The Shapes of Abstraction in Data Structure Diagrams

Devamardeep Hayatpur
Department of Cognitive Science and Design Lab
University of California, San Diego
La Jolla, California, USA
dshayatpur@ucsd.edu

Brian Hempel
Department of Cognitive Science and Design Lab
University of California San Deigo
La Jolla, California, USA
bhempel@ucsd.edu

Richard Lin
University of California, Los Angeles
Los Angeles, California, USA
richardlin@ucla.edu

Haijun Xia
Department of Cognitive Science and Design Lab
University of California, San Diego
San Diego, California, USA
haijunxia@ucsd.edu

Abstract

Tools to inspect runtime state, like print statements and debuggers, are an essential part of programming. Yet, a major limitation is that they present data at a fixed, low level of abstraction which can overload the user with irrelevant details. In contrast, human drawings of data structures use many illustrative visual abstractions to show the most useful information. We attempt to bridge the gap by surveying 80 programmer-produced diagrams to develop a mechanical approach for capturing visual abstraction, termed *abstraction moves*. An abstraction move selects data objects of interest, and then revisualizes, simplifies, or annotates them. We implement these moves as a diagramming language for JavaScript code, named *Chisel*, and show that it can effectively reproduce 78 out of the 80 surveyed diagrams. In a preliminary study with four CS educators, we evaluate its usage and discover potential contexts of use. Our approach of mechanically moving between levels of abstraction in data displays opens the doors to new tools and workflows in programming education and software development.

CCS Concepts

• **Human-centered computing** → *Information visualization.*

Keywords

programming, abstraction, graphical representations

ACM Reference Format:
Devamardeep Hayatpur, Brian Hempel, Richard Lin, and Haijun Xia. 2025. The Shapes of Abstraction in Data Structure Diagrams. In *CHI Conference on Human Factors in Computing Systems (CHI '25), April 26–May 01, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3706598.3713723

1 Introduction

Understanding code is hard. We have to simulate, in our mind, how a machine would execute each instruction; how data will be transformed across the life cycle of the program's execution. To help us, we rely on a variety of mechanical aids, like print statements and debuggers, to see the runtime state. In these tools, we see data in a concrete display. As an example, while debugging, we might see an array update from values `[3,2,1,4,3,8,0,1,9]` to `[0,3,2,1,4,3,8,1,9]`. Then, it is up to us to deduce the abstract behavior of the code from these values, that is, the smallest element has moved to the front. Understanding the behavior from this display is overwhelming. It both (a) *shows too much* as most of the array is irrelevant to the operation, and (b) *does not show enough* as the points of interest, like the smallest element, are not called out. In other words, it does not display the most useful information.

In contrast, *human* notations of data, like drawings and diagrams, make use of illustrative abstractions [15]. We might have drawn the array update as: `[3,2,…0`$_{(min)}$`…]` to `[0`$_{(min)}$`,3,2,…]`, abbreviating superfluous elements and labeling the smallest value '*min*' to illustrate that it is has been pushed to the front.

This work seeks to bridge the gap in abstraction between concrete displays of data in programming tools and conceptual drawings. We conducted a content analysis of 80 programmer-produced diagrams sampled from real-world practice and instruction. We deduced step-by-step ways in which these diagrams are abstracted compared to a counterpart concrete display. We term these steps *abstraction moves*, which are comprised of two parts: a *selection* that describes the piece of data being abstracted, and one of three abstraction strategies: a *revisualization* (*e.g.* changing the visual layout from a list to a grid), *simplification* (*e.g.* abbreviating the data), or *annotation* (*e.g.* putting a label on part of the data).

We then used the discovered abstraction moves to design a JavaScript-based diagramming language, named *Chisel*, for creating data structure diagrams through incremental shifts in abstraction. For example, to construct the earlier notation of the array, we can first *select ranges* of the concrete data `[3,2,1,4,3,8,0,1,9]` and *abbreviate* them, `[3,2,…,0,…]`, then *select the minimum element*, `[3,2,…,0,…]` and *label* it: `[3,2,…,0`$_{(min)}$`,…]`. The advantage of creating diagrams through *Chisel* is that it can:

(1) *be dynamic to changes in the data,* if the array is updated, the process can be followed again, and the diagram would reflect the update (*e.g.* `[5,1,…,-1`$_{(min)}$`,…]`).
(2) *be flexible across levels of abstraction,* we can easily add and take away moves (*e.g.* no longer elide the start, `[3,2,1,4,3,8,0`$_{(min)}$`,…]`).

ACM ISBN 979-8-4007-1394-1/25/04
https://doi.org/10.1145/3706598.3713723

(3) *provide a visible record of abstraction,* it describes assumptions about what is considered essential and non-essential about the data, which can then be scrutinized, adapted, and revised.

Using *Chisel*, we were able to reproduce visual abstractions in 78/80 diagrams sampled in our content analysis, and in a preliminary study with three computer science instructors and one teaching assistant, we found encouraging initial results on usability of *Chisel* and potential contexts of use.

2 Related Work

Visual abstraction transforms information into visual representations by abstracting away its idiosyncrasies to simplify, highlight, and summarize it [38]. Below, we survey the use of visual abstractions in programming and existing solutions to authoring diagrams.

2.1 Roles of Visual Abstraction in Programming

Programmers use visuals prolifically: they draw their software's design, data structures, code, and its execution [3, 8, 20, 30, 42], and instructors rely on diagrams such as *"list as a sequence of boxes," "control flow as a graph," "memory as a stack," etc.* [16]. Petre [35] finds that programmers omit information to reason across designs, and Mangano et al. [30] find that software designers shifted between levels of abstraction in their sketches to *"focus on particular aspect of the design by omitting non-relevant details."* Fong et al. [17] studied diagrams of linked lists made by a community of Youtubers, some of whom shifted between different levels of abstraction. These works document the use of diagrams at varying levels of abstraction, but do not provide a clear vocabulary to operationalize visual abstraction, which this work aims to contribute.

2.2 Visualizations of Runtime State

Interfaces to inspect runtime state, like traditional debuggers, read-eval-print-loop (REPLs), and print statements, are essential tools for programmers to understand and write code. Program visualization tools like Python Tutor [19], Projection Boxes [26], and CrossCode [21] automatically provide richer visualizations of runtime values during execution. However, across these tools, data is shown at a fixed low-level of abstraction: concrete values are shown in full detail, with little possibility of customization.

Abstracted displays of program output are under-explored, and thus, their design space is unclear. To our knowledge, two prior works on interactive systems which use abstract displays exist: Curry [11]'s PAD where users record executions on abstract data, and McDirmid [32]'s abstract notation for manipulating sequences. Outside of interactive systems, Wilhelm et al. conducted a series of projects to automatically infer and show invariants of linked data structures using static analysis, facilitating focus on relevant aspects of the data [6, 34, 40]. These works do not cover the whole suite of common data structures, nor was their design grounded in the notations and abstractions programmers use in practice.

2.3 Data Visualization Frameworks

General purpose editors, like Figma, Inkscape, and Illustrator, enable manual assembly of diagrams with direct manipulation on primitive marks and shapes on a canvas. However, graphical editors can quickly become tedious for making, updating, and exploring design alternatives [29]. To mitigate this tedium, programmatic visualization frameworks facilitate diagramming by, *e.g. scene manipulation*, like D3 [5], or with *declarative* specifications, like Bluefish [36]. Penrose [43], a diagramming tool for mathematical diagrams separates domain knowledge from the visual representation, which allows for reusable and extendable representations. The closest prior system to our target usage is Lau and Guo [25]'s Data Theater, which creates explorable explanations of Python code by mapping runtime values onto graphical objects using a declarative specification.

The salient difference between *Chisel* and existing visualization frameworks is that *Chisel* is designed around diagrams of data structures and abstractions on them. To achieve the same outcome in, *e.g.*, Penrose, a user would need to build a domain and style library that implements Chisel itself. The relationship between Chisel and Bluefish or D3 is similar. A more subtle difference is that *Chisel* uses mutating operations to gradually arrive at a diagram from a concrete display of data, which implies that diagrams cannot be encoded directly at the appropriate level of abstraction. We speculate that for illustrating conceptual ideas which are not parameterized by concrete data, our approach will provide little benefit. Instead, a *Chisel* program specifies a design space of diagrams that cuts across multiple levels of abstractions down to the concrete, which can prove helpful when concrete data is meaningful (*e.g.* when debugging) or when concrete values can scaffold understanding. We further sketch out appropriate places to use *Chisel* in subsection 4.3.

3 Abstraction Moves

To discover an initial set of abstraction moves—*i.e.* visual operations that incrementally transform a concrete display of data to a diagrammatic representation—we conducted a content analysis of a wide-ranging corpus of data structure diagrams.

