Lenses for Partially-Specified States

Kazutaka Matsuda[1][0000−0002−9747−4899] (✉), Minh Nguyen[2][0000−0003−3845−9928], and Meng Wang[2][0000−0001−7780−630X]

[1] Tohoku University, Sendai 980-8579, Japan.
kztk@tohoku.ac.jp
[2] University of Bristol, Bristol BS8 1TH, United Kingdom.

Abstract. A bidirectional transformation is a pair of transformations satisfying certain well-behavedness properties: one maps source data into view data, and the other translates changes on the view back to the source. However, when multiple views share a source, an update on one view may affect the others, making it hard to maintain correspondence while preserving the user's update, especially when multiple views are changed at once. Ensuring these properties within a compositional framework is even more challenging. In this paper, we propose *partial-state lenses*, which allow source and view states to be partially specified to precisely represent the user's update intentions. These intentions are partially ordered, providing clear semantics for merging intentions of updates coming from multiple views and a refined notion of update preservation compatible with this merging. We formalize partial-state lenses, together with partial-specifiedness-aware well-behavedness that supports compositional reasoning and ensures update preservation. In addition, we demonstrate the utility of the proposed system through examples.

1 Introduction

Bidirectional transformations provide mechanisms to propagate updates between multiple data representations by translating changes in the transformed result back to the original form. This allows for synchronization across various data structures and has numerous applications, including classical view updating [4,7,16,21], co-development of printers and parsers [32,36,52,57,58], spreadsheets that support edits to the computed formula results [10,28,51], reflexive test case generators that can recover choices from a generated value [15], live programming [38,55,56], and model synchronization in model-driven engineering [46,53,54].

Despite their usefulness, developing bidirectional transformations requires effort because programmers must be aware of behavior in both directions. Moreover, such transformations are often expected to satisfy certain properties to be considered "bidirectional". Accordingly, many programming languages and frameworks have been proposed to facilitate them [5,6,12,13,14,18,19,23,24,26,30,33,34,35,37,40,48,49,50]. Among these systems, the most prominent is the lens framework [12,13].

In the lens framework, a bidirectional transformation—or lens—consists of two functions: $get \in S \to V$ maps a *source* state $s \in S$ to its *view* state $get\ s \in V$,

and $put \in S \times V \to S$ combines s and an updated view v' to return an updated source, $put\ (s, v')$. Since get is not always injective, put requires the original source s to reconstruct the information lost during the get transformation. A lens, a get/put pair, is called well-behaved ("bidirectional") if it satisfies the following two properties.[3]

$$put\ (s, v') = s' \implies get\ s' = v' \qquad \text{(consistency)}$$
$$get\ s = v \implies put\ (s, v) = s \qquad \text{(acceptability)}$$

Intuitively, consistency requires that any updates to the view are preserved by put-then-get. On the other hand, acceptability ensures that if there is no update to the view, then there is no update to the source. The lens framework is outstanding for its *compositional*, *correct-by-construction* approach, allowing programmers to build complex, well-behaved lenses by composing simpler ones and language designers to extend the system without breaking the guarantee.

Since their first appearance, lenses have been extended in various ways, but several research challenges remain. One key problem is the handling of multiple views. When multiple views share the same source, and a user updates one of them, we expect the other views to be adjusted accordingly while preserving the original change in the view that was directly modified. However, allowing this behavior breaks traditional well-behavedness properties, particularly *consistency*. Specifically, the view state v' that a user provides may differ from the final state $get\ (put\ (s, v'))$ after synchronization, because an update (involved in v') to one view will be propagated to the other view via the shared source. More generally, when multiple views are updated simultaneously, these updates are merged and propagated through the source, meaning no single updated view is guaranteed to remain unchanged. The technical challenges are: (1) characterizations of update preservation (for *consistency*) and of no update (for *acceptability*), and (2) an appropriate *compositional* notion of well-behavedness based on the characterizations. To the best of our knowledge, existing approaches either rely on global (non-compositional) properties or adopt weaker properties without update preservation. For example, Hu et al. [24] and Mu et al. [40] propose the duplication lens to support multiple views with the desired behavior, but they abandon the preservation of updates (consistency). As a remedy, Hidaka et al. [18] propose a weaker consistency property called WPutGet, where the modified and final views can differ, but both must result in the same update to the original source; this neither preserves the very update the user provided, nor is it compositional. As far as we are aware, using existing generalizations/extensions to lenses does not resolve this issue. Many of them (e.g., [1,2,11,19,20]) require the updated view to be identical to the view of the updated source, which is too strong to accommodate the above behavior of multiple views. Some categorical generalizations [8,44] assume a form of the PutPut law [13], which intuitively states that put must preserve update composition. However, in the literature of view updating, this law is considered too strong to permit practical transformations (see, e.g., [13,25]).

[3] We use the terminology used in Bancilhon and Spyratos [4]. Foster et al. [12,13] called the acceptability and consistency laws GetPut and PutGet, respectively.

To specify whether an update is preserved in the presence of multiple views sharing a source, we need a clear notion to represent the intentions underlying the user's updates. Consider the simplest situation where a source is simply copied into two views. For example, if the source state is 5, the corresponding view is $(5, 5)$. Suppose that this view is changed to $(7, 5)$. If the user's intention is just to change the first view from 5 to 7, without caring about the second, we can safely propagate the update to obtain the updated source 7 whose view is $(7, 7)$, where the user's intention is preserved. In contrast, if the user's intention is to change the pair of views to $(7, 5)$, there is no way to propagate the change to the source while preserving the user's intention. However, if we represent the updated state as $(7, 5)$, the intention is ambiguous. In general, we can have simultaneous updates on multiple views with independent intentions. In such cases, we try to merge these intentions into one.

This observation naturally suggests representing these update intentions by using partial orders, where comparing two update intentions $u_1 \leq u_2$ means that u_1 is subsumed by, or preserved in u_2. For example, in the previous example, we can define $\Omega \leq n$ for any n, where Ω represents an unspecified number, and $(7, \Omega) \leq (7, 7)$ indicates that the intention of updating only the first copy to 7 is preserved in the view $(7, 7)$. Additionally, partial orders provide a natural notion of merging two update intentions: the least upper bound or the join $(\vee)$. The join operator is partial in general, but this partiality is rather desirable to model conflicting updates: recall that, if we intend to update one copy of a source to 7 and another to 5, no source update can reflect these conflicting intentions.

A similar idea appears in conflict-free replicated datatypes (CRDTs) [45], which are used to synchronize distributed replicas that are updated in an independent and concurrent manner without coordination. In state-based CRDTs, replica states form a join-semilattice, where the $(\vee)$ operator is used to merge updates. As long as updates make states greater than or equal to the original states, the commutative, associative, and idempotent properties of the join guarantee that every replica reaches a consistent state without rollbacks. Sending out whole states appears impractical, but it is known that it suffices to send delta states u, where an update is described as $s \vee u$ [3]. Although requiring join-semilattices and monotonic updates is too strong for our purposes in general, this connection suggests two advantages of handling partially ordered domains. First, our bidirectional transformations can now handle CRDTs to enlarge potential applications. Second, we can borrow some ideas from concrete CRDTs to design update intentions for concrete bidirectional transformation examples.

Equipped with partial orders to represent update intentions, a remaining challenge is to establish an appropriate notion of ordering-aware well-behavedness that is suitable for compositional reasoning. The well-behavedness should be independent of concrete representations of partially-specified states, to support both simple partially-specified states that just allow complete unspecifiedness and partially-specified states tailored to concrete datatypes like CRDTs.

In this paper, we propose a novel bidirectional transformation framework, which we call partial-state lenses. Its key feature lies in lenses that can operate

on partially-ordered update intentions, which we call partially-specified states. Leveraging the partial ordering, our framework provides compositional well-behavedness that guarantees preservation of updates across multiple views sharing a source, independently of concrete representations of partially-specified states. Specifically, we make the following contributions.

- We illustrate our key idea with a concrete scenario (Sect. 2). After motivating the source-sharing problem with it, we observe a limitation of classical lenses: the lack of a domain capable of expressing the user's intent, namely, the intent of what an update should finally result in. We then demonstrate how having partially-specified states addresses this issue.
- We formalize the partial-state lens framework, which incorporates compositional well-behavedness concerning partial specifiedness (Sect. 3). This is the first lens framework that achieves all of: (1) view-to-view update propagation in multiple views via a shared source, (2) guaranteed preservation of user's updates, (3) compositional reasoning, and (4) interoperability with the classical lens framework [12, 13].
- We demonstrate the utility of the proposed framework by revisiting the scenario and informal solution from Sect. 2, showing how our framework can model the scenario through the composition of concrete lenses (Sect. 4).
- We give a recipe to encode updates into partially-specified states, demonstrating that partial-state lenses are hybrids of state-based systems and operation-based systems [2, 11, 20] (Sect. 5).

Finally, we discuss related work (Sect. 6) and conclude the paper (Sect. 7). Proofs of key claims, as well as additional statements and discussions, are provided in the extended version [31]. We also provide mechanized proofs in Agda[4] and a prototype implementation in Haskell.[5]

2 Motivating Scenario: Shared-Source Problem

In this section, we will provide an overview of our approach using a motivating example.

2.1 A Motivating Scenario

Consider managing tasks (to-dos) using two views: one (ONGOING) for all ongoing tasks and the other (TODAY) for all tasks due today. We assume that these tasks may be updated individually, maybe because the views correspond to different panes in the same application or because these tasks are shared with others. This example models a situation where bidirectional transformations are used to extract (and transform) data for application components to manage. Such an

[4] https://github.com/kztk-m/ps-lenses-agda
[5] https://github.com/kztk-m/ps-lenses-hs

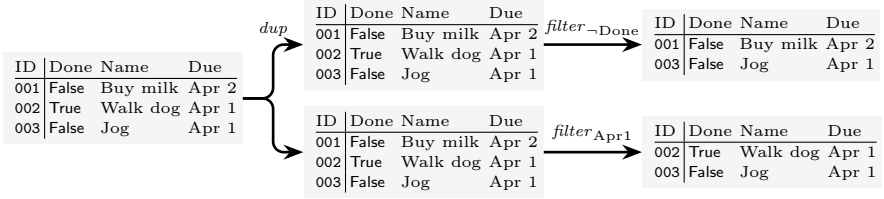


Fig. 1: Forward (*get*) behavior of composition of lenses *dup* and $\mathit{filter}_{\neg\mathrm{Done}}/\mathit{filter}_{\mathrm{Apr1}}$

application is a common introductory example in Web application programming.[6] We could go further to extract smaller pieces of data for smaller components (such as a task name for a text box), which would be preferable, but for simplicity here we consider these two, coarser-grained views.

