Fast Direct Manipulation Programming with Patch-Reconciliation Correspondence

PARKER ZIEGLER, University of California, Berkeley, USA
JUSTIN LUBIN, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, Berkeley, USA

Direct manipulation programming gives users a way to write programs without directly writing code, by using the familiar GUI-style interactions they know from direct manipulation interfaces. To date, direct manipulation programming systems have relied on two core components: (1) a *patch* component, which modifies the program based on a GUI interaction, and (2) a *forward evaluator*, which executes the modified program to produce an updated program output. This architecture has worked for developing short-running programs—i.e., programs that reliably execute in <1 second—generating outputs such as SVG and HTML documents. However, direct manipulation programming has not yet been applied to long-running programs (e.g., data visualization, mapping), perhaps because executing such programs in response to every GUI interaction would mean crossing outside of interactive speeds. We propose extending direct manipulation programming to long-running programs by pairing a standard *patch* component (**`patch`**) with a corresponding *reconciliation* component (**`recon`**). **`recon`** directly updates the program *output* in response to a GUI interaction, obviating the need for forward evaluation.

We introduce corresponding **`patch`** and **`recon`** procedures for the domain of geospatial data visualization and prove them sound—that is, we show that the output produced by **`recon`** is identical to the output produced by forward-evaluating a **`patch`**-modified program. **`recon`** can operate both incrementally and in parallel with **`patch`**. Our implementation of our **`patch`**-**`recon`** instantiation achieves a 2.92× median reduction in interface latency compared to forward evaluation on a suite of real-world geospatial visualization tasks. Looking forward, our results suggest that *patch-reconciliation correspondence* offers a promising pathway for extending direct manipulation programming to domains involving large-scale computation.

CCS Concepts: • **Human-centered computing** → **User interface programming**; • **Software and its engineering** → **Graphical user interface languages**; **Integrated and visual development environments**.

Additional Key Words and Phrases: direct manipulation, direct manipulation programming, reconciliation, patch-reconciliation correspondence, cartokit, geospatial data

ACM Reference Format:
Parker Ziegler, Justin Lubin, and Sarah E. Chasins. 2025. Fast Direct Manipulation Programming with Patch-Reconciliation Correspondence. *Proc. ACM Program. Lang.* 9, PLDI, Article 175 (June 2025), 26 pages. https://doi.org/10.1145/3729278

1 Introduction

Direct manipulation programming systems integrate the point-click-modify interactions of graphical user interfaces (GUIs) with the flexibility and expressiveness of programming. Typically contrasted with command-line interfaces, direct manipulation interfaces [57] let users "act on displayed objects of interest using physical, incremental, and reversible actions whose effects are immediately

Authors' Contact Information: Parker Ziegler, peziegler@cs.berkeley.edu, University of California, Berkeley, Berkeley, California, USA; Justin Lubin, justinlubin@berkeley.edu, University of California, Berkeley, Berkeley, California, USA; Sarah E. Chasins, schasins@cs.berkeley.edu, University of California, Berkeley, Berkeley, California, USA.

ACM 2475-1421/2025/6-ART175
https://doi.org/10.1145/3729278

visible on the screen [56]." In the context of programming, direct manipulation typically means displaying an always-visible program *and* an always-visible program output. In addition to textual edits, the programmer can interact with graphical interface elements (e.g., menus, buttons, drop-downs, color pickers). As in other direct manipulation settings, the effects of these interactions should be "immediately visible on the screen." In contrast to non-programming settings, this means the user should see effects on two artifacts: the program and the program output.

A key challenge in the design of direct manipulation programming systems is the synchronization of program and output, often formalized as a "round-tripping" property [22]. Several existing systems (e.g., Sketch-n-Sketch [38], BiOOP [64]) maintain program-output synchrony via a bidirectional semantics, which introduces a *backward evaluation* relation defining how GUI actions propagate to the source program. Synchrony is preserved by subsequently forward-evaluating the updated program to produce the updated output. Implicit in this architecture is the assumption that this loop—backward evaluation followed by forward evaluation—is fast enough to maintain interactive speeds in a GUI.

While this assumption may hold for the domains these systems have explored to date (SVG [14, 29, 30], HTML documents [38, 64]), some tasks are longer-running—e.g., tasks involving large-scale computation over data. Forward evaluation in these contexts can be extremely expensive. Consider, for example, a simple program for rendering a data visualization such as a scatterplot or choropleth map. If a programmer manipulates display attributes of the visualization's marks, such as the color scheme or stroke width, forward evaluation will involve re-parsing and storing the underlying data in memory, re-computing scales to map data points to onscreen values, and re-drawing graphical marks. If the data happens to be stored remotely in a database or at an API endpoint, forward evaluation introduces the additional penalty of re-fetching over the network on every execution. If a given GUI action only affects a small subset of the data, forward evaluation will waste resources re-rendering many unaffected marks.

Given these challenges, we ask: *How can we avoid the performance cost of forward evaluation while ensuring that the program and output remain in agreement?* This paper aims to present an answer in the form of *patch-reconciliation correspondence*, a strategy for supporting direct manipulation programming that avoids forward-evaluating the updated program in response to every direct manipulation interaction. We instantiate patch-reconciliation correspondence in a direct manipulation programming system for data-intensive geospatial visualization.

Key Insight. Forward evaluation is a sensible synchronization mechanism because it guarantees exact program-output correspondence by construction. In lieu of running sophisticated and computationally expensive program analyses to check this property, we can assure it by simply evaluating the program. The problem, however, is that forward evaluation is too coarse; for a given output modification, it may repeat a lot of work that is not needed for synchronization. In fact, many output modifications can be distilled into very small transformations on the current program output. We introduce an approach for implementing a direct manipulation programming system by combining (1) a standard **`patch`** process for updating a program with (2) a reconciliation (**`recon`**) process for updating a program output. Importantly, we use **`patch`** and **`recon`** not only to prove our system sound, but also to *implement* our instantiation. By showing that **`patch`** and **`recon`** are conjugate—that is, that patching and then evaluating a program produces the same value as directly reconciling the program output—we eliminate the need for forward evaluation altogether; instead, we can rely exclusively on parallel applications of **`patch`** and **`recon`**.

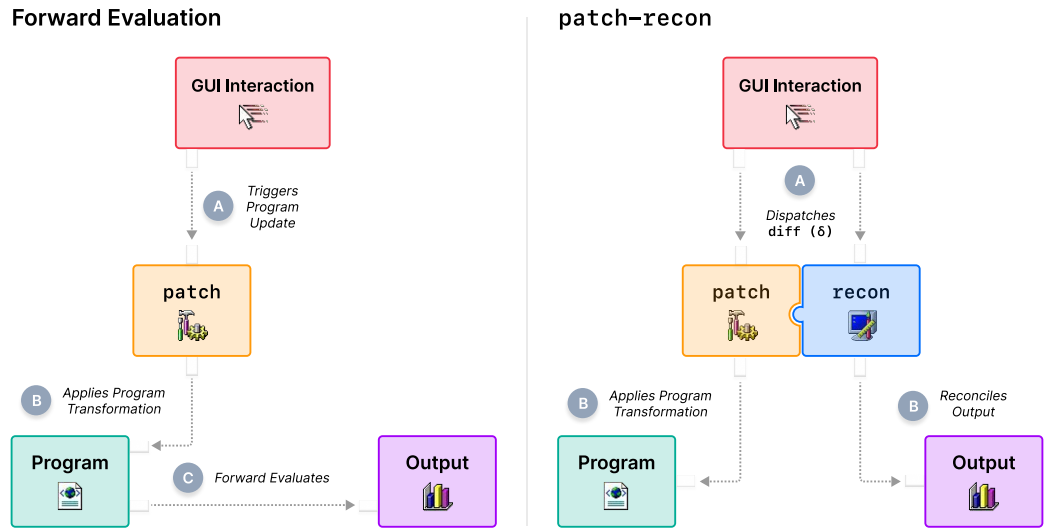


Fig. 1. **Comparing direct manipulation programming approaches.** In systems that use **Forward Evaluation** (including all prior direct manipulation programming systems), (A) a GUI interaction triggers a program update, (B) a **patch** function (e.g., backwards evaluation [38], fusion [66]) applies a synthesized program transformation to the program, and (C) the system forward-evaluates the program to produce the updated output. In contrast, our system uses a **patch-recon** approach, in which (A) a GUI interaction dispatches a **diff** (δ) and (B) **patch** and **recon** operate in parallel on this same **diff**. **patch** generates an updated program while **recon** generates a corresponding updated output. Proving correspondence between **patch** and **recon** is key to enabling this approach.

Contributions. This paper contributes:

(1) A strategy for program-output synchronization in direct manipulation programming systems, *patch-reconciliation correspondence*, that obviates the need for forward evaluation (Section 3).
(2) An instantiation of reconciliation (**recon**) for the domain of data-intensive geospatial visualization (Section 4). We prove reconciliation sound and demonstrate its equivalence to **patch** followed by forward evaluation.
(3) An implementation of this instantiation, and an evaluation on (1) the 80 reconciliation events required to reproduce six real-world maps published by national newsrooms, and (2) organic, in situ use of the system over a 30-day period (Section 6). Performance evaluation reveals that our patch-reconciliation approach can offer performance benefits for direct manipulation programming; we observed a 2.92× median reduction in interface latency and a 28.06× median speedup on code execution time compared to forward evaluation.

2 Overview

To demonstrate the key ideas of our patch-reconciliation approach (hereafter **patch-recon**) and its effect on performance in a direct manipulation programming system, we consider a concrete geospatial visualization example. Note this is just one instantiation of the technique; we show in Section 7.1 that **patch-recon** applies to direct manipulation programming systems targeting general-purpose programming languages, including those built on the λ-calculus with general recursion.

For now, suppose we are programming a scrollable, zoomable map showing wildfires in the United States using data from the National Interagency Fire Center's API[1]. We want the map to reveal where fires occurred (by county), the acreage each fire burned, and the fire's root cause (e.g., human activity, natural occurrence). We will assume that we are working within a direct manipulation environment in which a series of layers are placed atop a base map layer in order to create a visualization, as is standard for geospatial analyses.

patch *and* **recon**. To begin our exploration of **patch** and **recon**, we focus on a single interaction in our direct manipulation programming process. To set the stage, suppose the direct manipulation system has already fetched the geospatial data for wildfire perimeters and county boundaries in the United States. Suppose further that we have already added two visual layers to the output map in our system: (1) a layer of polygons whose boundaries are defined by the data for recorded wildfires and (2) a layer of polygons whose boundaries are defined by the data for U.S. counties.

Now we perform a GUI interaction to transform the wildfires layer into a *proportional symbol layer*. Rather than rendering polygons that represent the extent of each wildfire, a proportional symbol layer renders circles positioned at the geographic center of each wildfire; the size of the circle is proportional to the total acreage burned. (The bottom-right map of Figure 2 shows an example of a proportional symbol layer.) Executing this transformation is complex and expensive; the system needs to (1) iterate over all wildfire polygons in the dataset, (2) compute their geographic centers, (3) compute a scale function mapping the `acreage_burned` property to circle sizes, (4) evaluate that scale on all wildfires in the dataset, and (5) render the resulting marks to the screen.

