Sculpin: Direct-Manipulation Transformation of JSON

Joshua Horowitz*
University of Washington
Seattle, Washington, USA
joshuah@alum.mit.edu

Devamardeep Hayatpur*
University of California, San Diego
La Jolla, California, USA
dshayatpur@ucsd.edu

Haijun Xia
University of California, San Diego
La Jolla, California, USA
haijunxia@ucsd.edu

Jeffrey Heer
University of Washington
Seattle, Washington, USA
jheer@uw.edu

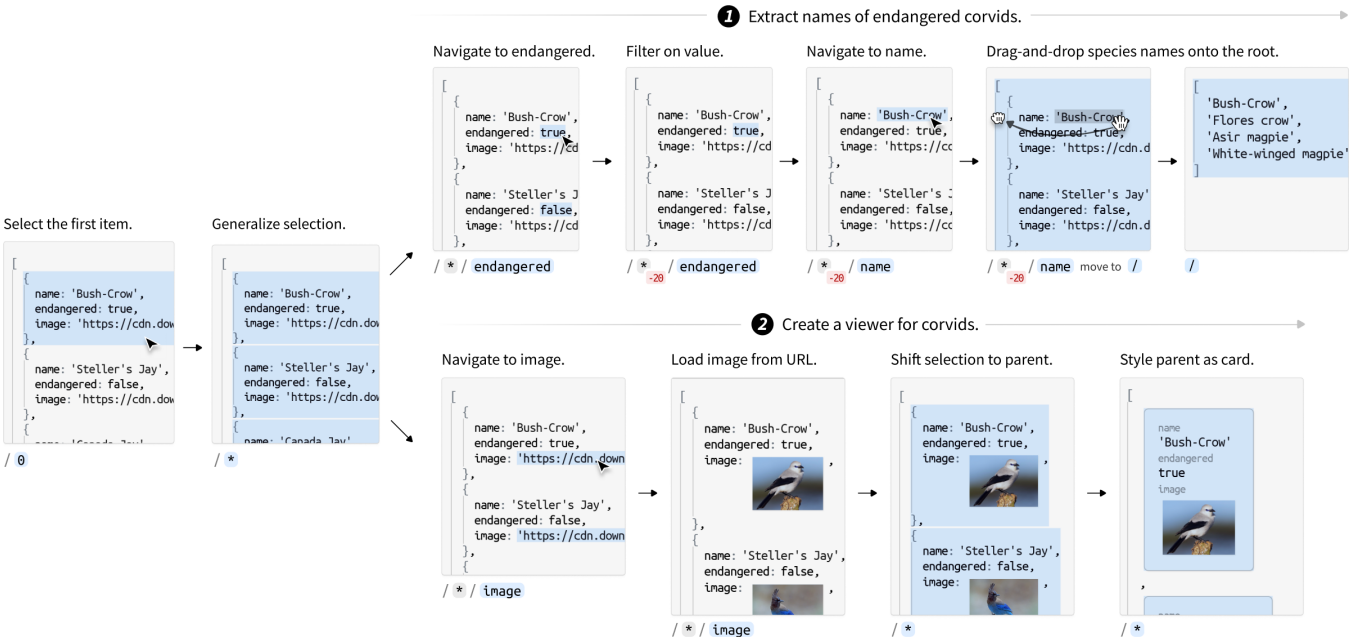

Figure 1: Sculpin is a programming-by-demonstration (PbD) system for JSON. Sculpin is versatile: it can wrangle and extract sub parts of JSON (❶), or create interfaces atop it (❷). In Sculpin, a user refines a selection and uses it to perform actions on the data underneath. User actions are recorded into a timeline, which forms a reusable program.

ABSTRACT

Many end-user programming tasks require programmatically processing JSON, wrangling it from one format to another or building interactive applications atop it. But end-users are impeded by the indirectness and steep learning curve of textual code. We present *Sculpin*, a direct-manipulation environment supporting a broad range of JSON-transformation tasks. A user of Sculpin transforms JSON data step by step, recording a program in the process. Sculpin makes three design commitments to ensure directness and versatility: (1) steps are small and precise, not inferred; (2) steps are general-purpose and open to re-appropriation; (3) steps operate on JSON itself, rather than on a limited intermediate representation. To support these commitments, Sculpin introduces a mechanism of sculptable selections: the user can direct their action by guiding a selection on top of the data through small steps like generalization and hierarchical navigation. Sculpin also extends JSON with embedded interface elements like form inputs and buttons, allowing applications to be sculpted incrementally from source data. We demonstrate the breadth and directness of Sculpin in use-cases ranging from wrangling data to building applications. We evaluate Sculpin through a heuristic analysis, situating it in a broad space of programming systems and surfacing limitations such as difficulties editing preexisting programs.

*Both authors contributed equally to this research.

UIST '25, September 28-October 1, 2025, Busan, Republic of Korea

ACM ISBN 979-8-4007-2037-6/2025/09
https://doi.org/10.1145/3746059.3747651

CCS CONCEPTS

• **Human-centered computing → Graphical user interfaces**; • **Software and its engineering → Integrated and visual development environments**; **Application specific development environments**.

KEYWORDS

end-user programming, programming by demonstration, direct manipulation

ACM Reference Format:
Joshua Horowitz, Devamardeep Hayatpur, Haijun Xia, and Jeffrey Heer. 2025. Sculpin: Direct-Manipulation Transformation of JSON. In *The 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25), September 28-October 1, 2025, Busan, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3746059.3747651

1 INTRODUCTION

JSON (JavaScript Object Notation) is a flexible, lightweight representation of data that has emerged as a pervasive standard in the Web era. Data ranging from API messages to document formats to domain-specific programming languages are often represented as JSON. Consequently, programmatically transforming JSON (processing it, visualizing it, and building interfaces atop it) is a high-leverage skill for both professional software engineers and technically-engaged end-users. These transformations are usually performed with conventional programming languages which are powerful but indirect: the user writes symbolic code (*e.g.* JavaScript) to manipulate data rather than directly act on it. This can present learning barriers for end-user programmers [31, 42], and limit the directness and visibility of programming [2, 43, 45].

In contrast to coding, editors for everyday computer activities like creating slide shows, drawing, and video editing feel controllable and live because they operate by principles of direct manipulation (DM): users transform data through operations that produce "immediately visible" effects on a "continuous representation of the object of interest" [50]. These editors are versatile, suited to a broad range of tasks. For example, a vector graphics editor can be used to directly annotate a chart, or to lay out furniture on a floor plan.

How might we design a versatile, direct-manipulation editor for programmatically transforming JSON? A long line of work in *programming by demonstration* (PbD) systems has sought to bring benefits of DM to programming. PbD systems enable users to specify programs by step-by-step demonstrations, and cover domains such as web scraping [11], text [40], data visualization [48], tabular data [29], and interactive systems [10]. However, these systems often (1) undermine directness through automatic inference, making the effects of actions unpredictable and error recovery difficult [33], and (2) are not versatile, as they are designed to streamline specific workflows (*e.g.* web-scraping [11]) rather than operate on a broader medium (*e.g.* JSON) that can support multiple workflows.

To overcome these limitations, we built *Sculpin*, a direct-manipulation programming system for transforming JSON data. Sculpin follows three design commitments to achieve directness and versatility. To ensure directness, Sculpin's operations are (1) small and precise, avoiding leaps of inference. To ensure versatility, Sculpin's operations are (2) general-purpose and reappropriable and (3) defined directly on their underlying medium (JSON), rather than on an intermediate or parallel representation.

In Sculpin, users transforms JSON data step by step, recording a program in the process. The user precisely directs their actions by sculpting a selection on top of the data through small steps like generalization and hierarchical navigation. The JSON medium they work on is also extended to include interface elements like form inputs and buttons, so Sculpin can be used to continuously transform data into interfaces, using the same mechanisms that are used to transform data into new data (Figure 1).

We evaluate Sculpin through three demonstrations: transforming JSON from one format into another, building an API-backed search interface, and crafting a document-backed TODO list application. These demonstrations show that Sculpin can be used for realistic programming tasks through continuous and familiar direct-manipulation interactions. To more systematically evaluate Sculpin, we apply a heuristic analysis using the "Technical Dimensions of Programming Systems" (TDPS) taxonomy introduced by Jakubovic et al. [28], which situates Sculpin in a space of related programming systems. Among other findings, it highlights Sculpin's tight feedback loops, but points to unresolved challenges around editing programs after writing them and specifying higher-level, semantic steps. We close by speculating how our design commitments might guide the creation of novel programming systems on diverse media which share Sculpin's expressivity and directness.