For example, for the set s_{tl} of whole tasks on the right, its corresponding views are v_{og} and v_{dt} (assuming today is Apr 1). (We use table-like notations solely for readability; we do not make any assumptions on their actual representations at this point.)

$$s_{\mathrm{tl}} = \left(\begin{array}{lllll} \mathrm{ID} & \mathrm{Done} & \mathrm{Name} & \mathrm{Due} \\ \hline 001 & \mathsf{False} & \mathrm{Buy\ milk} & \mathrm{Apr\ 2} \\ 002 & \mathsf{True} & \mathrm{Walk\ dog} & \mathrm{Apr\ 1} \\ 003 & \mathsf{False} & \mathrm{Jog} & \mathrm{Apr\ 1} \end{array}\right)$$

$$v_{\mathrm{og}} = \left(\begin{array}{lllll} \mathrm{ID} & \mathrm{Done} & \mathrm{Name} & \mathrm{Due} \\ \hline 001 & \mathsf{False} & \mathrm{Buy\ milk} & \mathrm{Apr\ 2} \\ 003 & \mathsf{False} & \mathrm{Jog} & \mathrm{Apr\ 1} \end{array}\right)$$

$$v_{\mathrm{dt}} = \left(\begin{array}{lllll} \mathrm{ID} & \mathrm{Done} & \mathrm{Name} & \mathrm{Due} \\ \hline 002 & \mathsf{True} & \mathrm{Walk\ dog} & \mathrm{Apr\ 1} \\ 003 & \mathsf{False} & \mathrm{Jog} & \mathrm{Apr\ 1} \end{array}\right)$$

Then, we consider connecting the source with the two views by bidirectional transformations. Suppose that the connections from the whole source to these views are given by $\mathit{filter}_{\neg\mathrm{Done}}$ and $\mathit{filter}_{\mathrm{Apr1}}$. Instead of considering them individually, we can bundle them by using the duplication lens [24, 40]. The entire transformation is hence given as a composition of *dup* and $\mathit{filter}_{\neg\mathrm{Done}}/\mathit{filter}_{\mathrm{Apr1}}$, whose forward behavior is depicted in Fig. 1. In the forward direction, *dup* simply copies the source, so that lenses to be composed can work on their copies. In the backward direction, *dup* merges updates on the copies—we leave the concrete definition for now, as how to provide a suitable one for compositional reasoning is a subject of discussion in this section. Theoretically, *dup* enables us to focus on a single lens instead of considering interactions among lenses. Practically, thanks to the internalization, we can introduce sharing dynamically in the transformation, allowing us to compose lenses involving sharing. A *dup*-like operation is particularly useful in giving a compositional semantics to (non-linear) bidirectional programming languages [18, 37].

2.2 Issues

Consider adding a new task "Buy egg" to the ONGOING view, as v'_{og} on the right. After propagating this update by the backward

$$v'_{\mathrm{og}} = \left(\begin{array}{lllll} \mathrm{ID} & \mathrm{Done} & \mathrm{Name} & \mathrm{Due} \\ \hline 001 & \mathsf{False} & \mathrm{Buy\ milk} & \mathrm{Apr\ 2} \\ 003 & \mathsf{False} & \mathrm{Jog} & \mathrm{Apr\ 1} \\ 004 & \mathsf{False} & \mathrm{Buy\ egg} & \mathrm{Apr\ 1} \end{array}\right)$$

[6] See, for example, https://todomvc.com/, though with different views.

transformation (*put*) and by the subsequent
forward transformation (*get*), the added tuple is expected to appear in the original source and the TODAY view as follows:

$$s'_{\mathrm{tl}} = \left(\begin{array}{l|lll} \mathsf{ID} & \mathsf{Done} & \mathsf{Name} & \mathsf{Due} \\ \hline \mathsf{001} & \mathsf{False} & \text{Buy milk} & \text{Apr 2} \\ \mathsf{002} & \mathsf{True} & \text{Walk dog} & \text{Apr 1} \\ \mathsf{003} & \mathsf{False} & \text{Jog} & \text{Apr 1} \\ \mathsf{004} & \mathsf{False} & \text{Buy egg} & \text{Apr 1} \end{array}\right) \qquad v'_{\mathrm{dt}} = \left(\begin{array}{l|lll} \mathsf{ID} & \mathsf{Done} & \mathsf{Name} & \mathsf{Due} \\ \hline \mathsf{002} & \mathsf{True} & \text{Walk dog} & \text{Apr 1} \\ \mathsf{003} & \mathsf{False} & \text{Jog} & \text{Apr 1} \\ \mathsf{004} & \mathsf{False} & \text{Buy egg} & \text{Apr 1} \end{array}\right).$$

To achieve this behavior, the backward behavior of *dup* must take $(s'_{\mathrm{tl}}, s_{\mathrm{tl}})$ in addition to the original source s_{tl} to return s'_{tl}. Existing *dup* lenses achieve this by detecting which copies are updated by comparing them with the original source [24] or by marking explicitly which view is updated [40]. However, this approach has two issues. First, this particular behavior violates the consistency property [4, 13] presented in Sect. 1, which states *preservation of updates*; we updated the views to $(v'_{\mathrm{og}}, v_{\mathrm{dt}})$ but finally obtained $(v'_{\mathrm{og}}, v'_{\mathrm{dt}})$ after *put*-then-*get*. To the best of our knowledge, no notion of update preservation exists that is compositional and allows such view-to-view update propagation via a shared source. Second, the approach does not scale to cases where two views are updated simultaneously. Simultaneous updating introduces a further difficulty: even when focusing on a single view, the consistency property no longer holds in general. Even without simultaneous updating, stating the consistency by focusing on a single view is unsatisfactory because it complicates the reasoning: we need to perform aliasing analysis to prevent copies from being mixed up in transformations, as otherwise we cannot project "a view" from the final result.

2.3 Approach: Partial-State Lenses

The above issues stem from the fact that mere states are not expressive enough to represent the user's intentions. For example, if we can state that what is changed in the views $(v'_{\mathrm{og}}, v_{\mathrm{dt}})$ is the insertion of the task ID `004` to the first view, we can state the update preservation as the inserted task is present after the synchronized state $(v'_{\mathrm{og}}, v'_{\mathrm{dt}})$. To state update preservation, such intentions must be compared, and to support simultaneous updating, they must come with a merge operation. This suggests that such intentions must be partially-ordered.

We call these intentions *partially-specified states* and allow our lenses to handle them in addition to ordinary *proper* states. One may think that partially-specified states are hybrids of states and updates (also called deltas [11], edits [20] or operations [39, 45]). Like updates, they can express finer intentions—for example, distinguishing an insertion of a single task from the entire state change—while, as states, they allow update preservation to be stated simply by comparing them using the partial order. A similar idea has been adopted in delta-state CRDTs [3], which are also partially-ordered and mix advantages of deltas and states, where the merge operation is guaranteed to be total—this is not the case for us as it is too strong for our purposes. Unlike CRDTs, where distributed replicas are updated asynchronously without coordination, the execution of *put* is often

coordinated and synchronous, which makes failing a feasible option, compared with silent resolution of competing updates.

Let us go back to the motivating example in Sect. 2.1. Leaving the formal definition of partially-specified states used for this example to Sect. 4, here we will illustrate its intuitive behavior for partially-specified states. When a user adds a task with ID 004 "Buy egg" due Apr 1 to the ONGOING view and leaves the TODAY view, if the user's intention underlying the update is to keep the added task in the ONGOING view, the intention is expressed as the pair $(w_{\mathrm{og}}, \Omega)$ with

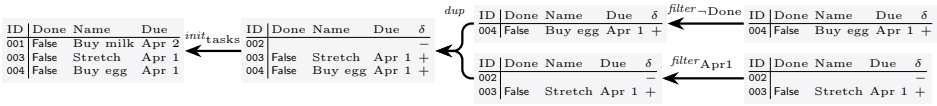
$$w_{\mathrm{og}} = \left(\begin{array}{c|c|c|c|c} \mathrm{ID} & \mathrm{Done} & \mathrm{Name} & \mathrm{Due} & \delta \\ \hline 004 & \mathsf{False} & \mathrm{Buy\ egg} & \mathrm{Apr\ 1} & + \end{array}\right).$$

Here, for visual simplicity, we represent (strictly) partially-specified states as tables with an extra field δ for update marks. The $(+)$ mark indicates the intention that the corresponding tuple must be present; i.e., it requests upsertion of the marked tuples. We also have the $(-)$ mark that indicates that the corresponding tuple must not be present; $(-)$-marked tuples only hold IDs because the information is sufficient for deletion. Formally, these intentions are modeled by ordering between a partially-specified state and a proper state. Strictly partially-specified states are ordered by the subset inclusion, and the least element is denoted by Ω,[7] which corresponds to the empty table and intuitively means "anything".

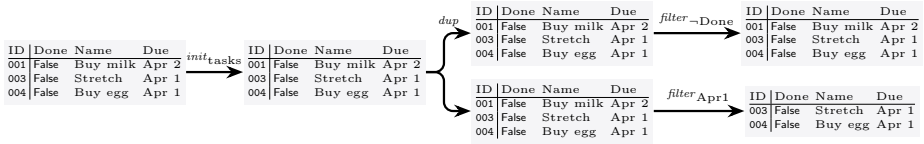
The backward behaviors of $\mathit{filter}_{\neg \mathrm{Done}}$ and $\mathit{filter}_{\mathrm{Apr1}}$ simply pass w_{og} and Ω through. They are then merged by *dup* using $\vee$ as $w_{\mathrm{og}} \vee \Omega = w_{\mathrm{og}}$; recall that Ω is the least element, which is the unit of $\vee$. This result indicates that the $(+)$-marked task in w_{og} should be inserted into s_{tl} (which results in s'_{tl}) to accommodate the view update $(w_{\mathrm{og}}, \Omega)$.

We may end the backward transformation here, assuming some external process to reflect w_{og} into s_{tl}, by leveraging the update aspect of a partially-specified state w_{og}. Instead, similarly to *dup* that internalizes the source sharing, we also internalize this updating process as a lens in our framework. This not only simplifies our formalization but also enables a compositional design of such update reflection. Specifically, we use a lens called a *ps-initiator* whose *get* embeds proper states into partially-specified states and whose *put* (tries to) reflect a partially-specified state into the original proper state, implementing the semantics of a partially-specified state as an update. Let us write $\mathit{init}_{\mathrm{tasks}}$ for a particular ps-initiator for this example, whose *put* is designed to take, for example, $(s_{\mathrm{tl}}, w_{\mathrm{og}})$ to return s'_{tl}. In general, its *put* for (s, w) updates or inserts tasks marked $(+)$ in w into s, depending on whether a task with the same ID already exists in s (i.e., its *put* upserts the tasks marked $(+)$ in w into s), and then deletes tasks marked by $(-)$. This behavior of the *put* of $\mathit{init}_{\mathrm{tasks}}$ respects the above-mentioned meaning of partially-specified states; formally, its *put* is defined so that $w \leq s'$ holds for s' with $\mathit{put}\ (s, w) = s'$—in particular, $w_{\mathrm{og}} \leq s'_{\mathrm{tl}}$. Now the source is updated to s'_{tl}. The views of s'_{tl} are $(v'_{\mathrm{og}}, v'_{\mathrm{dt}})$. We have $w_{\mathrm{og}} \leq v'_{\mathrm{og}}$ for the reason we have explained earlier, and $\Omega \leq v'_{\mathrm{dt}}$ as Ω is the least element. Then, we have

[7] To avoid confusion with $\bot$ often used to denote undefinedness of a partial function, we use Ω to denote the least element (if exists) in partially-specified states.