Under **patch**-**recon**, this GUI interaction triggers two *parallel* operations:

(1) The **patch** operation applies a small program transformation, which we call a **diff**, to the system's current program. In our example, this involves generating—though not executing—the code for steps 1 through 5 above.
(2) The **recon** operation interprets this same **diff** and applies its *effects* to the output map. In our example, this involves computing steps 1 through 5 above.

It is critical to note that although these operations share a correspondence—**patch** updates the program while **recon** models the effect of that update on the output—they execute *independently*. Compare this to forward evaluation, in which the system must wait for **patch** to perform the program update before evaluating it to a new output map. Figure 2 provides an illustrated example of **patch**-**recon** for this interaction.

Beyond parallelization, **patch**-**recon** carries additional benefits in this scenario. Because the wildfires data is already stored in memory, the computation to transform the polygon layer to a proportional symbol layer can begin immediately without re-fetching data from the API. Additionally, because the computation only affects the wildfires layer, the system does not need to modify the program representation or rendered marks of the counties layer. Compare this again to forward evaluation, in which the system would re-fetch, re-parse, and reload the wildfires dataset from its API endpoint. Additionally, because forward evaluation does not reuse results from prior executions, the system must re-render the unmodified counties layer. This carries a separate cost of re-fetching, re-parsing, and reloading the counties data, in addition to re-instantiating the map.

This interaction alone demonstrates the core problem that reconciliation addresses. As we continue to make small output modifications (e.g., mapping the color of each circle to the dataset's `cause` property, reducing opacity to increase the visibility of overlapping circles), forward evaluation repeats *all prior computation*. Moreover, program update and program evaluation must always occur *sequentially*. In contrast, **patch**-**recon** operates *incrementally* and *in parallel*. Each new

[1] https://data-nifc.opendata.arcgis.com/

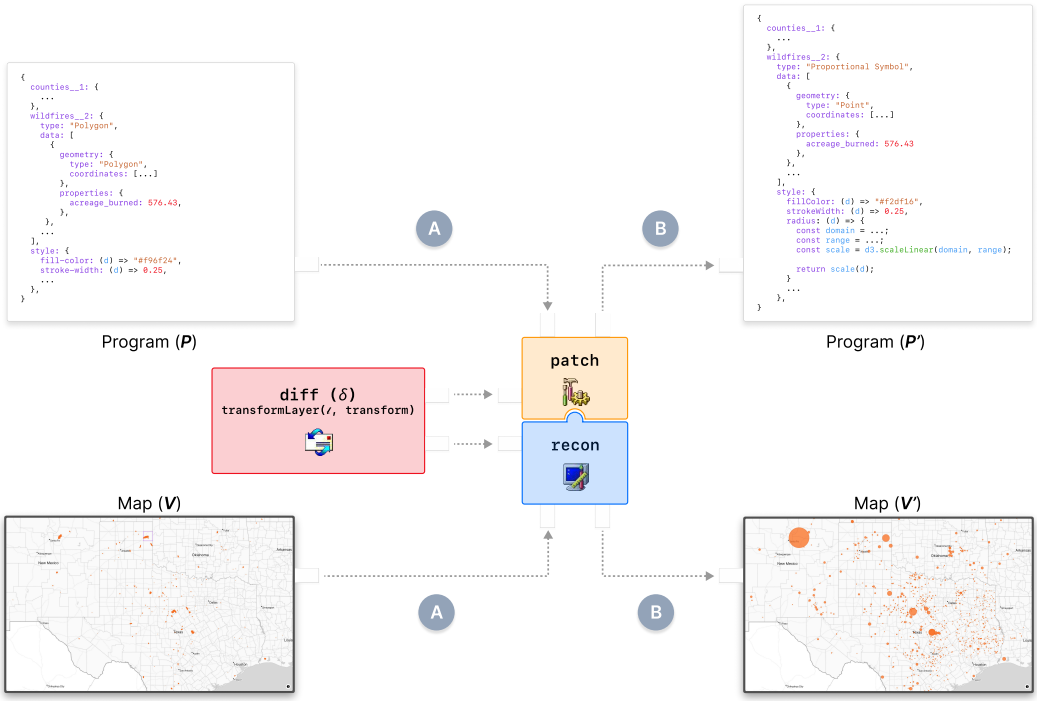


Fig. 2. **Our patch-recon approach applied to a geospatial visualization example.** (A) A GUI interaction triggers (1) the **patch** operation, supplying a **diff** and our program, P, as inputs, and (2) the **recon** operation, supplying the same **diff** and our map, V, as inputs. (B) **patch** applies the **diff** to produce the updated program, P', while **recon** interprets the **diff** to produce the updated map, V'. The program is never forward evaluated to produce the updated map, which visualizes wildfires using a proportional symbol layer.

output modification produces only a small **diff** from the prior program, and the **patch** and **recon** functions operate on this shared **diff** to synchronize the program and output, respectively.

Why not cache data? If data fetching, parsing, and storage is a primary bottleneck for forward evaluation of data-intensive programs, why not just cache the data? Indeed, as described in the example above, caching data in memory is a key factor in making our reconciliation implementation efficient. However, while this strategy would amortize the costs of data access, it would not eliminate all redundant computation performed by forward evaluation. To see why, let us extend our example above to a second GUI interaction.

Let us say the programmer performs a GUI interaction to map each circle's color to its corresponding wildfire's `cause` property, allowing them to identify which wildfires were human-caused versus naturally occurring. With caching in place, forward evaluation could skip re-fetching and re-parsing the data; however, we would still incur the cost of translating cached data into an efficient data structure for rendering. In our setting, this process—known as tiling—can be extremely expensive, particularly in a resource-constrained environment like a web browser. Worse yet, we pay the penalty for tiling on both the wildfires layer and the counties layer, even though the latter did not change as part of the output update. In contrast, reconciliation can sidestep tiling by modifying rendered marks in place, scoping the modification to the wildfires layer. This technique involves altering and re-running only the shader function that styles marks, which is comparatively cheap.

If caching at the data level is not sufficient, we may be tempted to lower our caching approach to an even finer granularity, such as the layer or mark level. In fact, this is one way of viewing what reconciliation does! As a system developer adds more caching, they essentially start implementing reconciliation in an ad hoc manner. The drawback of an ad hoc caching approach, however, is the lack of a soundness guarantee, which is a core contribution of this work. For a deeper discussion of how a **patch-recon** approach differs from traditional caching approaches, see Section 7.4.

patch-recon *Correspondence.* Prior direct manipulation programming systems have relied on forward evaluation to guarantee program-output agreement by construction. Since a language implementation is already available, reusing this machinery is a straightforward choice. But when forward evaluation is too expensive to achieve interactive speeds, we need some other mechanism to synchronize the program and output. We propose the pairing of **patch** and **recon** as an alternative strategy to achieve program-output agreement without the associated performance cost.

The key to our approach is a proof of the correspondence between these two functions, which we formalize in a *soundness* theorem (Sections 3 and 4). Absent soundness, we have no guarantee that the effects of **recon** on the output are captured by the effects of **patch** on the program, and vice versa. In this context, every GUI interaction would have the potential to lead to divergence of the program and output. Consider our working example above. Imagine that the transition to a proportional symbol layer succeeds on the **recon** side, but **patch** applies an incorrect corresponding program transformation. If we attempted to execute the generated program, we would get either a different output map or, in a pathological case, a runtime error. As we continue to make modifications within the system, the problem would only compound. For example, because **patch** would apply successive program updates assuming the transition was already encoded in the program, it could introduce code that modifies non-existent layer properties, performs impossible data transformations, or tries to execute a myriad of other degenerate actions. As we will see in the next section, the solution to these problems is soundness, which guarantees that the output produced by **recon** is identical to the output produced by forward evaluating the **patch**ed program.

3 Problem Statement

In this section, we formalize the constraints on the reconciliation function and establish the foundations for soundness.

Definition 3.1. For purposes, a **language** $\mathcal{L}$ has:

(1) A set of programs $\mathsf{Prog}_{\mathcal{L}}$.
(2) A set of values $\mathsf{Val}_{\mathcal{L}}$.
(3) A semantics $\mathbf{eval} : \mathsf{Prog}_{\mathcal{L}} \to \mathsf{Val}_{\mathcal{L}}$.
(4) A set of diffs $\mathsf{Diff}_{\mathcal{L}}$.
(5) A syntactic diffing operation $\mathbf{patch} : \mathsf{Diffs}_{\mathcal{L}} \times \mathsf{Prog}_{\mathcal{L}} \to \mathsf{Prog}_{\mathcal{L}}$.

We omit the $\mathcal{L}$ subscript when clear from context.

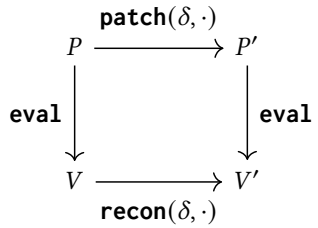
*Definition 3.2 (**Problem Statement**).* A **reconciliation function** is a function

$$\mathbf{recon} : \mathsf{Diff} \times \mathsf{Val} \to \mathsf{Val}$$

such that, for any diff δ,

$$\mathbf{eval}(\mathbf{patch}(\delta, P)) = \mathbf{recon}(\delta, \mathbf{eval}(P)).$$

Graphically, the following square must commute.



4 Reconciliation for Direct Manipulation Programming

We now instantiate our problem statement in the context of geospatial visualization. In addition to our discussion of how **patch**-**recon** can apply to general-purpose programming languages with recursion (Section 7.1), we will also later discuss how this particular instantiation is representative of many existing languages for tasks that are amenable to direct manipulation (Section 7.2).

4.1 Syntax, Semantics, and Patches

In the following sections, we define the syntax and semantics of language $\mathcal{L}_{ck}$ as well as a notion of **patch**es, following the structure of Definition 3.1.

Programs. Figure 3 defines the set of programs, $\mathsf{Prog}_{\mathcal{L}_{ck}}$, in the language. Intuitively, a program $p \in \mathsf{Prog}_{\mathcal{L}_{ck}}$ comprises a dictionary of map layer definitions.

Values. The set of values, $\mathsf{Val}_{\mathcal{L}_{ck}}$, in the language is the set of **maps**, as defined in Figure 3. A **map** M comprises a set of graphical **marks**. Each mark represents a graphical depiction of a single data point in a dataset and consists of a mark type (mt), associated layer id, associated datapoint, and associated mapping of channels (attributes like fill-color or stroke-width) to functions that render these channels.

Semantics. Programs in our language $\mathcal{L}_{ck}$ are dictionaries of map layer definitions; thus, to introduce the semantics **eval** of our language $\mathcal{L}_{ck}$, we first define layer evaluation **evalLayer** (Figure 4, left). Intuitively, layer evaluation takes a layer $\ell \in P$ and evaluates it to a set of rendered marks. As part of this process, a function, getMT, interprets a layer's type, T, and returns the corresponding mark type, mt, for the set of rendered marks. We can then define map evaluation **eval** (Figure 4, right), which evaluates all layers to the set of all marks.