In summary, this paper contributes:

(1) *Sculpin*, a programming interface enabling direct-manipulation transformation of a hybrid medium of JSON data and interface elements.
(2) Evaluations of Sculpin through demonstrations and a heuristic analysis.

2 RELATED WORK

2.1 Programming by Demonstration

A rich line of work in *programming by demonstration* (PbD) attempts to make programming possible for end-users by letting them demonstrate an action and generating a reusable program from the demonstration. PbD systems have been explored for tasks like scraping web pages [11], wrangling data [4, 29], creating charts [37, 46, 48] and editing text [40]. Sculpin draws on many mechanisms that are well-established in the PbD literature. For example, Sculpin displays a history of user actions as a visible timeline (as in Chimera [32], Smart Bookmarks [26], Drawing Dynamic Visualizations [54], and commercial CAD software like Autodesk Fusion 360 [3]), and user intent is specified by directing selections (see §2.3). Below, we describe three ways Sculpin positions itself among PbD systems to achieve directness and versatility.

Small, precise steps. Direct manipulation (DM) is an interaction paradigm which maximizes *directness* – the "qualitative feeling that one is directly engaged with control of the objects – not with the programs, not with the computer, but with the semantic objects of our goals and intentions" [27]. In a DM interface, changes to objects are incremental and immediately visible [50]. In contrast, many PbD systems undermine directness by performing large-step *inference*

of user intent from under-specified or implicit user actions using heuristic or machine-learning techniques. Some of these systems, like Eager [13], Metamouse [38], and Chimera [32], let users work with conventional interfaces and then infer patterns from their actions. Other systems, like FlashFill [20] and Wrex [15], are better described as *programming by example* (PbE) as they synthesize code entirely from input-output examples. In all these cases, inference complicates a feeling of direct control over media, because a user's actions no longer translate into predictable effects. When too much control is handed off to an inference-driven system, the user takes on the role of a manager who acts out steps for an employee to follow and then monitors the employee's work to ensure it matches their intent. The resulting distance between the user, the system's action, and the medium undermines the "confidence and mastery" that direct manipulation ought to produce [50].[1]

Instead of using inference, some PbD systems present novel palettes of interactions with which users can unambiguously express generalizable intent. This includes the earliest PbD system, Pygmalion [51], and more recent work like Gneiss [8] and Victor's "Drawing Dynamic Visualizations" demo [54]. Sculpin follows this approach in order to ensure users stay in control, introducing mechanisms like "sculptable selections" in lieu of inference.

General-purpose steps. PbD systems are often task specific. For example, there is a rich line of work in web-scraping with direct manipulation [11], and a separate line of work exploring how web pages can be modified by direct manipulation [36]. Sculpin explores the possibility of designing tools fit for a broader range of tasks by centering actions on a medium – in Sculpin's case, JSON. A user of Sculpin can perform a broad range of tasks with the same underlying mechanisms. This unity provides downstream benefits like unexpected re-appropriations and skill transfer between tasks.

As an example of the possibilities that open up by working with general-purpose operations in prior work, we can compare DM chart authoring systems (Lyra [48], Charticulator [46] and Data Illustrator [37]) to Victor's "Drawing Dynamic Visualizations" demo (DDV) [54]. Although Victor frames DDV as a tool for making data visualizations, it actually provides general-purpose operations on vector graphics (shapes) rather than higher-level operations specific to the domain of data visualizations (marks, encodings, axes). As a result, it can not only be used to plot datasets, but also to graphically define simulations of dynamical systems; he demonstrates an "implementation of the spring equation, specified entirely geometrically" [54]. By providing general-purpose operations, DDV explores a broader space of possibilities, though it may take more steps than more specialized tools to accomplish the same goals.

Working directly on a medium. Inspired by Bostock et al.'s [6] principle of *representational transparency*, we seek to provide direct access to our underlying medium (JSON) rather than replacing it with intermediate representations. Representational transparency makes it easier to maintain full expressivity within a medium. Replacing a medium with a new representation produces opportunities for elements to be lost. For instance, Gneiss [9, 10] enables processing JSON data by re-representing the data into a hierarchical spreadsheet structure. But this intermediate representation leads to limitations: Gneiss cannot process JSON with certain structures, like record-sets stored in keyed objects rather than arrays, and it cannot produce precisely-structured JSON for downstream processes. Representational transparency also provides benefits for accessibility: users can leverage their familiarity with the original representation while working with the system.

[1]The impact of inference on directness exists on a spectrum, and can be relieved through strategies like keeping inference to smaller steps and establishing shared representations between users and inference engines [23].

2.2 JSON

Sculpin works on the medium of JSON data. We are interested in JSON partially because of its *ecological* importance. JSON has emerged as a lingua franca in the Web era. JSON is found in API requests and responses, as a storage format in "NoSQL" databases, and as a format for domain-specific languages [5, 39]. Furthermore, runtime data in scripting languages like JavaScript and Python often takes the form of data very much like JSON: assemblies of arrays and records containing primitives like strings and numbers. Because of the frequency and breadth of its use, we believe JSON (or comparable "semi-structured data" like XML [1]) is a critical target for end-user programming tools.

The most common way to create programs that work on JSON is with traditional programming languages like JavaScript and Python. Domain-specific languages are also available for working on JSON [7, 14, 18]. We share some concepts with these languages, such as selecting paths with wildcards and parallel operations on sets, though we operationalize them as DM features rather than as language features. Some systems explore DM programming atop JSON or similar hierarchical formats. The one most similar to our approach is Gneiss [9, 10], which we discussed in §2.1. SIEUFERD [4] applies a similar "nested relational model" approach [34], and works on relational data rather than JSON itself.

2.3 Selections and Selectors

Selections are a universal part of direct manipulation systems. The basic concept of selection has been extended and enriched over time, to include multiple simultaneously editable selections [41], generalization of selections according to specific attributes [24, 56], and the ability to navigate selections through structures [55]. Sculpin's sculptable selections make use of all these developments.

Query languages like JSONPath [19, 22], XPath [47], and CSS selectors [16] allow sets of nodes in a tree to be selected using paths containing wildcards. Sculpin uses a similar approach, though we extend it to support branching patterns and storing the results of "filter" actions. We also use the internal structure of these patterns to drive the behavior of structural actions (see §4.5.1).

3 SCULPIN IN ACTION

We introduce Sculpin with a scenario: Alex is visiting Los Angeles during an art festival. The online website offers a list interface to search for exhibitions, but not a map, which is what Alex needs to find interesting exhibitions as they travel the area. Alex finds a JSON API response backing the website which has data for all exhibitions. Alex knows that if they could transform this into a GeoJSON format, then they can easily feed it into an online tool.

Figure 2: Sculpin can be used to reshape data: Alex turns JSON from a festival website into GeoJSON that can generate a map.

Let's follow Alex as they use Sculpin to turn their data into the desired GeoJSON format:[2]

- Alex's data starts as a list of exhibitions, where each exhibition contains an array of lat-lng objects. Alex's first goal is to flatten this data into a single list of lat-lngs with exhibition titles. This can be done in two stages: first copy the titles down into the lat-lng objects, and then bring these objects up to the top level. Alex selects the first exhibition's title (❶), and, in response, Sculpin surfaces a *command palette* menu for possible next operations. Alex uses the palette to *generalize* their selection across to the remaining titles (❷). Upon generalizing, Sculpin offers Alex an opportunity to alter their generalization direction (indicated with a light box). In this case, the default generalization to all titles is already what Alex wants.
- With the titles selected, Alex drags them into the first location (❸). Sculpin then gives Alex an opportunity to generalize the drop target to all entries in the `locations` arrays, rather than just the first – which they take (❹).
- Alex would like to extract the location objects as a flat list. They shift the selection to the parent objects (❺) and then drag those objects out onto the root node (❻).
- Finally, Alex selects each location (❼) and uses the formula input entry to transform the object into GeoJSON with a small JavaScript template (❽). Alex could have done this transformation in smaller direct-manipulation steps, but with the data already in the right place, they finds it easier to type out a formula. With data as GeoJSON, Alex then copies it into an online tool to view a map of the art festivals.

At each step, the JSON data was visible, and manipulations were directly and unambiguously performed on the concrete values. Alex was able to draw on their existing knowledge of interaction idioms like mouse-based direct manipulation and keyboard shortcuts. As Alex interacted with the data, each step was recorded into a timeline similar to the one in Figure 3 Ⓐ. If new exhibitions are posted, they can re-run this timeline with the new data to create a new GeoJSON. Sculpin is also capable of creating styled interfaces from JSON. Figure 3 Ⓑ shows an alternate workflow where Alex groups festival exhibitions by theme and then displays each exhibition's image in place. This example also uses Sculpin's scratch space feature, which allows extra data to be attached to any object.

