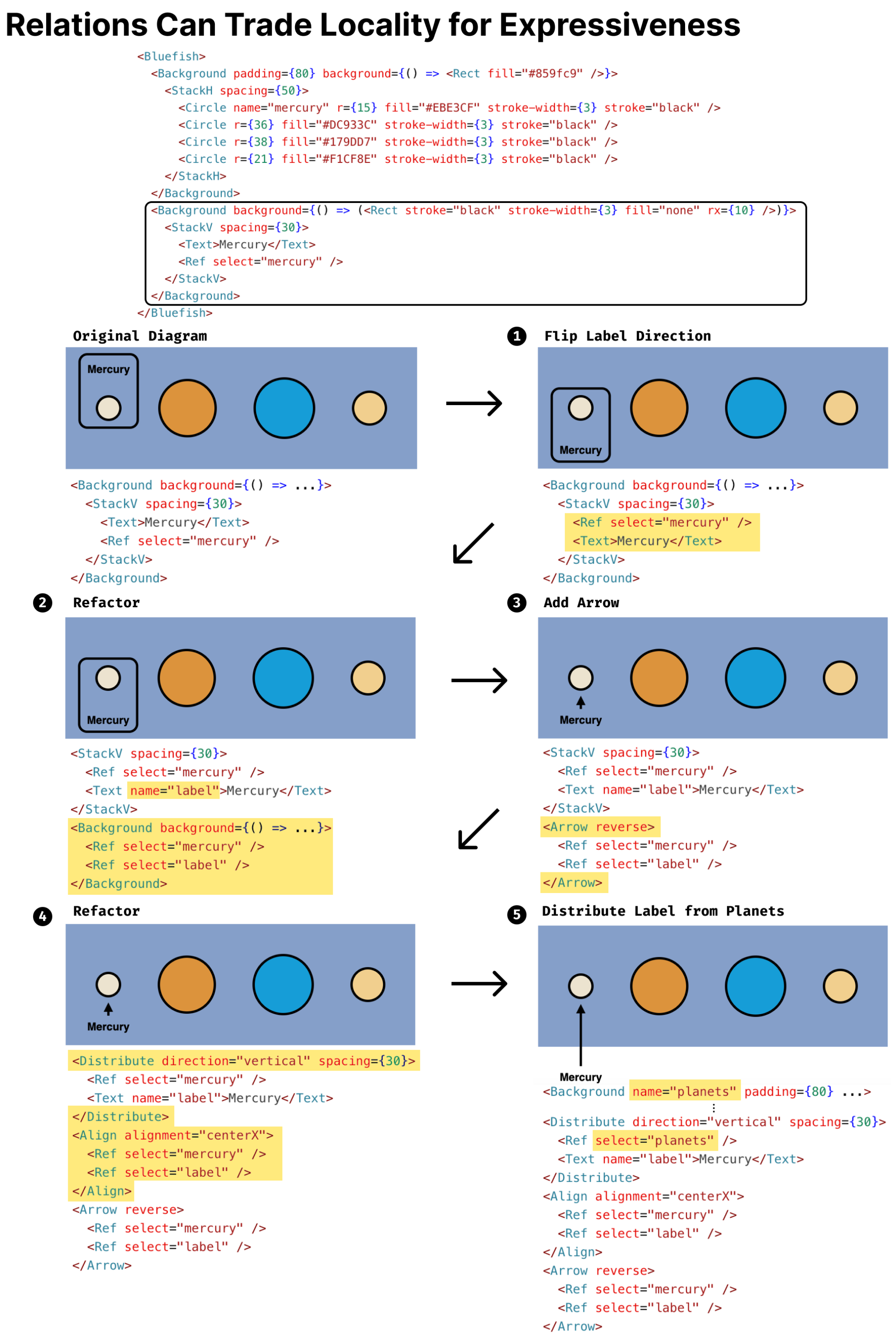
Nodes. Each node in the scenegraph corresponds to a `Layout` or `Ref` tag instantiated in JSX. `Layout` nodes hold information necessary for rendering the corresponding element (e.g., visual styles) as well as computing layout. Layout information includes a partially defined bounding box and any transformations needed to position and size this node based on higher-level nodes.

Hierarchy and Adjacencies. Nodes are assembled into a hierarchy following the nesting structured established by the JSX specification. An adjacency relation is established for every `Ref` element: a node is instantiated in the hierarchy for the `Ref`, and it links to the referenced node as an adjacency. As a result, and unlike general compound graphs where adjacencies can connect any pair of nodes, adjacencies in Bluefish always originate at leaf nodes (i.e., `Refs` are self-closing tags rather than block or container tags). Figure 5 depicts the scenegraph for the UI specification and the relational scenegraph for the Bluefish specification from Figure 2.

5.2 Computing Layout by Conservatively Extending UI Tree-Based Local Propagation

Bluefish extends the layout architecture adopted by modern UI layout engines including those underlying CSS [44], SwiftUI [5, 52], and Jetpack Compose [3]. This architecture is a form of tree-based *local propagation* [72]. Local propagation has a storied history in UI toolkits [8, 14, 36, 60, 61, 69, 70, 84] and is straightforward to implement in reactive dataflow runtimes.

UI layout needs to be fast yet support different, specialized algorithms like flex layout, line-breaking, and grids. To balance performance and expressiveness, UI layouts execute in one pass over the scenegraph, and each node can contain arbitrary code. Each node in the scenegraph has an associated layout algorithm, and layout commences at the scenegraph root. When a node's layout algorithm is evaluated, it invokes the algorithms of its children by proposing a width and height for each child. Once the children are laid out, they return their actual sizes and the parent may place each child in its local coordinate system. This local information-passing approach can express many kinds of layouts. For example, to implement flex layout each child may optionally specify its flex factor. During layout, the parent flex node can read its children's flex factors and distribute its free space proportionally to each child.



```
<Bluefish>
  <Background padding={80} background={() => <Rect fill="#859fc9" />}>
    <StackH spacing={50}>
      <Circle name="mercury" r={15} fill="#EBE3CF" stroke-width={3} stroke="black" />
      <Circle r={36} fill="#DC933C" stroke-width={3} stroke="black" />
      <Circle r={38} fill="#179DD7" stroke-width={3} stroke="black" />
      <Circle r={21} fill="#F1CF8E" stroke-width={3} stroke="black" />
    </StackH>
  </Background>
  <Background background={() => (<Rect stroke="black" stroke-width={3} fill="none" rx={10} />)}>
    <StackV spacing={30}>
      <Text>Mercury</Text>
      <Ref select="mercury" />
    </StackV>
  </Background>
</Bluefish>
```

Original Diagram

```
<Background background={() => ...}>
  <StackV spacing={30}>
    <Text>Mercury</Text>
    <Ref select="mercury" />
  </StackV>
</Background>
```

1 Flip Label Direction

```
<Background background={() => ...}>
  <StackV spacing={30}>
    <Ref select="mercury" />
    <Text>Mercury</Text>
  </StackV>
</Background>
```

2 Refactor

```
<StackV spacing={30}>
  <Ref select="mercury" />
  <Text name="label">Mercury</Text>
</StackV>
<Background background={() => ...}>
  <Ref select="mercury" />
  <Ref select="label" />
</Background>
```

3 Add Arrow

```
<StackV spacing={30}>
  <Ref select="mercury" />
  <Text name="label">Mercury</Text>
</StackV>
<Arrow reverse>
  <Ref select="mercury" />
  <Ref select="label" />
</Arrow>
```

4 Refactor

```
<Distribute direction="vertical" spacing={30}>
  <Ref select="mercury" />
  <Text name="label">Mercury</Text>
</Distribute>
<Align alignment="centerX">
  <Ref select="mercury" />
  <Ref select="label" />
</Align>
<Arrow reverse>
  <Ref select="mercury" />
  <Ref select="label" />
</Arrow>
```

5 Distribute Label from Planets

```
<Background name="planets" padding={80} ...>
                     ⋮
<Distribute direction="vertical" spacing={30}>
  <Ref select="planets" />
  <Text name="label">Mercury</Text>
</Distribute>
<Align alignment="centerX">
  <Ref select="mercury" />
  <Ref select="label" />
</Align>
<Arrow reverse>
  <Ref select="mercury" />
  <Ref select="label" />
</Arrow>
```

Figure 4: Relations can trade locality for expressiveness. (1) With the original spec, one can flip the direction of the label. (2-3) After denesting the `Background` and `StackV` relations one can replace the `Background` with an `Arrow`. (4-5) After breaking up `StackV` into `Align` and `Distribute`, one can space the label relative to the planets background while keeping it aligned with Mercury.