Full program evaluation is quite expensive in our context. Given the large sizes of geospatial datasets (typically 10–500 MB, consisting of tens of thousands to hundreds of thousands of features), the geometric complexity of rendered marks in a layer, and the linear time complexity of evaluation with respect to the number of data points across all layers, avoiding full program evaluation where possible is critical for performance.

Diffs. Our definition of **diff**s $\delta_{\mathcal{L}_{ck}}$ is included in Figure 3. We briefly describe each **diff** below.

(1) setChannel(id, C, fn) includes a layer id, a channel (C) to add or modify on the associated layer's style (S) and a function (fn) mapping each datum (d) in the layer to a stylistic value for the supplied channel.
(2) removeChannel(id, C) includes a layer id and a channel (C) to remove from the associated layer's style (S).

$$
\begin{aligned}
\textbf{Prog } P &::= \{\mathsf{id} \mapsto \ell\} \\
\textbf{Layers } \ell &::= \langle \mathsf{type} : T, \mathsf{data} : D, \mathsf{style} : S \rangle \\
\textbf{Layer Types } T &::= \mathsf{Point} \mid \mathsf{Line} \mid \mathsf{Polygon} \mid \mathsf{Choropleth} \mid \mathsf{Proportional\ Symbol} \mid \mathsf{Dot\ Density} \\
\textbf{Data } D &::= [\mathsf{d}_i, \ldots, \mathsf{d}_N]^{N \geq 1} \\
\textbf{Channel } C &::= \mathsf{fill\text{-}color} \mid \mathsf{stroke\text{-}width} \mid \mathsf{thresholds} \mid \mathsf{classifier} \mid \mathsf{min\text{-}radius} \mid \mathsf{dot\text{-}value} \mid \ldots \\
\textbf{Style } S &::= \{C \mapsto \mathsf{fn}\} \\
\\
\textbf{Maps } M &::= \{m_1, \ldots, m_N\}^{N \geq 0} \\
\textbf{Mark } m &::= \mathtt{mark}(\mathsf{mt}, \mathsf{id}, \mathsf{d}, S) \\
\textbf{Mark type } \mathsf{mt} &::= \mathsf{Point} \mid \mathsf{Line} \mid \mathsf{Polygon} \\
\\
\textbf{Diff } \delta &::= \mathsf{setChannel}(\mathsf{id}, C, \mathsf{fn}) \\
&\quad \mid \mathsf{removeChannel}(\mathsf{id}, C) \\
&\quad \mid \mathsf{addLayer}(\mathsf{id}, \ell) \\
&\quad \mid \mathsf{removeLayer}(\mathsf{id}) \\
&\quad \mid \mathsf{transformLayer}(\mathsf{id}, \mathsf{transform})
\end{aligned}
$$

Fig. 3. **The definition of programs in $\mathcal{L}_{ck}$.** d refers to an individual datum within a layer. id refers to a layer's unique string identifier. fn is an abstract function that takes a layer datum d as input and returns a corresponding stylistic value for a given channel.

$$
\textsc{Eval-Layer}\ \frac{\mathsf{mt} = \mathsf{getMT}(T)}{\mathtt{evalLayer}(\mathsf{id}, \langle T, D, S \rangle) = \{\mathtt{mark}(\mathsf{mt}, \mathsf{id}, \mathsf{d}, S) \mid \mathsf{d} \in D\}}
\qquad
\textsc{Eval-Map}\ \frac{}{\mathtt{eval}(P) = \bigcup_{\mathsf{id} \in P} \mathtt{evalLayer}(\mathsf{id}, P[\mathsf{id}])}
$$

Fig. 4. Forward evaluation for $\mathcal{L}_{ck}$.

(3) addLayer(id, ℓ) includes a *fresh* layer id and a *fresh* layer definition (ℓ).
(4) removeLayer(id) includes an *existing* layer id to use for targeted layer removal.
(5) transformLayer(id, transform) includes a layer id and a transform function ($\langle T, D, S \rangle \to \langle T', D', S' \rangle$) to apply to the associated layer. transform operates in practice by mapping a subroutine, $\mathsf{transform}_1 : \langle T, \mathsf{d}, S \rangle \to \langle T', \mathsf{d}', S' \rangle$, over all $\mathsf{d} \in D$.

Our formalism assumes a validity constraint on **diff**s, namely that (1) id is guaranteed to exist in P for setChannel, removeChannel, removeLayer, and transformLayer and (2) id is guaranteed to *not* exist in P for addLayer. Our implementation (Section 5) enforces this constraint.

Patch. The $\mathtt{patch}_{\mathcal{L}_{ck}}$ function applies a $\mathtt{diff}\ \delta_{\mathcal{L}_{ck}}$ to an existing $p \in \mathsf{Prog}_{\mathcal{L}_{ck}}$ to yield a new $p' \in \mathsf{Prog}_{\mathcal{L}_{ck}}$. Intuitively, **patch** modifies the minimal portion of the program AST based on the value of the **diff** δ. We define $\mathtt{patch}_{\mathcal{L}_{ck}}$ in Figure 5.

$$\textsc{Patch/Set-Channel}\quad \frac{P(\mathsf{id}) = \langle T, D, S\rangle}{\mathbf{patch}(\mathsf{setChannel}(\mathsf{id}, C, \mathsf{fn}), P) = [\mathsf{id} \mapsto \langle T, D, [C \mapsto \mathsf{fn}]S\rangle]P}$$

$$\textsc{Patch/Remove-Channel}\quad \frac{P(\mathsf{id}) = \langle T, D, S\rangle}{\mathbf{patch}(\mathsf{removeChannel}(\mathsf{id}, C), P) = [\mathsf{id} \mapsto \langle T, D, S \setminus \{C\}\rangle]P}$$

$$\textsc{Patch/Add-Layer}\quad \frac{}{\mathbf{patch}(\mathsf{addLayer}(\mathsf{id}, \langle T, D, S\rangle), P) = [\mathsf{id} \mapsto \langle T, D, S\rangle]P}$$

$$\textsc{Patch/Remove-Layer}\quad \frac{}{\mathbf{patch}(\mathsf{removeLayer}(\mathsf{id}), P) = P \setminus \{\mathsf{id}\}}$$

$$\textsc{Patch/Transform-Layer}\quad \frac{P(\mathsf{id}) = \langle T, D, S\rangle \qquad \mathsf{transform}(T, D, S) = \langle T', D', S'\rangle}{\mathbf{patch}(\mathsf{transformLayer}(\mathsf{id}, \mathsf{transform}), P) = [\mathsf{id} \mapsto \langle T', D', S'\rangle]P}$$

Fig. 5. Our implementation of **patch** for $\mathcal{L}_{ck}$, a syntactic program update.

$$\textsc{Recon/Set-Channel}\quad \frac{M' = \{\mathbf{mark}(\mathsf{mt}, \mathsf{id}, \mathsf{d}, [C \mapsto \mathsf{fn}]S) \mid \mathbf{mark}(\mathsf{mt}, \mathsf{id}, \mathsf{d}, S) \in M\} \qquad M'' = \{\mathbf{mark}(\mathsf{mt}, \mathsf{id}', \mathsf{d}, S) \mid \mathbf{mark}(\mathsf{mt}, \mathsf{id}', \mathsf{d}, S) \in M, \mathsf{id}' \neq \mathsf{id}\}}{\mathbf{recon}(\mathsf{setChannel}(\mathsf{id}, C, \mathsf{fn}), M) = M' \cup M''}$$

$$\textsc{Recon/Remove-Channel}\quad \frac{M' = \{\mathbf{mark}(\mathsf{mt}, \mathsf{id}, \mathsf{d}, S \setminus \{C\}) \mid \mathbf{mark}(\mathsf{mt}, \mathsf{id}, \mathsf{d}, S) \in M\} \qquad M'' = \{\mathbf{mark}(\mathsf{mt}, \mathsf{id}', \mathsf{d}, S) \mid \mathbf{mark}(\mathsf{mt}, \mathsf{id}', \mathsf{d}, S) \in M, \mathsf{id}' \neq \mathsf{id}\}}{\mathbf{recon}(\mathsf{removeChannel}(\mathsf{id}, C), M) = M' \cup M''}$$

$$\textsc{Recon/Add-Layer}\quad \frac{\mathbf{evalLayer}(\ell) = M'}{\mathbf{recon}(\mathsf{addLayer}(\ell, M)) = M \cup M'}$$

$$\textsc{Recon/Remove-Layer}\quad \frac{M' = \{\mathbf{mark}(\mathsf{mt}, \mathsf{id}', \mathsf{d}, S) \mid \mathbf{mark}(\mathsf{mt}, \mathsf{id}', \mathsf{d}, S) \in M, \mathsf{id}' \neq \mathsf{id}\}}{\mathbf{recon}(\mathsf{removeLayer}(\mathsf{id}), M) = M'}$$

$$\textsc{Recon/Transform-Layer}\quad \frac{M' = \{\mathbf{mark}(\mathsf{getMT}(T'), \mathsf{id}, \mathsf{d}', S') \mid \mathbf{mark}(\mathsf{mt}, \mathsf{id}, \mathsf{d}, S) \in M, \mathsf{transform}_1(\mathsf{getMT}^{-1}(\mathsf{mt}), \mathsf{d}, S) = T', \mathsf{d}', S'\} \qquad M'' = \{\mathbf{mark}(\mathsf{mt}, \mathsf{id}', \mathsf{d}, S) \mid \mathbf{mark}(\mathsf{mt}, \mathsf{id}', \mathsf{d}, S) \in M, \mathsf{id}' \neq \mathsf{id}\}}{\mathbf{recon}(\mathsf{transformLayer}(\mathsf{id}, \mathsf{transform}), m) = M' \cup M''}$$

Fig. 6. Our implementation of **recon** for $\mathcal{L}_{ck}$, a semantic value update.

4.2 Reconciliation Function

With our framework in place, we now define our reconciliation function $\mathbf{recon}_{\mathcal{L}_{ck}}$ (Figure 6). $\mathbf{recon}_{\mathcal{L}_{ck}}$ applies a **diff** $\delta_{\mathcal{L}_{ck}}$ to an existing $M \in \mathsf{Val}_{\mathcal{L}_{ck}}$ to yield a new $M' \in \mathsf{Val}_{\mathcal{L}_{ck}}$. Intuitively, **recon** modifies the minimal number of marks on the map based on the value of the **diff** δ.

patch-recon *Correspondence by Example.* Imagine we have the following program containing two layers, a Point layer and a Choropleth layer.[2]

$$P = \{\mathsf{id}_1 \mapsto \langle \text{Point}, D_1, S_1\rangle, \mathsf{id}_2 \mapsto \langle \text{Choropleth}, D_2, S_2\rangle\}$$

When evaluated by $\textbf{eval}_{\mathcal{L}_{ck}}$, this program yields a set of **marks**:

$$M = \{\textbf{mark}(\text{Point}, \mathsf{id}_1, \mathsf{d}, S_1) \mid \mathsf{d} \in D_1\} \cup \{\textbf{mark}(\text{Polygon}, \mathsf{id}_2, \mathsf{d}, S_2) \mid \mathsf{d} \in D_2\}$$

Now, we trigger a GUI interaction to remove the stroke-width channel from *just* the Point layer. This produces the following **diff**:

$$\delta = \text{removeChannel}(\mathsf{id}_1, \text{stroke-width})$$

$\textbf{patch}_{\mathcal{L}_{ck}}$ and $\textbf{recon}_{\mathcal{L}_{ck}}$ commence operation on this **diff** in parallel. $\textbf{patch}_{\mathcal{L}_{ck}}$ (instantiated below) identifies the associated layer, ℓ_1, in the program via dictionary lookup and removes the stroke-width channel from S_1. Notice that the Choropleth layer remains untouched.