4 SCULPIN

Below, we describe Sculpin's interface and implementation in full.

4.1 Interface

Figure 3 illustrates an annotated overview of Sculpin's interface, which consists of:

(Ⓐ) *Timeline*: User actions are recorded into a linear timeline, which represents the program the user is building. A user can survey past JSON and selection states by hovering over steps in the timeline.

[2]Some interface elements and data properties have been simplified for brevity.

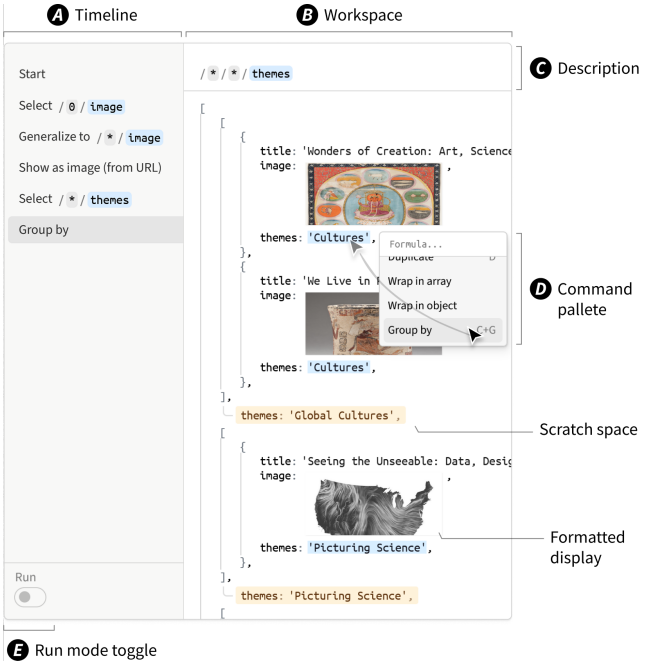

Figure 3: An alternative workflow where images of exhibitions are shown within Sculpin. The exhibitions are grouped by theme, which has added a scratch space containing the theme under each group.

(Ⓑ) *Workspace*: The workspace provides a single, focal view of Sculpin's *medium* (JSON data with extensions, see below) and a *selection* overlay on it.

(Ⓒ) *Description*: A description of the current selection is shown at the top as feedback for the action being performed.

(Ⓓ) *Command palette*: Next to the selection overlay, Sculpin shows a context menu containing a list of available actions on the current selection, and a formula text entry which allows input of JavaScript code for an "Apply formula" action.

(Ⓔ) *Run mode toggle*: When Sculpin is used to define an interface (as shown in §5.1 and §5.2), this toggle allows the user to enter a mode where this interface is available for interaction and program-editing features are disabled.

4.2 Medium

Sculpin's medium is JSON, extended in two ways. First, we add the ability to attach styles and user-interface elements to JSON nodes. With this extension, Sculpin can transform JSON not only into new JSON, but also into formatted displays and even interactive applications. Second, we provide *scratch space*: the ability to attach keys and values to not just objects but also arrays and primitives. When Sculpin is used to produce JSON output, the scratch space will not be included; it is solely for intermediate computations. These two extensions fit inside of JSON structures as lightweight augmentations, rather than replacing JSON with higher-level representations – preserving our commitment to representational transparency.

4.3 Selection

Sculpin uses a subject-verb metaphor for interaction. To specify the subject of their actions, the user sculpts a *selection*. When an action is performed, it acts on the selected part of the medium.

Selections in Sculpin support *multiple selection*, building off the multiple-selection mechanisms that have long been available in text editors [41]. When an action is applied to a selection with multiple items, it applies to the items in parallel. This obviates the need for loops in many cases, keeping the timeline linear and spreading computation over space instead of time. To maintain full expressivity, it is important that larger structural actions that draw upon many selected items at once (like "Group by") be similarly "loopable" from a single selection. To make this possible in Sculpin, we represent selections with a tree of patterns we call a *structured selection* (§4.5.1).

Users may sometimes want to select gaps – like the space between two array elements – where a part could be inserted by a future action. Here, we use the concept of a "seam": a selectable gap between parts. (In Figure 2 ❸, Alex drags titles onto a seam to insert them into objects.)

4.4 Actions

Sculpin provides two types of actions. A user selects a subject—their direct object of interest—by sculpting a selection using *selection actions*, and then operates on the parts underneath the selection with *medium actions*.

Selection actions. Selections are a familiar pattern from direct manipulation interfaces. In Sculpin, users also edit their selections in small steps, which lets them specify their intent unambiguously (without inference) and directly (on a concrete medium, not a symbolic representation thereof). Four primary types of selection actions work together to fulfill this role. *Concrete selection* selects a part of the JSON tree or a seam (space between values) directly, as in a conventional editor. Typically, this is followed by *generalization*: expanding from the particular selection to some broader category by abstracting over some part of it. (In Figure 2, we show these in steps ❶-❷.) It is often later important to *navigate* a selection through the medium. (Step ❺ is an example of this, moving from the location the title was dropped at to its parent.) Finally, *filtering* means narrowing a selection to a subset based on the data underneath the items. (This is shown in Figure 1 ❶.) Figure 4 Ⓐ provides a visual summary of selection actions and gestures to invoke them.

Medium actions. With a selection in hand, a user can specify edits to the medium using either a *per-item* action, where each selected item changes in the same way and does not alter surrounding structure, or a *structural* action, where larger changes to the medium occur. Figure 4 Ⓑ illustrates medium actions and the gestures to invoke them. Selection is maintained after medium actions, allowing actions to be chained.

4.5 Implementation

Sculpin is a web application, developed with React in TypeScript. It can be run as a standalone tool, or embedded into other web applications with an iframe. Demos of Sculpin are available online.[3]

[3] http://sculpin-uist25.github.io/paper.html

Programs created with Sculpin are represented in a JSON format which stores the steps recorded in the timeline. For example, the program shown in Figure 3 is represented as:

```
[
  { type: "MoveSelectionTo", path: [0, "image"] },
  { type: "GeneralizeSelection", pattern: [0] },
  { type: "ShowAs", show_as: "ImageFromURL" },
  { type: "MoveSelectionTo", path: [0, "themes"] },
  { type: "GroupBy" },
]
```

Sculpin interprets this program format directly. This interpreter logic could be split off for "headless" use of Sculpin programs.

4.5.1 Structured Selections. Sculpin uses a query-language-like structure called *structured selections* to represent selections. Our motivating example in §3 shows this structure at the top of each screenshot, like `/*/title` in Figure 2 ❷. This is a patterned path consisting of a wildcard `*`, meaning "take every key", followed by a concrete key `title`. Sculpin resolves each such path into a concrete set of nodes shown highlighted in the data. Sculpin also uses this structure to guide the behavior of structural operations. For instance, steps Figure 2 ❸-❹ construct a drag from `/*/title` to `/*/locations/*/title`. Because these share a leading `/*`, the drag becomes a loop of drags for each $/x$. The target contains an additional wildcard, so a nested loop is added: each $/x/$`title` is dragged to each $/x/$`locations`$/y/$`title`. This automatic looping behavior extends across the actions Sculpin supports and is essential to Sculpin's expressivity.

As described so far, structured selections match a subset of the JSONPath query language [19, 22], with novel semantics for actions like drag-and-drop. Sculpin adds two features: branching selections, which make patterned paths into patterned trees, and recording data-specific "exclusions" onto wildcards to support filtering. These features were not used in Figure 2, but are used in the following demos (§5). Appendix A further describes structured selections.

Note that selections in Sculpin are defined as patterns starting from the root of the JSON tree, proceeding a fixed number of steps. While they can refer to data at any fixed depth in the tree, they cannot refer to data at an *unknown* depth in the tree. This becomes a limitation if a user wishes to refer to arbitrarily-nested recursive structures, as might occur in ASTs of program code.

5 FURTHER DEMONSTRATIONS

We first demonstrated Sculpin in the map-making example in Figure 2. To further evaluate its expressive range, we now present two more demonstrations of it in use. To cover more ground, we will go into less step-by-step detail than we did in Figure 2.

5.1 Lightweight App: Image Quilt

With Sculpin, users can transform JSON not just into different JSON, but into user-facing interfaces. To do so, they use actions that apply styles to nodes or turn them into interface elements (*e.g.* turning booleans into checkboxes). The resulting interface, available in Sculpin's run mode, can be used to edit the original data: operations on output elements *bidirectionally* change the input.