(a) Backward (*put*), reusing original sources from Fig. 1; all involved lenses are lawful.



(b) Forward (*get*) after the Backward Execution (*put*) in Fig. 2a

Fig. 2: Composition of partial-state lenses *dup* and $\mathit{filter}_{\neg\mathrm{Done}}$/$\mathit{filter}_{\mathrm{Apr1}}$ for simultaneous updates

$(w_{\mathrm{og}}, \Omega) \leq (v'_{\mathrm{og}}, v'_{\mathrm{dt}})$ by point-wise ordering, stating that the update intention $(w_{\mathrm{og}}, \Omega)$ is in fact preserved.

This approach extends easily to simultaneous updates. Suppose that the user also deletes the task with ID 002 and changes the name of the task with ID 003 to "Stretch" on the TODAY view, simultaneously with the update w_{og} on the ONGOING view. Suppose also that the underlying intention of the update is exactly the deletion and the task name change. Then, we can represent the user's intention as $(w_{\mathrm{og}}, w_{\mathrm{dt}})$ with

$$w_{\mathrm{dt}} = \left(\begin{array}{l|llll|l} \mathrm{ID} & \mathrm{Done} & \mathrm{Name} & \mathrm{Due} & & \delta \\ \hline 002 & & & & & - \\ 003 & \mathsf{False} & \mathrm{Stretch} & \mathrm{Apr\ 1} & & + \end{array}\right).$$

Again, the backward behaviors of $\mathit{filter}_{\neg\mathrm{Done}}$ and $\mathit{filter}_{\mathrm{Apr1}}$ do nothing interesting and return $(w_{\mathrm{og}}, w_{\mathrm{dt}})$ intact. Then, the *put* of *dup* merges the two updates as:

$$w_{\mathrm{merged}} = w_{\mathrm{og}} \vee w_{\mathrm{dt}} = \left(\begin{array}{l|llll|l} \mathrm{ID} & \mathrm{Done} & \mathrm{Name} & \mathrm{Due} & & \delta \\ \hline 002 & & & & & - \\ 003 & \mathsf{False} & \mathrm{Stretch} & \mathrm{Apr\ 1} & & + \\ 004 & \mathsf{False} & \mathrm{Buy\ egg} & \mathrm{Apr\ 1} & & + \end{array}\right)$$

After that, the *put* of $\mathit{init}_{\mathrm{tasks}}$ reflects w_{merged} into s_{tl}, which results in:

$$s''_{\mathrm{tl}} = \left(\begin{array}{l|lll} \mathrm{ID} & \mathrm{Done} & \mathrm{Name} & \mathrm{Due} \\ \hline 001 & \mathsf{False} & \mathrm{Buy\ milk} & \mathrm{Apr\ 2} \\ 003 & \mathsf{False} & \mathrm{Stretch} & \mathrm{Apr\ 1} \\ 004 & \mathsf{False} & \mathrm{Buy\ egg} & \mathrm{Apr\ 1} \end{array}\right)$$

Accordingly, by the forward transformation, the views are changed to:

$$v''_{\mathrm{og}} = \left(\begin{array}{l|lll} \mathrm{ID} & \mathrm{Done} & \mathrm{Name} & \mathrm{Due} \\ \hline 001 & \mathsf{False} & \mathrm{Buy\ milk} & \mathrm{Apr\ 2} \\ 003 & \mathsf{False} & \mathrm{Stretch} & \mathrm{Apr\ 1} \\ 004 & \mathsf{False} & \mathrm{Buy\ egg} & \mathrm{Apr\ 1} \end{array}\right) \quad v''_{\mathrm{dt}} = \left(\begin{array}{l|lll} \mathrm{ID} & \mathrm{Done} & \mathrm{Name} & \mathrm{Due} \\ \hline 003 & \mathsf{False} & \mathrm{Stretch} & \mathrm{Apr\ 1} \\ 004 & \mathsf{False} & \mathrm{Buy\ egg} & \mathrm{Apr\ 1} \end{array}\right)$$

This backward and forward behavior is depicted in Fig. 2. Observe again that w_{og} and w_{dt} are preserved in v''_{og} and v''_{dt}, respectively.

2.4 Additional Advantage: Fine-Grained Descriptions of Updates

As we mentioned earlier, a partially-specified state lies in between a (proper) state and an update. This dual aspect allows our state-based framework to enjoy some advantages of operation-based systems [2,11,20,39]. Our *put* can distinguish updates that have the same net effect on states, because partially-specified states can convey information that is not visible from proper states.

Recall that the motivating example models the extraction of data managed in a to-do management application. Thus, it is desirable if the views support operations such as task insertion, task renaming, task completion and task postponing. However, this is known to be difficult in purely state-based systems. For example, in a state-based system like classical lenses [13], we cannot distinguish task completions from task deletions for the ONGOING view as their net effect on states is the same; both completion and deletion will remove the task from the ongoing view. Similarly, for the TODAY view, we cannot distinguish task postponing from task deletions for the same reason. The need to distinguish such updates is a key motivation for operation-based systems [11,20,39], where *put* transforms updates (or operations) rather than states.

Partially-specified states can also tackle this particular challenge by extending partially-specified states. For the ONGOING view, or the view of $\mathit{filter}_{\neg\mathrm{Done}}$, we consider the mark (✓) in addition, which intuitively requires completion of the tasks, but in the view of $\mathit{filter}_{\neg\mathrm{Done}}$, where every task is ongoing, they can only result in removal of the marked tasks. An example of our new partially-specified state is shown in Fig. 3. The *put* behavior of $\mathit{filter}_{\neg\mathrm{Done}}$ converts (✓)-marks to (+)-marks, reflecting the intention of the (✓) mark; recall that the (+) mark indicates upsertion. The behavior of $\mathit{init}_{\mathrm{tasks}}$ and *dup* is unchanged. For example, for the original source s_{tl} and the updated view in Fig. 3, the *put* of $\mathit{init}_{\mathrm{tasks}}$ returns the tasks in Fig. 4, where the task with `001` is deleted and the one with `003` is completed. Similarly, for the TODAY view, we consider the mark (P), which intuitively means postponing; the mark does not convey the information of postponing days, as it is carried in the task itself. Similarly to the (✓) case, (P)-marked tuples specify that the marked tuples will not be presented in the view of $\mathit{filter}_{\mathrm{Apr1}}$. Again, in the *put* execution, $\mathit{filter}_{\mathrm{Apr1}}$ converts (P) to (+), requesting postponing.

ID	Done	Name	Due	δ
001				−
003	True	Jog	Apr 1	✓

Fig. 3: A partially-specified state involving completion and deletion

ID	Done	Name	Due
002	True	Walk dog	Apr 1
003	True	Jog	Apr 1

Fig. 4: The update source corresponding to the update in Fig. 3

A caveat is that, when we compare such partially-specified states with proper states, we cannot escape from the gap in expressiveness of user's intention between them. For example, on the view of $\mathit{filter}_{\neg\mathrm{Done}}$, we cannot distinguish deletions and completions via proper states as no completed tasks are allowed on the view. This means, on that view, $w \leq t$ for a completion intention w and a proper state t where the task to be completed in w is not present. This is intrinsic: the

difference between completion and deletion in the view of $\mathit{filter}_{\neg\mathrm{Done}}$ cannot be expressed in proper states. However, this does not directly mean our lenses can garble such intentions. We prevent this by design: we allow partially-specified states also in *get*, which comes with other benefits (Sect. 2.5).

2.5 Partially-Specified States in *get*

In Sect. 2.3, we primarily focused on how *put* makes use of partially-specified states, as demonstrated in Fig. 2a. The *get* direction becomes relevant when discussing compositional reasoning of well-behavedness, which concerns statements about the behavior of *put*-then-*get* and *get*-then-*put*.

Since the main purpose of partially-specified states is to improve *put*'s behavior in handling multiple views, it is natural to ask why we care about them in *get*. This design has both theoretical and practical benefits.

Simplicity is one of the most apparent theoretical benefits. If *get* operates only on proper states and *put* propagates only partially-specified states, the original and updated states would belong to different domains, as $\mathit{get} \in S \to V$ and $\mathit{put} \in S \times P \to Q$ similarly to update-update lenses [2], where P and Q are sets of partially-specified states corresponding to the sets S and V of proper states, respectively. This, however, complicates well-behavedness (round-tripping) as we cannot pass the *put* results directly to *get* and vice versa. We may need ways to connect S to P and V to Q, and lenses must preserve such connections as well.

This simplicity leads to a benefit that is both theoretical and practical: reusability. Our lenses follow classical lenses [12, 13] computationally: in our framework, a lens between S and V is a pair of transformations $\mathit{get} \in S \to V$ and $\mathit{put} \in S \times V \to S$ operating on the same domains (we shall ignore partiality of these transformations for now). The only difference is that we consider partially-specified states and partial ordering in S and V when reasoning about well-behavedness of lenses. Thanks to this design, we can reuse many lenses and lens combinators from the existing literature, though with the proof obligation of our version of well-behavedness. Lens composition is such an example. In a special case, a well-behaved lens in the classical lens framework [12, 13] is also a well-behaved lens in our framework. Another benefit is that we can reuse existing lens representations for our lenses. For example, we can tuple [9, 22] the transformations to bundle common computations for the source in *get* and *put* [47]. We might also use van Laarhoven lenses [27, 41] as used in the `lens` library in Haskell, or profunctor optics [43].

The strength of the guarantee is another important benefit. If we restrict *get* to work only on proper states, the update preservation can only be stated by comparing partially-specified states with proper states, and hence struggles to reject lenses that garble finer intentions that can only be represented in partially-specified states, such as one that replaces (✓)-marked tuples with (−)-marked ones in *put*. If *get* of such a lens works also on partially-specified states, especially for those that the *put* returns, we can find that the *put* fails to preserve the user's intention by comparing the updated view and the view of the updated source.

Another practical benefit of partially-specified states in *get* is allowing competing updates on multiple views with certain precedence; in this case, our lenses will handle (delta-state [3]) CRDTs [45], which are, roughly speaking, partially-ordered states among which $\vee$ is total (i.e., *put* of *dup* never fails).

In exchange for these advantages, our framework requires programmers to define *get* for partially-specified states. This would not be overly burdensome, as we define *put* (s, v) to return a partially-specified state with the knowledge that the information is sufficient for *get* to return a meaningful result for v.

3 Lenses for Partially-Specified States

In this section, as the main contribution of this paper, we describe our proposed lens framework, *partial-state lenses* (ps-lenses, for short), that can handle unspecified states like v_{og} and v_{dt} in Sect. 2. The key point of our formalization is that it supports compositional reasoning, similarly to the original lens framework [12,13].

3.1 Preliminaries: Classical Lenses

We begin by reviewing the classical lens framework [12,13], on which our framework is based. As mentioned in Sect. 2.5, the main difference lies in the laws and structures on domains. In this paper, we consider partial functions between sets, rather than complete partial orders (CPOs) for simplicity and because *put* can fail for various reasons, such as conflicting updates on copies. Although this in effect rules out general recursions and allows only terminating functions, we believe that our discussions could be extended to CPOs straightforwardly.