Table 1: List of diagram sources. Count denotes the number of total diagrams collected from each source.

Source	Use Case	Modality	Count
The Linux Kernel [1]	Codebase	ASCII	57
Chromium	Codebase	ASCII	17
RFCs 1–500[2]	Design Standard	ASCII	69
Algorithm Design [24]	Textbook	Graphics	16
Intro. to Algorithms [10]	Textbook	Graphics	87
Crafting Interpreters [33]	Textbook	Graphics	66
Software Design by Example [41]	Textbook	Graphics	34
MIT Intro. to Algorithms [13]	Lecture	Drawing	40
UW Applied Algorithms [2]	Lecture	Graphics	86
UIUC Algorithms [14]	Lecture	Drawing	60

3.1 Methodology

3.1.1 Data Collection. We collected a corpus of diagrams from ten real-world sources spanning different use cases and modalities (Table 1). We limited our scope to pictures of common data

[1]The ASCII diagrams from Linux and Chromium use data collected in [20].
[2]The first 500 IETF RFCs [23], which are technical internet standards from 1969—1993.

structures: numbers, strings, lists, graphs, records, pointers, and combinations of them. More specialized representations like those embedded in Euclidean space (*e.g.* plots, or geometric shapes) were filtered out. Repeated visualizations of the same style from the same source were also filtered out. Our goal was not to produce counts or frequency statistics, but to have a diverse sample of ordinary data structure representations to describe visual abstraction.

3.1.2 Sampling. Twenty diagrams were sampled at a time, two from each of the ten sources. To strike a balance between uniform sampling that reflect the diagrams in our dataset as well as to capture their variance, we sampled one diagram randomly and one selected by the first author to prioritize types of diagrams that have not yet been analyzed.

3.1.3 Coding. The first author annotated each diagram by annotating its data structure and the moves required to move from a plausible concrete display[3] to the diagram (*e.g.* "hide all values in the list"). After each round of coding, the remaining authors reviewed the annotations, and the first author iterated on the codebook based on their feedback. The process was concluded when no more variations were observed ($N = 80$). The first author then performed a round of deductive coding by revisiting the older codes, and re-annotating them with the derived codebook. The outcome of this analysis provided the broad framework and categories of abstraction moves. (Appendix A includes examples of diagrams annotated with the codebook.)

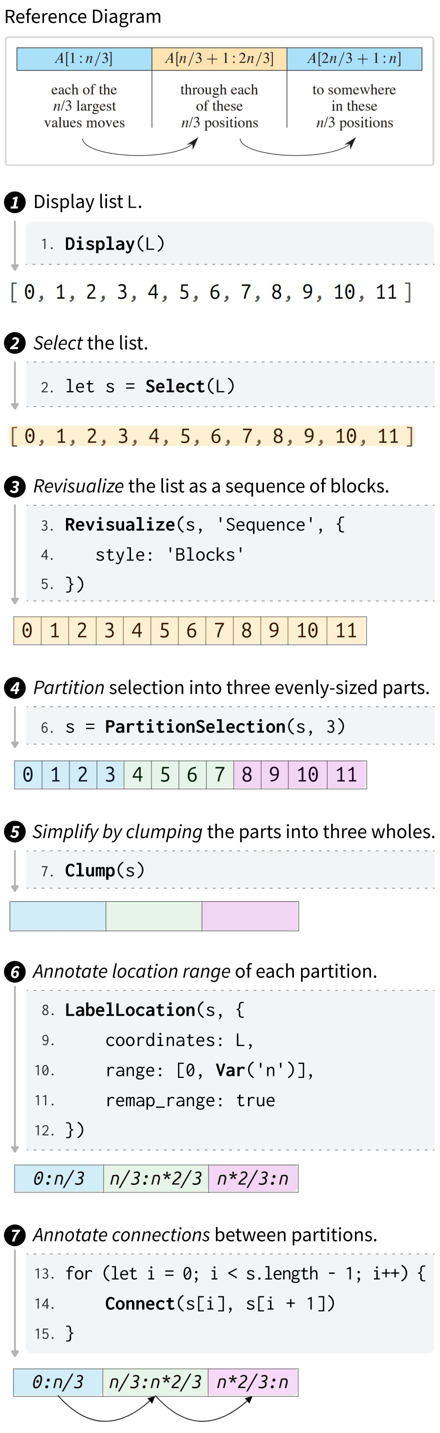
3.1.4 Formalizing. We implemented the abstraction moves discerned in the codebook as operations in a JavaScript-based diagramming language, *Chisel*, for visualizing runtime state. We constructed the prototype inductively by reproducing each of the 80 diagrams in the content analysis and maintained a close mapping between the qualitative codes and the operations in the language. As such, we were able to (a) evaluate if the basic categories in the content analysis and our syntax are logically consistent and expressive, (b) capture idiosyncrasies not detailed in the qualitative description but required when actually visualizing diagrams (*e.g.* visual styles), and (c) apply the framework generatively to create new diagrams.

3.2 Building Diagrams with Abstraction Moves

To demonstrate how abstraction moves can be used to specify a diagram, Figure 1 provides an illustrated workflow of partially recreating a diagram from our corpus of array partitions (Figure 1, top). Specifically, the diagram displays a list divided into groups of $n/3$ (where n is the length of the list), with each group labeled with its location. We start with a generic display ❶, and then:

- ❷ ...*select* the list; selections are shown with a colored overlay,
- ❸ ...*revisualize* the diagram to be shown as blocks instead of comma-separated-values,
- ❹, ❺ ...mutate the selection by *partitioning* it, and then *simplify* the partitions by clumping together the values into three larger sections,
- ❻, ❼ ...*annotate* locations of those sections onto the array and add connections between the sections.

[3]The first author made an educated guess to underlying representation of concrete data using the context surrounding the diagram.


Reference Diagram
A[1 : n/3]
A[n/3 + 1 : 2n/3]
A[2n/3 + 1 : n]
each of the n/3 largest values moves
through each of these n/3 positions
to somewhere in these n/3 positions

❶ Display list L.

```
Display(L)
```

```
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ]
```

❷ *Select* the list.

```
let s = Select(L)
```

```
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ]
```

❸ *Revisualize* the list as a sequence of blocks.

```
Revisualize(s, 'Sequence', {
   style: 'Blocks'
})
```

❹ *Partition* selection into three evenly-sized parts.

```
s = PartitionSelection(s, 3)
```

❺ *Simplify by clumping* the parts into three wholes.

```
Clump(s)
```

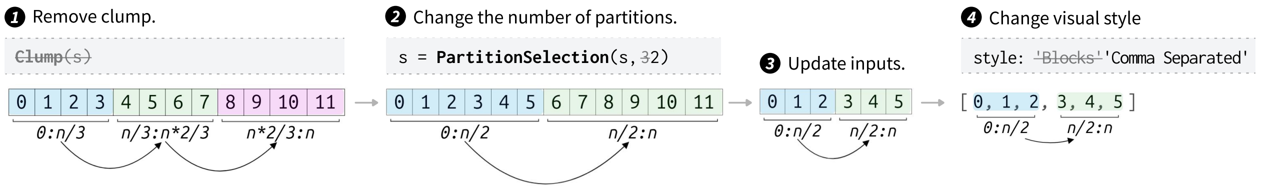
❻ *Annotate location range* of each partition.

```
LabelLocation(s, {
    coordinates: L,
    range: [0, Var('n')],
    remap_range: true
})
```

❼ *Annotate connections* between partitions.

```
for (let i = 0; i < s.length - 1; i++) {
    Connect(s[i], s[i + 1])
}
```

Figure 1: A partial recreation of a diagram from Cormen et al. [10]'s Introduction to Algorithms (top), where it was used to illustrate the worst-case lower bound of insertion sort. We assume that the list is stored in a variable L, and its length stored in n.



~~Clump(s)~~

s = **PartitionSelection**(s, ~~3~~2)

style: ~~'Blocks'~~'Comma Separated'

Figure 2: Variations on array partitions in Figure 1: ❶ removes a simplification; ❷ updates the number of partitions to two; ❸ halves the length of the input list; and ❹ updates the initial visual style to be a comma separated list instead of blocks.

Having reconstructed the diagram through abstraction moves, we can navigate between levels of abstraction (by adding, removing, and varying moves), as well as different states of data (Figure 2).

3.3 Overview of Abstraction Moves

Below we describe abstraction moves and *Chisel*'s syntax.

3.3.1 Display. To initially render a diagram, *Chisel* uses **`Display`**(d), which takes data as input and adds a diagram to the scene. It supports ordinary data structures: primitives (strings, numbers, booleans, nulls), combinations of them (arrays and objects), and variables.

$$\textbf{Data } d := num \mid bool \mid string \mid null \mid array \mid object \mid variable$$

`Display` provides visuals resembling a typical debugger...