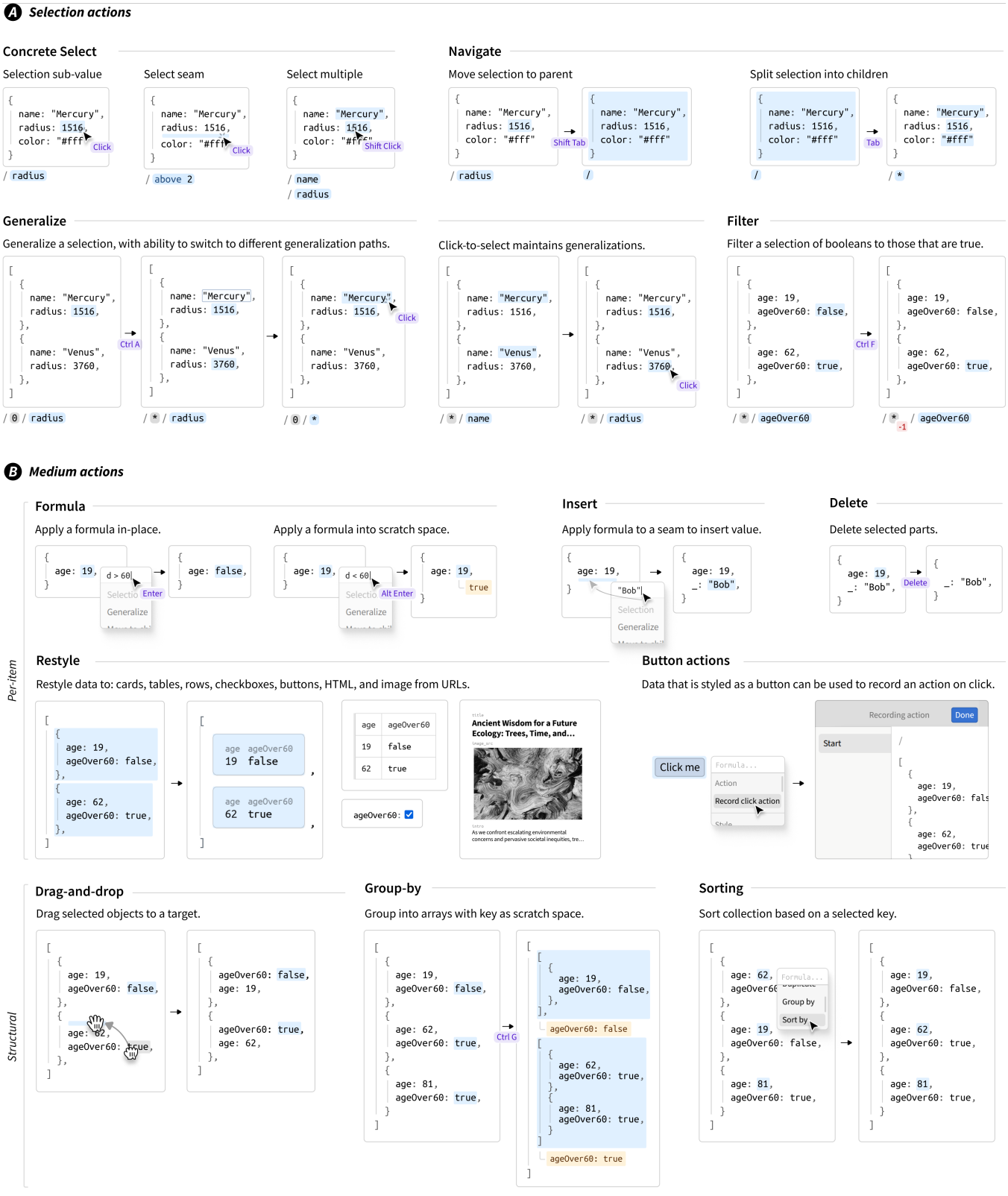

Figure 4: A summary of selection actions (A) and medium actions (B) in Sculpin.

Figure 5: Sculpin can make lightweight interfaces. Above, we create an interactive gallery application that shows artwork from the Art Institute of Chicago [53] matching a provided query. (Recreation of example from Horowitz and Heer [25].)

In Figure 5, we create an interactive image gallery of artworks using the Art Institute of Chicago API [53], demonstrating how a lightweight interface can be built up entirely within Sculpin.

The example begins by duplicating the query text (❶) to keep it available for editing later. Once the text is adapted into a URL, this URL can be fetched with an action that replaces it with the HTTP response (❷). As the response is sculpted, parts of it can be restyled (❹, ❺) using natural extensions of data-editing actions in Sculpin. Once the interface is constructed, we can interact with it by entering run mode where input values that are transformed into interface elements (without modification) can be edited. As such, the query text is editable (the system has tracked that it originated in a piece of the input). Changing the text updates the input and reruns the program (❻), producing a reactive interface.

Note that even after restyling, Sculpin maintains a close-to-data, 'wires sticking out' aesthetic. For example, object keys are shown even in the final interface. While a version of Sculpin could provide tight control over styling, we pursue this aesthetic to make handles on data visible for future modification, as we discuss in §8.2.

5.2 Document-Backed App: TODO List

Figure 6 illustrates how Sculpin can be used to make a TODO-list app atop a JSON document. We take as our premise a team building a shared TODO list app, where each item is assigned to a member.

This demonstration exercises a new feature of Sculpin: programming button presses. To add functionality on the Add todo button, we enter a mode to record a click action on the button (❺, ❻). In this mode, our actions are recorded onto a separate timeline which will act on the input data whenever the button is clicked. The same data shaping functionalities of Sculpin are available for this task. We drag the text into the list items and use a formula to reshape it into a TODO item format (❼, ❽). Later, in run mode, pressing the button adds a new item into the list (❿).

For this application to be useful, it must be embedded in an infrastructure that persists JSON documents. For this purpose, we used Patchwork [35], a platform for shareable data and tools comparable to Webstrates [30]. We elaborate on how open platforms like these could open up more uses for Sculpin in §8.2.

6 HEURISTIC EVALUATION

While our demonstrations of Sculpin show its range and directness, more systematic evaluation is necessary to characterize its limits. Towards that end, this section analyzes Sculpin according to the "Technical Dimensions of Programming Systems" (TDPS) taxonomy introduced by Jakubovic et al. [28]. The first two authors independently analyzed Sculpin dimension-by-dimension. Our discussion here synthesizes their results.

Feedback loops. This dimension refers to how systems bridge the gulfs of evaluation and execution [27]. Several aspects of Sculpin help bridge the gulf of evaluation: small-step actions make feedback loops short, and local actions focus attention on what must be reviewed after an action. Sculpin's formula editor could be improved: it provides no feedback until a formula is fully entered and applied. Sculpin's approach to bridging the gulf of execution is to show *all available data* in a single view so users can see precisely what they have to work with. Sculpin also shows *all available actions* in the