A (partial) (asymmetric) *classical lens* ℓ between a "source" set S and a "view" set V is a pair of partial functions $\mathit{get} \in S \to V$ and $\mathit{put} \in (S \times V) \rightharpoonup S$. Here, we used $\rightharpoonup$ to emphasize that *put* can be partial, while we assume the totality of *get* for simplicity. We write $\mathsf{Lens}\ S\ V$ for the set of all lenses $\ell = (\mathit{get}, \mathit{put})$ between S and V, and $\mathit{get}\ \ell$ and $\mathit{put}\ \ell$ for their first and second components. Intuitively, $\mathit{get}\ \ell \in S \to V$ accesses a view from a source s; and $\mathit{put}\ \ell \in (S \times V) \rightharpoonup S$ propagates a view update, which changes $\mathit{get}\ \ell\ s$ to some v', into a source update, which in turn changes s to $\mathit{put}\ \ell\ (s, v')$. Since *put* may be partial, the update propagation may fail.

Not all lenses are reasonable ones. The following laws establish the bidirectionality of lenses [4, 13, 16]. To provide a basis for our later discussions, we accompany these with informal properties (P) that they ensure, which we will aim to achieve for partial-state lenses.

$$\forall s, s' \in S, \forall v' \in V.\ \mathit{put}\ \ell\ (s, v') = s' \implies \mathit{get}\ \ell\ s' = v' \qquad \text{(consistency)}$$

"Updates to the source's view are preserved in the updated source." (P1)

$$\forall s \in S, \forall v \in V.\ \mathit{get}\ \ell\ s = v \implies \mathit{put}\ \ell\ (s, v) = s \qquad \text{(acceptability)}$$

"No updates to the source's view means no updates to the source." (P2)

Intuitively, the consistency property, also known as PutGet [13], states that the updated source s' (that *put* returns) correctly reflects the updated view v'

(that *put* takes) in the sense that its view (that we *get*) matches with v'. The acceptability property, also known as GetPut [13], states that, if there is no update on the view (that we *get*), there is no update (resulting from *put*) on the source. The two round-tripping properties are collectively called *well-behavedness*, and lenses satisfying them are called *well-behaved*. Sometimes, an additional law called PutPut [13] is considered, which intuitively states that *put* preserves compositions of updates. We ignore the law, as it is generally considered too strong [13,25] for state-based systems; we revisit this law in Sect. 6.

The acceptability implies an important property, stability, which is called PutGetPut in the literature [40].

$$\forall s_0, s \in S, \forall v \in V.\ put\ \ell\ (s_0, v) = s \implies put\ \ell\ (s, get\ \ell\ s) = s \qquad \text{(stability)}$$
$$\text{“One round-trip leads to a stable source and view”} \qquad \text{(P3)}$$

When $put\ \ell\ (s_0, v)$ succeeds, resulting in an updated source s, and then we extract its view with $get\ \ell\ s$, the resulting pair $(s, get\ \ell\ s)$ is in a *stable state*: any further *get* or *put* with them has no effect. In other words, one round-trip—*put* then *get*—is enough to synchronize the two.

3.2 Domains: Partially-Ordered Sets with Identical Updates

As mentioned in Sects. 1 and 2, we use partially-ordered sets (posets, for short) for states, which are called partially-specified states and come with the notion of update preservation defined by the partial ordering $\leq$. We also assume that our domains (source/view spaces) are equipped with a set of identical updates I_s relative to s in order to state our law(s) corresponding to acceptability.

Definition 1. A domain $P = (S, \leq, I)$, which we call an *i-poset*, is a triple where $(P, \leq)$ is a poset and $I \subseteq S \times S$ is a reflexive subset of $(\leq)$.

For $P = (S, \leq, I)$, we write $|P|$ for S and write I_s for $\{s' \mid (s', s) \in I\}$. The intuition behind the requirement $I \subseteq (\leq)$ is that $s \leq s'$, which means that s is less specified than s', can also be regarded as s representing no update with respect to s'; conceptually, I is a sub-partial-order of $\leq$ that compares identical updates. We do not require the transitivity of I just because we do not use the property in our formal development. The difference between the two relations matters when a partially-specified state represents more than a set of possible proper states. When P has the least element Ω (or P is lower-bounded), we also assume that $\Omega \in I_s$ for any s, meaning that the update that specifies nothing can work as an identical update to any state.

Simple examples of i-posets include discrete posets where $(\leq) = (=)$, which also implies $I = (=)$, and posets obtained by adding the distinct least element to a discrete poset, where the least element Ω represents “anything”. When partially-specified states just represent sets of possible proper states, an i-poset $(2^S, (\supseteq), (\supseteq))$ obtained from the power set of a given set S of proper states provides maximum flexibility. One may exclude $\emptyset$, which represents the ill-specification, from the state space in favor of failures of *put*. The join $(\cap)$ becomes partial, then. In these simple examples, I is the same as $\leq$, but the difference between the two becomes evident in non-trivial examples like below.

Example 1 (Delimiting Identical Updates). Often, a partially-specified state assumes a certain condition on its original (proper) state. In such a case, the difference between $\leq$ and I matters. To illustrate this, consider key-value mappings represented as partial functions in $K \rightharpoonup V$ for a key set K and a value set V. Consider partially-specified states that represent deletion requests of mappings, which are naturally represented by a subset D of K. We can easily merge two deletion requests by the set union, and whether a deletion D is reflected in f is formalized in $D \cap \mathrm{dom}(f) = \emptyset$.

There are some possibilities for this domain to permit both proper state f and partially-specified state D. A choice is $P_1 = ((K \rightharpoonup V) \uplus 2^K, (\leq), (\leq))$, where $(\leq)$ is the smallest reflexive relation that includes (D, D') with $D \subseteq D'$ and (D, f) with $D \cap \mathrm{dom}(f) = \emptyset$. However, since P_1 permits too many "identical updates", such as K (which intuitively means "delete everything") for $f = \emptyset$, it is difficult or complicated to define lawful (well-behaved in the sense of Definition 4) lenses, which must preserve identical updates (ps-acceptability). A more practical choice is $P_2 = ((K \rightharpoonup V) \uplus 2^K, (\leq), I)$, where I is the smallest reflexive relation that includes (D, D') with $D \subseteq D'$ and (D, f) with $D = \emptyset$, ruling out deleting non-existing IDs in I_f. □

We can even encode updates [2, 11, 20, 39] as partially-specified states by pairing them with their starting points. See Sect. 5.1 for the details.

Since there is a variety of domains as we have seen, our formalization is designed to be independent of any concrete representation of partially-specified states: the only assumption on the domains (sets of source and view states) is that they form i-posets. (In contrast, some concrete primitive lenses and combinators, such as those illustrated in Sect. 2, require concrete i-poset or impose further assumptions to ensure functionality and/or well-behavedness.) One might then wonder why we do not define their actions on proper states like updates. We do, but instead of integrating this into the definition of our domains, we internalize it using certain primitive lenses called ps-initiators (Sect. 3.7).

3.3 (Lawless) Partial-State Lenses

Recall that the primary goal of this paper is to give a formal treatment of lenses that handle partially-specified data. Now that we have formalized (possibly-)partially-specified data, we are ready to define (lawless-versions of) partial-state lenses.

Definition 2 ((Lawless) Partial-State Lenses). For i-posets P and Q, a *partial-state lens* (ps-lens) ℓ between P and Q is a pair of functions $get : |P| \to |Q|$ and $put : |P| \times |Q| \rightharpoonup |P|$. □

We write $\mathsf{Lens}^{\leq}\ P\ Q$ for the set of partial-state lenses $\ell = (get, put)$ between P and Q, and $get\ \ell$ and $put\ \ell$ for their first and second components. Definition 2 suggests that, without laws, partial-state lenses are no different from the classical lenses [13]. In what follows, we sometimes simply call ps lenses "lenses" unless confusion arises; they indeed are lawless lenses.

We show some examples of partial-state lenses (which indeed are lawful).

Example 2 (Identity Lens). Our simplest lens is the identity lens $idL \in \mathsf{Lens}^{\leq}\ P\ P$, defined as: $get\ idL\ s = s$ and $put\ idL\ (\_, v) = v$. □

Example 3 (Constant Lenses). Sometimes, a part of a view data is determined regardless of a source, such as element names in XML views. The constant lens $constL_a \in \mathsf{Lens}^{\leq}\ P\ Q$ for lower-bounded P (i.e., having the least element Ω) with $a \in |Q|$ is a useful building block for such a situation.

$$get\ constL_a\ \_ = a \qquad put\ constL_a\ (\_, v) = \Omega \text{ if } v \in I_a$$

Here, $v \in I_a$ instead of $v = a$ is intentional, and is required for the lens to be lawful; intuitively, this ensures that it works when composed with lenses that have a similar behavior to constant lenses. This lens is interesting for two reasons. First, its *put* is purposely partial (unless $|Q| = \{a\}$). Second, by returning the least element Ω, it explicitly states that the source can be anything. □

The next partial-state lens is more involved, and assumes an additional structure on an i-poset for it to be well-behaved.

Definition 3 (Duplicable). We call an i-poset P *duplicable* if it has a designated partial operator $\oplus \in |P| \times |P| \rightharpoonup |P|$ (called the *merge operator* for P) satisfying both of the following conditions: $x \oplus y = z$ implies $x \vee y = z$, and for any x, y and z, if $x \in I_z$ and $y \in I_z$, then $x \oplus y \in I_z$ holds. □

That is, $\oplus$ soundly computes joins, and must be total and closed (and thus must coincide with $\vee$) for identical updates to a fixed state. It follows that $\oplus$ is also idempotent. A sufficient condition for duplicability is that $\oplus$ is complete ($(\oplus) = (\vee)$), and associative, in the sense that $x \oplus (y \oplus z)$ and $(x \oplus y) \oplus z$ coincide also in their definedness as well as their outcomes. For simple i-posets, this condition is easy to meet, but is often laborious or difficult to establish for tailored ones. The duplicability is preserved by the basic constructions of i-posets, such as $P \times Q$, $P + Q$, and P_Ω (the i-poset obtained from P by adding a new least element $\Omega \notin |P|$), assuming point-wise operations for $\leq$, I, and $\oplus$.

Example 4. Our duplication lens $dup \in \mathsf{Lens}^{\leq}\ P\ (P \times P)$ for duplicable P is defined as:

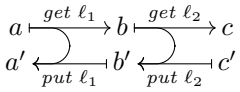
$$get\ dup\ s = (s, s) \qquad put\ dup\ \big(\_, (s_1, s_2)\big) = s_1 \oplus s_2$$

Here, the view data is simply a pair of source data. The *get* is trivial, returning a view that contains two identical copies of the original source. For *put*, we replace the source with the join of two (possibly) updated copies; the informal idea is that, by joining partially-specified states, their intentions, such as addition or deletion of particular tasks as seen in Sect. 2.3, are merged into a unified one. For example with $P = \{1, 2\}_\Omega \times \{1, 2\}_\Omega = \{(\Omega, \Omega), (1, \Omega), (\Omega, 2), (1, 2)\}$, with the point-wise ordering induced from $\Omega \leq 1, 2$, meaning that Ω is a "no-op" for the corresponding component, and with the merge operator $(\oplus) = (\vee)$ for which Ω is the unit, we have $put\ dup\ \big(s, (1, \Omega), (\Omega, 2)\big) = (1, \Omega) \oplus (\Omega, 2) = (1, 2)$.