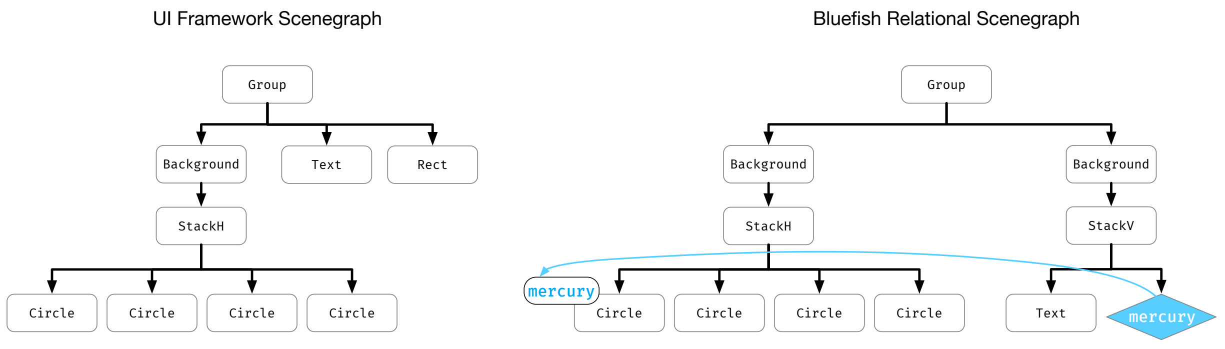


Figure 5: The tree-structured and relational scenegraphs corresponding to Figure 2. Notice that Bluefish's scenegraph retains more information than a tree-structured scenegraph. Bluefish's scenegraph represents the `StackV` and `Background` relations between the label and the planet. In the traditional scenegraph the `StackV` relation between the label and the planet is missing, and the `Background` relation has been reduced to a `Rect` component.

Instead of local propagation, many visualization and diagramming frameworks use a different strategy, a *global solver* [6, 24, 37, 53, 62, 64, 67, 78, 83, 90] such as gradient descent, linear programming, or SMT. Whereas a local solver specifies how values flow through a constraint graph, a global solver specifies a constraint language that all layout constraints must be written in such as differential programs for gradient descent, linear inequalities and an objective function for linear programming, or quantifier-free non-linear real arithmetic for SMT. Because global solvers solve all constraints simultaneously, they can tackle very complex layout problems that cannot be solved by local propagation. Indeed, we implemented an early version of Bluefish using the Cassowary linear programming solver [7]. However, in doing so, we identified a series of tradeoffs at odds with our design goals of composability and extensibility. First, a global solver increases viscosity for diagram authors: it can be difficult to localize layout bugs because the solver reasons about all constraints at once and a node's layout can, by design, be a function of a highly non-local set of inputs. Second, while global solvers increase expressiveness by supporting a larger class of layout problems, they actually limit extensibility: common domain-specific algorithms for domains like trees [82] and graphs [30] rely on custom imperative code that cannot be easily translated to or integrated with a global solver's constraint language. In contrast, Bluefish layout problems can be debugged more easily as layout information only flows locally. Moreover, Bluefish is able to integrate any external layout algorithm simply by instantiating it as a node in the scenegraph. These benefits are extended directly from UI layout architectures.

Local propagation does present some limitations — namely, it does not provide special support for continuous optimization problems or complicated simultaneous constraints. Many International Math Olympiad geometry problems, for example, can only be drawn by solving a circular system of geometry constraints [48]. Diagrams involving knots are well-suited to gradient-descent schemes [91]. Nevertheless, such domain-specific solvers could be embedded as special nodes in the Bluefish architecture. In this way, Bluefish serves more as a layout fabric than a solver itself. It is concerned with the interface between nodes more than the language those nodes' layouts are written in.

Algorithm 1: The layout algorithm for `StackV`

```
   Data: alignment, spacing
XY foreach node ∈ subnodes do node.layout()
Y  y←0
   foreach node ∈ subnodes do
X      x = switch alignment do
           case left do 0
           case centerX do $-$node.width/2
           case right do $-$node.width
       end
XY     node.place(x, y)
Y      y += node.height + spacing
   end
   return {
X  w: maxBy(subnodes, 'width'),
Y  h: sumBy(subnodes, 'height') + spacing · (|subnodes| − 1)
   }
```

5.2.1 A Running Example: Equivalent `StackV` Specifications. To make a framework with low viscosity, we want to support authoring any given graphic representation in many different ways. This property increases the malleability of the language, because a specification can be rewritten into many equivalent forms where each form may be adjacent to different specifications with new meanings. In Section 4.3 we introduced two patterns for rewrites of this kind: denesting relations and splitting them apart. We would like a layout engine where the following three specifications are equivalent.

A purely hierarchical specification:

```
<StackV>
  <Rect width={10} height={20} />
  <Rect width={30} height={10} />
</StackV>
```

A denested specification:

```
<Rect name="a" width={10} height={20} />
<Rect name="b" width={30} height={10} />
<StackV>
   <Ref select="a" />
   <Ref select="b" />
</StackV>
```

A denested specification where `StackV` has been split apart:

```
<Rect name="a" width={10} height={20} />
<Rect name="b" width={30} height={10} />
<Distribute direction="vertical">
   <Ref select="a" />
   <Ref select="b" />
</Distribute>
<Align alignment="centerX">
   <Ref select="a" />
   <Ref select="b" />
</Align>
```

We can work backwards from these equivalences to design a layout semantics that ensures these equivalences as much as possible.

5.2.2 The `StackV` Layout Algorithm. Algorithm 1 (based on one provided by Jetpack Compose [3]), gives pseudocode for `StackV`'s layout algorithm. `StackV` takes as input an `alignment` (`left`, `centerX`, or `right`) and a `spacing` between elements in pixels. It then calls the `layout` algorithm of each of its children who determine their own sizes to be used later. Next, `StackV` places each of its children in its local coordinate space. The `x` coordinate refers to the left edge of the child. The `y` coordinate refers to the top edge of the child and is initialized to 0. Each child is placed horizontally based on the `alignment` parameter. In each `alignment` case, 0 is used as the guideline to which all left edges, horizontal centers, or right edges are aligned.[3] Next, the node is placed and the next top edge is calculated by moving `spacing` pixels below the previous node. This repeats for each child. Finally, the width and height of the `StackV` are returned for use by its own parent.

5.2.3 Lazy Materialization of Coordinate Transforms. To ensure that Bluefish is a conservative extension of UI architectures, layout algorithms like Algorithm 1 must work correctly even when some of their children are references, such as when denesting `StackV`.

When the `layout` method is called on a `Ref` node, it triggers reference resolution to instead return information about the bounding box pointed to by the reference. To resolve a reference, we have to transform the bounding box of the referent into the reference's coordinate frame. We accomplish this by walking the scenegraph between the reference and the referent through their least common ancestor. However, it is often the case that one or more of these intermediate nodes does not have a defined coordinate transform of its own. For example, since the `Ref` children of `StackV` are resolved during `StackV`'s own layout algorithm. `StackV`'s transform is not yet known. In these cases, we must *materialize* intermediate coordinate transforms. Figure 6 depicts lazy coordinate transform materialization during the layout of our running example. When `StackV` attempts to set `a`'s `x` position, its own transform is not

[3]This guideline is in `StackV`'s local space, so the choice of 0 is arbitrary.