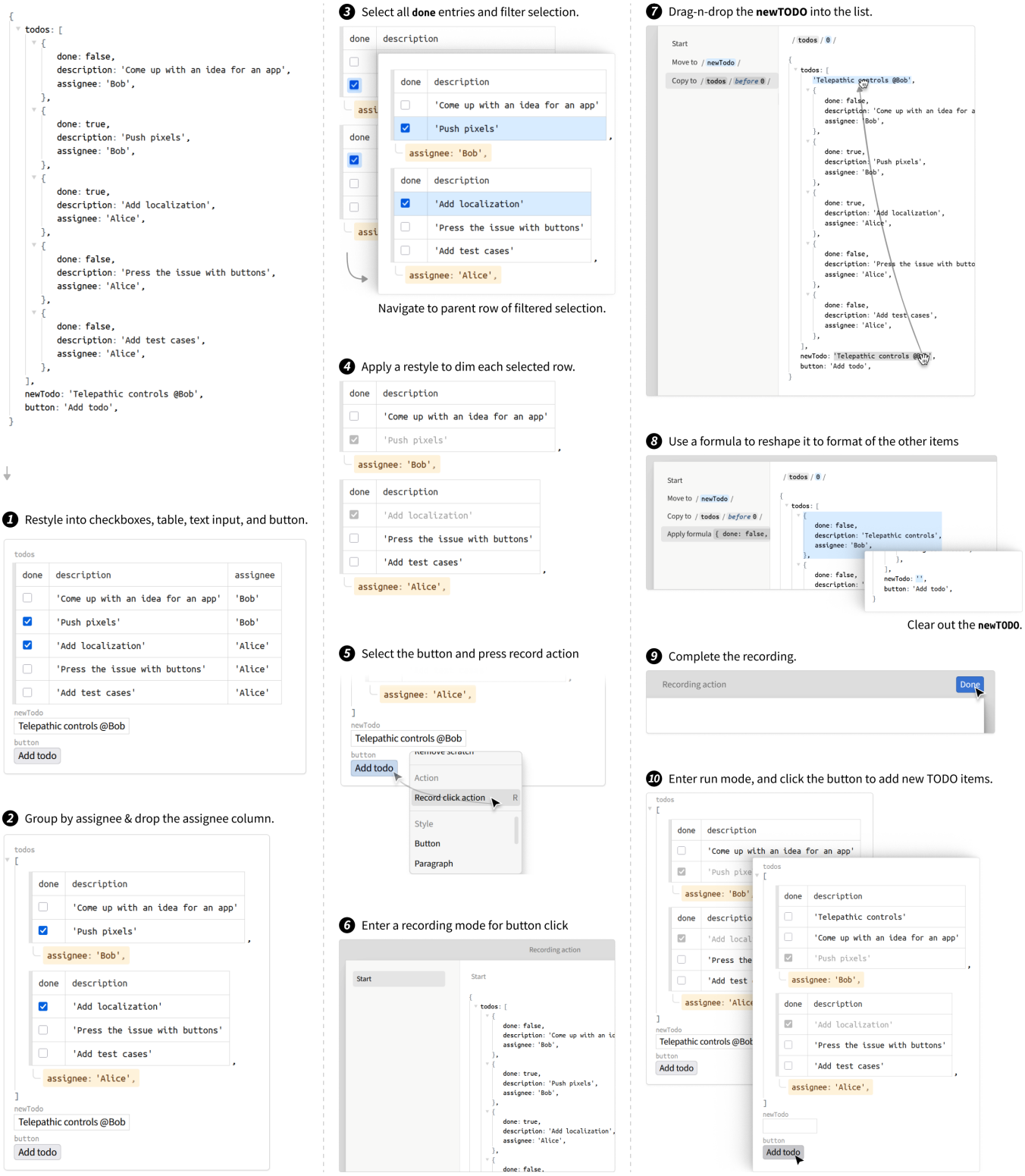

Figure 6: A TODO list being built with Sculpin. Steps ❶–❺ restyle the list into an interface that groups tasks by assignee and dims completed tasks. Steps ❻–❾ record an action for the "Add todo" button: adding a new task and clearing the text input. (Step ❹ uses a 'dim' restyle action that lowers opacity of selected elements.)

command palette. This provides guidance roughly comparable to a method auto-complete feature in a conventional coding environment. We also note that Sculpin's emphasis on small-step actions may present challenges for bridging the gulf of execution. It may be difficult for users to bridge the "semantic distance" between their larger goals and the small steps Sculpin provides [27].

Modes of interaction. Sculpin was deliberately designed to minimize the use of modes. For instance, drag-and-drop interactions with generalized targets were originally specified by sculpting the target of the drag-and-drop in a mode. We abandoned this approach in favor of the current approach in which drag-and-drop occurs immediately (without generalization) and the target can then be generalized modelessly. Sculpin does have separate modes for "recording" and "running". While this distinction serves a clear function (should edits affect the program or the input data?), blurring these lines could open up new possibilities.

Conceptual integrity vs openness. "Conceptual integrity" means employing "*unified concepts* that may *compose orthogonally* to generate diversity" [28]. Sculpin's design is built around several conceptual unifications: Actions are used to modify both media *and* selections. Medium actions allow both changes to data *and* restylings. Action timelines are used to make application views out of data *and* to define the behavior of button presses.

Factoring of complexity. "Factoring of complexity" asks what means a programming system provides for hiding details inside reusable components. In practical use of Sculpin, users will likely want to define, use, and share higher-level operations – a feature Sculpin currently lacks. We imagine higher-level operations like this could be defined inside our system as timelines to be called from timelines, like functions in traditional programming languages. Questions remain about the detailed design of such a feature, such as how multiple "arguments" could be provided to such an operation, and how a continuous, direct-manipulation feeling can be maintained as operations grow in size.

Level of automation. While many PbD systems are based around inferring intent from user actions, Sculpin is designed to let users express intent unambiguously (§2.1). The only place Sculpin deviates from this discipline is in its "Generalize" action, which uses heuristics to pick a default axis for generalization. This feels innocuous to us, as the system simultaneously reveals alternative axes and makes it easy for the user to switch – the heuristic inference is not a bottleneck, but a transparent, optional shortcut.

Error response. A Sculpin program might run into errors in two different situations: (1) A user records an action that produces an error: Because the timeline in Sculpin can be freely navigated, the user can undo the action and try again. (2) New input comes into a program and causes an error in an intermediate step: As we discuss further in §7, Sculpin does not yet have the means to modify intermediate steps in programs.

Learnability. The learnability of Sculpin is untested. That said, we have reasons to think it may fare well: Sculpin builds on familiar direct-manipulation techniques like drag-and-drop. Also, Sculpin provides fast feedback for user actions, so users can quickly learn the consequences of their actions. The "conceptual integrity" of Sculpin (described above) will aid learners by "collapsing stacks", using the same concepts and interactions to craft user interfaces as it uses to transform data between formats. On the other hand, Sculpin's generality may make it harder for learners to adapt it to their needs, compared to a more task-specific system [42].

7 LIMITATIONS

A complete version of Sculpin would provide additional actions (*e.g.* joins[4]). However, certain questions remain unresolved.

How can programs be edited in the middle? A significant limitation in Sculpin is the inability to edit intermediate parts of a program. A Sculpin program can only be edited by appending actions to the end of the timeline or by undoing actions. Many situations call for adjusting actions partway through a program (*e.g.* adapting a program to respond to an error triggered by new data). There are easy ways to begin to remedy this, like adding a movable "cursor" in the timeline to allow actions to be inserted at any point, but this raises new questions. For instance, what happens if a modification breaks the part of a program after the modification?[5]

How might users better work with unpredictable data? Like most PbD systems, Sculpin works best in situations with *predictable inputs*. A Sculpin user builds a program atop a representative input, which guides the actions they apply. If new inputs arrive with different structures, the program is unlikely to work correctly. Established PbD techniques, like examining multiple inputs in parallel, might ameliorate this problem [44].

How can loops and control flow be specified? Sculpin's multiple-selection system makes it easy to apply an action across items in collections, equivalent to a functional "map" operation. However, Sculpin does not have a imperative-style looping construct – its timeline is always linear. There are certainly contexts in which "map" is not enough; say, implementing a custom aggregation routine that must loop over a list while maintaining state. This could be addressed by adding loops to the timeline [54] or using recursive calls [17], but these both complicate Sculpin's linear timeline, sacrificing some measure of directness. Fortunately, general-purpose loops may not be as important to end-user programmers as higher-level operations like "map", which Sculpin supports well through multiple selections: "End-user programmers most often use systems where they do *not* write loops. Instead, they use vector-based operations — doing something to a whole dataset at once" [21].

How might Sculpin be used to build varied interfaces? When transforming JSON into a user-facing interface, a Sculpin user can only express a limited range of interfaces – those consisting of editable primitive values and script-invoking buttons. Sculpin does not currently support common interface patterns like: (1) direct manipulations that change the structure of collections, like dragging cards on a kanban board, (2) mouse events like hovering, (3) controllable reactivity, such as debouncing, throttling, and memoization.

[4]Gneiss [10] and SIEUFERD [4] both show how the results of joins can be represented as nested data which is easy to represent in JSON.
[5]Bakke and Karger [4] characterize difficulty with edits as a shared limitation of what they call "algebraic" interfaces, a category to which Sculpin belongs.