Patch/Remove-Channel

$$\frac{P(\mathsf{id}_1) = \langle \text{Point}, D_1, S_1\rangle}{\textbf{patch}(\text{removeChannel}(\mathsf{id}_1, \text{stroke-width}), P) = [\mathsf{id}_1 \mapsto \langle \text{Point}, D_1, S_1 \setminus \{\text{stroke-width}\}\rangle]P}$$

Meanwhile, $\textbf{recon}_{\mathcal{L}_{ck}}$ (instantiated below) identifies the current set of associated marks by id ($\{\textbf{mark}(\mathsf{id}_1, \text{Point}, \mathsf{d}, S_1) \mid \mathsf{d} \in D_1\}$) and modifies the marks to remove the stroke-width channel. Notice again that marks associated with the Choropleth layer (id_2) remain unchanged.

Recon/Remove-Channel

$$\frac{\begin{array}{c}M' = \{\textbf{mark}(\text{Point}, \mathsf{id}_1, \mathsf{d}, S_1 \setminus \{\text{stroke-width}\}) \mid \textbf{mark}(\text{Point}, \mathsf{id}_1, \mathsf{d}, S_1) \in M\} \\ M'' = \{\textbf{mark}(\text{Polygon}, \mathsf{id}_2, \mathsf{d}, S_2) \mid \textbf{mark}(\text{Polygon}, \mathsf{id}_2, \mathsf{d}, S_2) \in M, \mathsf{id}_1 \neq \mathsf{id}_2\}\end{array}}{\textbf{recon}(\text{removeChannel}(\mathsf{id}_1, \text{stroke-width}), M) = M' \cup M''}$$

The following theorem establishes that the new set of marks produced by $\textbf{recon}_{\mathcal{L}_{ck}}$—that is, the new map—is equivalent to the map that would be produced by **eval**'ing the **patch**'ed program (Definition 3.1). For brevity, we provide a proof of this theorem in Appendix A.

Theorem 4.1 (Soundness). $\textbf{recon}_{\mathcal{L}_{ck}}$ *is a reconciliation function.*

5 Implementation

We implement $\mathcal{L}_{ck}$, $\textbf{eval}_{\mathcal{L}_{ck}}$, $\textbf{patch}_{\mathcal{L}_{ck}}$, and $\textbf{recon}_{\mathcal{L}_{ck}}$ described in Section 4 in a direct manipulation programming environment for geospatial analysis and visualization, `cartokit`. `cartokit` is implemented in 9,526 lines of TypeScript and Svelte, and its source code is publicly available at https://github.com/parkerziegler/cartokit. A deployment of the `cartokit` programming environment is available at https://alpha.cartokit.dev.

Our implementations of **patch** and **recon** account for 1,713 lines (≈18%) of the codebase. Much of this code is devoted to specializing the same core logic for all visualization channels (currently 21) our system supports. In fact, **patch-recon** for a single **diff** tends to be quite compact in our setting. For example, the entirety of **patch-recon** for updating the point-size channel (i.e., $\delta = \text{setChannel}(\text{id}, \text{point-size}, n)$), is implemented in only 14 lines of TypeScript. A small minority are much larger. In particular, **patch-recon** for $\delta = \text{transformLayer}(\text{id}, \text{transform})$, which handles advanced cases like the example described in Section 2, constitutes 1,020 of the overall 1,713 lines (≈59.5%); most of this code implements specific geospatial data transformation algorithms.

[2] A choropleth layer associates a geographic region with a color based on the value of a particular data property for that region. See Figure 7 for an example.

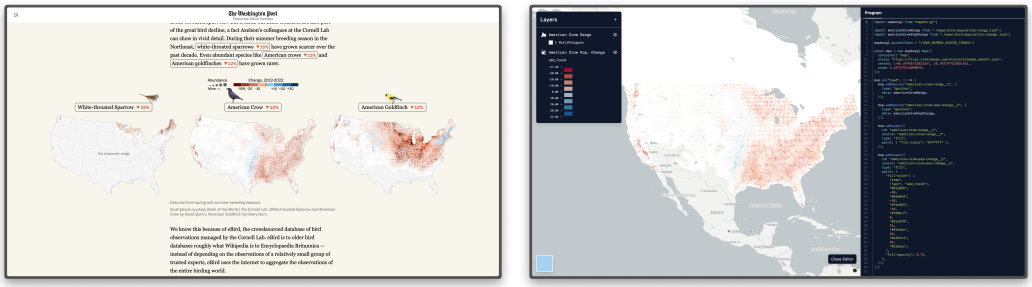


Fig. 7. **An example map from our benchmark suite (left) alongside the `cartokit` reproduction (right).** This example choropleth map, taken from "Bird populations are declining. Some are in your neighborhood" published in The Washington Post, shows the change in American Crow abundance across the United States from 2012-2022. The benchmark workflow associated with this map includes 17 **`recon`**-triggering actions.

6 Evaluation

To assess the performance impacts of reconciliation on direct manipulation programming, we designed our empirical evaluation around two core research questions:

RQ1. How does reconciliation affect performance relative to forward evaluation, if at all?
RQ1a. Does reconciliation result in greater speedups for longer-running computations?
RQ2. How fast is reconciliation when updating real-world outputs with real-world datasets?

We investigated these questions through two studies using our instantiation of the **`patch-recon`** approach in `cartokit`. In Study 1, we addressed **RQ1** and **RQ1a** by measuring and comparing reconciliation's performance against forward evaluation while reproducing six maps published by two national newsrooms: The Washington Post[3] and The New York Times[4]. In Study 2, we answered **RQ2** by instrumenting `cartokit`'s production deployment to capture reconciliation's run time performance in situ on organic, real-world use.

6.1 Study 1: Comparing Reconciliation vs. Forward Evaluation Performance on Six Real-World Benchmarks

6.1.1 Benchmark Suite. We selected six maps published by two national newsrooms as targets for replication using the following criteria:

(1) **Data availability**. GeoJSON[5] data for the map had to be provided, publicly available, or computable from the data sources listed with the map.
(2) **Recency.** The map had to have been published after January 1, 2023.

Table 1 includes details of each benchmark map. Figure 7 shows an example of one of our benchmark maps alongside its reproduction in `cartokit`.

6.1.2 Setup. We used v0.5.2 of `cartokit` to reproduce each map in our benchmark suite, generating one *workflow* per map. Each workflow is composed of a sequence of *actions*, which correspond to GUI interactions that trigger reconciliation (in the **`patch-recon`** condition) or forward evaluation (in the forward evaluation condition). Each action results in both an updated output and an updated $\mathcal{L}_{ck}$ program; `cartokit` additionally compiles the updated $\mathcal{L}_{ck}$ program to JavaScript for display

[3] https://www.washingtonpost.com
[4] https://www.nytimes.com/spotlight/graphics
[5] GeoJSON is a standard interchange format used to encode geospatial data for web applications. The GeoJSON specification is available at: https://datatracker.ietf.org/doc/html/rfc7946

Table 1. **Benchmark suite. LOC** reports the number of lines of code in the final JavaScript program generated by `cartokit` for the given benchmark. **Action #** reports the number of **`recon`**-triggering actions required to reach the target map. **Data (**MB**)** shows the size of the map's datasets, in megabytes.

ID	Article Title	Newsroom	LOC	Action #	Data (MB)
1	"Maps of the April 2024 Total Solar Eclipse"	The New York Times	54	13	8.2
2	"You're not crazy. Spring is getting earlier. Find out how it's changed in your town."	The Washington Post	42	13	127
3	"Winter is warming almost everywhere. See how it's changed in your town."	The Washington Post	40	11	249.4
4	"A boat went dark. Finding it could help save the world's fish."	The Washington Post	47	12	2.7
5	"Bird populations are declining. Some are in your neighborhood."	The Washington Post	60	17	7
6	"Will global warming make temperature less deadly?"	The Washington Post	47	14	15
		Median	**47**	**13**	**11.6**

to the user. When run in order, the sequence of actions in a workflow produces the final target map and program for the workflow. Across our six workflows, there was a total of 80 unique actions. We automated our workflows as Playwright [41] tests, which are publicly available at https://github.com/parkerziegler/cartokit/tree/v0.5.2/tests/workflows.

We executed both reconciliation and forward evaluation for each of the 80 actions in Google Chrome 130.0.6723.70 on a laptop running macOS 14.6.1 with a 2.3 GHz Quad-Core Intel Core i7 processor and 32GB RAM. To capture reconciliation execution times, we instrumented the source code of `cartokit`'s reconciliation algorithm with the browser's native Performance API [17]. To capture forward evaluation execution times, we instrumented `cartokit`-generated programs with identical calls to the browser Performance API. For each action, we measured 10 executions of the corresponding reconciliation event and 10 forward evaluations of the corresponding program.

Importantly, measuring *only* code execution time does not give the full picture of how long it takes for a new output to display. Given that we want to achieve interactive speeds to facilitate direct manipulation programming, we also measure when the user interface reaches a quiescent state after each reconciliation event (in the **`patch-recon`** condition) or each forward evaluation (in the forward evaluation condition). We call this metric *time-to-quiescent* (hereafter TTQ). We used the `idle` event of `cartokit`'s map rendering library, MapLibre GL JS [35], as the endpoint for TTQ measurement. The `idle` event is fired when all currently requested tiles (that is, data) have rendered on the map.