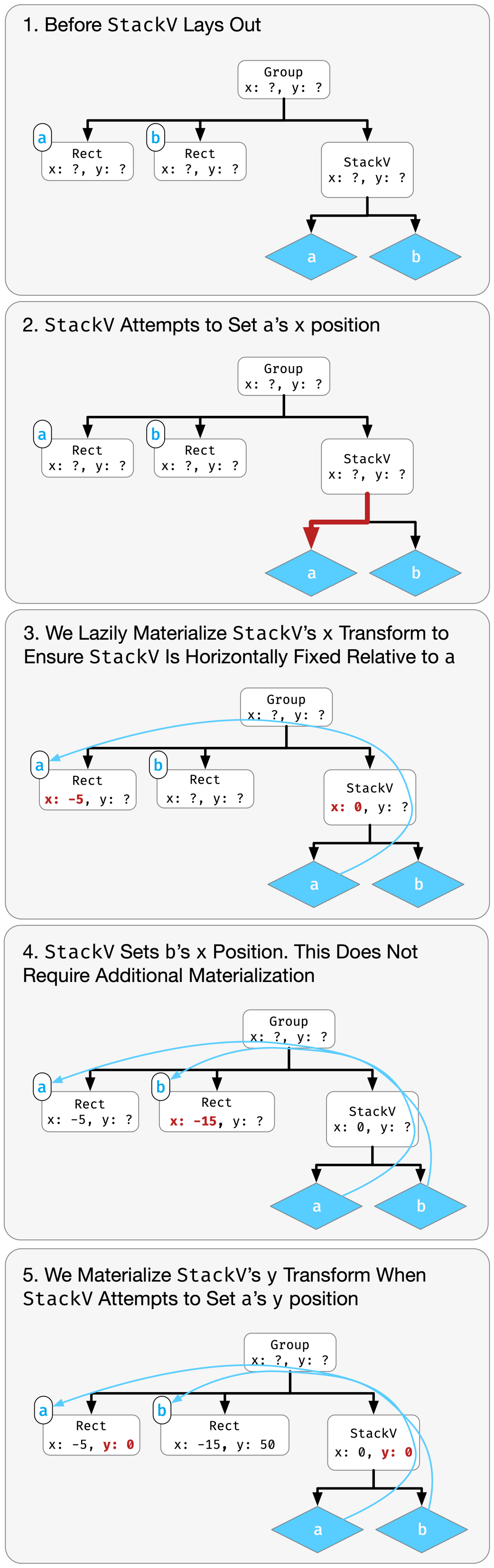


Figure 6: When a bounding box dimension is requested via a `Ref`, intermediate transforms are lazily materialized. This ensures the dimension is well-defined relative to the requester.

yet known. We thus default `StackV`'s `x` transform to the identity transform. Materializing this transform ensures that the horizontal position of `StackV` is fixed relative to `a`, which helps guarantee that all of `StackV`'s children are actually vertically stacked. By deferring transform materialization lazily until a value is requested, we help ensure that the specification is as flexible as possible. If transforms were defaulted eagerly, the position of every object would be fixed before layouts could set them.

```
type BBox<T> = {
  left?: T; centerX?: T; right?: T;
  top?: T; centerY?: T; bottom?: T;
  width?: T; height?: T;
};

type Transform<T> = { translate: { x?: T; y?: T } };

type ScenegraphNode = LayoutNode | RefNode;

type LayoutNode = {                      type RefNode = {
  type: "layout";                          type: "ref";
  bbox: BBox<number>;                      refId: Id;
  transform: Transform<number>;            parent: Id | null;
  children: Id[];                        };
  parent: Id | null;
  bboxOwners: BBox<Id>;
  transformOwners: Transform<Id>;
};
```

Figure 7: The TypeScript type specification for Bluefish's relational scenegraph. Bolded sections are extensions to a typical tree-structured scenegraph. Specifically, we (i) make bounding box and coordinate transform fields optional; (ii) track ownership of individual fields; and (iii) add a `RefNode` type for adjacency relations.

5.2.4 Relaxing Node Ownership to Dimension Ownership. The `StackV` layout in Algorithm 1 can be cleanly separated into a horizontal `Align` and a vertical `Distribute` relation by using the lines labeled `X` and `Y`, respectively, because the logic for each axis are essentially disjoint. Though splitting `StackV`'s layout is straightforward, it creates an architectural problem. In order to split `StackV` in two, we must allow multiple relations to modify a single node.

Typically in a UI layout engine a node is owned by a single parent and only that parent's layout may modify the node. As a result, the relation established by that parent (such as a `StackV`) can never be mutated. This makes UI specifications *declarative*: a relation like `StackV` always corresponds to a vertical stack in the diagram. We want to preserve this correspondence when a node has more than one parent. To account for this multiplicity, instead of a parent node owning an entire child node, a parent node owns specific dimensions of a child node's bounding box. Figure 7 summarizes the modifications to a tree-structured scenegraph datatype required to implement bounding box ownership. Bluefish throws an error if another layout tries to write to a dimension that is already owned, which guards against overconstrained layouts (such as aligning an element to two different elements that have already been placed). Tracking the specific owner (rather than just whether or not a property is owned) allows us to determine the two layouts that conflict. Overconstrained layouts occur frequently when editing a diagram, but problems tend to be easy to localize with access to ownership information and when relations are added one at a time.

6 EXAMPLE GALLERY

To evaluate the strengths and limitations of Bluefish, we constructed a gallery of example diagrams in collaboration with a creative coder Figure 8. As there are no well-established diagram taxonomies, we instead decided to collect diagrams that are highly complex and that run the spectrum of common diagram structures including tables, overlapping containments (e.g. Venn diagrams), trees, graphs, and lists. These examples are inspired by existing diagrams across computer science, physics, math, and cooking. We created them using only the primitives in the Bluefish standard library with a few exceptions where we take advantage of a special `LayoutFunction` relation to sidestep current limitations of the library. Live examples and code are available in the supplemental material.

Table 1 lists the diagrams, their domains, the Gestalt relations they use, and their render times. We first describe two examples in detail (Section 6.1) that we use for our comparisons in Section 7. They illustrate how typical Bluefish specifications are constructed. We then identify three general limitations of our current abstractions that we discovered when creating our gallery (Section 6.2). Finally, we conducted a preliminary performance evaluation by comparing Bluefish to existing baseline implementations of three diagrams from our gallery (Section 6.3). We find that Bluefish scales linearly with the size of its scenegraph. Bluefish is asymptotically faster than Penrose on the *Insertion Sort* diagram and less than ten times slower compared to the original D3-based implementations of the *Python Tutor* and *Ohm Parse Tree* diagrams.

6.1 Selected Examples

6.1.1 Insertion Sort [85, 90]. This diagram traces the steps of the insertion sort algorithm and was originally created for the Penrose [90] example gallery. We compare our specification to Penrose's in Section 7.1. The bordered region represents the sorted part of the array, and the arrow shows the insertion of the next element into the sorted region. In Bluefish we encapsulate this diagram as an element that takes an unsorted array:

```
<InsertionSort
  array={[43, 9, 15, 95, 5, 23, 75]}
/>
```

This diagram is a good example of a deeply nested Bluefish specification built entirely with the standard library primitives. `InsertionSortDiagram` creates a `StackV` of `InsertionSortSteps` then places labels using a series of `StackH` relations:

```
const InsertionSort = withBluefish(props => {
  const insertionSortSteps =
    computeInsertionSortSteps(props.array);
  return (
    <Group>
      <StackV spacing={15}>
        <For each={insertionSortSteps}>
          {(data, i) => (
            <InsertionSortStep
              name={i()}
              stage={i()}
              data={data} />
          )}
```