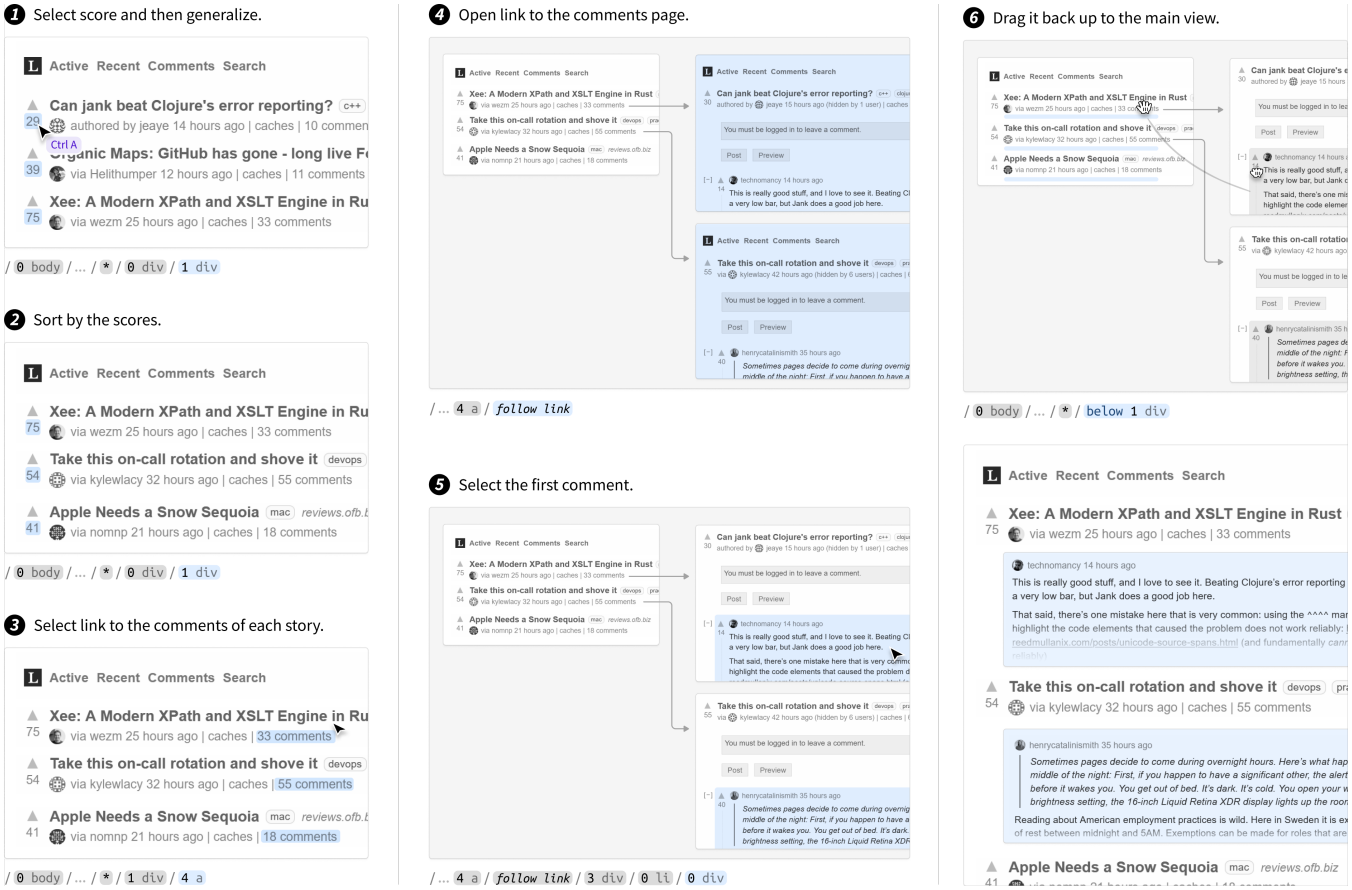

Figure 7: An imagined system modeled after Sculpin but applied to web pages rather than JSON, sorting links on a news aggregator (❶–❷) and pulling in comments from linked pages (❸–❻).

How might Sculpin be more usable? As a prototype, there are numerous ways Sculpin's usability could be improved. Promising directions include: (1) *Feedforward.* Previewing the effects of actions (like drags) would make it easier for users to fluidly explore and perform actions. (2) *Working with selections symbolically.* A Sculpin user usually interacts with a selection as a concrete overlay on JSON. However, in some cases it may be easier to edit a selection's symbolic representation as a tree of patterns. (3) *Animations.* Sculpin aspires to create the feeling of a material changing continuously. Animations (*e.g.* of drag operations) could support this. (4) *Controlling focus.* When data becomes complex, it becomes essential to find ways to focus the view on parts of interest. (5) *Organizing actions.* Actions in Sculpin are accessed through a long list. Different means of accessing and triggering actions (such as toolbars) might make them more organized and discoverable.

8 DISCUSSION & FUTURE WORK

8.1 Media beyond JSON

We believe the approach Sculpin takes to scripting transformations of JSON might be applicable to media beyond JSON. As an example, Figure 7 shows a sketch of how Sculpin's mechanisms (like sculptable, structured selections and drag-and-drop) could be repurposed to a medium consisting of web pages connected by links. While this figure shows a user customizing an existing web page to add features, the system we imagine here would not be specific to this task. Like Sculpin, it would provide medium-wide general-purpose operations, and could also support other tasks on web pages, like building scripts for web scraping (as in Rousillon [11]).

Sculpin's approach generalizes smoothly to new media because it preserves its subject medium. Many PbD systems instead substitute newly invented representations for the original representations of a medium's conventional editor, mixing programmatic structures into their media. As an example, Gneiss [9, 10] persistently stores user actions like sorts and groupings inside its spreadsheet representation. Approaches like these are medium-specific, and challenging to generalize to new media. In contrast, Sculpin provides programmatic power through overlays (like the selection & timeline) that are generic across media. For this reason, we believe that a Sculpin-like system may facilitate direct-manipulation programming on diverse media beyond JSON, from text documents to vector graphics.

8.2 Blending Data and Interfaces

Sculpin blends together data and interfaces into a single medium.[6] This is useful when building a traditional interface, as data structures act as "scaffolding" for parts of the interface that haven't been built yet. It also opens up further applications:

(1) Sculpin could incrementally style output data displayed in a live programming system like a computational notebook (*e.g.* certain values become headings, others become rows).
(2) Hybrids between between data and interface can act as purpose-specific tools. A colleague using Patchwork [35] wanted to retrieve text from a comment that had been deleted off a document. Using an embedding of Sculpin within Patchwork, we were able to quickly craft an alternative interface that showed all comments (including deleted ones) in a table, making it easy to find the desired comment. In this context, tools made with Sculpin can route around limitations of a primary editor – a hint of "malleable software" [35, 52].
(3) Interfaces built with Sculpin might maintain and expose enough structure that they could be interacted with programmatically. For instance, a user using the "TODO" app (§5.2) might use Sculpin's generalization features to batch-select todos associated with a given user and check them all in a single action. They would do this not with a feature implemented by the interface author, but using a general, open-ended feature of Sculpin. This relies on our "wires-sticking-out" approach to interfaces: even when data is shaped into an interface, the structure of the data is still visible to the user and accessible to programmatic action.

8.3 Bidirectional programming

Bidirectional programming is a form of PbD in which demonstrations perform edits on ordinary, human-editable code [12]. This approach has a number of possible benefits – it offers a clear representation of the underlying program and affords an alternative route for editing it. This ameliorates a number of classic issues with PbD articulated by Lau [33]. Sculpin's current implementation is subject to these classic issues. Although Sculpin exposes a representation of the program in the form of a timeline, it is not designed as an independently legible artifact. Furthermore, as discussed in §7, this timeline representation is not directly editable. A bidirectional approach might resolve these issues, though it may be complex to implement, as Sculpin's construction and use of selections does not map line-by-line to idiomatic code in a language like Python.

9 CONCLUSION

This work contributes *Sculpin*, a programming-by-demonstration system for transforming JSON data into new forms and interactive applications. Sculpin aims for directness and versatility by offering operations that: *are small & precise*, *are general-purpose*, and *are defined directly on the medium*. To achieve these goals, Sculpin uses a novel mechanism called sculptable selections where a user specifies the target of their action in small, deliberate steps. Sculpin also extends JSON to a hybrid medium with interface elements, so that its mechanisms can be used to incrementally transform data into interfaces. Our examples of Sculpin in use and a heuristic analysis demonstrate Sculpin's directness and versatility, while revealing the need for new techniques to make Sculpin programs fluidly editable. The principles that underlay Sculpin's design seem well-suited to generalization to new media beyond JSON, which may help bring direct-manipulation programming to diverse end-user contexts.

[6] We were inspired by Schmitz's demonstration [49] of gradually building up an interface from a JSON skeleton.

ACKNOWLEDGMENTS

We thank Brian Hempel, Matthew Beaudouin-Lafon, Alice Chung, Amy Ko, Edward Misback, Josh Pollock, Emilia Rosselli Del Turco, Fuling Sun, and the anonymous reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation under Grant No. 2432644. This work was inspired by the vision and optimism of Yoshiki Schmitz, and we dedicate it to his memory.