□

Finally, we can also define the *composition* $\ell_1 \fatsemi \ell_2$ of two partial-state lenses $\ell_1 : \mathsf{Lens}^{\leq}\ P\ Q$ and $\ell_2 : \mathsf{Lens}^{\leq}\ Q\ R$ in the standard manner [13] as below:

$$\begin{aligned} &\mathit{get}\ (\ell_1 \fatsemi \ell_2)\ a = \mathit{get}\ \ell_2\ (\mathit{get}\ \ell_1\ a) \\ &\mathit{put}\ (\ell_1 \fatsemi \ell_2)\ (a, c') = \mathit{put}\ \ell_1\ \big(a, \mathit{put}\ \ell_2\ (\mathit{get}\ \ell_1\ a, c')\big) \end{aligned}$$



The composition is associative and has the identity *idL* in Example 2.

3.4 Laws: Partial-State Consistency and Acceptability

Now, we focus on the laws for partial-state lenses. We first adapt consistency and acceptability [4,13] of classical lenses into the setting of partial-state lenses, while keeping their direct informal intentions (P1 and P2). We will leave stability (P3) for the moment; as we will see soon, P1 and P2's extensions do not guarantee P3 in our setting, and we require an additional property.

Partial-State Consistency We first focus on consistency (P1)as it is directly related to the lens *dup* that plays a central role in our motivating example.

Recall that *dup* introduces the source sharing, which causes the violation of consistency due to merging of updates on the copies. For example, we have $\mathit{put}\ \mathit{dup}\ (s, ((1, \Omega), (\Omega, 2))) = (1, 2)$ for any s and the view $((1, 2), (1, 2))$ of the updated source is different from either of the updated views $(1, \Omega)$ and $(\Omega, 2)$. As mentioned in Sect. 2.3, we use $\leq$ to state update preservation: $\mathit{put}\ \ell\ (s, v') = s'$ must imply $v' \leq \mathit{get}\ \ell\ s'$, meaning that the update intention in the updated view v' is preserved in the view $\mathit{get}\ \ell\ s'$ of the updated source. To make the property compositional, we ensure $v' \leq \mathit{get}\ \ell\ s''$ for any s'' with $s' \leq s''$, ensuring that ℓ can be precomposed with lenses of the same kind. In summary, partial-state consistency for $\ell \in \mathsf{Lens}^{\leq}\ P\ Q$ is formalized as follows, satisfying P1.

$$\forall s, s' \in |P|, \forall v' \in |Q|.\ \mathit{put}\ \ell\ (s, v') \leq s' \implies v' \leq \mathit{get}\ \ell\ s' \qquad \text{(ps-consistency)}$$

Partial-State Acceptability Then, we adapt acceptability (P2).It is true that the original version is compositional and leads to the stability as requested, but it is too strong for our purpose for the following two reasons.

- We prefer *put* to return the smallest possible results, so that further merging by $\vee$ is more likely to succeed. (This is not true for a general poset but typically holds for our domains.) An extreme example is constL_{42} in Example 3, where $\mathit{put}\ \mathit{constL}_{42}\ (s_0, 42) = \Omega$, saying that the lens does not contain the updated source at all and any merge with Ω always succeeds as $\Omega \vee s = s \vee \Omega = s$.
- The original law hampers natural behavior of a lens as an update translator. Recall that, the *put* of the filter lens $\mathit{filter}_{\neg \mathrm{Done}}$ in Sect. 2.3 passes insertion requests, i.e., (+)-marked tuples, on the view into the source intact. This seemingly innocuous behavior violates the original law, because the original source may contain update requests which are not representable on the view, such as insertion requests of completed tasks. Its *get* or *put* must filter out or reject such updates for update preservation.

Thus, we adapt the original acceptability law to these behaviors. Following the informal requirement P2, which says that no-update, or identical updates, are preserved by *put*, we give partial-state acceptability as follows.

$$\forall s \in |P|, \forall v \in |Q|.\ v \in I_{get\ \ell\ s} \implies put\ \ell\ (s, v) \in I_s \qquad \text{(ps-acceptability)}$$

The original version corresponds to the case where I is equality ($=$), or equivalently $I_s = \{s\}$ for any s, on both domains.

Partial-State Weak Well-Behavedness So far, we have obtained partial-state versions of consistency and acceptability, which are the standard round-tripping properties used in this literature [4,13,16]. We collectively call the two properties *weak well-behavedness*. We call it "weak" because it does not cover stability, which we will recover in Sect. 3.5 by adding a law. Although it is "weak", when ($\leq$) is ($=$), which implies I is also ($=$) as it is a reflexive subset of ($\leq$), the weak well-behavedness coincides with the original well-behavedness. In this sense, the weak well-behavedness is a clean generalization of the original.

We state the following two properties of weakly well-behaved ps lenses.

Lemma 1. *Both ps-acceptability and ps-consistency are closed under the lens composition.* □

Lemma 2. *For a weakly well-behaved ps-lens ℓ, get ℓ is monotone.* □

Unlike *get*, our *put* is not necessarily monotone in either argument. Typically, ps-initiators are not monotone in the second argument, as they often try to keep the original source information as much as possible. For example, for $init_{\text{tasks}}$ in Sect. 2, observe that $put\ init_{\text{tasks}}\ (s_{\text{tl}}, \Omega) = s_{\text{tl}}$ while $put\ init_{\text{tasks}}\ (s_{\text{tl}}, w_{\text{merged}}) = s''_{\text{tl}}$ with $s_{\text{tl}} \not\leq s''_{\text{tl}}$. We also have a lens whose *put* is not monotone in the first argument; however, we have found no practical reason to have such lenses—they are allowed just because our requirements (laws) are minimal.

3.5 Additional Law: Partial-State Stability

Now we have laws ps-acceptability and ps-consistency that satisfy properties P2 and P1, respectively. Our next goal is to find a law that adapts stability to achieve P3, while also supporting compositional reasoning. The stability law is important. It says that one round-trip—updating the source with *put*, and then acquiring its view with *get*—captures the entire update process needed to reach a stable state $(s, get\ \ell\ s)$: any further *get* or *put* on the pair has no effect.

Issue: Non-Stability As noted earlier, weak well-behavedness alone does not guarantee stability. The acceptability property implies stability, but, in our case, we only have a weaker form of acceptability, which makes stability an issue. The challenge for partial-state lenses is that, *put* may result in a partially-specified state, like with the lens $constL_{42}$. By using ps-acceptability, we have

$put\ \ell\ (s, get\ \ell\ s) \in I_s$, which implies $put\ \ell\ (s, get\ \ell\ s) \leq s$, but it can happen that $put\ \ell\ (s, get\ \ell\ s) < s$. Although $constL_{42}$ happens to be stable,the underlying behavior—that *put* can return a smaller state than its original source—can be exploited to make unstable lenses, as illustrated below.

Example 5 (Unstable Lens). Let $\mathbf{1}$ be the singleton i-poset whose only element is $(\,)$ and N be the set $\{\underline{0}, \underline{1}, \underline{2}, \dots\}$ where $\underline{i} \leq \underline{j}$ if and only if $i \leq j$, and $I = (\leq)$.[8] Intuitively, an element $\underline{n} \in N$ represents a precedence of updates (without payload). Consider a lens $bad : \mathsf{Lens}^{\leq}\ N\ \mathbf{1}$ defined as:

$$get\ bad\ \_ = (\,) \qquad put\ bad\ (s, (\,)) = \begin{cases} \underline{0} & \text{if } s = \underline{0} \\ \underline{k} & \text{if } s = \underline{k+1} \end{cases}$$

Since $put\ bad\ (\underline{n}, (\,))$ with non-zero n results in $\underline{n-1}$, an n-fold application of $put\ bad\ (-, (\,))$ is needed for $\underline{n}$ to reach a fixed point $\underline{0}$, failing to fulfill P3.

Regardless, the lens *bad* still satisfies both ps-acceptability and ps-consistency; the former reduces to $put\ bad\ (s, (\,)) \in I_s$, which is obvious, and the latter is trivial as the only possible view is the unit value $(\,)$. Hence, weak well-behavedness is not enough to rule out this undesirable lens. □

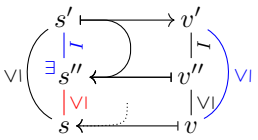
Remark. If the final source is discrete—having only concrete states, which is often desirable—stability already holds. This is because $(\leq) = I = (=)$ in a discrete domain, and thus ps-acceptability ensures $put\ \ell\ (s', get\ \ell\ s') = s'$. However, the *put* of an unstable lens may not always provide a "definitive answer", demonstrated by Example 5 where performing multiple round-trips can gradually refine the source. Thus, for the composition of unstable lenses, while the overall *put* could fail, extra local round-trips might lead to success in propagating updates. We also note that, as mentioned in Sect. 2.5, having strictly partially-specified states in the final source is useful when a lens is supposed to accept any simultaneous updates on multiple views by using CRDTs [3, 45].

Partial-State Stability Thus, we need an additional property for P3. The issue with directly enforcing the stability law is that it conflicts with compositional reasoning, a core principle underlying lenses as an approach to bidirectional transformations. It is known that stability by itself is not closed under composition (see Matsuda et al. [34, Appendix B]).

Since ps-acceptability already ensures $put\ \ell\ (s', get\ \ell\ s') \in I_{s'}$, which implies $put\ \ell\ (s', get\ \ell\ s') \leq s'$, it suffices to enforce $s' \leq put\ \ell\ (s', get\ \ell\ s')$ to entail stability. To enforce this compositionally, assuming ps-consistency and ps-acceptability, we require the following law.

$$\begin{aligned} &\forall s_0, s, s', s'' \in |P|.\, \forall v, v'' \in |Q|. \\ &\quad \big((put\ \ell\ (s_0, v) = s \leq s') \ \wedge\ (v \leq v'' \in I_{get\ \ell\ s'}) \ \wedge\ (put\ \ell\ (s', v'') = s'') \big) \\ &\quad \Longrightarrow\ s \leq s'' \end{aligned} \qquad \text{(ps-stability)}$$

[8] Hence, N is the set of natural numbers with the standard ordering. We do not use $\mathbb{N}$ here, as 1 and 2 in $\mathbb{N}$ are incompatible in specifiedness in the usual sense.

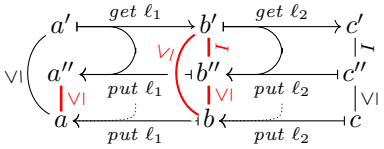


This law may be easier to understand with the diagram on the right, where the concluded parts are colored red, and the parts derived from ps-acceptability and ps-consistency are colored blue. Here, we write s for $put\ \ell\ (s_0, v)$ and v' for $get\ \ell\ s'$. We place greater elements in upper rows to highlight ordering among elements. (We omit s_0 as it is irrelevant for the ordering, and denote the omitted inputs by dotted lines in the diagram.)