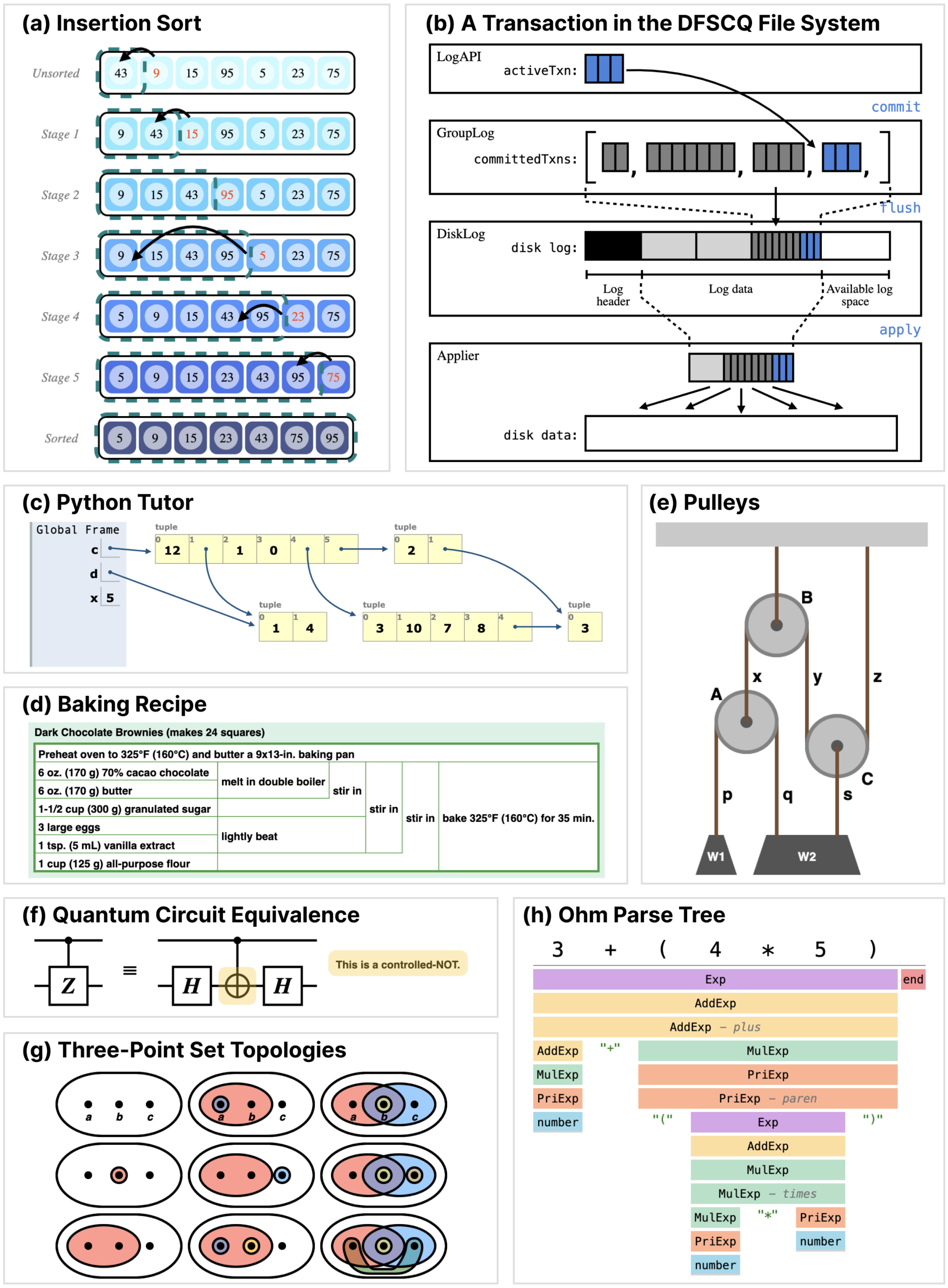


Figure 8: To evaluate Bluefish's expressiveness, we created a diverse example gallery drawn from several domains including (a–c, h) computer science, (d) cooking, (e–f) physics, and (g) math. Code and live examples are available in supplemental materials.

Diagram	Domain	Relations				Render time (ms)
		Alignment	Uniform density	Connectedness	Common region	
(a) Insertion Sort [85, 90]	CS	✓	✓	✓	✓	163.56
(b) DFSCQ File System [6, 16]	CS	✓	✓	✓	✓	155.24
(c) Python Tutor [34]	CS	✓	✓	✓	✓	149.74
(d) Baking Recipe [17]	Cooking	✓	✓	-	✓	99.70
(e) Pulleys [49]	Physics	✓	✓	✓	✓	95.18
(f) Quantum Circuit [42]	Physics	✓	✓	✓	✓	68.99
(g) Three-Point Set Topologies [59]	Math	✓	✓	-	✓	129.58
(h) Ohm Parse Tree [23]	CS	✓	✓	-	✓	174.10

Table 1: The domains, relations, marks, and render times of the diagrams in Figure 8. The examples demonstrate coverage over the four Gestalt relations supported by Bluefish's standard library. All examples run in less than 175ms.

```
          </For>
        </StackV>
        <For each={insertionSortSteps}>
          {(_, i) => (
            <StackH spacing={20}>
              <LabelText>
                {label(i(), props.array.length)}
              </LabelText>
              <Ref select={i()} />
            </StackH>
          )}
        </For>
      </Group>
  ))
```

Like InsertionSort, InsertionSortStep is a custom element. It specifies a StackH of custom ArrayEntry elements inside a Background, then uses a custom DashedBorder relation that specializes Background to surround the sorted region of the array, and finally uses an Arrow relation to show the movement of each element into the sorted region.

6.1.2 DFSCQ File System [6, 16]. This diagram describes the life of a transaction in the DFSCQ file system. The diagram was originally created in Inkscape and recreated in the Basalt diagramming framework to test the limits of its expressiveness [6]. We compare our specification to Basalt's in Section 7.2. As with the *Insertion Sort* diagram, the *DFSCQ File System* diagram's specification uses several custom elements and relations composed of Bluefish standard library primitives. For example, the top level specification consists of a StackV containing four custom TitledBackground relations interspersed with custom ActionText elements like these:

```
<TitledBackground title="LogAPI">
  <StackH>
    <BoxedAlign alignment="centerRight"
                width={200}>
      <Text font-family="monospace"
            font-weight={300}
            font-size="18">
        activeTxn:
      </Text>
    </BoxedAlign>
    <Blocks
      colors={["#4582DE", "#4582DE", "#4582DE"]}
      name="activeTxnBlock" />
  </StackH>
</TitledBackground>
<ActionText text="commit" />
```

TitledBackground is a custom relation composed of Bluefish standard library primitives:

```
const TitledBackground = withBluefish(props => (
  <Align alignment="topLeft">
    <Text font-family="serif" font-weight={300}
          font-size="20" x={10} y={4}>
      {props.title}
    </Text>
    <Background padding={30}>
      <Align alignment="centerLeft">
        <Rect height={0} width={680}
              fill="transparent" />
        {props.children}
      </Align>
    </Background>
  </Align>
))
```

After these have been placed, we place the various lines and labels that cut across this hierarchy including the arrows connecting neighboring TitledBackgrounds. To create the fanned arrows, we create a StackH of invisible Rect marks to represent different regions of the disk and connect each arrow to a different region. We ran into limitations with Bluefish for aligning the widths of the backgrounds across the four rows (Section 6.2). To address this limitation, we used a special LayoutFunction relation for more expressive bounding box constraints.

6.2 General Limitations

We encountered three recurring limitations while making our gallery.

Width and Height Alignment. We needed to align widths and heights of elements in several of our diagrams. For example, in the *Baking Recipe* diagram, the widths of all the backgrounds in the ingredients column must be the same size, and similarly for the backgrounds in the *DFSCQ File System* diagram. At first, aligning sizes may appear to be a straightforward extension of

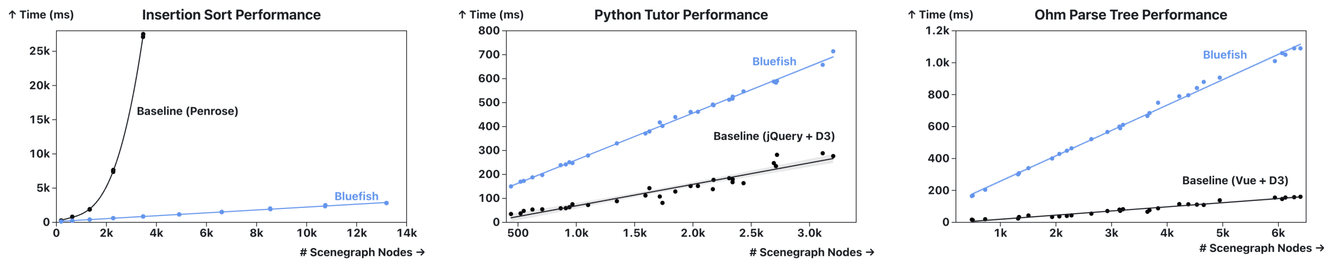


Figure 9: We evaluated Bluefish's performance on three examples against their original implementations. On *Insertion Sort*, Bluefish scales linearly with its scenegraph size whereas Penrose scales superlinearly. Bluefish is roughly 2x and 6x slower than the original D3-based implementations of the *Python Tutor* and *Ohm Parse Tree* diagrams, respectively.

the `Align` relation. But unlike vertical and horizontal positions, which can often be any value, the width and height of an element typically must be large enough to contain their child elements. For example, the `Background` behind each ingredient should not be smaller than the text it contains. Aligning the size of multiple elements therefore requires determining their minimum sizes before performing layout. UI frameworks circumvent this problem by allowing a parent to query its children's preferred sizes before performing layout. We could adopt a similar approach.