REFERENCES

[1] Serge Abiteboul. 1997. Querying semi-structured data. In *Database Theory — ICDT '97*, Foto Afrati and Phokion Kolaitis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–18.
[2] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax To Textual Code. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA* (2020). https://doi.org/10.1145/3428290
[3] Autodesk. 2025. Fusion Help | Use the Timeline. https://help.autodesk.com/view/fusion360/ENU/?guid=ASM-USE-TIMELINE.
[4] Eirik Bakke and David R. Karger. 2016. Expressive Query Construction through Direct Manipulation of Nested Relational Results. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, San Francisco California USA, 1377–1392. doi:10.1145/2882903.2915210
[5] Marcel Borowski, Luke Murray, Rolf Bagge, Bager Kristensen, Arvind Satyanarayan, and Clemens Nylandsted Klokmose. 2022. Varv: Reprogrammable Interactive Software as a Declarative Data Structure. *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (2022).
[6] M. Bostock, V. Ogievetsky, and J. Heer. 2011. D^3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309. doi:10.1109/TVCG.2011.185
[7] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoč. 2017. JSON: Data model, Query languages and Schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 123–135. doi:10.1145/3034786.3056120
[8] Kerry Shih-Ping Chang and Brad A. Myers. 2014. Creating Interactive Web Data Applications with Spreadsheets. In *Symposium on User Interface Software and Technology (UIST)*.
[9] Kerry Shih-Ping Chang. 2016. *A Spreadsheet Model for Using Web Services and Creating Data-Driven Applications*. Ph. D. Dissertation. Carnegie Mellon University.
[10] Kerry Shih-Ping Chang and Brad A. Myers. 2016. Using and Exploring Hierarchical Data in Spreadsheets. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 2497–2507. doi:10.1145/2858036.2858430
[11] Sarah E. Chasins, Maria Mueller, and Rastislav Bodík. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *The 31st Annual ACM Symposium on User Interface Software and Technology, UIST 2018, Berlin, Germany, October 14-17, 2018*, Patrick Baudisch, Albrecht Schmidt, and Andy Wilson (Eds.). ACM, 963–975. doi:10.1145/3242587.3242661
[12] Ravi Chugh. 2016. Prodirect manipulation: bidirectional programming for the masses. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 781–784. doi:10.1145/2889160.2889210
[13] Allen Cypher. 1991. EAGER: Programming Repetitive Tasks by Example. In *Conference on Human Factors in Computing Systems (CHI)*. https://doi.org/10.1145/108844.108850
[14] Stephen Dolan. 2025. jq. https://jqlang.org/.
[15] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unifed Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. *Conference on Human Factors in Computing Systems (CHI)* (2020).
[16] Elika J. Etemad and Tab Atkins Jr. 2022. Selectors Level 4. https://www.w3.org/TR/selectors-4/. W3C Working Draft, 11 November 2022.

[17] Elliot Evans and Josh Horowitz. 2024. An invitation into Droste's Lair. https://vezwork.github.io/drostes-lair-post/.

[18] Daniela Florescu and Ghislain Fourny. 2013. JSONiq: The History of a Query Language. *IEEE Internet Computing* 17 (2013), 86–90.

[19] Stefan Gössner, Glyn Normington, and Carsten Bormann. 2024. JSONPath: Query Expressions for JSON. *RFC* 9535 (2024), 1–62. https://api.semanticscholar.org/CorpusID:268023987

[20] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Symposium on Principles of Programming Languages (POPL)*.

[21] Mark Guzdial. 2025. CS doesn't have a monopoly on computing education: Programming is for everyone. https://computinged.wordpress.com/2025/03/05/cs-doesnt-have-a-monopoly-on-computing-education-programming-is-for-everyone/.

[22] Stefan Gössner, Glyn Normington, and Carsten Bormann. 2024. JSONPath: Query Expressions for JSON. RFC 9535. doi:10.17487/RFC9535

[23] Jeffrey Heer. 2019. Agency plus automation: Designing artificial intelligence into interactive systems. *Proceedings of the National Academy of Sciences* 116, 6 (Feb. 2019), 1844–1850. doi:10.1073/pnas.1807184115

[24] 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* (Florence, Italy) *(CHI '08)*. Association for Computing Machinery, New York, NY, USA, 959–968. doi:10.1145/1357054.1357203

[25] Joshua Horowitz and Jeffrey Heer. 2023. Engraft: An API for Live, Rich, and Composable Programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 72, 18 pages. doi:10.1145/3586183.3606733

[26] Darris Hupp and Robert C. Miller. 2007. Smart bookmarks: automatic retroactive macro recording on the web. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (Newport, Rhode Island, USA) *(UIST '07)*. Association for Computing Machinery, New York, NY, USA, 81–90. doi:10.1145/1294211.1294226

[27] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1985. Direct Manipulation Interfaces. *Hum. Comput. Interact.* 1, 4 (1985), 311–338. doi:10.1207/s15327051hci0104_2

[28] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7, 3 (Feb. 2023), 13. doi:10.22152/programming-journal.org/2023/7/13 tdops.

[29] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Vancouver BC Canada, 3363–3372. doi:10.1145/1978942.1979444

[30] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. *Webstrates*: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, Charlotte NC USA, 280–290. doi:10.1145/2807442.2807446

[31] Amy J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*. IEEE, Rome, Italy, 199–206. doi:10.1109/VLHCC.2004.47

[32] David Kurlander and Steven Feiner. 1992. A history-based macro by example system. In *Proceedings of the 5th annual ACM symposium on User interface software and technology*. ACM, Monteray California USA, 99–106. doi:10.1145/142621.142633

[33] Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* (2009). http://www.aaai.org/ojs/index.php/aimagazine/article/view/2262

[34] Mark Levene and George Loizou. 1994. The nested universal relation data model. *J. Comput. System Sci.* 49, 3 (1994), 683–717. doi:10.1016/S0022-0000(05)80076-5 30th IEEE Conference on Foundations of Computer Science.

[35] Geoffrey Litt, Josh Horowitz, Peter van Hardenberg, and Todd Matthews. 2025. *Malleable Software: Restoring User Agency in a World of Locked-Down Apps*. Technical Report. Ink & Switch. https://www.inkandswitch.com/essay/malleable-software/ Accessed: 2025-06-10.

[36] Geoffrey Litt, Daniel Jackson, Tyler Millis, and Jessica Ayeley Quaye. 2020. End-user software customization by direct manipulation of tabular data. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2020, Virtual, November, 2020*. ACM, 18–33. doi:10.1145/3426428.3426914

[37] Zhicheng Liu, John Thompson, Alan Wilson, Mira Dontcheva, James Delorey, Sam Grigg, Bernard Kerr, and John Stasko. 2018. Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–13. doi:10.1145/3173574.3173697

[38] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. 1989. Metamouse: Specifying Graphical Procedures by Example. In *Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[39] Andrew M Mcnutt. 2022. No Grammar to Rule Them All: A Survey of JSON-style DSLs for Visualization. *IEEE Transactions on Visualization and Computer Graphics* 29 (2022), 160–170.

[40] Robert C. Miller. 2002. *Lightweight structure in text*. Ph. D. Dissertation. Carnegie Mellon University.

[41] Robert C. Miller and Brad A. Myers. 2001. Interactive Simultaneous Editing of Multiple Text Regions. In *USENIX Annual Technical Conference, General Track*. 161–174.

[42] Bonnie A. Nardi. 1993. *A small matter of programming: perspectives on end user computing*. MIT Press, Cambridge, MA.

[43] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3453483.3454059

[44] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming - Design and Implementation of An Integration of Live Examples Into General-purpose Source Code. *Art Sci. Eng. Program.* 3, 3 (2019), 9. doi:10.22152/programming-journal.org/2019/3/9

[45] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives On Liveness. *The Art, Science, and Engineering of Programming Journal* (2019). doi:10.22152/programming-journal.org/2019/3/1

[46] Donghao Ren, Bongshin Lee, and Matthew Brehmer. 2019. Charticulator: Interactive Construction of Bespoke Chart Layouts. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (Jan. 2019), 789–799. doi:10.1109/TVCG.2018.2865158

[47] Jonathan Robie, Michael Dyck, and Josh Spiegel. 2017. XML Path Language (XPath) 3.1. https://www.w3.org/TR/xpath-3/. W3C Recommendation, 21 March 2017.