(1) ...*primitives* are shown with their value,
(2) ...*arrays* are shown as horizontal comma separated values, if it contains one or more objects, then it is shown as a vertical list labeled with its indices,
(3) ...*objects* are shown as an indented hierarchy,
(4) ...*variables* are shown with their name above the their value.

3.3.2 Selections. Selections are ways to refer to subsets of data for later revisualizing, simplifying, or annotating. A selection can be to a data value (*e.g.* first item of a list), a contiguous span (*e.g.* a row of items in a grid), or to a union of other selections (*e.g.* the first and last item in a list).

$$\textbf{Selection } sel := \text{data selection} \mid \text{span selection} \mid \text{union selection}$$

The basic selection operator accepts multiple data values to be selected: **`Select`**$(d_1, d_2, \dots)$. The selections can then be *merged* into contiguous spanning selections using **`SpanSelection`**(sel). We also provide selections to select subparts of data, which, along with the basic selection strategies, are illustrated in Figure 3 Ⓐ.

3.3.3 Revisualizations. We categorized data structures being visualized as graphs, sequences, and grids:

(1) A node link *graph*, used to show data structures such as graphs or a pointer data structure. As specific cases of graphs, *trees* and indented *hierarchies*, can visualize objects like binary trees.
(2) *Sequences*, to visualize ordered collections, *e.g.* arrays, strings, bits, and memory layouts.
(3) *Grids*, to visualize 2D collections, matrices, or data tables.

Chisel provides **`Revisualize`**(sel, graph | tree | hierarchy | sequence | grid), to revisualize a selection under different visual forms. Supported revisualizations are illustrated in Figure 3 Ⓑ.

Data formats. For data structures where JavaScript does not define a format, *Chisel* provides a canonical format...

(1) ...*graphs* can be made from (a) objects with both `vertices` (a list of objects) and `edges` (a list of vertex index pairs) attributes or (b) constructed as a pointer graph from any key-value object or an array.
(2) ...*trees* can be made from objects with a `children` (a list of objects) attribute, or attributes `left` and `right`.
(3) ...*grids* can be created from two dimensional arrays or from tabular data (*i.e.* a list of objects which share attributes).

Arranging sub-diagrams. Some diagrams in our corpus used layouts within diagrams, *e.g.*, aligning two arrays in parallel. *Chisel* supports this in an ad-hoc way by collecting the sub-diagrams to be organized into a larger data structure and revisualizing it. This basic approach accommodates the diagrams in our content analysis—but in practice these arrangements could be specified more explicitly by *e.g.* using relational constraints as in Bluefish [36].

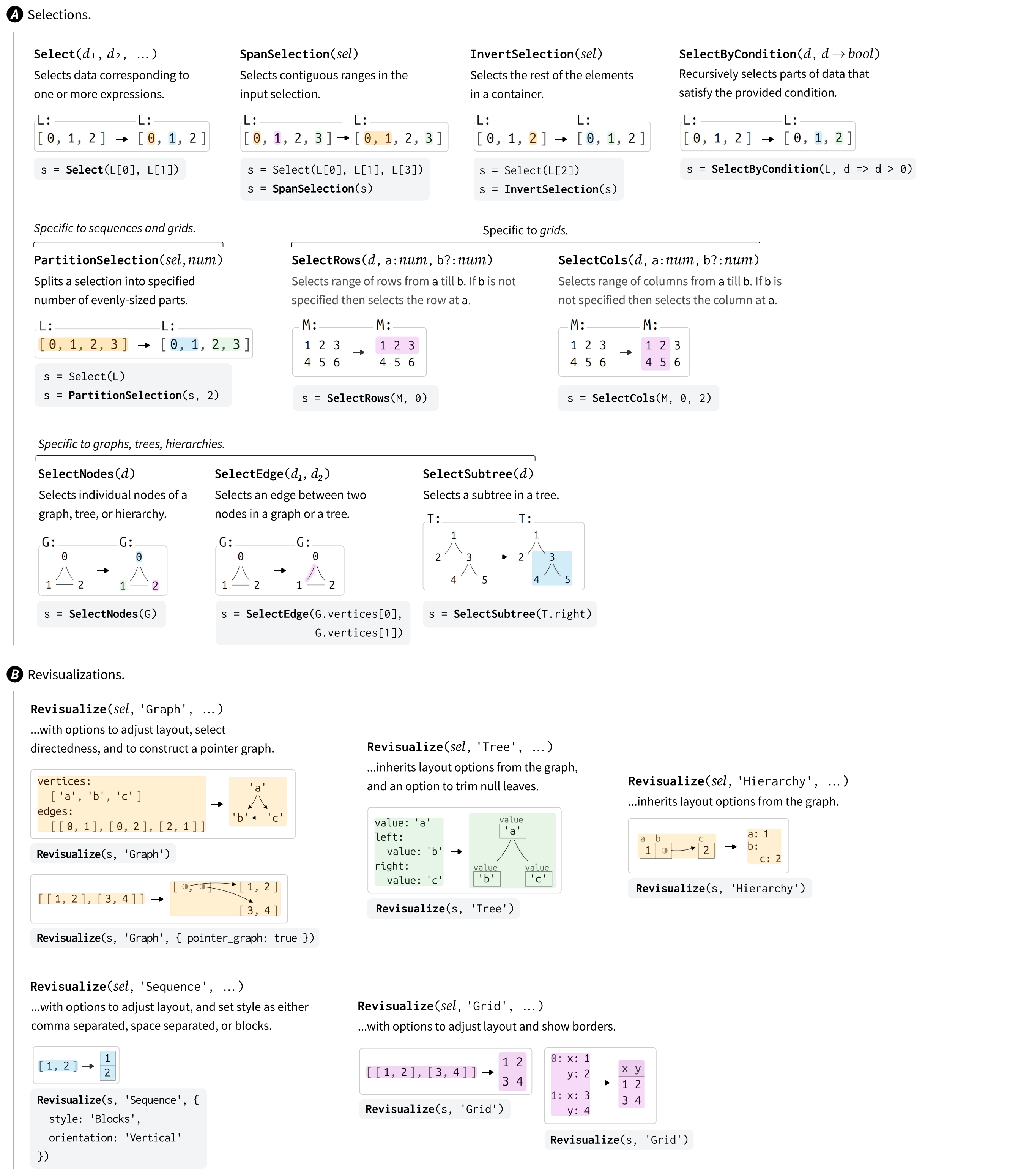
3.3.4 Simplifications. Simplifications describe ways information is omitted in the display. We categorized four types of simplifications.

(1) *Hide* the displayed value.
(2) *Clump* a span of values into one shape. The size of the clump is proportional to the number of values underneath it (*e.g.* a clump of two values will be smaller than that of ten values).
(3) *Abbreviate* is similar to clump, but collapses a span of values into an ellipses with a fixed size and shape.
(4) *Fragment* isolates data within a larger container by explicitly hiding the edges of the container.

Figure 4 Ⓒ provides the signatures and illustrates examples of each simplification strategy as implemented in *Chisel*.

3.3.5 Annotations. Annotations add information to the display:

(1) *Styles* update the appearance of a value, *e.g.* changing its background color, adding outlines, etc. *Chisel* uses CSS style declarations to modify appearance of elements. In addition to styling individual data values, an *encircled region* around multiple data values is also available for styling.
(2) *Labels* add text around (or inline with) the selected data. These are versatile: they can attached be to an identifier or name, a property satisfied by the selected data (*e.g.* putting '$x \neq 0$' next to non-zero elements), a textual description, etc. Labels also annotate a property of selected data, such as its index locations (e.g. 0, 1, or named like $n-1$), or the *length* of a spanning selection (*e.g.* 5, or named like n). *Chisel* supports


A Selections.
Specific to sequences and grids.
Specific to grids.
Specific to graphs, trees, hierarchies.
B Revisualizations.