Precise Alignment and Spacing. We ran into limitations when specifying more precise element positioning. For example, the Bluefish standard library does not including a `Padding` relation, so we sometimes used a `Stack` with an invisible `Rect` to shift elements. This workaround could be encapsulated in a custom relation. Similarly, `Align`'s vertical alignments — `top`, `centerY`, and `bottom` — are not sufficient for more precise alignments with text. One often wants to align to text's visual baseline rather than the bottom of its bounding box. We worked around this problem by manually nudging text slightly in several examples. UI frameworks provide extensible guideline abstractions for working with text and images. We could adapt these solutions to Bluefish, and we have already prototyped this feature.

Boundary Curve Abstraction. We also ran into limitations of Bluefish's bounding box shape abstraction. For example, when labelling the pulley in the *Pulleys* diagram and the points in *Three-Point Set Topologies*, we manually adjusted the text to avoid intersecting other shapes. Similarly, we manually constructed the concave set in the bottom right of the *Three-Point Set Topologies* diagram to avoid overlapping the purple region. These nudges cannot be done automatically, because Bluefish represents shapes during layout using axis-aligned bounding boxes, which are too coarse for shapes like circles or paths. These examples suggest the need for a boundary curve abstraction. Such an abstraction would allow a user to specify that two shapes should be nested or be made disjoint with greater precision than our bounding box model. It would also support precise labeling along curved lines and arrows. We have made experimental extensions to the system that probe this idea, and we believe this is a promising future approach.

6.3 Performance

Though performance was not the primary focus of Bluefish's design, we conducted a preliminary evaluation to assess the potential impact of Bluefish's expressiveness on performance. Every example in our gallery renders in under 175ms. Because Bluefish executes each layout node once, we hypothesized that Bluefish's performance scales linearly with the number of scenegraph nodes. To test this, we ran the *Insertion Sort*, *Python Tutor*, and *Ohm Parse Tree* diagrams with different input data since they have existing data-driven baseline implementations. The *Insertion Sort* diagram was originally written in Penrose. The *Python Tutor* and *Ohm Parse Tree* diagrams were both originally written in a UI framework using D3. We evaluated these diagrams on an Apple M1 Pro SoC with 32GB of RAM using Chrome Version 126. We used the console's performance analysis to measure the total time required to layout and render each diagram.

Figure 9 visualizes the results of our performance testing. We found that the render time for all three diagrams scaled linearly with the number of nodes in the scenegraph. Compared to Penrose on *Insertion Sort*, Bluefish scales linearly while Penrose scales superlinearly. Bluefish is roughly 2x slower than the original *Python Tutor* implementation and roughly 6x slower than the original *Ohm Parse Tree* implementation. These results suggest the expressiveness of Bluefish's relation abstraction preserves the linear performance scaling of local propagation and UI layout architectures. Future work can improve the constant factor overhead and investigate incremental layout performance to facilitate real-time interaction and animation use cases.

7 COMPARISON TO OTHER COMPOSITIONAL APPROACHES

In this section, we use our selected examples from Section 6.1 to compare Bluefish to two recent diagramming frameworks researchers have developed. This complements the comparison to UI frameworks we conducted in Section 3.

7.1 Penrose: Substance + Style

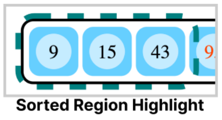
Penrose is a programming language for creating mathematical diagrams [90]. It features three languages: *Substance* for specifying the content of a diagram, *Domain* for defining the content primitives, and *Style* for visualizing the Substance specification.

Bluefish

```
const InsertionSortStep = withBluefish((props) => {
  const { array, move: [from, to] } = props.iterationData;
  const entryNames = array.map((entry) => createName(entry));

  return (
    <Group>
      ...
      <DashedBorder>
        <Ref select={entryNames[0]} />
        <Ref select={entryNames[from - 1]} />
      </DashedBorder>
      ...
    </Group>)})

const DashedBorder = withBluefish((props) => (
  <Background
    padding={8}
    background={() => (
      <Rect fill="none" stroke="teal" stroke-width={4}
            rx={12} stroke-dasharray="12" />)}>
    {props.children}
  </Background>))
```



Penrose

Substance (generated with Python)

```
Array a4        Group p4          Label g22 $9$
Element g22     inGroup(g22,p4)   Label g23 $15$
Element g23     inGroup(g23,p4)   Label g24 $43$
Element g24     inGroup(g24,p4)   Label g25 $95$
Element g25     inGroup(g25,p4)   Label g26 $5$
Element g26     ...               Label g27 $23$
Element g27                       Label g28 $75$
Element g28                       ...
...
```

Style

```
collect Element e into es
where inGroup(e, p)
foreach Group p {
  vectorV groupCenter = [
    average(listof u from es),
    average(listof v from es)
  ]
  p.rect = Rectangle {
    center: groupCenter
    height: maxList(listof v from es) -
            minList(listof v from es) +
            global.boxSize*1.15
    width: maxList(listof u from es) -
            minList(listof u from es) +
            global.boxSize*1.15
    fillColor: colors.none
    strokeColor: colors.green
    strokeStyle: "dashed"
    strokeWidth: 2.8
    cornerRadius: 1
  }
}
```

Figure 10: A comparison between Bluefish and Penrose's specifications of the dashed border in the *Insertion Sort* diagram. Bluefish uses a declarative, component-based approach while Penrose's Style language draws inspiration from CSS and uses low-level bounding box calculations.

7.1.1 Language Design. Figure 10 compares a snippet of the Bluefish and Penrose specifications of the sorted region highlight in the *Insertion Sort* diagram. The Bluefish code uses a custom `DashedBorder` relation that encapsulates a customized `Background` relation. This relation is then used to contain the first and last entry of the sorted region. The Penrose Substance file (generated automatically from a Python script) establishes the elements and relations visualized in the diagram. These include an `Array`, the array's `Element`s, a `Group` of the sorted elements, the `inGroup` relation between elements and the group, and `Label`s for the array elements. The Style program *selects* the elements of the group, collecting them into a variable `es`, and constructs a `Rectangle` that contains them.

The specifications differ primarily in how the code is organized. Bluefish colocates related data (via props) and display logic. Penrose colocates all of the data and all of the display logic in Substance and Style files, respectively. By colocating data and logic, Bluefish allows a user to encapsulate reusable pieces as custom relations like `DashedBorder`. Furthermore, Bluefish encapsulates low-level bounding box calculations behind primitive relations. By separating data and display logic, Penrose allows a user to more easily restyle an entire diagram. For mathematical domains like Euclidean geometry, which have a fixed set of primitive elements and relations, this separation is especially useful. It also frees the Substance language from conforming to a component-based syntax, which allows it to more easily match math notation.

The differences between Penrose and Bluefish stem directly from the inspirations for each system. Penrose's Substance and Style languages are loosely inspired by HTML and CSS, which similarly separate content and display logic into two DSLs. Meanwhile, Bluefish is inspired by UI component frameworks like React, which are specifically designed to couple related HTML, CSS, and JS together [39] as well as take advantage of the expressiveness of a general-purpose host language [40].

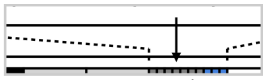
7.1.2 Layout Engine. Penrose's layout engine uses L-BFGS, a global solver. The *Insertion Sort* diagram, while deeply nested, does not contain constraints that show the full power of Penrose. This engine can easily encode constraints that are useful for geometry like ensuring the angles of a triangle are at least 30 degrees, that labels do not overlap, or that arbitrary shapes are contained inside a circle.

As a result of its more powerful layout engine, Penrose can express more perceptual relations than Bluefish including geometric relations like line-line intersection. Bluefish's standard library and internal node abstractions would have to be significantly extended to support these relations. Even then, it would not be easy to incorporate minimum angle requirements into a local propagation solver, because they often must be solved globally. Future work may investigate whether how to integrate global solvers as sublanguages within Bluefish to extend its relational expressiveness.

7.2 Basalt: Components + Constraints

Basalt is a diagramming DSL embedded in Python [6]. It is a modern exemplar of languages that extend a component model with a flexible constraint system [8, 36, 60, 61]. Basalt authors create Python classes similar to UI framework components. However, they can also author *constraints* to relate information between components.