6.1.3 Results. **Reconciliation outperformed forward evaluation, yielding a median speedup of 2.92× for TTQ and 28.06× for code execution time.**

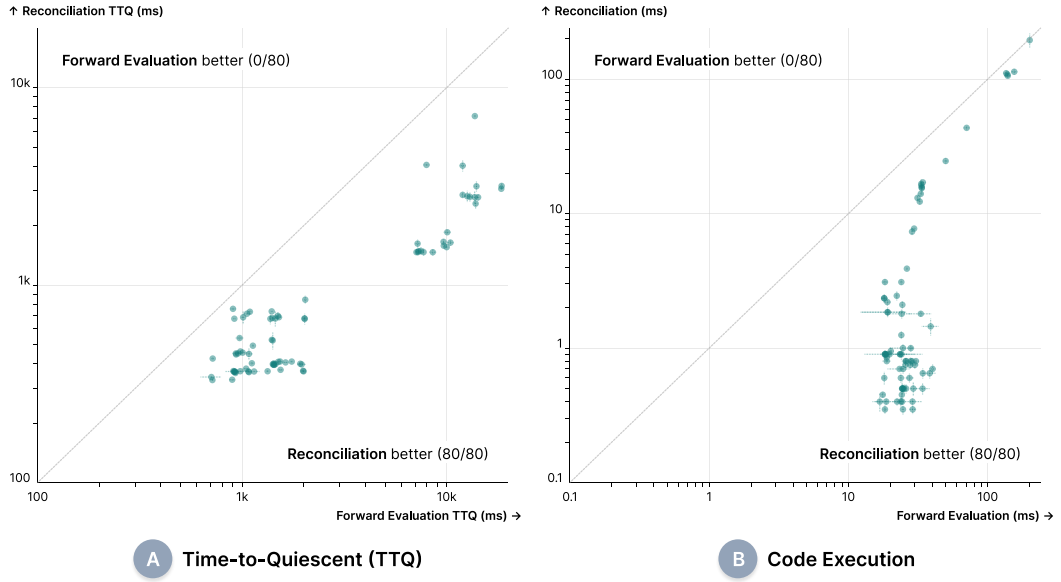


Fig. 8. **Comparing forward evaluation and reconciliation across all benchmarks.** (A) shows the time-to-quiescent (TTQ) run times of each approach while (B) shows the code execution run time of each approach. Each point reports the median run time across 10 executions. Error bars (dashed) show the standard error ($\frac{\sigma}{\sqrt{n}}$) along both axes. Points below the diagonal represent actions for which reconciliation outperformed forward evaluation. Points above the diagonal represent actions for which forward evaluation outperformed reconciliation.

TTQ. Across the 80 actions in our six benchmark workflows, reconciliation led to speedups on all of them. Of these speedups, 70 were by 2× or more and 38 by 3× or more. Recall that TTQ measures the time it takes for the output map to reach an idle state, signaling that the update has fully propagated to pixels rendered onscreen. These results suggest that reconciliation more than *halves* interface latency in comparison with forward evaluation. Figure 8A shows the median TTQ run times for forward evaluation and reconciliation; error bars (dashed) represent the standard error ($\frac{\sigma}{\sqrt{n}}$). The cumulative speedup on the benchmark suite as a whole was 3.87×.

Code Execution Times. Speedups were similarly consistent on code execution times as TTQ; across the 80 actions from our six workflows, we observed that reconciliation sped up all of them. Additionally, these speedups were often considerably more dramatic; of the 80 speedup instances, 56 (70%) were by an order of magnitude (10×) or more. Figure 8B shows the median code execution run times for forward evaluation and reconciliation; error bars (dashed) again represent the standard error. The cumulative speedup on the benchmark suite as a whole was 3.19×.

Speedups from reconciliation increased as forward evaluation TTQ increased; that is, longer-running programs tended to see greater latency reduction from reconciliation.

We observed a positive correlation between forward evaluation TTQ and speedups from reconciliation (Spearman's rank correlation coefficient of 0.732). This indicates that reconciliation is especially helpful (produces higher speedups) for tasks that have long forward evaluation times. Figure 9 shows forward evaluation TTQ plotted against speedup.

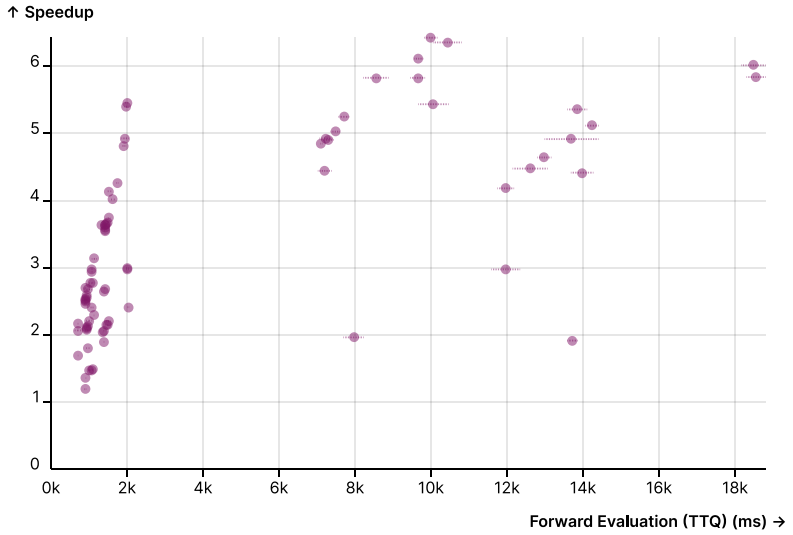


Fig. 9. **Comparing forward evaluation TTQ against speedup from reconciliation.** Each point reports the median run time of forward evaluation TTQ on the x-axis and the speedup attributed to reconciliation on the y-axis. Error bars (dashed) show the standard error for forward evaluation observations. The two measures are positively correlated (Spearman's rank correlation coefficient of 0.732).

6.2 Study 2: Measuring Reconciliation Performance In Situ

6.2.1 Setup. To extend our observation of reconciliation's performance beyond the benchmark suite, we instrumented the production deployment of `cartokit` to capture performance in situ on organic real-world use over a 30-day period. Importantly, our instrumentation captures no information about users or their data; we only record the system's **`recon`** run time, **`patch`** run time, and the size of generated programs (but not the contents of the programs themselves). Thus, we cannot report any information on the size of user datasets, the number of interactions in a user session, or the number of user sessions that occurred. Additionally, we do not have any information on browser usage, operating system usage, or RAM capacity on users' machines. In total, we collected 153 reconciliation traces.

The goal of this study was to identify whether real users' observed **`recon`** run times were in the same range as those observed in our benchmark study. Beyond addressing this question, we cannot learn much from this data. For example, we do not know whether collected traces came from a few long-running sessions or many shorter sessions, or whether users were working with large or small datasets. Lacking this information on the diversity and complexity of user workloads, we cannot make more definitive claims about reconciliation's performance in the general case. However, if production traces displayed similar performance characteristics to traces from our benchmark suite, it would provide some signal that our benchmarks realistically capture production use. In situ use may also test interaction sequences that our benchmarks did not exercise.

Capturing TTQ in a production context was unfortunately not possible. This is due to the fact that rendering updates enqueued by reconciliation may be batched by MapLibre GL JS, meaning that multiple reconciliation events may produce only a single `idle` event. (For our benchmark evaluation, we waited for the `idle` event to fire before triggering the next interaction, ensuring one `idle` event per output update.) Thus, we report here only the code execution time of the reconciliation algorithm.

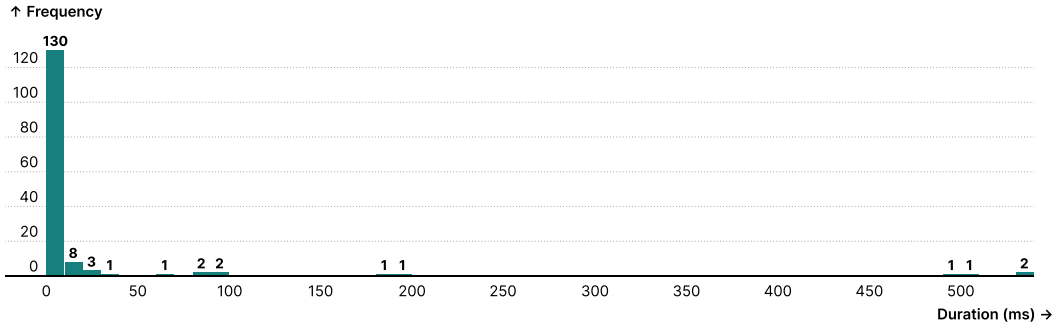


Fig. 10. **Distribution of reconciliation code run execution times from production traces.** Reconciliation times are aggregated into bins of 10ms. A large majority of reconciliation times (130/153) fall within the 0-10ms range.

6.2.2 Results. **In production, reconciliation achieved a median code execution time of 1.1ms, closely mirroring performance in our benchmark suite.** Median reconciliation code execution time across the benchmark suite was 0.9ms, suggesting the performance we observe in our benchmark suite was relatively representative of in situ use. Of the 153 traces, 130 of them (≈85%) took less than 10ms and 147 of them (≈96.1%) took less than 100ms. The remaining traces tended to involve output updates that are inherently expensive, such as transitioning to a Dot Density layer or computing new statistical breaks for a Choropleth layer. Such updates require full linear scans of the dataset. Figure 10 shows the distribution of reconciliation's code execution times in production.

6.3 Time-to-Quiescent vs. Code Execution Times

A key difference between our evaluation and evaluations in prior work on direct manipulation programming systems (e.g., [38, 64, 65]) is the choice to measure time-to-quiescent in addition to code execution times as a performance metric. Our goal with this decision was to assess reconciliation's impact on interface latency, which we believe is a stronger indicator of the interactivity of a direct manipulation programming system. Moreover, as our evaluation revealed, code execution times were not always accurate predictors of TTQ. For example, Workflow 3-A4 had the highest median reconciliation code execution time of any action (195.35ms), yielding only a 1.03× speedup over forward evaluation (200.55ms). However, the same action actually witnessed a 3.89× TTQ speedup from reconciliation (3177.10ms) compared to forward evaluation (12347.40ms). Figure 11 plots reconciliation's code execution time against its TTQ run time. The Spearman's rank correlation coefficient between the two measures is 0.070, indicating that the measures are not correlated. Results like this suggest that measuring code execution times alone may lead to (1) false conclusions about how fast a direct manipulation programming system is from a user's perspective and (2) erroneous claims about the classes of interactions that are relatively fast or slow.

One plausible explanation for the absence of TTQ measurement in prior work is that generated outputs are inexpensive to compute, and so there is little (if any) difference between code execution times and TTQ for these contexts. Indeed, modern web browsers are extremely efficient at rendering HTML and SVG. In contrast, geospatial visualization—with its tens of megabyte dataset sizes, asynchronous tile generation algorithms, and aggressive use of the GPU—is significantly more resource-intensive. Going forward, if the community attempts to extend direct manipulation to longer-running computations, it is possible that the distinction between quiescent times and execution times will become more important.

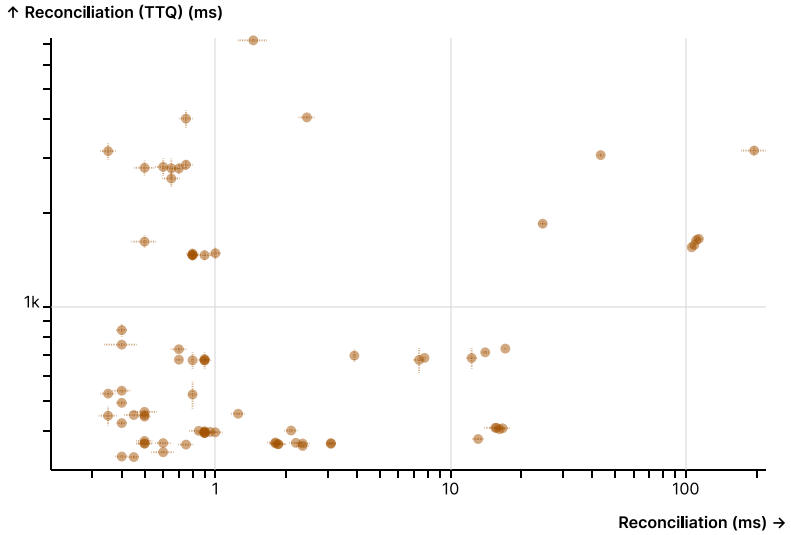


Fig. 11. **Comparing reconciliation code execution time to reconciliation TTQ run time.** Each point reports the median run time of each measure across 10 executions. Error bars (dashed) show the standard error ($\frac{\sigma}{\sqrt{n}}$) along both axes. The two measures have no discernible correlation (Spearman's rank correlation coefficient of 0.070).