Select(d_1, d_2, …)
Selects data corresponding to one or more expressions.
```
s = Select(L[0], L[1])
```

SpanSelection(sel)
Selects contiguous ranges in the input selection.
```
s = Select(L[0], L[1], L[3])
s = SpanSelection(s)
```

InvertSelection(sel)
Selects the rest of the elements in a container.
```
s = Select(L[2])
s = InvertSelection(s)
```

SelectByCondition(d, $d \rightarrow bool$)
Recursively selects parts of data that satisfy the provided condition.
```
s = SelectByCondition(L, d => d > 0)
```

PartitionSelection(sel, num)
Splits a selection into specified number of evenly-sized parts.
```
s = Select(L)
s = PartitionSelection(s, 2)
```

SelectRows(d, a:num, b?:num)
Selects range of rows from a till b. If b is not specified then selects the row at a.
```
s = SelectRows(M, 0)
```

SelectCols(d, a:num, b?:num)
Selects range of columns from a till b. If b is not specified then selects the column at a.
```
s = SelectCols(M, 0, 2)
```

SelectNodes(d)
Selects individual nodes of a graph, tree, or hierarchy.
```
s = SelectNodes(G)
```

SelectEdge(d_1, d_2)
Selects an edge between two nodes in a graph or a tree.
```
s = SelectEdge(G.vertices[0],
               G.vertices[1])
```

SelectSubtree(d)
Selects a subtree in a tree.
```
s = SelectSubtree(T.right)
```

Revisualize(sel, 'Graph', …)
...with options to adjust layout, select directedness, and to construct a pointer graph.
```
vertices:
  ['a', 'b', 'c']
edges:
  [[0, 1], [0, 2], [2, 1]]
```
```
Revisualize(s, 'Graph')
```
```
Revisualize(s, 'Graph', { pointer_graph: true })
```

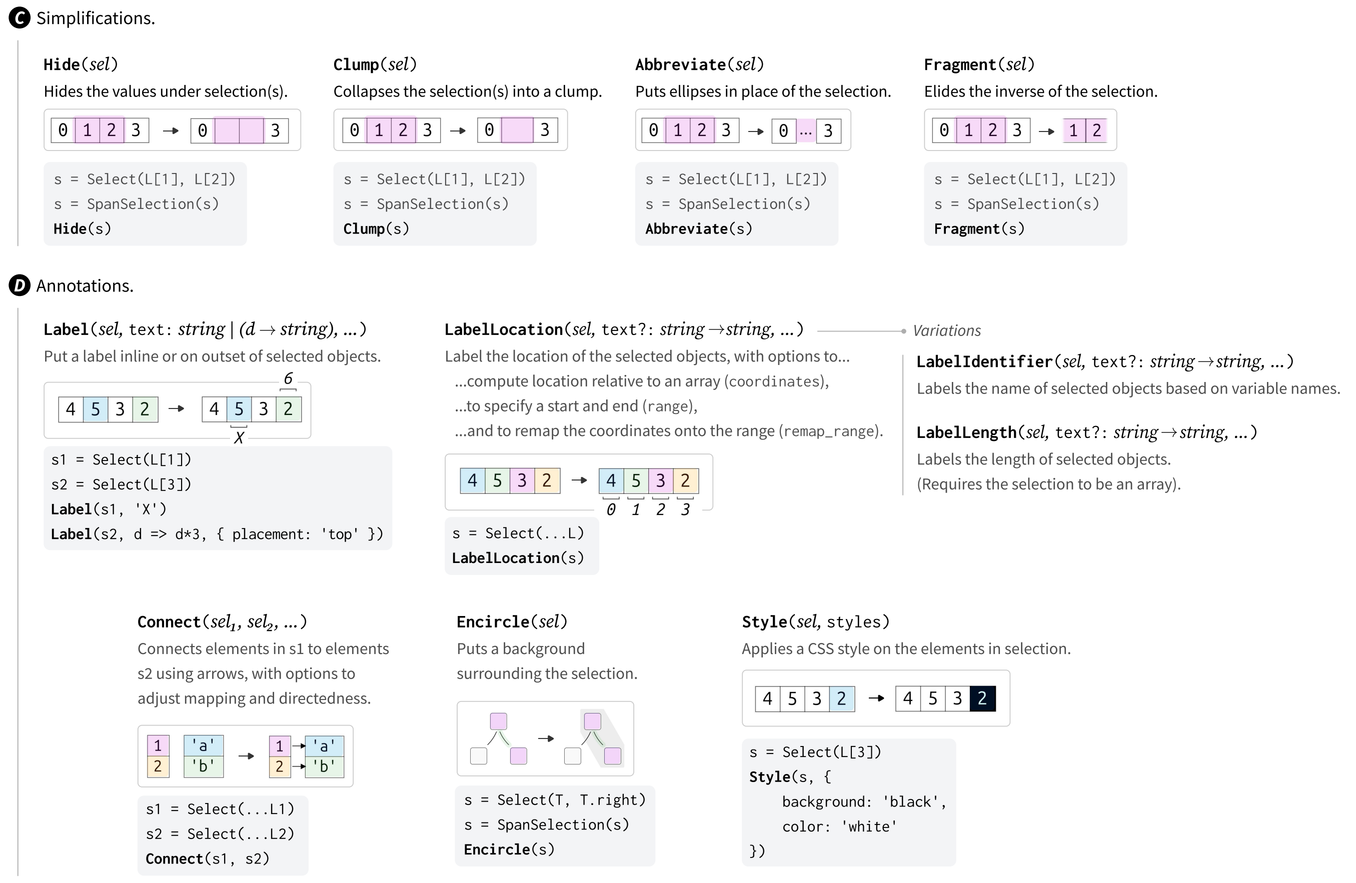
Revisualize(sel, 'Tree', …)
...inherits layout options from the graph, and an option to trim null leaves.
```
value: 'a'
left:
  value: 'b'
right:
  value: 'c'
```
```
Revisualize(s, 'Tree')
```

Revisualize(sel, 'Hierarchy', …)
...inherits layout options from the graph.
```
Revisualize(s, 'Hierarchy')
```

Revisualize(sel, 'Sequence', …)
...with options to adjust layout, and set style as either comma separated, space separated, or blocks.
```
Revisualize(s, 'Sequence', {
  style: 'Blocks',
  orientation: 'Vertical'
})
```

Revisualize(sel, 'Grid', …)
...with options to adjust layout and show borders.
```
Revisualize(s, 'Grid')
```
```
Revisualize(s, 'Grid')
```

Figure 3: Function signatures and examples of the supported selections (A) and revisualizations (B).



Hide(*sel*)
Hides the values under selection(s).

```
s = Select(L[1], L[2])
s = SpanSelection(s)
Hide(s)
```

Clump(*sel*)
Collapses the selection(s) into a clump.

```
s = Select(L[1], L[2])
s = SpanSelection(s)
Clump(s)
```

Abbreviate(*sel*)
Puts ellipses in place of the selection.

```
s = Select(L[1], L[2])
s = SpanSelection(s)
Abbreviate(s)
```

Fragment(*sel*)
Elides the inverse of the selection.

```
s = Select(L[1], L[2])
s = SpanSelection(s)
Fragment(s)
```

Label(*sel*, text: *string* | (*d* → *string*), ...)
Put a label inline or on outset of selected objects.

```
s1 = Select(L[1])
s2 = Select(L[3])
Label(s1, 'X')
Label(s2, d => d*3, { placement: 'top' })
```

LabelLocation(*sel*, text?: *string* → *string*, ...)
Label the location of the selected objects, with options to...
...compute location relative to an array (coordinates),
...to specify a start and end (range),
...and to remap the coordinates onto the range (remap_range).

```
s = Select(...L)
LabelLocation(s)
```

LabelIdentifier(*sel*, text?: *string* → *string*, ...)
Labels the name of selected objects based on variable names.

LabelLength(*sel*, text?: *string* → *string*, ...)
Labels the length of selected objects.
(Requires the selection to be an array).

Connect(*sel*$_1$, *sel*$_2$, ...)
Connects elements in s1 to elements s2 using arrows, with options to adjust mapping and directedness.

```
s1 = Select(...L1)
s2 = Select(...L2)
Connect(s1, s2)
```

Encircle(*sel*)
Puts a background surrounding the selection.

```
s = Select(T, T.right)
s = SpanSelection(s)
Encircle(s)
```

Style(*sel*, styles)
Applies a CSS style on the elements in selection.

```
s = Select(L[3])
Style(s, {
    background: 'black',
    color: 'white'
})
```

Figure 4: Function signatures and examples of simplifications (C) and annotations (D).

adding custom labels, as well as specialized functions for labeling locations, names, and lengths of the selected items.

(3) *Connection* annotations add a visual link (*e.g.*, an arrow) between data selections. For example, to annotate data movement, or to visualize a connection between data structures (*e.g.*, a shared pointer). Connections can exist between different diagrams (e.g. before-after snapshots of a data structure) or within a diagram (*e.g.* data movement of values). In *Chisel*, connection annotations are simplified as connecting arrows (which can either be directed or not).

Figure 4 D shows each annotation strategy implemented in *Chisel*.

3.4 Reproducing Observed Diagrams

Chisel's syntax was constructed inductively by reproducing each of the 80 observed diagrams. The first author then annotated each code from the content analysis with it either being successfully reproduced, partially reproduced (*i.e.* an alternate representation that captures the same meaning), or not reproduced at all (*i.e.* unable to represent the meaning of the code).