The law is designed to be preserved under composition. The conditions $s \leq s'$ and $v'' \in I_{get\ \ell\ s'}$ (rather than $s = s'$ and $v'' = get\ \ell\ s'$) mirror the designs of ps-consistency and ps-acceptability. In contrast, the condition $v \leq v''$, which is enforced by the composition, is crucial to make the law practical; otherwise, even $idL \in \mathsf{Lens}^{\leq}\ P\ P$ in Example 2 does not satisfy the property for general P (e.g., take $s_0 = \Omega$, $v = 1$, $s' = s = 1$, and $v'' = \Omega$ to get $s'' = \Omega$). The definedness of $put\ \ell\ (s', v'')$ appears as a requirement instead of a conclusion of the law, which does not make a difference as it is implied by ps-acceptability.

Lemma 3. *For ℓ_1 and ℓ_2 that satisfy all of ps-acceptability, ps-consistency, and ps-stability, $\ell_1 \fatsemi \ell_2$ also satisfies ps-stability.*

Proof. Consider the diagram on the right. This diagram illustrates the execution of $put\ (\ell_1 \fatsemi \ell_2)\ (a_0, c) = a$, $get\ (\ell_1 \fatsemi \ell_2)\ a' = c'$, and $put\ (\ell_1 \fatsemi \ell_2)\ (a', c'') = a''$, with $a \leq a'$ and $c \leq c'' \leq c'$. Here, we use thick red arrows for the parts to be concluded.



We have $b \leq b'$ by the ps-consistency of ℓ_1 and $a \leq a'$. Then, we use the ps-stability of ℓ_2 to obtain $b \leq b''$. We also have $b'' \in I_{b'}$ by the ps-acceptability of ℓ_2 and $c'' \leq c'$. Then, we have $a \leq a''$, as required, by the ps-stability of ℓ_1.□

Lemma 4. *If a partial-state lens ℓ satisfies all of ps-acceptability, ps-consistency, and ps-stability, it also satisfies stability.*

Proof (Sketch). We use ps-stability with $s' = s$ and $v'' = get\ \ell\ s$ to have $s \leq s''$, where ps-consistency ensures $v \leq get\ \ell\ s$ and acceptability ensures that $s'' = put\ \ell\ (s, get\ \ell\ s)$ is defined, with satisfying $s'' \in I_s$, which implies $s'' \leq s$. Then, we have $s = s'' = put\ \ell\ (s, get\ \ell\ s)$ by antisymmetry. □

Now, we are ready to define well-behavedness of partial-state lenses.

Definition 4 (Well-Behavedness of Partial-state Lenses). A partial-state lens ℓ is well-behaved if it is weakly well-behaved and satisfies ps-stability. □

In other words, a well-behaved partial-state lens satisfies ps-acceptability, ps-consistency and ps-stability.

Theorem 1. *The well-behavedness is closed under composition. That is, $\ell_1 \fatsemi \ell_2$ is well-behaved when ℓ_2 and ℓ_1 are.* □

When P of $\ell \in \mathsf{Lens}^{\leq}\ P\ Q$ is discrete (i.e., $(\leq) = I = (=)$), ps-stability is implied by ps-acceptability; in this case, well-behavedness and weak well-behavedness coincide. When Q is discrete in addition, our (weak) well-behavedness coincides with the classical well-behavedness, showing that our formalization is a *clean extension* of the classical lens framework.

Lemma 5. *For $\ell \in \mathsf{Lens}^{\leq}\ P\ Q$ with discrete P, ℓ is weakly well-behaved if and only if it is well-behaved.* □

Lemma 6. *A classical lens $\ell \in \mathsf{Lens}\ A\ B$ is well-behaved if and only if $\ell \in \mathsf{Lens}^{\leq}\ (A, =, =)\ (B, =, =)$ is well-behaved as a ps-lens.* □

3.6 Examples of Well-Behaved Partial-State Lenses

Then, we show some examples of the well-behaved partial-state lenses.

Example 6 (Identity Lens). The identity lens $idL \in \mathsf{Lens}^{\leq}\ P\ P$ in Example 2 is well-behaved for any i-poset P. □

Example 7 (Constant Lens). The constant lens $constL_a \in \mathsf{Lens}^{\leq}\ P\ Q$ for lower-bounded P in Example 3 is well-behaved. □

Example 8 (Duplication Lens). The duplication lens $\delta \in \mathsf{Lens}^{\leq}\ P\ (P \times P)$ in Example 4 is well-behaved if P is duplicable. Its well-behavedness follows from the properties of $\oplus$ (and $\vee$): ps-acceptability holds because $\oplus$ is total and closed for identical updates to a fixed state, ps-consistency holds because $\oplus$ soundly implements $\vee$, and ps-stability holds because the sound behavior of $\oplus$ and the join is monotone. □

Nonexample 9. The lens *bad* in Example 5 is not well-behaved as it violates ps-stability. Take $s_0 = \underline{2}$ for which $s = put\ bad\ (s_0, ())) = \underline{1}$, and take $s' = s$, then we have $put\ bad\ (s', ())) = put\ bad\ (\underline{1}, ())) = \underline{0} \not\geq \underline{1} = s = s'$, violating the property. □

Example 10 (Untagging). Many primitive lenses that we so far have seen do not use their first arguments in *put*. Of course, this is not always the case. The following lens $unTagS \in \mathsf{Lens}^{\leq}\ (P + P)\ P$ is such an example.

$$\begin{array}{ll} get\ unTagS\ (\mathsf{InL}\ s_1) = s_1 & put\ unTagS\ (\mathsf{InL}\ \_)\ s = \mathsf{InL}\ s \\ get\ unTagS\ (\mathsf{InR}\ s_2) = s_2 & put\ unTagS\ (\mathsf{InR}\ \_)\ s = \mathsf{InR}\ s \end{array}$$

Observe that its *put* determines the tag only by using the source information; this is why the lens is called $unTagS$. The lens $unTagS$ is well-behaved.

The lens $unTagS$ serves as a building block of conditional branching. Due to space limitation, we omit the concrete definitions, but the idea is: we examine a condition by using a lens $\mathsf{Lens}^{\leq}\ S\ (A + B)$, perform computation on each case by lenses $\mathsf{Lens}^{\leq}\ A\ C$ and $\mathsf{Lens}^{\leq}\ B\ C$ to obtain $C + C$, and then finally erase the tag by $unTagS$. This behavior corresponds to *ccond* in the classical lens [13]. □

3.7 Relating Partial-Ordering and Updates: Ps-Initiators

We give the formal definition of ps-initiators. Although they are no different from other lenses in our framework, their general form illustrates the relationship among partially-specified states, proper states, updates, and partial orders.

As mentioned informally in Sect. 2.3, the definitions of ps-initiators $\mathit{init} \in \mathsf{Lens}^{\leq}\ S\ P$ with $S \subseteq P$ for a discrete S share the same pattern: $\mathit{get}\ \mathit{init}\ s = s$ and $\mathit{put}\ \mathit{init}\ (s, v) = v \cdot s$. Namely, its *get* embeds proper states S in (possibly) partially-specified states P, and its *put* implements the semantics of partially-specified states as updates via $(\cdot) \in P \to S \rightharpoonup S$. Since S is discrete, for *init* to be well-behaved, it suffices to satisfy ps-acceptability and ps-consistency, which reduce to the following two properties:

$$\forall v \in |P|, s \in |S|.\ v \in I_s \Longrightarrow v \cdot s = s \qquad \text{(u-acceptability)}$$
$$\forall v \in |P|, s, s' \in |S|.\ v \cdot s = s' \Longrightarrow v \leq s' \qquad \text{(u-consistency)}$$

Intuitively, $v \in I_s$ means that v represents no update with respect to s—this is what u-acceptability ensures. The u-consistency property states that the application $v \cdot s$ of an update v to a state s must, if it succeeds, preserve the intention given as v.

4 Scenario Revisited: Formal Implementation

We are now ready to revisit the example from Sect. 2 and formalize the solution that was presented informally. We first define the version corresponding to the example in Sect. 2.3 (Sects. 4.1–4.3), and then extend it for the finer-grained updates presented in Sect. 2.4 (Sect. 4.5).

Before defining concrete domains (Tasks, DT) involved in the transformation, we first present its overall structure as below.

$$\mathsf{Lens}^{\leq}\ \mathsf{Tasks}\ (\mathsf{DT} \times \mathsf{DT}) \ni \ell_{\mathrm{task}} = \mathit{init}_{\mathrm{tasks}} \fatsemi \mathit{dup} \fatsemi (\mathit{filter}_{\neg \mathrm{Done}} \times \mathit{filter}_{\mathrm{Apr1}})$$

The above involves the following lenses and lens combinators (as well as $\fatsemi$).

$$\mathit{dup} \in \mathsf{Lens}^{\leq}\ P\ (P \times P) \qquad (\times) \in \mathsf{Lens}^{\leq}\ P_1\ Q_1 \to \mathsf{Lens}^{\leq}\ P_2\ Q_2 \to \mathsf{Lens}^{\leq}\ (P_1 \times P_2)\ (Q_1 \times Q_2)$$
$$\mathit{init}_{\mathrm{tasks}} \in \mathsf{Lens}^{\leq}\ \mathsf{Tasks}\ \mathsf{DT} \qquad \mathit{filter}_{\neg \mathrm{Done}} \in \mathsf{Lens}^{\leq}\ \mathsf{DT}\ \mathsf{DT} \qquad \mathit{filter}_{\mathrm{Apr1}} \in \mathsf{Lens}^{\leq}\ \mathsf{DT}\ \mathsf{DT}$$

Here, P is an arbitrary duplicable i-poset, and P_1, P_2, Q_1, and Q_2 are arbitrary i-posets. Among them, we have already seen the definition of the *dup*lication lens in Sect. 3 and confirmed its well-behavedness. In what follows, after defining the domains involved, we will show concrete definitions of $\mathit{init}_{\mathrm{tasks}}$, $\mathit{filter}_{\neg \mathrm{Done}}$, $\mathit{filter}_{\mathrm{Apr1}}$, and $(\times)$, and confirm that the lenses ($\mathit{init}_{\mathrm{tasks}}$ and the filters) are well-behaved and that the lens combinator $(\times)$ preserves well-behavedness.