Bluefish



```
const DashedFunnel = withBluefish((props) => {
  const bottomTick1 = createName("bottomTick1");
  const bottomTick2 = createName("bottomTick2");
  const topTick1 = createName("topTick1");
  const topTick2 = createName("topTick2");
  return (
    <Group>
      <StackV spacing={6} alignment="left">
        <Ref select={props.selectTopLeft} />
        <Path name={topTick1} d="M 0 0 L 0 20" stroke-dasharray="5" />
      </StackV>
      <StackV spacing={5} alignment="right">
        <Ref select={props.selectTopRight} />
        <Path name={topTick2} d="M 0 0 L 0 20" stroke-dasharray="5" />
      </StackV>

      <StackV spacing={5} alignment="left">
        <Path name={bottomTick1} d="M 0 0 L 0 20" stroke-dasharray="5" />
        <Ref select={props.selectBottomLeft} />
      </StackV>
      <StackV spacing={5} alignment="right">
        <Path name={bottomTick2} d="M 0 0 L 0 20" stroke-dasharray="5" />
        <Ref select={props.selectBottomRight} />
      </StackV>

      <Line stroke-dasharray="5">
        <Ref select={topTick1} />
        <Ref select={bottomTick1} />
      </Line>
      <Line stroke-dasharray="5">
        <Ref select={topTick2} />
        <Ref select={bottomTick2} />
      </Line>
    </Group>
  );
})
```

```
<DashedFunnel
  selectTopLeft="bigleftbracket"
  selectTopRight="bigrightbracket"
  selectBottomLeft="blocks1"
  selectBottomRight="blocks2"
/>
```

Basalt

```
class Explode(Wrapper):
    def __init__(self):
        style = Style({
            Property.stroke: RGB(0x000000),
            Property.stroke_dasharray: [unit / 3, unit / 8]
        })
        self.top_left = Point()
        self.top_right = Point()
        self.bottom_left = Point()
        self.bottom_right = Point()
        # left
        l1 = Line(self.top_left.x, self.top_left.y,
                  self.top_left.x, self.top_left.y + unit,
                  style=style, name="explode.l1")

        l2 = Line(self.top_left.x,    self.top_left.y + unit,
                  self.bottom_left.x, self.bottom_left.y - unit,
                  style=style, name="explode.l2")

        l3 = Line(self.bottom_left.x, self.bottom_left.y - unit,
                  self.bottom_left.x, self.bottom_left.y,
                  style=style, name="explode.l3")

        # right
        r1 = Line(self.top_right.x, self.top_right.y,
                  self.top_right.x, self.top_right.y + unit,
                  style=style, name="explode.r1")

        r2 = Line(self.top_right.x, self.top_right.y + unit,
                  self.bottom_right.x, self.bottom_right.y - unit,
                  style=style, name="explode.r2")

        r3 = Line(self.bottom_right.x, self.bottom_right.y - unit,
                  self.bottom_right.x, self.bottom_right.y,
                  style=style, name="explode.r3")

        super().__init__(Group([l1, l2, l3, r1, r2, r3]))
explode = Explode()
explode_constraints = Group([explode], [
    explode.top_left.x == blocks.bounds.left,
    explode.top_left.y == blocks.bounds.bottom + unit/2,
    explode.top_right.x == blocks.bounds.right,
    explode.top_right.y == blocks.bounds.bottom + unit/2,
    explode.bottom_left.x == dl_body.disk[-2].bounds.left,
    explode.bottom_left.y == dl_body.disk[-2].bounds.top - unit/2,
    explode.bottom_right.x == dl_body.disk[-2].bounds.right,
    explode.bottom_right.y == dl_body.disk[-2].bounds.top - unit/2,
])
```

Figure 11: A comparison between Bluefish and Basalt's specifications of the dashed funnel lines connecting neighboring rows in the *DFSCQ* diagram. Bluefish uses a more declarative abstraction while Basalt uses low-level constraints.

7.2.1 Language Design. Figure 11 compares a snippet of the Bluefish and Basalt specifications for the dashed funnel linkages in the *DFSCQ* diagram. The Bluefish relation takes as input four names that are used to relatively position the funnel. The Basalt code instead sets up four abstract `Point`s inside the `Explode` component and aligns them to other components using constraints defined outside the component.

The Bluefish specification is more declarative. Rather than using bounding box and point constraints, the Bluefish code uses `StackV`s to position the funnel. The Basalt specification, while lower level, is more malleable. The `Explode` component merely defines `Point`s that represent the corners of the bounding box and `Line`s related to those corners. It leaves the positioning of the corner `Point`s for later. While the constraints could be moved inside the `Explode` component to look more like the Bluefish specification, the Bluefish specification must explicitly take in the corners as dependencies. To allow Bluefish to move the corner dependencies outside, we would have to introduce a way to position elements relative to their enclosing container and have those relations run only after the container's size has been set.

This example highlights a viscosity tradeoff between the two systems. In Bluefish, authoring specifications is more high-level as a user can think in terms of relations like `Stack` and `Line`. However, extending Bluefish with new kinds of primitive relations often requires stepping down to the low-level layout API. On the other hand, systems like Basalt are more viscous for end-users, because they must deal with constraints. But creating custom constraints is much more straightforward, because the user is already working with an expressive, low-level API. Future work may explore whether the locality-expressiveness tradeoffs we highlighted in Section 4.3 could be extended to the level of bounding box and point constraints.

7.2.2 Layout Engine. Basalt uses Z3 [22], an SMT solver, to construct solutions to constraint problems. Z3 is very expressive and can handle circular constraints and nonlinear inequalities. This expressiveness leads to some of the low-viscosity properties of Basalt's design. However, nearly all of the constraints used to create the *DFSCQ* diagram are sparse linear equations similar to the ones shown in Figure 11. As a result, it may be possible to achieve the same functionality with a local propagation system like Bluefish's.

8 DESIGN REFLECTIONS WITH A PROFESSIONAL CREATIVE CODER

We built the example gallery in collaboration with a professional creative coder, Elliot Evans. Except for the Python Tutor diagram, Evans directed the implementation of the examples. Through discussions during and after building these examples, we surfaced two main insights about Bluefish's abstractions.

8.1 Relations provide a shallow learning curve for UI developers

In addition to providing conceptual simplicity, relaxing the component model makes Bluefish easier to understand for UI developers. Evans found that programming in Bluefish without using relations was very similar to using Tailwind. For example, consider Tailwind's `flex-row` specification versus Bluefish's `StackH`:

```
<div class="flex flex-row gap-5">
   <div>1</div>
   <div>2</div>
   <div>3</div>
</div>
```

```
<StackH>
   <Text>1</Text>
   <Text>2</Text>
   <Text>3</Text>
</StackH>
```

Evans first familiarized himself with Bluefish by using it as a UI layout engine only, without `Ref`. He then learned Bluefish's relations concept through a bridge example much like our example in Figure 2. Specifically, he started building the *Quantum Circuit Equivalence* diagram solely using nested hierarchies before placing the `Background` highlight using a `Ref`. After using `Background`, a traditional UI component, in an overlapping context, Evans used the `Line` relation, which has no direct UI component analog.

8.2 Bluefish specifications often move from hierarchical to diffuse

While Bluefish specifications often start like UI specifications — compact and hierarchical — Evans observed that his diagram specifications typically became more diffuse and relational over time. We demonstrate this pattern in Section 4.3.

Through language design conversations with Evans, we realized this behavior stems from our choice to unify different Gestalt relations with a shared abstraction. Relations can be composed to create new elements, which means relations must have bounding boxes. But while some relations (e.g., `Background`) have bounding boxes that are easy to define, the bounding boxes of other relations like `Arrow` are more ambiguous. Should `Arrow`'s bounding box contain the bounding boxes of the elements it connects? To facilitate easy and predictable switching between different relations, we decided that all relations' bounding boxes should contain their children. While this approach lowers editing viscosity, it requires users to denest specifications earlier than they may expect.