Table 2 provides a granular tally of replication success per code in the content analysis. We were able to accommodate for and replicate 78/80 diagrams. Figure 5 showcases some of the diagrams that were reproduced. A gallery of all diagrams reproduced, along

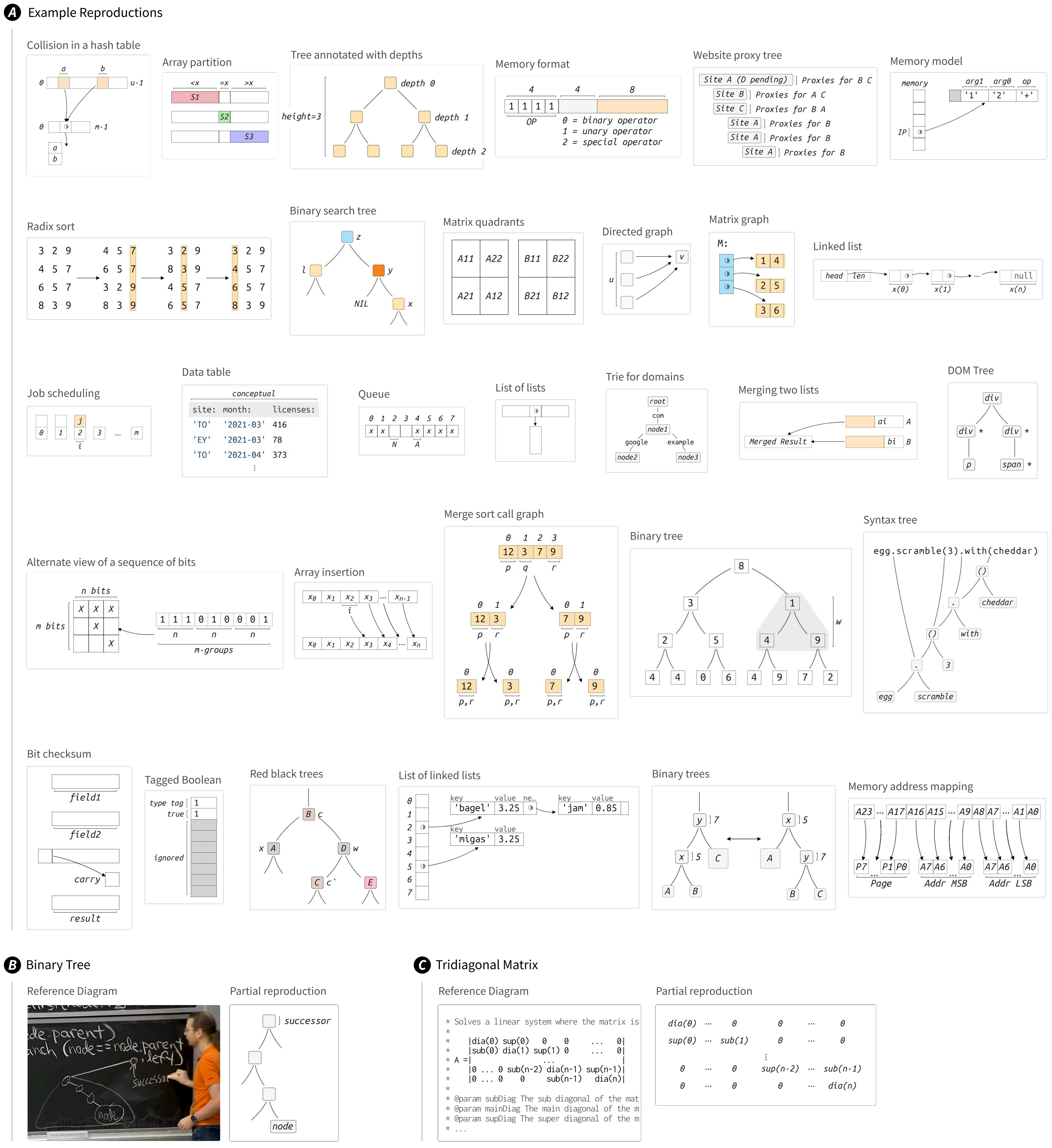
Table 2: Tally statistics on the codes reproduced by our system grouped by the primary data structure in the diagram.

Data	#	Revisualize	Simplify	Annotate
Graph	15	16/16 (100%)	17/17 (~100%)	23/27 (~85%)
Tree	17	18/22 (~80%)	16/18 (~90%)	31/33 (~95%)
Array	18	25/26 (~95%)	17/18 (~95%)	30/31 (~95%)
2D Array	7	11/12 (~90%)	10/11 (~90%)	11/11 (100%)
Memory	23	31/32 (~95%)	27/30 (90%)	56/60 (~95%)

with the codes is provided at: https://abstraction-moves.github.io/. *Chisel* specifications contained 21 lines of code on average across the replicated diagrams.

3.4.1 *Aspects of diagrams that were not reproduced.*

Two diagrams contained visuals that we were not able to reproduce. Figure 5 B is a diagram of a binary tree where a branch in the left subtree is visible inside a clumped subtree. *Chisel* does not support showing a fragment of data inside a container that has been simplified. Figure 5 C shows a matrix where the first two and the last two rows contain abbreviated columns (*i.e.* `0...0`), but those abbreviation don't carry



```
* Solves a linear system where the matrix is
*
*    |dia(0) sup(0)  0    0      ...   0|
*    |sub(0) dia(1) sup(1) 0     ...   0|
* A =|                ...               |
*    |0 ... 0 sub(n-2) dia(n-1) sup(n-1)|
*    |0 ... 0    0     sub(n-1)   dia(n)|
*
* @param subDiag The sub diagonal of the mat
* @param mainDiag The main diagonal of the m
* @param supDiag The super diagonal of the m
* ...
```

Figure 5: A Example gallery of reproduced diagrams. At the bottom are two diagrams whose abstraction *Chisel* was unable to capture, along with the partial reproduction: B A binary tree diagram, from MIT's Introduction to Algorithms [13] shows a branch as part of a larger subtree, and C Tridiagonal matrix, from Chromium [9] which uses partial abbreviations on the rows.

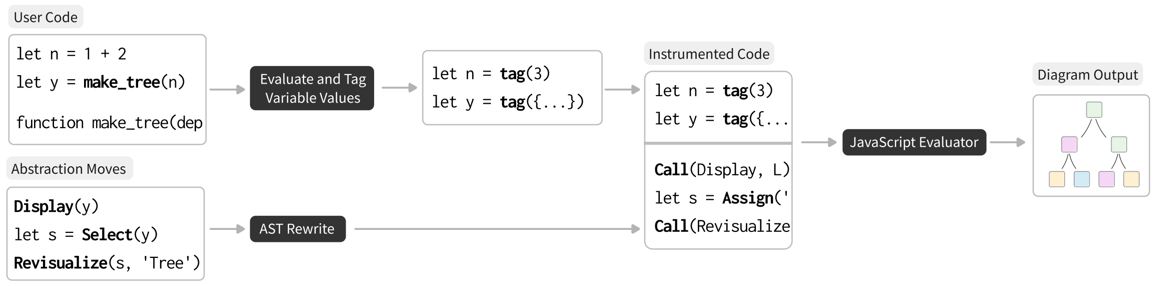


Figure 6: An overview of the pipeline of the prototype. We first execute the user's code and extract and tag variable values' which are joined with an instrumented version of the abstraction specification.

into the rest of the rows, which makes the abbreviation inconsistent with the grid visual structure.

3.5 Implementation Description

The prototype system is an in-browser JavaScript web application. Figure 6 provides an overview of the architecture. Below, we will detail two implementation details: data provenance tracking and data-to-visual mappings.

3.5.1 Data provenance tracking. User's data may not start in the format required to display it appropriately, *e.g.* a graph may be stored as an adjacency-matrix, while *Chisel* requires it to be a collection of vertices and edges. So, the user must first transform the data into the format accepted by *Chisel*. However, after the transform, it can still be preferable to perform selections in terms of the original structure, *e.g.* select a row in an adjacency matrix, and have that selection propagated to the visual representation. Therefore, we track data as it is transformed. *Chisel* rewrites variable assignments to box each value and sub-value in a wrapper that contains a unique identifier (see Figure 6), *e.g.* `[5,2,3]` becomes a data structure of objects: `Array`^{3}`(Num(5)`^{0}`,Num(2)`^{1}`,Num(3)`^{2}`)`. We use custom handlers to maintain data provenance, *e.g.* the expression `1+2` is rewritten to `BinaryExpression(Num(1)`^{0}`,Num(2)`^{1}`)` which evaluates to `Num(3)`^{0,1}. Operating on tagged data means that *Chisel* is a subset of JavaScript, which so far includes the core constructs (loops, conditionals), standard libraries on arrays (filters, slice, map), as well as the `Math` standard library.