7 Discussion

7.1 patch-recon Extends to General-Purpose Programming Languages

While Section 7.2 discusses how our choice of language in Section 4.1 is representative of existing languages for tasks amenable to direct manipulation, it is natural to wonder: What about a language more like the λ-calculus? In particular, can **patch-recon** work in languages with general recursion?

patch-recon can directly apply to general-purpose languages with recursion. The key insight is that the complexity of **patch-recon** is a function of the *user actions* that the system developer makes available, *not* the underlying language. As we will see in the following example, this is because the signature of **recon** in Definition 3.1 is **recon** : Diff × Val → Val and thus depends only on the definition of Diff and Val, not the underlying language Prog.

Example 7.1. Here we describe how to apply **patch-recon** to the direct manipulation programming system Sketch-n-Sketch [30], which uses an Elm-like functional programming language based on the λ-calculus with general recursion. Consider the program P below that constructs four differently-colored SVG `<circle>` elements with radius 5 at the coordinates $(0, 0)$, $(10, 10)$, $(20, 20)$, and $(30, 30)$:

```
List.indexedMap
  (\i c -> circle 5 (10 * i) (10 * i) c)
  ["turquoise", "violet", "steelblue", "indigo"]
```

(`List.indexedMap` is a function that is implemented using recursion.) Let's say a user changes the color of the second `<circle>` to `"red"` via the GUI. The novelty of prior work is defining a program transformation δ that, applied to this program, produces a new program P' evaluating

to the same set of <circle> elements, but with the color of the second <circle> set to "red." In our framing, this amounts to defining **patch**, which prior work has successfully tackled using ideas like bidirectional evaluation [38] and value provenance [30] (Section 8.1). We do not claim any new contributions in this space.

Rather, we require a new function **recon** that can immediately be applied to the current output to produce new output *without re-running the program.* This last stipulation—not re-running the program—differentiates the **patch-recon** technique from what is currently implemented in Sketch-n-Sketch. In Sketch-n-Sketch, changing the color of the <circle> will generate the program transformation δ above. This transformation creates a new program that is evaluated to get the new output. In other words, every single time a color is changed on a <circle>, Sketch-n-Sketch goes through the entire **patch-eval** cycle (the upper-right path in our commutative square in Definition 3.1). The same situation occurs every time an element is moved, duplicated, etc.

We can bypass this by defining a reconciliation function **recon** here. In this case, **recon** would simply set the fill attribute of the selected SVG circle element to "red" directly. For moving, **recon** would directly update the cx and cy attributes of the <circle> element; for duplication, **recon** would simply clone the <circle> element and place it within the same subtree in the DOM.

Example 7.2. Let us now complicate the example above. Say we have the following program P'' in Sketch-n-Sketch, which introduces a variable, color, that we reference in our iterated list.

```
color = "violet"
List.indexedMap
  (\i c -> circle 5 (10 * i) (10 * i) c)
  ["turquoise", color, color, color]
```

In this case, if we change the color of the second <circle> to "red" in the GUI, we likely expect the system to update the string bound to color from "violet" to "red" (as opposed to updating just the second element in the list). Zhang et al. [66] demonstrated how to implement **patch** to capture these semantics, but how would we implement **recon** for this case? In this formulation, we are obligated to update <circle> elements beyond those the user has actively selected.

One strategy to address this problem is to tag output values with provenance information. For example, in cartokit, marks carry both the id of the layer they belong to as well as functions for re-computing channel values based on their associated data. Thus, when a single channel on a single mark is updated, **recon** can easily identify the full set of affected marks and, for each mark, re-execute the function of the modified channel. In a setting like our example, we can imagine using expression provenance [2] to derive similar information, such that each <circle> "knows" that its fill value is derived from the color variable (e.g., by tagging it with a shared identifier). **recon** can then use this information to re-evaluate the fill of affected <circle> elements whenever the color variable changes.

Ultimately, the key insight to take away from this example is that **recon** is enabled by carrying some computation (e.g., cartokit's channel functions) to output values for evaluation as needed. This general approach may be useful for implementing **recon** when **patch** is more sophisticated.

Falling Back on Forward Evaluation. Of course, **diff**s can be even more complex than the two examples above. Sketch-n-Sketch can famously create a Koch snowflake fractal parameterized by recursive depth using only direct manipulation on the output [30]. In this extreme situation, a corresponding **diff** would introduce recursion into a defined function, which can essentially be considered a whole program rewrite. We view it as improbable that reconciliation will work

for such transformations; the program simply does need to be re-run. However, in such cases, **patch-recon** degrades gracefully to prior state of the art; if a **recon** function cannot be defined for a certain **diff**, the system can fall back on simply running forward evaluation. In other words, if the lower-left path of the commutative square in Definition 3.1 is not available, we can always fall back to the upper-right path.

7.2 Representativeness of the cartokit Language

By design, the language $\mathcal{L}_{ck}$ we introduce in Section 4.1 is quite similar to a large number of existing languages for tasks amenable to direct manipulation. These languages include, for example, Vega [52, 59] and Tableau's VizQL [27]. Overall, this kind of visualization language has had broad adoption, and there are a tremendous number of them—[39] recently identified 57, many with substantial real-world use. We provide concrete examples of how **patch-recon** applies to two of these languages; the approach is essentially the same as that for our language $\mathcal{L}_{ck}$.

Example 7.3. Consider Vega-Lite [51, 60], a language for interactive charts. Here is a simple example of a (partial) Vega-Lite specification for a stacked area chart:

```
{
  "encoding": {
    "x": {
      "timeUnit": "yearmonth", "field": "date",
      "axis": {"format": "%Y"}
    },
    "y": {
      "aggregate": "sum", "field": "count"
    },
    "color": {
      "field": "series",
      "scale": {"scheme": "category20b"}
    }
  }
}
```

Interestingly, this language is quite similar to $\mathcal{L}_{ck}$ in structure and abstraction level, hinting at how we can implement **patch-recon**. Let us imagine the user changes the color scheme of this visualization to a different categorical color scheme (e.g., `"observable10"`) via a GUI interaction. In this case, **patch** would simply update the `color.scale.scheme` value in the program to `"observable10."` Now, to define **recon**, we take a similar approach to what we do in `cartokit`:

(1) Create a scale function mapping the domain of the variable visualized in the `color` channel (`"series"`) to the range of discrete colors in the selected scheme.
(2) Obtain references to all rendered marks. For Vega-Lite, these are SVG elements.
(3) For each rendered mark:
   (a) Call the scale function, passing the mark's corresponding data value as the argument to obtain the mark's new color.
   (b) Update the mark's `fill` and `stroke` attributes.

Crucially, using **patch-recon** would not entail re-evaluating the entire Vega-Lite spec using its interpreter; rather, we would precisely update only the relevant portion of the program and the

relevant attributes of rendered SVG elements in the DOM. Proving soundness of **patch-recon** in this domain would take a proof very similar to `cartokit`'s proof (see Appendix A).

Example 7.4. Beyond data visualization, other domains amenable to direct manipulation (such as visual graphics creation and image processing) use similar languages. For our next concrete example, we will describe **patch-recon** for a direct manipulation programming system targeting Mermaid [40], a popular diagramming language. Imagine we had a simple flowchart diagram in such a system as follows:

```
flowchart LR
  id1{{First node}}
  id2{{Second node}}
  id1 -> id2
```

If we wanted to update the shape of the second node in the diagram from a hexagon (indicated by the curly braces on the third line) to a parallelogram, **patch** could change the third line of the above program to `id2[/Second node/]` (brackets and forward slashes indicate parallelogram). A **recon** function would update the `points` and `transform` attributes of the rendered SVG polygon, similar to the Sketch-n-Sketch example from Section 7.1. In this case, because the **diff** is simple, **recon** is also simple. **patch-recon** avoids the need for full program re-evaluation, which could be particularly important for a diagram with hundreds or thousands of nodes.

7.3 When should we use patch-recon vs. forward evaluation?

A **patch-recon** approach is (probably) harder to implement than an approach that centers on running forward evaluation to keep program and output in agreement. For situations where forward evaluation is sufficiently fast, the corresponding **patch-recon** implementation may be more complex for a system developer to reason about and more heavyweight than its advantages justify. We suggest using **patch-recon** in cases where programs are long-running; in cases where programs are short-running, we recommend using the standard approach of **patch** followed by forward evaluation. In our view, **patch-recon** does not replace forward evaluation. Rather, it offers an alternative for settings where forward evaluation would make a direct manipulation programming environment infeasible.

7.4 When should we use patch-recon vs. caching?

In Section 2, we argued that implementing an ad hoc caching scheme effectively approaches **patch-recon** in the limit, albeit without the soundness guarantee. Going beyond correctness, we believe there may be additional performance and maintainability benefits associated with **patch-recon**.

To build this intuition, let us revisit the example from Section 4, where a GUI interaction triggers removal of the stroke-width channel from a single layer on the map. Notice here that **patch-recon**'s update strategy is both extremely fine-grained and operates in place; it *only* modifies the stroke-width channel of marks in the target layer while leaving other channels and layers untouched. Moreover, as previously discussed, **patch** and **recon** run in parallel in this context—there is no ordering constraint on their execution. Now, imagine that we replaced **patch-recon** with a layer-level caching scheme that only re-evaluated modified layers while leaving the remainder intact. On the surface, this may sound like an equivalent approach! In practice, however, this would be a variation on the standard technique coupling a **patch** procedure with forward evaluation, with the refinement that forward evaluation would reuse cached marks from unmodified layers. Notice that this strategy still encounters two familiar issues: sequential execution and redundant computation.

Concretely, an implementation would need to (1) throw away all rendered marks associated with the modified layer, (2) update the layer in the program via **patch**, and finally (3) re-evaluate the layer to a *new set* of rendered marks, despite the fact that many channels on the marks (e.g., fill-color, fill-opacity) did not change. For large data (e.g., our implementation handles up to 1 million records), the difference between in-place updates and partial re-evaluation produces a significant increase in interface latency; we expect other direct manipulation programming systems targeting large data or long-running programs will face the same issue. In short, an ad hoc caching scheme could get us *some* performance improvement over forward evaluation, but it would certainly leave some performance wins on the table.