[48] Arvind Satyanarayan and Jeffrey Heer. 2014. Lyra: An Interactive Visualization Design Environment: Lyra: An Interactive Visualization Design Environment. *Computer Graphics Forum* 33, 3 (June 2014), 351–360. doi:10.1111/cgf.12391

[49] Yoshiki Schmitz. 2025. *yoshiki on X: "I noticed when frontend work that I write a lot of code just iterating over data structures and extracting text from them into text nodes. I thought: what if we just rendered everything upfront, and then told the computer how we want it to look?" / X*. https://x.com/yoshikischmitz/status/1176643658591793153

[50] Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* (August 1983).

[51] David Canfield Smith. 1975. *Pygmalion: A Creative Programming Environment*. Ph. D. Dissertation. Stanford University.

[52] Philip Tchernavskij. 2019. *Designing and Programming Malleable Software*. Ph. D. Dissertation. Université Paris-Saclay, École doctorale n°580 Sciences et Technologies de l'Information et de la Communication (STIC).

[53] The Art Institute of Chicago. 2025. *Public API | The Art Institute of Chicago*. https://www.artic.edu/open-access/public-api

[54] Bret Victor. 2013. Drawing Dynamic Visualizations. https://vimeo.com/66085662. Presented at the Stanford HCI seminar on February 1, 2013.

[55] Philippe Voinov, Manuel Rigger, and Zhendong Su. 2022. Forest: Structural Code Editing with Multiple Cursors. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Auckland, New Zealand) *(Onward! 2022)*. Association for Computing Machinery, New York, NY, USA, 137–152. doi:10.1145/3563835.3567663

[56] Haijun Xia, Bruno Araujo, and Daniel Wigdor. 2017. Collection Objects: Enabling Fluid Formation and Manipulation of Aggregate Selections. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 5592–5604. doi:10.1145/3025453.3025554

A SELECTION MECHANISM

This appendix provides more details on Sculpin's selection mechanism, previously introduced in §4.5.1

Extensions. Sculpin makes two extensions to simple JSONPath-like paths: First, rather than proceeding linearly down a path, structured selections can branch at any point, forming a tree of patterns. For instance, a structured selection on `[{a: 1, b: 2}]` might start with a wildcard / `*`, followed by a branch to / `a` in parallel with a branch to / `b`. We notate this selection as / `*` / `a` `b` .

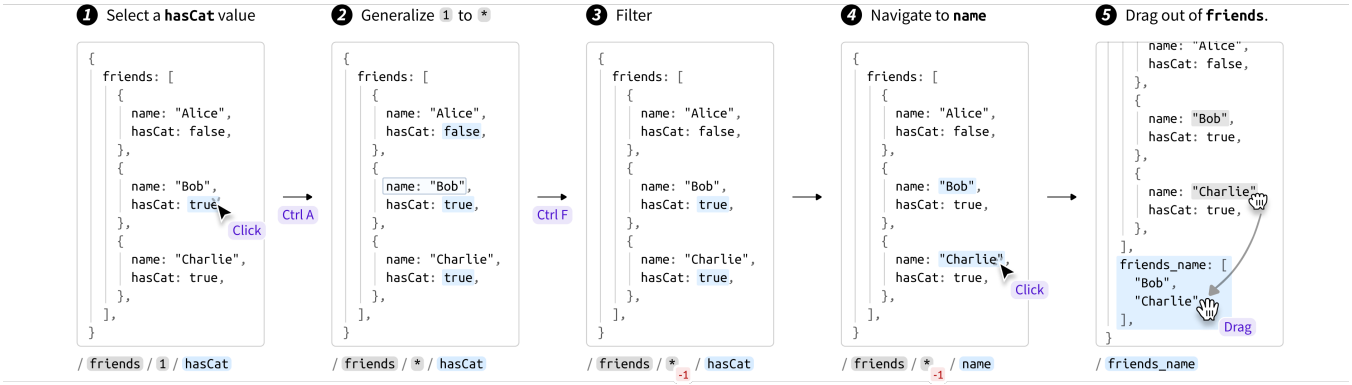

Figure 8: Using Sculpin to make a list of friends with cats, illustrating how structured selections guide the action.

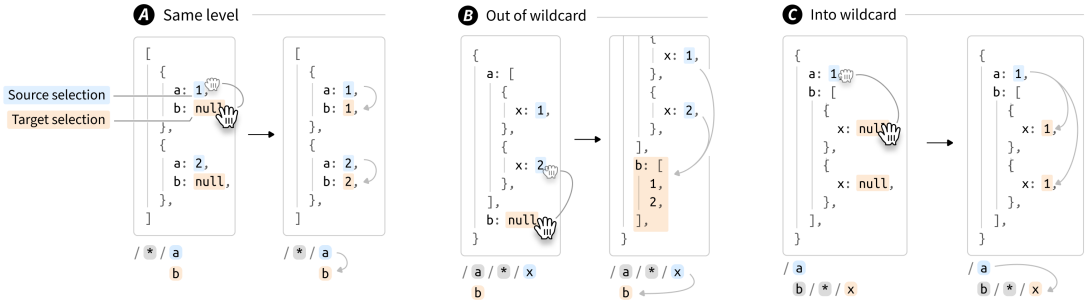

Figure 9: Drag-and-drop actions in Sculpin can A move values, B gather them together, or C spread them apart, depending on the structure of the source and target selections.

Second, wildcard steps can mark some paths in the data as "excluded", in order to represent the results of a data-based filter operation.[7] We summarize exclusions in our notation with a subscript marking how many paths have been excluded: / *-5 / a. These two extensions of JSONPath can interact: if a selection contains multiple paths branching after a wildcard, exclusions recorded on that wildcard will apply to all the branches.

Walkthrough. To further describe how structured selections enable structural navigation and sculpting operations, Figure 8 shows how a user might use Sculpin to make a list of friends with cats:

(1) They select a node in the data, which sets the selection to / friends / 1 / hasCat, a path of definite steps.

(2) Next, the user hits Ctrl-A to "Generalize" the selection. This action converts a single definite step in the pattern-path into a wildcard step. There are two possible generalizations: / friends / * / hasCat and / friends / 1 / *. (A third option, / * / 1 / hasCat would be possible, but this adds no additional nodes to the selection, so Sculpin does not surface it.) Sculpin observes that / friends / * / hasCat adds more nodes to the selection than / friends / 1 / *, so it picks this generalization by default. The user can switch which generalization they want by clicking one of the boxes that appear around nodes on other axes of generalization.

(3) The user hits Ctrl-F to "Filter" the selection. This filters out `false` values, resulting in exclusions on the wildcard: / friends / *-1 / hasCat.

(4) The user wants to access the names of these cat-having friends, so, they click one of these names. This invokes Sculpin's "parallel navigation" logic. The clicked node is at the path / friends / 2 / name. When moving to this from the existing selection / friends / *-1 / hasCat, Sculpin re-uses as much of the existing selection as possible, starting from the root. This results in the selection / friends / *-1 / name, maintaining the wildcard filter from before.

(5) Finally, the user drags the selections out to a seam at the end of the top-level object after "friends". If the drag target shared structure with the source, it would produce a loop of drags, but here that doesn't happen: the target is interpreted as a single node. The selected items are gathered together into an array and dropped into the target.

Drag-and-drop. The most versatile action in Sculpin is drag-and-drop. Figure 9 shows three different behaviors that can be created with drag-and-drop, depending on the structure of the source and target. In A, the source and target share a wildcard prefix, but without any wildcards below this prefix. This results in a loop of one-to-one drags. In B, the source has a wildcard the target doesn't. This results in a many-to-one drag, "extracting" content from the

[7] As an edge case, filtering a value that is not under a wildcard results in a exclusion mark on the root node.

source structure. In ⓒ, the target has a wildcard the source doesn't. This results in a one-to-many drag.

The behavior of drag-and-drop extends further than these basic examples. For instance, steps ❸-❹ of Figure 2 construct a drag from /`*`/`title` to /`*`/`locations`/`*`/`before 0`. Since the two paths share a leading wildcard step, a loop is created: for each child node in /`*`, a drag is performed from its /`title` to its /`locations`/`*`/`before 0`, which works as a one-to-many drag like Figure 9 ⓒ.

Designing selections and actions in Sculpin, we have been careful to make sure that any operation can be looped in any way by building the right kind of selection. For instance, the "Sort by" action receives a selection marking keys by which to sort elements of an array. If the keys are placed under the items of a single array, "Sort by" will sort that single array, but if they are under multiple arrays, "Sort by" will sort all these arrays simultaneously. If structural actions did not allow loops like this, the result ing system would lack expressivity.