3.5.2 Technical Architecture. The prototype is a JavaScript web application developed using TypeScript. We use acorn [1] for parsing JavaScript into an AST, which is then traversed and rewritten. We use Kiwi [28], a global constraint solver, for computing the diagram layout, and math.js' [12] algebra system to display named indices relative to the actual data-length. There are some notable implementation limitations:

(1) Visual layouts are not done as carefully as they could be. For example, the graph layout renders a tree, and then adds back-edges for cycles. In practice, a graph rendering algorithm should be used (e.g. graphviz [18]).
(2) The current implementation requires a specific operator order (revisualization, simplification, then annotation operations), to avoid failure states. This is not inherit to abstraction moves, but a limitation of the current prototype.
(3) Potential optimizations for performance. The diagram in Figure 2 ❶ renders within $\sim 200ms$ on a twelve item array, and takes over $\sim 5000ms$ to render a fifty item array on our machine. This poor scaling in render time is caused by the global constraint layout solver and is limiting since visual abstractions are perhaps the most helpful when making sense of large amounts of data. An alternative is to use a local constraint solver along with a separate engine for handling global constraints when needed, as in Bluefish [36].

3.6 Initial Results on User Experience and Usage Scenarios

Our core contribution is a description of visual abstractions in data structure diagrams, and a reification of those descriptions into *Chisel*. The syntax is intentionally interpretive and not grounded in any particular context of use or task. To understand the experience of working with the language and discover possible contexts of use, we conducted a preliminary study with four introductory computer science (CS) educators. We selected CS educators as they will likely find the most immediate relevance from *Chisel* since they may be manually designing diagrams of data structures and 7/10 diagram sources in our content analysis were from educational contexts.

3.6.1 Participants. We recruited three CS instructors (P_1, P_2, P_4) and one graduate student teaching assistant (P_3), see Table 3. Participation was voluntary without compensation. Only P_2 had prior experience with using JavaScript.

3.6.2 Study Procedure. Each study lasted 60 minutes, and consisted of three parts: (a) a tutorial of the system with four worked examples, (b) a guided construction of a diagram,[4] and (c) a semi-structured interview on the shortcomings/successes of the notation and its potential uses. The study is limited: the sample size is small and the the session was closely shepherded by the experimenter. Our goal was not necessarily to teach *Chisel* in the span of 30–40min (not including interview time), but to get participants' initial reactions to using it and understand its potential contexts of use.

3.6.3 Usability Issues. In general, we were encouraged by the use of *Chisel* in the short span of the study, P_4 even found it to be *"pretty intuitive most of the time...I thought it was kind of fun!"* All

[4]For the instructors, P_1, P_2, P_4, the target diagram was preselected from their slides from a recent class, for P_3 this was selected from our corpus.

Table 3: Participant demographics. Teaching includes experience as a teaching assistant.

ID	Title	Teaching Experience	Subjects of recently taught courses
P_1	Associate Teaching Professor	10 years	Intro to Python, Intro to Data Science, Data Science in Practice
P_2	Assistant Teaching Professor	11 years	Practice and Application of Data Science, Data Visualization
P_3	Graduate student	4 years	Intro to Python
P_4	Professor, Teaching Stream	35 years	Intro to Programming (in Python), Software Design (in Java)

participants found the syntax to be reasonable, *"from an API [standpoint], the names make sense"* (P_1), but would need more time to get comfortable with it. There were multiple breakdowns during the usage which we summarize below.

Difficulty discerning units of selection. A selection in *Chisel* can span over multiple items (*e.g.* `[1,2,3,4]`) or be a union of multiple items (*e.g.* `[1,2,3,4]`). All participants encountered difficulties disambiguating if the selection was to individual items or to a span: *"am I selecting two things in a list or am I selecting one list with two things?"* (P_2). P_3 and P_4 also encountered situations where they used an incompatible unit of selection as input to an operation. P_3 suggested that confusion between individual and spanning selections may be resolved with better scaffolding when introducing the syntax[5], and integrating type signatures of the operations into the editor to learn valid inputs into the various moves. P_2 suggested adding more explicit visual cues, *e.g.* to use a *"visual indicator...like corners around [the selection]"* if it is a spanning selection.

Lack of alternate paths and flexibility in authoring. Some strategies that participants reached for were unsupported. For example, instead of using `PartitionSelection` to split a large selection into smaller evenly sized groups, P_4 wanted to *"start with all of the values in L selected separately and then try to group them,"* and P_3 wished to index into data by indexing into a selection, *"I wanted to do* `s[0]`*...and so then I was just stuck."* Similarly, while navigating the diagrams at this level of description was appreciated, *"it's cool how like the grids and trees things just work out-of-the-box...as an instructor, I'm like, the more hard coded things you can give me, the better, because I don't want to spend my time like pixel pushing the diagrams"* (P_2), P_1 and P_2 also expressed concerns of not having control to tweak the diagrams, *"I wonder how often the arrows actually do what I want?"* (P_1). These concerns suggest building on a general-purpose diagramming library as a base for *Chisel*, which can be extendable to new diagrams and workflows. We discuss opportunities for doing so in subsection 4.1.

3.6.4 Usage scenarios. We also asked participants if *Chisel* is relevant to their teaching and its potential uses (if any).

Creating diagrams offline. All participants mentioned potential for *Chisel* to help them prepare lecture slides. For P_2, the flexibility of picking the revisualization is particularly helpful: *"often times when we teach graphs...sometimes it's an adjacency list, sometimes it's like a dictionary with vertices and edges."* P_4 mused on using it to compare algorithms, when *"we're teaching you insertion and selection sort, let's visualize the invariants of the two, and then we [as a class] can visually compare them."* P_3 and P_4 also mentioned using it to typeset diagrams for use in print:

> *"I would totally see using this to generate diagrams for a paper I'm writing...having it automatically constructed makes it, I think, well worth the effort because I might want to change my input but keep the same view of the output...that's what got me excited at the very beginning of this, because I was thinking back to the diagrams we drew for the Java textbook that we wrote."* (P_4)

Creating diagrams interactively. P_2 and P_4 proposed to use *Chisel* interactively during lecture. P_2 would like to experiment with diagramming alongside the students as a way to scaffold their learning:

> *"I think students would benefit a lot from being able to move really slowly...if [Chisel] worked really well, and I could make diagrams like very quickly or live even then I can imagine myself commenting out parts of this and then re-commenting it back to gradually show like one row [in a data table] being added at a time."* (P_2)

P_4 would use *Chisel* to test different inputs live in class, *"it's the what if questions that always happen. What if the list isn't a length that's divisible by three? What if we've got an extra element? What if the smallest item is at the front?"* We also floated to P_2 and P_4 why existing program visualization tools, like Python Tutor [19], did not cover these use cases. To which, they responded that the level of description and visual encoding was fixed: it would *"just show you the before and after"* (P_2), and is *"just a memory model, it's not a visualization tool."* (P_4).

4 Discussion and Future Work

We have presented abstraction moves, ways to incrementally shift a concrete data display into a customized diagram. Below, we reflect on limitations of our description and potential usage contexts.

4.1 Limitations of Translating a Content Analysis to a DSL

When designing *Chisel*, we aimed to maintain a close mapping between the qualitative coding and the syntax. Translating qualitative codes into operations may have helped with the high coverage over the coded diagrams and provided meaningful names, but it has also meant that similar operations and their variations were not unified to form a smaller and more composable API. Below, we speculate on potential areas of unification to reduce the API surface.

[5]The first example in the tutorial was the same as Figure 1, which implicitly creates three span selections through `PartitionSelection`.

4.1.1 Selections. There are ten different functions for specifying selections, six of which are for specific data structures. All these functions are, broadly, different ways of selecting subparts of a data structure. Instead of using specialized functions, we may use a more generic method which generalizes to data types outside the ones supported, *e.g.* using algebraic data types to specify selections via pattern matching.

4.1.2 Simplifications and annotations. All four simplification functions are closely linked to each other. *Abbreviate* and *Hide* are slight variations on *Clump*, and *Fragment* effectively clumps the inverse of the current selection. Moreover, the relationship between simplifications and annotations is not yet clear. For example:

(1) We initially included a category for *replacing* a data value with a label, which was later decomposed to a simplification (*Hide*) plus annotate (*Label*).
(2) We could have assumed that all lists show indices by default and those indices must be simplified away (instead of added on with *Label Location*).
(3) We could have conceptualized *Encircle* as a container that is yet to be simplified into a clump.