In addition, it is unclear that such a scheme would be easier to implement or evolve than **patch-recon**. Concretely, realizing the layer-level caching approach would require (1) building, maintaining, and invalidating a cache of rendered marks, (2) maintaining a mapping of program segments to cache entries, and (3) implementing a cache-aware partial evaluator. Moreover, it may be difficult to adapt this approach over time if we want to change cache granularity; for example, lowering our caching logic to the mark level would involve a full re-implementation. **patch-recon** encourages a different view from the start: for a given GUI interaction, one wants to make the smallest possible change to bring the program and output into agreement. In our experience developing `cartokit`, this made the engineering goal clear—it shifted us from thinking about what *may* be reused (caching) to determining what *must* be updated (**patch-recon**).

Ultimately, ad hoc caching may be a good choice when a system developer needs modest performance improvements over forward evaluation to reach their interface latency goals. However, we believe that a **patch-recon** structure is more likely to guide system developers to a performant and maintainable implementation from the outset.

8 Related Work

8.1 Improving patch

Existing research on direct manipulation programming systems has focused primarily on the problem of propagating changes in the program output to changes in the program itself; that is, improving the **patch** function [8, 14, 23, 29, 31, 43, 54]. Some techniques heavily constrain the locations or types of **patch** updates, including livelits [43] (which constrains updates to holes) and early versions of Sketch-n-Sketch [14] (which constrain updates to numeric constants). Other techniques like bidirectional evaluation [38], delta fusion [66], value provenance tracing [30], and others [64, 65] have enabled more expressive program transformations, such as function creation, recursive calls, and expression hoisting. Collectively, these techniques are attempts at tackling the *view-update problem* [10] for GUIs as part of the broader research on bidirectional programming languages [12, 22, 42].

In contrast to this prior work, we do not introduce a new technique for propagating output changes to source programs. Rather, we see our **patch-recon** approach as complementary; we can combine any of the techniques described above with reconciliation as long as we continue to prove **patch-recon** correspondence.

8.2 Improving recon

Reconciliation is a form of incremental program evaluation; given a small program transformation in the form of a **diff**, reconciliation determines how to execute *just* the **diff** to update the current program output. In this section, we survey the prior work on incremental evaluation more generally. While a handful of related techniques focus on incrementalizing evaluation with respect to *program*

updates (akin to reconciliation), others focus on responding to updates in the *data* supplied as program input. Thus, we organize our discussion along this axis.

8.2.1 Incremental Evaluation with Program Updates. Reconciliation shares at least one core aim with prior work on incremental evaluation of evolving programs—reducing redundant recomputation. Given some change to a program, incremental evaluation techniques attempt to identify and re-evaluate *only* the portion of the program affected by the change. For example, incremental compilers [33, 45–47, 55] operate by recompiling only those statements that are either (1) modified by the programmer or (2) dependent on modified statements. Similarly, incremental program analysis techniques (e.g., for computing dominator trees [58] or points-to analysis [48]) compute updates to the analysis information based solely on the program change rather than rerunning the analysis from scratch [16, 63]. Incremental evaluators for logic programs [49, 50] construct memo tables mapping "calls" (subgoals) to their "answers" (provable instances); when the facts of a program change, only the affected calls must be recomputed. Interestingly, much of this literature cites improving the interactivity of programming systems as a core motivation—precisely the problem reconciliation aims to tackle in the direct manipulation context.

8.2.2 Incremental Evaluation with Program Input Updates. Another branch of work in incremental evaluation and self-adjusting computation focuses on rerunning only the necessary parts of a program when the program *input* changes [3, 4, 7, 13, 15]. Broadly, these techniques work by constructing (1) a dynamic dependency graph capturing dependencies between parts of the computation and (2) a change propagation algorithm that identifies and re-evaluates dependencies in response to input changes. When the result of re-evaluation does not yield a new output, related dependencies need not be re-evaluated; their prior outputs can simply be reused. This general strategy plays a critical role in modern web frameworks for constructing user interfaces (e.g., React [19], Svelte [28]), which include some strategy for incremental update of the DOM in response to changes in application state (e.g., virtual DOM [20], signals [15, 18]). This work also shares a close connection to the rich literature on incremental view maintenance in databases [1, 9, 24–26].

8.2.3 **`recon`***-Style Updates Outside of Programming Contexts.* Many GUIs use direct manipulation as their core interaction model but do not author programs (e.g., Photoshop [6], Illustrator [5], and others [21, 32]). Users leverage these interfaces to produce a particular output, but not a program that they can apply to other inputs. Most tools in this category respond to GUI actions by calculating a next output based on the current output. For example, in Photoshop, the system produces the next image by modifying the current image, not by starting from the original input image and rerunning the entire sequence of user-triggered modifications. In that this process presents a new value to the user by operating on an existing value, we can think of it as being a **`recon`**-style operation. Despite this surface resemblance, these tools are under no obligation to produce a program. This means they sidestep the central, animating obligation of our approach: to update a program and its output value in parallel, and to keep them in agreement without running the program in its entirety.

8.2.4 Reconciliation for Direct Manipulation Programming. We are not aware of any works that use **`recon`**-style approaches in the context of direct manipulation programming.

8.2.5 Summary. In contrast to the prior work on incremental evaluation, we do not introduce a new technique for implementing **`recon`**. Rather, we see our **`patch-recon`** approach as complementary; we can integrate some of the techniques described above into reconciliation as long as we continue to prove **`patch-recon`** correspondence.

8.3 patch-recon Correspondence

Our central contribution is to exploit **patch-recon** correspondence by using reconciliation as a fast means of achieving synchronization between programs and values. Our **patch-recon** correspondence is analogous to a classic class of theorems from the literature on bidirectional programming systems called PutGet [11, 22, 44, 61, 66], also known as Consistency [36, 37, 62] or UpdateEval [38]. At a high level, these theorems state that if a value v' is backward-evaluated onto a program e that evaluates to v, then the resulting program e' will evaluate to v'. Sometimes, this value is described as the result of a direct manipulation, either directly [38] or in terms of a syntactic patch obligation [66]. In contrast to our work, the mapping between v and v'—that is, reconciliation—is only used to reason about the correctness of these systems, as prior work has always assumed that simply evaluating e' to obtain v' is fast enough.

8.4 Improving Performance for Direct Manipulation Programming Systems

Although our chosen technique for extending direct manipulation programming to long-running programs is to eliminate forward evaluation via **patch-recon** correspondence, our higher-level goal is to make direct manipulation programming systems efficient for interactive, large-scale computation. We therefore share a goal with research that aims to speed up direct manipulation programming, even if it does not eliminate forward evaluation.

Most evaluations of direct manipulation programming systems in the literature focus on assessing expressiveness (that is, the ability of the system to produce target programs and outputs primarily or solely through direct manipulation [14, 30, 66]) rather than performance. Some papers include data on forward evaluation times [38, 64, 65], but generally emphasize backward evaluation of output updates to program updates. We are aware of only two works identifying repeated forward evaluation of modified programs as a barrier to achieving interactive speeds. Transmorphic [53], a direct manipulation programming system for GUI development, reduces the total time spent on forward evaluation by deferring full forward evaluation in cases where updates can be effectively emulated on the output. Similarly, [34] offers additional "speculative" visual feedback on output manipulations in order to defer running forward evaluation.

9 Conclusion

Direct manipulation programming systems have traditionally relied on forward evaluation to guarantee program-output agreement by construction. While this strategy has succeeded for short-running programs in domains like vector graphics and HTML documents, it has not allowed us to scale direct manipulation programming to authoring long-running programs. In these contexts, re-executing a complete program in response to every user interaction quickly crosses outside of interactive speeds. This work presents *patch-reconciliation correspondence*, a novel strategy for enforcing program-output agreement in direct manipulation programming systems that eliminates the need for forward evaluation altogether. We prove our **patch-recon** approach sound for a language and a set of direct manipulations interactions for the domain of geospatial data visualization, and implement this instantiation in a new direct manipulation programming system, `cartokit`. Our approach reduces interface latency on real-world program-authoring tasks, yielding larger speedups for longer-running programs. We feel **patch-recon** is an important first step in making direct manipulation programming feasible for interactively authoring long-running programs. We also hope this work unlocks new research opportunities in direct manipulation programming by expanding the paradigm's applicability to tasks previously considered beyond its reach.

Data Availability Statement

`cartokit` is freely available at https://alpha.cartokit.dev, and additional documentation for users is available at https://docs.cartokit.dev. The `cartokit` source code is also publicly available at https://github.com/parkerziegler/cartokit. We provide an archived snapshot of `cartokit` v0.5.2 and our full evaluation harness in amd64-compatible and arm64v8-compatible Docker images on Zenodo [67, 68].

Acknowledgments

We are extremely grateful to our anonymous OOPLSA and PLDI reviewers for their thoughtful and actionable feedback, and to our outstanding PLDI shepherd for their guidance and deep engagement with our work. We would also like to thank the members of PLAIT Lab and EPIC Data Lab at the University of California, Berkeley for their evergreen support and encouragement. Thanks are due, as well, to `cartokit`'s early adopters in the data journalism community. This work is supported in part by NSF grants FW-HTF 2129008 and CA-HDR 2033558, as well as by gifts from Google, G-Research, Adobe, and Microsoft. Chasins is a Chan Zuckerberg Biohub Investigator.

References

[1] Supun Abeysinghe, Qiyang He, and Tiark Rompf. 2022. Efficient Incrementialization of Correlated Nested Aggregate Queries using Relative Partial Aggregate Indexes (RPAI). In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 136–149. doi:10.1145/3514221.3517889

[2] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. 2012. A Core Calculus for Provenance. In *Principles of Security and Trust*, Pierpaolo Degano and Joshua D. Guttman (Eds.). Springer, Berlin, Heidelberg, 410–429. doi:10.1007/978-3-642-28641-4_22

[3] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, and Sam Westrick. 2020. Parallel Batch-Dynamic Trees via Change Propagation. In *28th Annual European Symposium on Algorithms (ESA 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 173)*, Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:23. doi:10.4230/LIPIcs.ESA.2020.2

[4] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive Functional Programming. *ACM Transactions on Programming Languages and Systems* 28, 6 (Nov. 2006), 990–1034. doi:10.1145/1186632.1186634

[5] Adobe. 2024. Adobe Illustrator. https://www.adobe.com/products/illustrator.html. Accessed: 2024-11-14.

[6] Adobe. 2024. Adobe Photoshop. https://www.adobe.com/products/photoshop.html. Accessed: 2024-11-12.