4.1 Domains

Roughly speaking, Tasks is a discrete poset of proper states, and DT additionally allows addition and deletion updates. For simplicity, we use these names to

denote both the i-posets and their underlying sets. Their definitions are similar to Example 1, but now we consider additions as well as deletions. We treat $t \in \mathsf{Tasks}$ as a partial function, i.e., $\mathsf{Tasks} \triangleq \mathsf{ID} \rightharpoonup (\mathsf{Bool} \times \mathsf{String} \times \mathsf{Date})$. Strictly partially-specified states have the form of (A, D) with $A \in \mathsf{Tasks}$ and $D \subseteq \mathsf{ID}$, with the restriction of $\mathrm{Disj}(D, \mathrm{dom}(A))$. Here, we write $\mathrm{Disj}(S, T)$ to mean that S and T are disjoint, i.e., $S \cap T = \emptyset$. Intuitively, (A, D) constrains that A must be present and D must not be present; $\mathrm{Disj}(D, \mathrm{dom}(A))$ ensures feasibility. Formally, we define $\mathsf{DT} \triangleq \mathsf{Tasks} \cup \{(A, D) \in \mathsf{Tasks} \times 2^{\mathsf{ID}} \mid \mathrm{Disj}(D, \mathrm{dom}(A))\}$, with the following ordering and identical updates (obvious reflexivity rules are omitted).

$$\frac{A \subseteq A' \quad D \subseteq D'}{(A, D) \leq (A', D')} \qquad \frac{A \subseteq t \quad \mathrm{Disj}(D, \mathrm{dom}(t))}{(A, D) \leq t} \qquad \frac{A \subseteq A' \quad D \subseteq D'}{(A, D) \in I_{(A', D')}} \qquad \frac{A \subseteq t}{(A, \emptyset) \in I_t}$$

We define its merge operator $\oplus$ as below.

$$t \oplus t = t \qquad t \oplus (A, D) = (A, D) \oplus t = t \text{ if } (A, D) \leq t$$
$$(A, D) \oplus (A', D') = (A \cup A', D \cup D') \text{ if } (A \cup A') \in \mathsf{Tasks} \wedge \mathrm{Disj}(D \cup D', \mathrm{dom}(A \cup A'))$$

The condition in the last line says that $(A \cup A', D \cup D')$ is a valid partially-specified state in DT. This $\oplus$ soundly implements the join in DT, and is total and closed for I_x for any x, which entails that DT is duplicable. Actually, the construction of $\leq$, I, and $\oplus$ here is an instance of the general recipe (Sect. 5.1) and the conditions for duplicability are easy to enforce (Lemma 7). This design of i-poset is inspired by the 2P-Set CRDT [45], which uses a pair (A, D) to represent the set $A \setminus D$.

An example of DT's element is w_{merged} in Sect. 2.3, where we used marks $(+)$ or $(-)$ in tables instead of using pairs (A, D). The partially-specified states w_{og} and w_{dt} are also elements in DT.

4.2 Ps-Initiator: $\mathit{init}_{\mathbf{tasks}}$

The ps-initiator $\mathit{init}_{\mathrm{tasks}}$ connects proper states in Tasks and partially-specified states in DT. We give it as an instance of a general form discussed in Sect. 3.7. Since we have defined its source and view domains (Tasks and DT), the remaining task is an appropriate definition of $(\cdot)$ that satisfies u-acceptability and u-consistency. This is straightforward: we define it as $t' \cdot \_ = t'$ and $(A, D) \cdot t = \{(k \mapsto v) \in (t \lhd A) \mid k \notin D\}$. Here, $t \lhd A$ upserts A into t. Formally, $(t \lhd A)(k)$ is defined as $A(k)$ if $k \in \mathrm{dom}(A)$ and otherwise as $t(k)$. For example, we have $w_{\mathrm{og}} \cdot s_{\mathrm{tl}} = s'_{\mathrm{tl}}$ and $w_{\mathrm{merged}} \cdot s_{\mathrm{tl}} = s''_{\mathrm{tl}}$.

The order of upsertion and deletion above does not matter as we required $\mathrm{Disj}(D, \mathrm{dom}(A))$. Requiring this is a design choice: without this requirement, more partially-specified states can be merged, but the notion of preservation of updates is weakened. Without it, we need to give precedence for A or D, which makes the non-precedent one a soft constraint (as it may be overwritten).

4.3 Filter Lenses

Next, we give the definition of $\mathit{filter}_{\neg\mathrm{Done}} \in \mathsf{Lens}^{\leq}\ \mathsf{DT}\ \mathsf{DT}$. We shall omit the one for $\mathit{filter}_{\mathrm{Apr1}}$ as it is similar. In the forward direction, the lens $\mathit{filter}_{\neg\mathrm{Done}}$ simply

filters ongoing tasks.

$$\mathit{get}\ \mathit{filter}_{\neg\mathrm{Done}}\ s = \begin{cases} t_{\mathrm{OG}} & \text{if } s = t \in \mathsf{Tasks} \\ (A_{\mathrm{OG}}, D) & \text{if } s = (A, D) \end{cases}$$

Here, given $t \in \mathsf{Tasks}$, we write t_{OG} for a mapping obtained by collecting all ongoing tasks in t, i.e., $t_{\mathrm{OG}} = \{(k \mapsto v) \in t \mid v = (\mathsf{False}, \_, \_)\}$. In the backward direction, if the updated view is a proper state, it must return a proper state. Otherwise, the successive backward execution could add more tasks to violate the update preservation. Specifically, it upserts t' into the original filtered-out tasks. If the updated view is a partially-specified state, it just returns the partially-specified state intact regardless of the source s.

$$\mathit{put}\ \mathit{filter}_{\neg\mathrm{Done}}\ (s, v) = \begin{cases} (t \setminus t_{\mathrm{OG}}) \lhd t' & \text{if } v = t' \in \mathsf{Tasks}_{\mathrm{OG}} \wedge s = t \in \mathsf{Tasks} \\ (A, D) & \text{if } v = (A, D) \wedge A \in \mathsf{Tasks}_{\mathrm{OG}} \end{cases}$$

We write $\mathsf{Tasks}_{\mathrm{OG}}$ for a set of partially-specified states that consists only of ongoing tasks, i.e., $\{t \in \mathsf{Tasks} \mid \forall k, t(k) = (\mathsf{False}, \_, \_)\}$. This *put* returns the updated view intact if it is a strictly partially-specified state. An intuition behind this behavior is that a strictly partially-specified state represents update requests and addition/removal requests on the filtered result can be translated into those on the original tasks.

This lens passes D intact regardless of whether they are on ongoing tasks or not. This is safe, in the sense that the lens is well-behaved. For ps-consistency, this only matters when we prove $(A, D) \leq \mathit{get}\ \mathit{filter}_{\neg\mathrm{Done}}\ t = t_{\mathrm{OG}}$ from $\mathit{put}\ \mathit{filter}_{\neg\mathrm{Done}}\ (s, (A, D)) = (A, D) \leq t$. Even when D contains removal requests for completed tasks, we still have $(A, D) \leq t_{\mathrm{OG}}$ (provided that $A \in \mathsf{Tasks}_{\mathrm{OG}}$, which is ensured by the *put*'s guard). Notice that, by $t_{\mathrm{OG}} \subseteq t$, $\mathrm{Disj}(D, \mathrm{dom}(t))$ implies $\mathrm{Disj}(D, \mathrm{dom}(t_{\mathrm{OG}}))$. Recall that D constrains that tasks with the IDs will not be present—it is allowed that more tasks will not be present (as long as A is contained). For ps-acceptability, there is no such concern as $(A, D) \in I_t$ simply enforces D to be the empty set.

4.4 Lens Combinator $\times$

The definition of $\times$ is no different from the one appearing in the literature [42]. Let $\ell_1 \in \mathsf{Lens}^{\leq}\ P_1\ Q_1$ and $\ell_2 \in \mathsf{Lens}^{\leq}\ P_2\ Q_2$ be lenses. Then, we define the lens $(\ell_1 \times \ell_2) \in \mathsf{Lens}^{\leq}\ (P_1 \times P_2)\ (Q_1 \times Q_2)$, which applies ℓ_1/ℓ_2 to each component of the pair in parallel, as below.

$$\begin{aligned} &\mathit{get}\ (\ell_1 \times \ell_2)\ (s_1, s_2) = (\mathit{get}\ \ell_1\ s_1, \mathit{get}\ \ell_2\ s_2) \\ &\mathit{put}\ (\ell_1 \times \ell_2)\ ((s_1, s_2), (v_1, v_2)) = (\mathit{put}\ \ell_1\ (s_1, v_1), \mathit{put}\ \ell_2\ (s_2, v_2)) \end{aligned}$$

Thanks to the point-wise ordering, the combinator $\times$ preserves well-behavedness.

4.5 Elaborated Domains for Finer-Grained Updates

Sect. 2.4 mentioned that we can support finer-grained update descriptions (strictly partially-specified states) u_1, u_2 that specify the same possible results, i.e., the

set of partially-specified states t with $u \leq t$ is the same for $u = u_1, u_2$. The ability to treat such partially-specified states differently is one of the strengths of operation-based systems [2, 11, 20, 39]. Here we show that such control is also possible in our system. This is achieved by modifying the filter lenses in ℓ_{task}.

Domains A natural approach is to make $\mathit{filter}_{\neg\text{Done}}$/$\mathit{filter}_{\text{Apr1}}$ convert completion/postponing requests to addition requests. To make this possible, we use separate domains for filter results: DT_{OG} and DT_{DT}, which support completion and postponing requests, respectively. Let us focus on DT_{OG} as the definition of DT_{DT} is similar. We define Δ_{OG} as a set of triples (A, C, D) where $A, C \in \mathsf{Tasks}$ are sets of tasks, and $D \subseteq \mathsf{ID}$ is a set of IDs, satisfying the following conditions: $A(k) = (\mathsf{False}, \_, \_)$ for all $k \in \mathrm{dom}(A)$, $C(k) = (\mathsf{True}, \_, \_)$ for all $k \in \mathrm{dom}(C)$, and $\mathrm{dom}(A)$, $\mathrm{dom}(C)$ and D are pairwise disjoint. Intuitively, the first two conditions say that A is an addition request for visible tasks in the view and C is for invisible tasks. The third condition asserts the feasibility. Then, we define $\mathsf{DT}_{\text{OG}} \triangleq \Delta_{\text{OG}}$ with the following $\leq$ and I.

$$\frac{A \subseteq A' \quad C \subseteq C' \quad D \subseteq D'}{(A, C, D) \leq (A', C', D')} \qquad \frac{A \subseteq t \quad \mathrm{Disj}\,(\mathrm{dom}(C) \cup D, \mathrm{dom}(t))}{(A, C, D) \leq t}$$

$$\frac{A \subseteq A' \quad C \subseteq C' \quad D \subseteq D'}{(A, C, D) \in I_{(A', C', D')}} \qquad \frac{A \subseteq t}{(A, \emptyset, \emptyset) \in I_t}$$

In Sect. 2.4, A, C, and D are represented by $(+)$, $(\checkmark)$, and $(-)$ marks in the table. This DT_{OG} is duplicable with $\oplus$ defined similarly to Sect. 4.1, but we shall omit the discussion as we do not use *dup* for the domain.

Modified $\mathit{filter}_{\neg\text{Done}}$ The modification to $\mathit{filter}_{\neg\text{Done}}$ is straightforward. Now, its view domain is DT_{OG} (and thus $\mathit{filter}_{\neg\text{Done}} \in \mathsf{Lens}^{\leq}\ \mathsf{DT}\ \mathsf{DT}_{\text{OG}}$). Its behavior on strictly partially-specified states is changed: *get* and *put* now interconvert an addition request $A \uplus C$ in DT and a pair of an addition request A and a completion request C in DT_{OG}, using the fact that $A \in \mathsf{Tasks}_{\text{OG}}$ and $C \in \mathsf{Tasks}_{\text{Comp}}$.

$$\mathit{get}\ \mathit{filter}_{\neg\text{Done}}\ (A \uplus C, D) = (A, C, D) \text{ where } A \in \mathsf{Tasks}_{\text{OG}}, C \in \mathsf{Tasks}_{\text{Comp}}$$
$$\mathit{put}\ \mathit{filter}_{\neg\text{Done}}\ (\_, (A, C, D)) = (A \uplus C, D)$$

(The definition of $\mathit{filter}_{\text{Apr1}} \in \mathsf{Lens}^{\leq}\ \mathsf{DT}\ \mathsf{DT}_{\text{DT}}$ is similar). The well-behavedness of this version is proved similarly to the previous version.