These suggest a more unified grammar for reducing and adding information. A useful theoretical framework for generalizing abstractions might be Brüggemann et al. [7]'s the *Fold*—a structure for describing interactive data visualizations which includes operations for *explication* (revealing information), *implicitation* (hiding information), and *complication* (adding dimensions of information).

4.1.3 Revisualizations. *Chisel* includes preset mappings of data structures to visual structures through different calls to `Revisualize`. This makes it brittle to any new visual types, *e.g.* something simple, like setting the height of a visual mark to a variable value is beyond the current syntax's reach. We may instead build on a grammar of graphics approach (*e.g.* similar to ggplot2 [39]), to provide a composable and more incremental approach for building graphics.

4.2 Comparison with Existing Classifications

Our description of abstraction moves reaffirms existing categorizations of diagrams. For example, we identified a small number of visual structures in data structure drawings (graphs, sequences, grids), which may be expanded to more, *e.g.* maps and plots [4]. Our description of annotation closely matches annotation categories uncovered by Head et al. [22]'s analysis of augmented math equations (*e.g. Label*, *Connection*, and *Encircle* correspond to their categories of *text labels*, *connector*, and *background* respectively). An under-explored category in prior work that is surfaced by our analysis are simplifications. To our knowledge, the closest notion is Hayatpur et al. [20]'s notion of *abstractions* in ASCII drawings, in which they categorize two broad strategies of abstraction as either *unpatterned elision* where the omitted content cannot be inferred (*e.g.* `[x, ... ,y]`) or *patterned elision* where the omitted content has an inherent order (*e.g.* `[1,2, ... ,10]`). Abstraction moves surfaces primitive operations which omit content (*e.g. Abbreviate*, *Hide*) to mechanize these broader strategies.

4.3 Workflows Suitable to Abstraction Moves

As mentioned earlier, the benefit of abstraction moves is that they are *dynamic to changes in the data, flexible across levels of abstraction*, and provide a *visible record of abstraction.* We now sketch how these qualities can be used in instruction and software development.

4.3.1 Instruction.

(1) *Dynamic to changes in the data.* To teach an algorithm, an instructor often makes a slide showing how the algorithm works on a small input data. However, this is an arduous and brittle process: the slide is manually created for each input data they wish to demonstrate. And, as mentioned by P_4, if a student asks about an edge case in class, the static slides are of no help. Instead, if the diagram was recorded as a sequence of abstraction moves, the instructor could improvise lessons by updating the concrete data live which updates the diagram.
(2) *Flexible across levels of abstraction.* To transfer knowledge from one situation to another, an instructor often present variations of a problem to communicate essential features of it by varying non-essential ones [27]. Abstraction moves provide the ability for the instructor to define what is essential and what is non-essential *interactively.* For example, in a merge sort algorithm, the diagram might start with concrete values, then the half-way point is labeled, and then the two halves are clumped. *Chisel* could be extended to support animated unfolding of the diagram in order to help scaffold understanding (this type of interaction is also implied by P_2's proposal for using *Chisel* to slowly reveal a data transformation).
(3) *A visible record of abstraction.* Persistence allows for curating and standardizing diagrams. After creating abstraction moves to illustrate one algorithm, an instructor can share it with their colleagues, and collaboratively develop pedagogical material.

4.3.2 Software Development.

(1) *Dynamic to changes in data.* Since the diagram is responsive to changes in data, it can be written once and then used to illustrate many different variations of the same data structure, *e.g.* to illustrate test cases or edge cases of an algorithm. Indeed, in their study of ASCII diagrams from open source codebases, Hayatpur et al. [20] found illustrating test cases as one of the key roles that diagrams play.
(2) *Flexible across levels of abstraction.* A diagram made with abstraction moves can become a unifying representation for high-level, conceptual documentation down to low-level debug views. An array might initially be illustrated at a high-level in the documentation with labels to relevant locations but mostly elided, and then later be reused for debugging by displaying the values in their full detail while keeping the labels to relevant locations.
(3) *A visible record of abstraction.* Abstraction moves can also enable development of debugger views. They may be packaged and shared in debugging tools (*e.g.* through GUI controls, letting other programmers easily apply them).

4.4 Future Work: Contextually Relevant Diagrams and Direct Manipulation

Using *Chisel* requires learning a syntax which may be difficult to grasp. Two ways in which abstraction moves might become more integrated into programming workflows can be through *recommendation systems* and *direct manipulation*:

4.4.1 Recommendation system for diagrams. To mitigate some tedium of writing *Chisel*, we could use the user's code that is being diagrammed (*e.g.* its variable names, indexes, scope) to infer contextually relevant diagrams as a starting point. This becomes more relevant if *Chisel* is integrated into an ordinary IDE where multiple diagrams co-exist in a code file and are attached to different code snippets. For example, variables declared in a scope can be automatically displayed. If an index into a list is used, *e.g.* `L[i]`, then a diagram where `i` is labeled on `L` can be suggested. Or, if the diagram is attached to a loop which reads one element from `L` at a time then the other elements can be elided. More general abstractions might also inferred by program analysis techniques like concolic execution [37] to collect abstract relations which can be mapped to visual abstractions.

4.4.2 Specifying abstraction moves via direct manipulation. Graphic editors like Powerpoint and Photoshop operate by principles of *direct manipulation* (DM). *Chisel* could profit from a DM metaphor rather than a programming languages one. For example, the user might directly scrub over a portion of the display using their cursor, and then open a menu to apply one of the abstraction moves. DM may eliminate the need to internalize syntax and lower the barrier to entry. Yet, open questions on the specific interactions remain:

(1) *Disambiguating interaction intent*, *e.g.* if the user scrubs over the first two elements of the array `L: [1,2,3,4]`, how can we disambiguate between `L[0:2]`, or `L[0:L.length/2]`?
(2) *Specifying control flow*, *e.g.* to select by a condition, like selecting only the non-null values in the diagram?

Two approaches to these problems can be to either (a) use automation, *e.g.* using program synthesis to guess intended operation from demonstrations [31], or (b) present both the syntax and a limited direct manipulation view which provides a reasonable trade-off between ease-of-use (from the DM metaphor) and expressiveness (from the syntax).

5 Conclusion

We have demonstrated how *abstraction moves*—shifts in a data's display—enable creating illustrative diagrams from concrete data values. Through a content analysis of 80 programmer-produced diagrams, we discovered three overarching abstraction strategies: *revisualizations*, *simplifications*, and *annotations*, as well as *selections*, which specify the objects to be abstracted. We then implemented abstraction moves in a JavaScript-based prototype that is able to reproduce 78/80 diagrams. The ability to make these customized views of data might enable new workflows in programming education, *e.g.* interactively switching between levels of abstraction during a lecture, and software development, *e.g.* building task-specific debug views that can be packaged and shared with others.

Acknowledgments

We are overwhelmingly grateful to the anonymous reviewers for their detailed reading and constructive feedback on the manuscript which have improved it immensely. We also thank Matthew Beaudouin-Lafon, Emilia Rosselli Del Turco, and Fuling Sun for being early readers of the manuscript. This material is based upon work supported by the National Science Foundation under Grant No. 2432644.