For example, consider the *Python Tutor* diagram. It depicts program state of a running Python program. In Bluefish we construct a top-level element that accepts a description of the stack and heap:

```
<PythonTutor
   stack={[
      { variable: "c", value: pointer(0) },
      { variable: "d", value: pointer(1) },
      { variable: "x", value: "5" }]}
   heap={[
      tuple("1", pointer(1), pointer(2)),
      tuple("1", "4"),
      tuple("3", "10")]}
   heapArrangement={[
      [0, null, null],
      [null, 1, 2]]}
/>
```

Each `pointer` corresponds to an `Arrow` in the diagram. We initially wanted to place the `Arrow` relations corresponding to stack pointers inside the stack element. This nesting would mirror the data structure driving the visualization. However, the `Arrow` relation's bounding box contains the heap object it points to. The `Arrow` must therefore be denested out of the stack element or else the stack would contain the `Arrow` and thus the heap object. The tradeoff of this early denesting is that switching the `Arrow` relation for a `Background` is a predictable, atomic edit. If the `Arrow` were nested and its bounding box did not contain its children, the user might be surprised that switching it to a `Background` would suddenly include a heap object in the stack.

9 DISCUSSION AND FUTURE WORK

In this paper, we presented Bluefish, a diagramming framework based on relations that is declarative, composable, and extensible. We have demonstrated how relaxing the component model transfers the benefits of UI framework design to diagramming without the need for completely new concepts like constraints. Relations allow users to smoothly trade the local affordances of hierarchical specification for the expressive affordances of adjacency. Our long-term goal is to make Bluefish both a usable tool and a research platform for investigating graphic representations from diagrams to documents to notation augmentations the way Vega-Lite has done for statistical graphics [71] and LLVM for compilers [51]. To support this goal, we have released Bluefish as an open source project at bluefishjs.org, and present several promising directions for future research and tool development.

Interactive and Animated Graphic Representations. In this paper, we explored formalisms of relations for *static* graphics. An immediate next step would be to consider how our abstractions could be extended to interactive and animated diagrams. First, there are temporal analogs to static Gestalt relations. For example, *common fate*, where elements travel in the same direction are grouped together, is alignment applied to velocity [87]. Similarly, we could think of Bluefish's `Distribute` as distributing elements along a time axis to stagger movements in time, and a temporal `Align` as unifying the start or end of multiple animations. Data visualization grammars have explored these temporal analogs. For example Gemini provides `concat` and `sync` operations for temporal distribution and alignment, respectively [46]. Animations may also be staged or nested, conveying information similar to *common region*, as in Canis/CAST [31, 32]. Or an animation may follow a path between two elements to represent a temporal *link* between them. There are analogs in interaction as well. Gestalt relations seem to manifest in interactions as *on-demand* relations. For example, brushing can be thought of as on-demand common region, and generalized selections [35] allow users to select sets of elements based on *similar attributes.*

Formalizing Visual Structure and Domain Semantics. While our standard library of relations covers a large number of use cases, many domains have different sets of primitive relations. For example, Euclidean geometry features relations like line-line intersection and perpendicular bisector. Similarly, specific features of a line convey semantic intent in sketched route maps [81]. A straight line means *"go down,"* a curved line means *"follow around,"* and a line with a sharp corner can signify a *"turn."* These primitives tie closely to the underlying semantics of the domains they visualize, synthetic planar geometry and routes, respectively. Mackinlay's expressiveness principle[56], Tversky's correspondence principle [80], and Kindlmann and Scheidegger's algebraic design process [47] suggest we may find many such mappings between graphics and domain semantics. The core idea underlying Bluefish is that more powerful formalisms of these correspondences not only lower authoring viscosity, but also capture more underlying semantic information for later analysis and processing.

Towards Richer Tools for Graphic Representations. Developing these formal mappings also enables more powerful tools for end-users. For instance, how might Bluefish's scenegraph — which explicitly encodes relationships between elements — be automatically retargeted for screen reader use, blending approaches found in tools such as Olli [12], which produces a hierarchical structure for navigating statistical graphics, and Data Navigator [25], which provides methods for navigating adjacency structures? Similarly, while diagramming environments such as StickyLines [18] have reified alignment and distribution, Bluefish's relations suggest the possibility of a more general, consistent interface for allowing end-users to directly manipulate Gestalt relations. Tools like Draco [58] and Scout [77] have explored automatic recommendations of statistical graphics and UIs, respectively, based on studies from the perceptual literature. By providing an explicit encoding of relations at the language level, we believe Bluefish can serve as the base for exploring diagramming recommendations based on the relative effectiveness of Gestalt relations.

ACKNOWLEDGMENTS

Thanks to Josh Horowitz for pointing us to modern UI local propagation layouts; to Tom George for his help on a previous iteration of the project; and to our anonymous reviewers for their thoughtful feedback that clarified our contributions. This work is supported by the National Science Foundation under Grant No. 1745302.

REFERENCES

[1] [n. d.]. p5.js. https://github.com/processing/p5.js.

[2] [n. d.]. Subform | Dynamic layout meets direct manipulation — subformapp.com. https://subformapp.com/.

[3] 2023. Custom layouts | Jetpack Compose. https://developer.android.com/jetpack/compose/layouts/custom.

[4] 2024. ConstraintLayout in Compose | Jetpack Compose. https://developer.android.com/jetpack/compose/layouts/constraintlayout.

[5] Dave Abrahams and John Harper. 2019. Building Custom Views with SwiftUI - WWDC19 - Videos - Apple Developer — developer.apple.com. https://developer.apple.com/videos/play/wwdc2019/237 and https://developer.apple.com/documentation/swiftui/building_custom_views_in_swiftui.

[6] Anish Athalye. 2019. Experiments in constraint-based graphic design. https://www.anishathalye.com/2019/12/12/constraint-based-graphic-design/

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

[8] Paul S Barth. 1986. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics (TOG)* 5, 2 (1986), 142–172.

[9] Michael Benedikt and Christoph Koch. 2009. XPath leashed. *ACM Computing Surveys (CSUR)* 41, 1 (2009), 1–54.

[10] Jacques Bertin. 1983. *Semiology of graphics.* University of Wisconsin press.

[11] Alan F Blackwell, Carol Britton, Anna Cox, Thomas RG Green, Corin Gurr, Gada Kadoda, Maria S Kutar, Martin Loomes, Chrystopher L Nehaniv, Marian Petre, et al. 2001. Cognitive dimensions of notations: Design tools for cognitive technology. In *Cognitive Technology: Instruments of Mind: 4th International Conference, CT 2001 Coventry, UK, August 6–9, 2001 Proceedings.* Springer Berlin Heidelberg, 325–341.

[12] Matt Blanco, Jonathan Zong, and Arvind Satyanarayan. 2022. Olli: An extensible visualization library for screen reader accessibility. *IEEE VIS Posters* 6 (2022).

[13] Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. 2002. XQuery 1.0: An XML query language. (2002).

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

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

[16] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles.* 270–286.

[17] Michael Chu. 2006. Dark Chocolate Brownies - Recipe File - Cooking For Engineers — cookingforengineers.com. https://www.cookingforengineers.com/recipe/158/Dark-Chocolate-Brownies.

[18] Marianela Ciolfi Felice, Nolwenn Maudet, Wendy E Mackay, and Michel Beaudouin-Lafon. 2016. Beyond snapping: Persistent, tweakable alignment and distribution with StickyLines. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology.* 133–144.

[19] James Clark, Steve DeRose, et al. 1999. XML path language (XPath).

[20] I.F. Cruz and P.S. Leveille. 2000. Implementation of a constraint-based visualization system. In *Proceeding 2000 IEEE International Symposium on Visual Languages.* 13–20. https://doi.org/10.1109/VL.2000.874345

[21] François Dagognet. 1973. Écriture et iconographie. (1973).

[22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 337–340.

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

[24] Tim Dwyer, Kim Marriott, and Michael Wybrow. 2009. Dunnart: A constraint-based network diagram authoring tool. In *Graph Drawing: 16th International Symposium, GD 2008, Heraklion, Crete, Greece, September 21-24, 2008. Revised Papers 16.* Springer, 420–431.

[25] Frank Elavsky, Lucas Nadolskis, and Dominik Moritz. 2023. Data Navigator: An accessibility-centered data navigation toolkit. arXiv:2308.08475 [cs.HC]

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

[27] Yuri Engelhardt and Clive Richards. 2021. A universal grammar for specifying visualization types. In *Diagrammatic Representation and Inference: 12th International Conference, Diagrams 2021, Virtual, September 28–30, 2021, Proceedings 12.* Springer, 395–403.