[7] Daniel Anderson, Guy E. Blelloch, Anubhav Baweja, and Umut A. Acar. 2021. Efficient Parallel Self-Adjusting Computation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 59–70. doi:10.1145/3409964.3461799

[8] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. 1989. A Two-View Approach to Constructing User Interfaces. *SIGGRAPH Comput. Graph.* 23, 3 (July 1989), 137–146. doi:10.1145/74334.74347

[9] Shivnath Babu and Jennifer Widom. 2001. Continuous Queries Over Data Streams. *SIGMOD Rec.* 30, 3 (Sept. 2001), 109–120. doi:10.1145/603867.603884

[10] F. Bancilhon and N. Spyratos. 1981. Update Semantics of Relational Views. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 557–575. doi:10.1145/319628.319634

[11] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: Resourceful Lenses for String Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 407–419. doi:10.1145/1328438.1328487

[12] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. 2006. Relational Lenses: A Language for Updatable Views. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '06)*. Association for Computing Machinery, New York, NY, USA, 338–347. doi:10.1145/1142351.1142399

[13] Yan Chen, Umut A. Acar, and Kanat Tangwongsan. 2014. Functional Programming for Dynamic and Large Data with Self-Adjusting Computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 227–240. doi:10.1145/2628136.2628150

[14] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*

(PLDI '16). Association for Computing Machinery, New York, NY, USA, 341–354. doi:10.1145/2908080.2908103

[15] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 411–422. doi:10.1145/2491956.2462161

[16] Alan Demers, Thomas Reps, and Tim Teitelbaum. 1981. Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. Association for Computing Machinery, New York, NY, USA, 105–116. doi:10.1145/567532.567544

[17] MDN Web Docs. 2024. Performance – Web APIs. https://developer.mozilla.org/en-US/docs/Web/API/Performance. Accessed: 2024-10-09.

[18] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. Association for Computing Machinery, New York, NY, USA, 263–273. doi:10.1145/258948.258973

[19] Facebook. 2024. React. https://react.dev/. Accessed: 2024-09-24.

[20] Facebook. 2024. Virtual DOM and Internals. https://legacy.reactjs.org/docs/faq-internals.html. Accessed: 2024-11-13.

[21] Figma. 2024. Figma. https://figma.com/. Accessed: 2024-11-12.

[22] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (May 2007), 17–es. doi:10.1145/1232420.1232424

[23] Koumei Fukahori, Daisuke Sakamoto, Jun Kato, and Takeo Igarashi. 2014. CapStudio: An Interactive Screencast for Visual Application Development. In *CHI '14 Extended Abstracts on Human Factors in Computing Systems (CHI EA '14)*. Association for Computing Machinery, New York, NY, USA, 1453–1458. doi:10.1145/2559206.2581138

[24] Timothy Griffin and Leonid Libkin. 1995. Incremental Maintenance of Views with Duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*. Association for Computing Machinery, New York, NY, USA, 328–339. doi:10.1145/223784.223849

[25] Ashish Gupta and Inderpal Singh Mumick. 1999. Materialized Views: Techniques, Implementations, and Applications. In *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, Cambridge, MA, USA, 145–157. doi:10.7551/mitpress/4472.001.0001

[26] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*. Association for Computing Machinery, New York, NY, USA, 157–166. doi:10.1145/170035.170066

[27] Pat Hanrahan. 2006. VizQL: A Language for Query, Analysis and Visualization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 721. doi:10.1145/1142473.1142560

[28] Rich Harris and Svelte Contributors. 2024. Svelte. https://svelte.dev/. Accessed: 2024-09-24.

[29] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. Association for Computing Machinery, New York, NY, USA, 379–390. doi:10.1145/2984511.2984575

[30] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 281–292. doi:10.1145/3332165.3347925

[31] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 654–664. doi:10.1145/3180155.3180165

[32] Jennifer Jacobs, Sumit Gogia, Radomír Měch, and Joel R. Brandt. 2017. Supporting Expressive Procedural Art Creation through Direct Manipulation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. Association for Computing Machinery, New York, NY, USA, 6330–6341. doi:10.1145/3025453.3025927

[33] Mark Kahrs. 1979. Implementation of an Interactive Programming System. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction (SIGPLAN '79)*. Association for Computing Machinery, New York, NY, USA, 76–82. doi:10.1145/800229.806956

[34] Richard Lin, Rohit Ramesh, Nikhil Jain, Josephine Koe, Ryan Nuqui, Prabal Dutta, and Bjoern Hartmann. 2021. Weaving Schematics and Code: Interactive Visual Editing for Hardware Description Languages. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. Association for Computing Machinery, New York, NY, USA, 1039–1049. doi:10.1145/3472749.3474804

[35] MapLibre. 2024. MapLibre GL JS. https://maplibre.org/maplibre-gl-js/docs/. Accessed: 2023-04-05.

[36] Kazutaka Matsuda and Meng Wang. 2015. Applicative Bidirectional Programming with Lenses. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. Association for Computing

Machinery, New York, NY, USA, 62–74. doi:10.1145/2784731.2784750

[37] Kazutaka Matsuda and Meng Wang. 2018. HOBiT: Programming Lenses Without Using Lens Combinators. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 31–59. doi:10.1007/978-3-319-89884-1_2

[38] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 127:1–127:28. doi:10.1145/3276497

[39] Andrew M. McNutt. 2023. No Grammar to Rule Them All: A Survey of JSON-style DSLs for Visualization. *IEEE Transactions on Visualization and Computer Graphics* 29, 1 (Jan. 2023), 160–170. doi:10.1109/TVCG.2022.3209460

[40] Mermaid Contributors. 2025. *Mermaid — Diagramming and Charting Tool*. https://mermaid.js.org/. Accessed: 2025-03-23.

[41] Microsoft. 2024. Playwright. https://playwright.dev/. Accessed: 2024-10-09.

[42] Keisuke Nakano, Zhenjiang Hu, and Masato Takeichi. 2009. Consistent Web Site Updating Based on Bidirectional Transformation. *International Journal on Software Tools for Technology Transfer* 11, 6 (Dec. 2009), 453–468. doi:10.1007/s10009-009-0124-3

[43] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. Association for Computing Machinery, New York, NY, USA, 511–525. doi:10.1145/3453483.3454059

[44] Hugo Pacheco, Zhenjiang Hu, and Sebastian Fischer. 2014. Monadic Combinators for "Putback" Style Bidirectional Programming. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM '14)*. Association for Computing Machinery, New York, NY, USA, 39–50. doi:10.1145/2543728.2543737

[45] Lori L. Pollock and Mary Lou Soffa. 1985. Incremental Compilation of Optimized Code. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '85)*. Association for Computing Machinery, New York, NY, USA, 152–164. doi:10.1145/318593.318629

[46] Patrick Rein, Robert Hirschfeld, and Marcel Taeumel. 2016. Gramada: Immediacy in Programming Language Development. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)*. Association for Computing Machinery, New York, NY, USA, 165–179. doi:10.1145/2986012.2986022

[47] Steven P. Reiss. 1984. An Approach to Incremental Compilation. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (SIGPLAN '84)*. Association for Computing Machinery, New York, NY, USA, 144–156. doi:10.1145/502874.502889

[48] Diptikalyan Saha and C. R. Ramakrishnan. 2005. Incremental and Demand-Driven Points-To Analysis using Logic Programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 117–128. doi:10.1145/1069774.1069785

[49] Diptikalyan Saha and C. R. Ramakrishnan. 2006. Incremental Evaluation of Tabled Prolog: Beyond Pure Logic Programs. In *Practical Aspects of Declarative Languages*, Pascal Van Hentenryck (Ed.). Springer, Berlin, Heidelberg, 215–229. doi:10.1007/11603023_15

[50] Diptikalyan Saha and C. R. Ramakrishnan. 2006. A Local Algorithm for Incremental Evaluation of Tabled Logic Programs. In *Logic Programming*, Sandro Etalle and Mirosław Truszczyński (Eds.). Springer, Berlin, Heidelberg, 56–71. doi:10.1007/11799573_7

[51] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 341–350. doi:10.1109/TVCG.2016.2599030

[52] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative Interaction Design for Data Visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. Association for Computing Machinery, New York, NY, USA, 669–678. doi:10.1145/2642918.2647360

[53] R. Schreiber, R. Krahn, D.H.H. Ingalls, and R. Hirschfeld. 2017. *Transmorphic: Mapping Direct Manipulation to Source Code Transformations*. Universitätsverlag Potsdam, Potsdam, Germany. https://books.google.com/books?id=88RADgAAQBAJ

[54] Christopher Schuster and Cormac Flanagan. 2016. Live Programming by Example: Using Direct Manipulation for Live Program Synthesis. In *LIVE Workshop*. Rome, Italy.

[55] Mayer D. Schwartz, Norman M. Delisle, and Vimal S. Begwani. 1984. Incremental Compilation in Magpie. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (SIGPLAN '84)*. Association for Computing Machinery, New York, NY, USA, 122–131. doi:10.1145/502874.502887

[56] Samyukta Sherugar and Raluca Budiu. 2016. Direct Manipulation: Definition. https://www.nngroup.com/articles/direct-manipulation. Accessed: 2024-11-14.

[57] Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (Aug. 1983), 57–69. doi:10.1109/MC.1983.1654471
[58] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. 1997. Incremental Computation of Dominator Trees. *ACM Trans. Program. Lang. Syst.* 19, 2 (March 1997), 239–252. doi:10.1145/244795.244799
[59] Vega Contributors. 2025. *Vega — A Visualization Grammar.* https://vega.github.io/vega/about/. Accessed: 2025-03-23.
[60] Vega-Lite Contributors. 2025. *Vega-Lite — A High-Level Grammar of Interactive Graphics.* https://vega.github.io/vega-lite/. Accessed: 2025-03-23.
[61] Janis Voigtländer. 2009. Bidirectionalization for Free! (Pearl). In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. Association for Computing Machinery, New York, NY, USA, 165–176. doi:10.1145/1480881.1480904
[62] Masaomi Yamaguchi, Kazutaka Matsuda, Cristina David, and Meng Wang. 2022. Synbit: Synthesizing Bidirectional Programs Using Unidirectional Sketches. *Formal Methods in System Design* 61, 2 (Dec. 2022), 198–247. doi:10.1007/s10703-023-00436-9
[63] Frank Kenneth Zadeck. 1984. Incremental Data Flow Analysis in a Structured Program Editor. *SIGPLAN Not.* 19, 6 (June 1984), 132–143. doi:10.1145/502949.502888
[64] Xing Zhang, Guanchen Guo, Xiao He, and Zhenjiang Hu. 2023. Bidirectional Object-Oriented Programming: Towards Programmatic and Direct Manipulation of Objects. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 83:230–83:255. doi:10.1145/3586035
[65] Xing Zhang and Zhenjiang Hu. 2022. Towards Bidirectional Live Programming for Incomplete Programs. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2154–2164. doi:10.1145/3510003.3510195
[66] Xing Zhang, Ruifeng Xie, Guanchen Guo, Xiao He, Tao Zan, and Zhenjiang Hu. 2024. Fusing Direct Manipulations into Functional Programs. *Proceedings of the ACM on Principles of Programming Languages* 8, POPL (Jan. 2024), 41:1211–41:1238. doi:10.1145/3632883
[67] Parker Ziegler, Justin Lubin, and Sarah E. Chasins. 2025. cartokit Docker Image. Zenodo. doi:10.5281/zenodo.15047320
[68] Parker Ziegler, Justin Lubin, and Sarah E. Chasins. 2025. cartokit Docker Image (Exact Version for Artifact Evaluation). Zenodo. doi:10.5281/zenodo.15079881

Received 2024-11-15; accepted 2025-03-06