References

[1] Acorn. 2024. Acorn. https://github.com/acornjs/acorn.
[2] Richard Anderson. 2013. CSEP 521: Applied Algorithms, Winter 2013 — courses.cs.washington.edu. https://courses.cs.washington.edu/courses/csep521/13wi/. [Accessed 12-09-2024].
[3] Sebastian Baltes and Stephan Diehl. 2017. Sketches and Diagrams in Practice. *CoRR* abs/1706.09172 (2017). arXiv:1706.09172 http://arxiv.org/abs/1706.09172
[4] Alan F. Blackwell and Yuri Engelhardt. 2002. A Meta-Taxonomy for Diagram Research. In *Diagrammatic Representation and Reasoning*, Michael Anderson, Bernd Meyer, and Patrick Olivier (Eds.). Springer, 47–64. https://doi.org/10.1007/978-1-4471-0109-3_3
[5] Mike Bostock. 2012. D3.js - Data-Driven Documents. http://d3js.org/
[6] Beatrix Braune and Reinhard Wilhelm. 2000. Focusing in Algorithm Explanation. *IEEE Trans. Vis. Comput. Graph.* 6, 1 (2000), 1–7. https://doi.org/10.1109/2945.841117
[7] Viktoria Brüggemann, Mark-Jan Bludau, and Marian Dörk. 2020. The Fold: Rethinking Interactivity in Data Visualization. *Digit. Humanit. Q.* 14, 3 (2020). http://www.digitalhumanities.org/dhq/vol/14/3/000487/000487.html
[8] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. 2007. Let's Go to the Whiteboard: How and Why Software Developers Use Drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '07)*. Association for Computing Machinery, New York, NY, USA, 557–566. https://doi.org/10.1145/1240624.1240714
[9] Chromium. 2008. Home — chromium.org. https://www.chromium.org/chromium-projects/. [Accessed 12-09-2024].
[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.
[11] Gael Alan Curry. 1978. *Programming by Abstract Demonstration*. Ph. D. Dissertation. USA. AAI7814420.
[12] Jos de Jong. 2013. mathjs. https://mathjs.org/.
[13] Erik Demaine, Dr. Jason Ku, and Prof. Justin Solomon. 2020. Introduction to Algorithms | Electrical Engineering and Computer Science | MIT OpenCourseWare — ocw.mit.edu. https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/. [Accessed 12-09-2024].
[14] Jeff Erickson. 2016. CS 473: Lecture Schedule — courses.grainger.illinois.edu. https://courses.grainger.illinois.edu/cs473/sp2016/lectures.html. [Accessed 12-09-2024].
[15] Judith E. Fan, Wilma A. Bainbridge, Rebecca Chamberlain, and Jeffrey D. Wammes. 2023. Drawing as a versatile cognitive tool. *Nature Reviews Psychology* 2, 9 (01 Sep 2023), 556–568. https://doi.org/10.1038/s44159-023-00212-w
[16] Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen M. Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, ITiCSE-WGR 2020, Trondheim, Norway, June 15-19, 2020*, Michail N. Giannakos, Guttorm Sindre, Andrew Luxton-Reilly, and Monica Divitini (Eds.). ACM, 21–50. https://doi.org/10.1145/3437800.3439202
[17] Morgan M Fong, Seth Poulsen, and Geoffrey L Herman. 2021. What's in a Linked List? A Phenomenographic Study of Data Structure Diagrams. In *ASEE Annual Conference and Exposition, Conference Proceedings*.
[18] Emden R Gansner. 2009. Drawing graphs with Graphviz. *Technical report, AT&T Bell Laboratories, Murray, Tech. Rep, Tech. Rep.* (2009).
[19] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Technical Symposium on Computer Science Education (SIGCSE)*.
[20] Devamardeep Hayatpur, Brian Hempel, Kathy Chen, William Duan, Philip Guo, and Haijun Xia. 2024. Taking ASCII Drawings Seriously: How Programmers Diagram Code. In *Conference on Human Factors in Computing Systems (CHI)*. https://doi.org/10.1145/3544548.3581390
[21] Devamardeep Hayatpur, Daniel Wigdor, and Haijun Xia. 2023. CrossCode: Multi-level Visualization of Program Execution. In *Conference on Human Factors in Computing Systems (CHI)*. https://doi.org/10.1145/3544548.3581390
[22] Andrew Head, Amber Xie, and Marti A. Hearst. 2022. Math Augmentation: How Authors Enhance the Readability of Formulas using Novel Visual Design Practices.

In *CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022*, Simone D. J. Barbosa, Cliff Lampe, Caroline Appert, David A. Shamma, Steven Mark Drucker, Julie R. Williamson, and Koji Yatani (Eds.). ACM, 491:1–491:18. https://doi.org/10.1145/3491102.3501932

[23] IETF. 2024. About RFCs. https://www.ietf.org/process/rfcs/.

[24] J Kleinberg. 2006. *Algorithm Design.* Addison Wesley.

[25] Sam Lau and Philip J Guo. 2020. Data Theater: A live programming environment for prototyping data-driven explorable explanations. In *Workshop on Live Programming (LIVE)*.

[26] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. *Conference on Human Factors in Computing Systems (CHI)* (2020).

[27] Mun Ling Lo. 2012. *Variation theory and the improvement of teaching and learning.* Göteborg: Acta Universitatis Gothoburgensis.

[28] Lume. 2024. Kiwi. https://github.com/lume/kiwi.

[29] 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 *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik (Eds.). ACM, 1–14. https://doi.org/10.1145/3313831.3376253

[30] Nicolas Mangano, Thomas D. LaToza, Marian Petre, and André van der Hoek. 2015. How Software Designers Interact with Sketches at the Whiteboard. *IEEE Trans. Software Eng.* 41, 2 (2015), 135–156. https://doi.org/10.1109/TSE.2014.2362924

[31] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin G. Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*, Celine Latulipe, Bjoern Hartmann, and Tovi Grossman (Eds.). ACM, 291–301. https://doi.org/10.1145/2807442.2807459

[32] Sean McDirmid. 2018. Tangible Abstraction. *SPLASH-I* (2018). A video of the system is at https://www.youtube.com/watch?v=6VmA_whVxPc.

[33] Robert Nystrom. 2021. *Crafting interpreters.* Genever Benning.

[34] Sascha A. Parduhn, Raimund Seidel, and Reinhard Wilhelm. 2008. Algorithm visualization using concrete and abstract shape graphs. In *Proceedings of the ACM 2008 Symposium on Software Visualization, Ammersee, Germany, September 16-17, 2008*, Rainer Koschke, Christopher D. Hundhausen, and Alexandru C. Telea (Eds.). ACM, 33–36. https://doi.org/10.1145/1409720.1409726

[35] Marian Petre. 2009. Insights From Expert Software Design Practice. In *The Art, Science, and Engineering of Programming Journal.* https://doi.org/10.1145/1595696.1595731

[36] Josh Pollock, Catherine Mei, Grace Huang, Elliot Evans, Daniel Jackson, and Arvind Satyanarayan. 2024. Bluefish: Composing Diagrams with Declarative Relations. (2024).

[37] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, Michel Wermelinger and Harald C. Gall (Eds.). ACM, 263–272. https://doi.org/10.1145/1081706.1081750

[38] Ivan Viola and Tobias Isenberg. 2018. Pondering the Concept of Abstraction in (Illustrative) Visualization. *IEEE Trans. Vis. Comput. Graph.* 24, 9 (2018), 2573–2588. https://doi.org/10.1109/TVCG.2017.2747545

[39] Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis.* Springer-Verlag New York. https://ggplot2.tidyverse.org

[40] Reinhard Wilhelm, Tomasz Müldner, and Raimund Seidel. 2001. Algorithm Explanation: Visualizing Abstract States and Invariants. In *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures (Lecture Notes in Computer Science, Vol. 2269)*, Stephan Diehl (Ed.). Springer, 381–394. https://doi.org/10.1007/3-540-45875-1_30

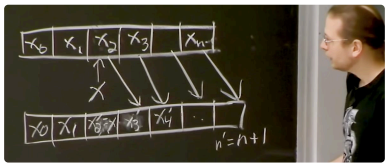
[41] Greg Wilson. 2022. Software Design by Example — third-bit.com. https://third-bit.com/sdxpy/. [Accessed 12-09-2024].

[42] Koji Yatani, Eunyoung Chung, Carlos Jensen, and Khai N. Truong. 2009. Understanding How and Why Open Source Contributors Use Diagrams in the Development of Ubuntu. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, USA) *(CHI '09)*. Association for Computing Machinery, New York, NY, USA, 995–1004. https://doi.org/10.1145/1518701.1518853

[43] 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 Trans. Graph.* 39, 4 (2020), 144. https://doi.org/10.1145/3386569.3392375

A Example codes

❶ Array insertion



Revisualization(s)

- ***Sequence**, arrays as horizontal sequences.*

Simplification(s)

- ***Hide** value of each item.*
- ***Abbreviate** sub ranges.*

Annotation(s)

- ***Label location** of all items.*
- ***Connect** arrows between items in the two arrays.*

❷ Document object model (DOM)

```
// The tree appears as following,
// with the starred nodes dirty:
//       div [relayout-common-ancestor]
//      /   \
//   *div  *div
//    /      /
// *div   *div
```

Revisualization(s)

- ***Tree**, a hierarchy shown as a tree.*

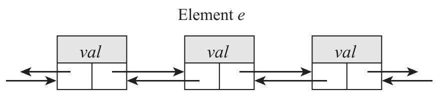
Simplification(s)

- ***Hide** value of each node.*

Annotation(s)

- ***Label** the type ("`div`") for each node.*
- ***Label** nodes that have a `dirty` property with a "`*`".*

❸ Linked list



Revisualization(s)

- ***Graph**, a linked list shown as a graph.*

Simplification(s)

- ***Hide** value of each node.*
- ***Fragment** of the larger linked list.*

Annotation(s)

- ***Label** "`val`" in place of each node's value.*
- ***Label** "Element e" above node e.*

Figure 7: ❶ An insertion operation into an array and the affect it has on positions of the existing elements from MIT 6.006 Introduction to Algorithms [13]; ❷ A data structure which describes the logical pieces of web page as nodes and objects from Chromium [9]; ❸ A fragment of a doubly linked list from Algorithm Design [24].