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

[29] Dagognet François. 1969. Tableaux et langages de la chimie: Essai sur la représentation.

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

[31] Tong Ge, Bongshin Lee, and Yunhai Wang. 2021. Cast: Authoring data-driven chart animations. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* 1–15.

[32] Tong Ge, Yue Zhao, Bongshin Lee, Donghao Ren, Baoquan Chen, and Yunhai Wang. 2020. Canis: A High-Level Language for Data-Driven Chart Animations. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 607–617.

[33] Pontus Granström. [n. d.]. *Diagrammar: Simply Make Interactive Diagrams.* YouTube. https://www.youtube.com/watch?v=gT9Xu-ctNqI

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

[35] Jeffrey Heer, Maneesh Agrawala, and Wesley Willett. 2008. Generalized selection via interactive query relaxation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* 959–968.

[36] Tyson R Henry and Scott E Hudson. 1988. Using active data in a UIMS. In *Proceedings of the 1st annual ACM SIGGRAPH symposium on User Interface Software.* 167–178.

[37] Jane Hoffswell, Alan Borning, and Jeffrey Heer. 2018. SetCoLa: High-Level Constraints for Graph Layout. In *Computer Graphics Forum*, Vol. 37. Wiley Online Library, 537–548.

[38] Danny Holten. 2006. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on visualization and computer graphics* 12, 5 (2006), 741–748.

[39] Honeypot. [n. d.]. *How A Small Team of Developers Created React at Facebook | React.js: The Documentary*. YouTube. https://www.youtube.com/watch?v=8pDqJVdNa44&t=2451s

[40] Pete Hunt. [n. d.]. *Pete Hunt: React: Rethinking best practices – JSConf EU*. YouTube. https://www.youtube.com/watch?v=x7cQ3mrcKaY

[41] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct manipulation interfaces. *Human–computer interaction* 1, 4 (1985), 311–338.

[42] Dintomon Joy, M Sabir, Bikash K Behera, and Prasanta K Panigrahi. 2020. Implementation of quantum secret sharing and quantum binary voting protocol in the IBM quantum computer. *Quantum Information Processing* 19 (2020), 1–20.

[43] David Kaiser. 2005. Physics and Feynman's Diagrams: In the hands of a postwar generation, a tool intended to lead quantum electrodynamics out of a decades-long morass helped transform physics. *American Scientist* 93, 2 (2005), 156–165.

[44] Ian Kilpatrick. 2022. CSS Layout API Explained. https://github.com/w3c/css-houdini-drafts/blob/main/css-layout-api/EXPLAINER.md.

[45] Hyeok Kim, Ryan Rossi, Fan Du, Eunyee Koh, Shunan Guo, Jessica Hullman, and Jane Hoffswell. 2022. Cicero: A declarative grammar for responsive visualization. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–15.

[46] Younghoon Kim and Jeffrey Heer. 2020. Gemini: A grammar and recommender system for animated transitions in statistical graphics. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2020), 485–494.

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

[48] Ryan Krueger, Jesse Michael Han, and Daniel Selsam. 2021. Automatically Building Diagrams for Olympiad Geometry Problems.. In *CADE*. 577–588.

[49] Jill H Larkin and Herbert A Simon. 1987. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science* 11, 1 (1987), 65–100.

[50] Bruno Latour. 1986. Visualization and cognition: Thinking with eyes and hands. *Knowledge and society: Studies in the sociology of culture past and present* 6, 1 (1986), 1–40.

[51] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE, 75–86.

[52] Paul Lettieri. 2022. Compose custom layouts with SwiftUI - WWDC22 - Videos - Apple Developer — developer.apple.com. https://developer.apple.com/videos/play/wwdc2022/10056/ and https://developer.apple.com/documentation/swiftui/composing_custom_layouts_with_swiftui.

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

[54] Zhicheng Liu, Chen Chen, Francisco Morales, and Yishan Zhao. 2021. Atlas: Grammar-based Procedural Generation of Data Visualizations. In *2021 IEEE Visualization Conference (VIS)*. IEEE, 171–175.

[55] Dor Ma'ayan, Wode Ni, Katherine Ye, Chinmay Kulkarni, and Joshua Sunshine. 2020. How domain experts create conceptual diagrams and implications for tool design. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–14.

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

[57] James Clerk Maxwell. 1911. Diagram.

[58] Dominik Moritz, Chenglong Wang, Gregory Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2019). http://idl.cs.washington.edu/papers/draco

[59] James R Munkres. 1999. *Topology* (2 ed.). Pearson, Upper Saddle River, NJ.

[60] Brad A Myers. 1991. Graphical techniques in a spreadsheet for specifying user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 243–249.

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

[62] Greg Nelson. 1985. Juno, a constraint-based graphics system. In *Proceedings of the 12th annual conference on Computer Graphics and Interactive Techniques*. 235–243.

[63] Don Norman. 2014. *Things that make us smart: Defending human attributes in the age of the machine*. Diversion Books.

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

[65] Clive Richards. 1984. *Diagrammatics*. Ph. D. Dissertation. Royal College of Art. Available from: diagrammatics.com.

[66] Clive Richards. 2002. The fundamental design variables of diagramming. *Diagrammatic representation and reasoning* (2002), 85–102.

[67] Kathy Ryall, Joe Marks, and Stuart Shieber. 1997. An interactive constraint-based system for drawing graphs. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*. 97–104.

[68] Grant Sanderson. 2018. Manim. https://github.com/ManimCommunity/manim.

[69] Michael Sannella. 1994. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*. 137–146.

[70] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. 1993. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software: Practice and Experience* 23, 5 (1993), 529–566.

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

[72] Guy Lewis Steele Jr. 1980. *The Definition and Implementation of a Computer Programming Language Based on Constraints*. Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB.

[73] Kozo Sugiyama. 2002. *Graph drawing and applications for software and knowledge engineers*. Vol. 11. World Scientific.

[74] Ivan E Sutherland. 1963. Sketchpad: A man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, spring joint computer conference*. 329–346.

[75] Masaki Suwa and Barbara Tversky. 2002. External representations contribute to the dynamic construction of ideas. In *Diagrammatic Representation and Inference: Second International Conference, Diagrams 2002 Callaway Gardens, GA, USA, April 18–20, 2002 Proceedings 2*. Springer, 341–343.

[76] Knut Sveidqvist and Sidharth Vinod. 2014. Mermaid. https://github.com/mermaid-js/mermaid.

[77] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J Ko. 2020. 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*. 1–13.

[78] Shin Takahashi, Satoshi Matsuoka, Ken Miyashita, Hiroshi Hosobe, and Tomihisa Kamada. 1998. A constraint-based approach for visualization and animation. *Constraints* 3 (1998), 61–86.

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

[80] Barbara Tversky. 2019. *Mind in motion: How action shapes thought*. Hachette UK.

[81] Barbara Tversky, Jeff Zacks, Paul Lee, and Julie Heiser. 2000. Lines, blobs, crosses and arrows: Diagrammatic communication with schematic figures. In *Theory and Application of Diagrams: First International Conference, Diagrams 2000 Edinburgh, Scotland, UK, September 1–3, 2000 Proceedings 1*. Springer, 221–230.

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

[83] Christopher J Van Wyk. 1982. A high-level language for specifying pictures. *ACM Transactions on Graphics (TOG)* 1, 2 (1982), 163–182.

[84] Bradley T Vander Zanden, Richard Halterman, Brad A Myers, Rich McDaniel, Rob Miller, Pedro Szekely, Dario A Giuse, and David Kosbie. 2001. Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 6 (2001), 776–796.

[85] Vennobennu. 2024. feat: Add array-manipulation diagram to gallery · Pull Request #1716 · penrose/penrose — github.com. https://github.com/penrose/penrose/pull/1716.

[86] Jörg von Engelhardt. 2002. *The language of graphics: A framework for the analysis of syntax and meaning in maps, charts and diagrams*. Yuri Engelhardt.

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

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

[89] Ryan Yates and Brent A Yorgey. 2015. Diagrams: a functional EDSL for vector graphics. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*. 4–5.

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

[91] Chris Yu, Henrik Schumacher, and Keenan Crane. 2021. Repulsive curves. *ACM Transactions on Graphics (TOG)* 40, 2 (2021), 1–21.