5 Partial-State Lenses as an Operation-Based System

This section shows that we can encode updates as i-posets, providing a key advantage of operation-based systems: fine-grained control on backward behavior.

5.1 State-Update Pairs

The basic idea to construct an i-poset is to pair updates with their origins, inspired by the encoding of operation-based CRDTs into state-based CRDTs [45]. Let S

be a set of (proper) states and U be a poset of updates.[9] We assume that U has a merge operator $\oplus_U$ that soundly implements the join in U, i.e., $u_1 \oplus_U u_2 = u$ implies $u_1 \vee u_2 = u$. We also assume that each $u \in U$ can be interpreted as a partial function $[\![u]\!] \in S \rightharpoonup S$, which intuitively represents the application of the update. Let us define $\mathrm{ran}_s(u) = \{s' \in S \mid \exists u'.\, u \leq_U u', [\![u']\!](s) = s'\}$, which intuitively denotes the set of possible results of u when the original state is s. Then, we define an i-poset $\mathsf{G}_{S,U} = (S \cup (S \times U), \leq, I)$, where $\leq$ and I are given by:

$$\frac{u \leq_U u'}{(s,u) \leq (s,u')} \qquad \frac{}{s \leq s} \qquad \frac{s' \in \mathrm{ran}_s(u)}{(s,u) \leq s'} \qquad \frac{u \leq_U u'}{(s,u) \in I_{(s,u')}} \qquad \frac{}{s \in I_s} \qquad \frac{[\![u]\!](s) = s}{(s,u) \in I_s}$$

Observe that $\leq$ is a partial order and I is a reflexive subset of $(\leq)$.

To support a ps-initiator, we then define $(\cdot) \in \mathsf{G}_{S,U} \to S \rightharpoonup S$ as: $s \cdot \_ = s$, and $(s,u) \cdot s' = [\![u]\!](s)$ if $s = s'$. The u-consistency and u-acceptability of this $(\cdot)$ follow from the definitions of $(\cdot)$, $(\leq)$, and I.

Supporting *dup* on $\mathsf{G}_{S,U}$ requires some additional structures. We first define the merge operator $\oplus_{S,U} \in \mathsf{G}_{S,U} \times \mathsf{G}_{S,U} \rightharpoonup \mathsf{G}_{S,U}$ as below.

$$s \oplus_{S,U} s' = s \text{ if } s = s' \qquad s' \oplus_{S,U} (s,u) = (s,u) \oplus_{S,U} s' = s' \text{ if } s' \in \mathrm{ran}_s(u)$$
$$(s,u) \oplus_{S,U} (s',u') = (s, u \oplus_U u') \text{ if } s = s'$$

Let us say that $\oplus_U$ respects $\mathrm{ran}_s(-)$ if $\mathrm{ran}_s(u_1) \cap \mathrm{ran}_s(u_2) \subseteq \mathrm{ran}_s(u)$ whenever $u_1 \oplus_U u_2 = u$. (The converse inclusion follows from the definition of $\mathrm{ran}_s(-)$.)

Lemma 7. *$\mathsf{G}_{S,U}$ is duplicable with the merge operator $\oplus_{S,U}$ if the following conditions hold:* (G1) *$\oplus_U$ respects $\mathrm{ran}_s(-)$ for any s,* (G2) *$\oplus_U$ is total on $\{u' \mid u' \leq_U u\}$ for any u, and* (G3) *$\oplus_U$ is total and closed on $\{u \mid [\![u]\!](s) = s\}$ for any s.* □

5.2 Eliminating States from Partially-Specified States

Pairing a state s with an update u is inefficient as updates are typically smaller than states. Using $\mathrm{ran}(u) = \bigcup_s \mathrm{ran}_s(u)$ in the premise of $u \leq s'$ makes $\oplus_{S,U}$ unsound in general, as it violates (G1), which is also a necessary condition for $\mathsf{G}_{S,U}$ to be duplicable. However, this also means that s can be omitted if either we will not use *dup* for $\mathsf{G}_{S,U}$, or $\oplus_{S,U}$ respects $\mathrm{ran}(u) = \bigcup_s \mathrm{ran}_s(u)$. The i-posets DT and $\mathsf{DT}_{\mathsf{OG}}$ in Sects. 4.1 and 4.5 are examples of the latter kind. They deviate slightly from the recipe; their Is are stricter than the recipe, and $(A,D) \cdot t$ is total, while the recipe assumes the partiality of $[\![(A,D)]\!]$.

6 Related Work

Partially-specified states have appeared in various forms in the bidirectional transformation literature. The original lens formalization [12,13] uses Ω to represent

[9] Unlike Ahman and Uustalu [2], Diskin et al. [11], we do not use a family of indexed sets to model updates, because we focus on their merging and applications. Using indexed sets makes the latter total, but merging is an intrinsically partial operation.

unavailable sources. This use of Ω, however, can be replaced by other means, such as the *create* function (e.g., [6,19]) which computes the updated source only from the updated view. A bidirectional programming language HOBiT [37] adopts partially-specified states for their treatment of value environments. Roughly speaking, they interpret a term-in-context $\Theta \vdash e : \sigma$ as a lens between value environments of type Θ and values of type σ, where its *put* execution specifies only the values of variables that occur in e. Such value environments, crucial for interpreting compound expressions like (x, y) compositionally, can be seen as partially-specified states; compound expressions can then be seen as a microscopic form of multiple views. Their well-behavedness is compositional, partial-order based, and similar to our weak well-behavedness except for their maximality requirement, which ensures that their *get* can only involve proper states. They have no law corresponding to ps-stability because their final source is essentially discrete due to external update reflection. This difference would reflect where and how partial specifiedness is allowed: in their framework, partially-specified states are only permitted as value environments and used to interpret terms-in-context. We also note their goal is to guarantee classical well-behavedness [12,13]; they did not support view-to-view update propagation via source-sharing.

A similar idea appears in Mu et al. [40], where special tags are used to identify updated parts. One form of tags is $*v$, which denotes that the part is updated and gives it precedence in the *put* execution of their duplication lens. In this sense, their $*$-ed values are more specified than $*$-less values. They also consider tags for structural list updates: $x\,^{+}\!: xs$ and $x\,^{-}\!: xs$ mark insertion and deletion of x, respectively.[10] The general recipe (Sect. 5.1) can handle their tagged lists with a slight modification. Since the origins of updates can be obtained from their tagged lists, the tagged lists can be regarded as state-and-update pairs. The ordering is then given by the backward behavior (say, *merge*) of their list duplication, namely, $x \leq y \triangleq (\mathit{merge}\ x\ y = y)$. A caveat is that we need to distinguish a list as a proper state from that as a partially-specified state, and appropriately restrict I so that *merge* becomes total on I_s for a proper state s. The removal of tags is performed by *put* of the ps-initiator.

In the field of databases, Hegner [17] discusses laws for bidirectionality that refer to ordering. Unlike ours, his ordering intuitively represents insertions and deletions. Even for the non-order-related subpart, the laws are much stronger than the well-behavedness for classical lenses. Besides acceptability and consistency, he also imposes conditions that make *put* transparent [16]: intuitively, any updates can be undone, synchronization timing (invocation of *put*) is irrelevant (PutPut), and whether $\mathit{put}\ (s, v)$ is defined depends only on $\mathit{get}\ s$ and v. By further requiring conditions on ordering, Hegner [17] shows the uniqueness of *put* for a given *get*.

The general recipe for encoding updates in partially-specified states (Sect. 5.1) provides us a basis for direct comparisons with other operation-based frameworks, such as delta lenses [11], update-update lenses [2], and edit lenses [20]. A crucial difference is that, in ours, both proper states and strictly partially-specified states

[10] Originally, the symbols $\oplus$ and $\ominus$ are used [40]. We use different symbols to avoid confusion with our use of $\oplus$.

can be mixed and compared, while they are separated in these frameworks. This is key to having ps-initiators, which purposely mix the two sorts of states (see the extended version [31] for more details, where it shows that a naive extension of delta lenses does not work). Update-update lenses [2] are a variant of delta lenses that only translate updates backward. This design rules out domains that consist only of strictly partially-specified states, such as CRDTs [3, 45], and weakens the update preservation guarantee. For example, the difference between completion and deletion on the ONGOING view in Sect. 2.4 may be ignored by their lenses, as their difference cannot be observed by comparing with proper states. Edit lenses [20] allow lenses to have internal states, generalizing delta lenses. While useful for both efficiency and expressiveness, such internal states would further complicate the discussion of laws in our setting.

These operation-based frameworks require update translation to preserve update compositions—a form of PUTPUT. Unlike the state-based case, this requirement does not make lenses impractical. In our setting, however, the preservation of composition in their sense is less meaningful, because the updated view may differ from the view of the updated source. We might express our operations in terms of compositions by treating $u_1 \leq u_2$ as $u_2 = u_1 \fatsemi u$ for some u, and defining the merge operator of updates by its pushout. Although this view would give a form of PUTPUTin our setting, it would complicate the formalization.

7 Conclusion

We introduced *partial-state lenses*, extending the classical lenses [12, 13] with partially-specified states. Partially-specified states are partially ordered, providing clear notions of update preservation and merging. This is crucial for handling multiple views that share a source. A key advantage of this framework is its support for compositional reasoning of order-aware well-behavedness, which is strong enough to guarantee the preservation of the user's updates.

A future direction is to discuss more primitives and combinators for various datatypes, such as lists and finite maps. For lists, we would base our work on Mu et al. [40]'s lists, as mentioned in Sect. 6. Another direction is to design a frontend system so that users can express their intentions as partially-specified states. An approach would be to connect the proposed framework to an Elm-like architecture, providing a GUI to issue updates as partially-specified states. A third direction is to provide a better programming interface, similarly to HOBiT [37] and its embedded version [29, Section 6] for the classical lenses [12, 13].

Acknowledgments. We thank Zihang Ye, who inspired the direction of this research. This work was partially supported by JSPS KAKENHI Grant Numbers JP20H04161, JP23K20379, JP22H03562 and JP23K24818, and EPSRC Grant *EXHIBIT: Expressive High-Level Languages for Bidirectional Transformations* (EP/T008911/1).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Data Availability The accompanying artifact is available at https://doi.org/10.5281/zenodo.18309769. It contains a prototype implementation of partial-state lenses in Haskell for reproducing the behavior of the lenses discussed in Sects. 2 and 4, and a mechanized formalization of definitions and statements in this paper. Their source code repositories are hosted at https://github.com/kztk-m/ps-lenses-hs and https://github.com/kztk-m/ps-lenses-agda.

References

1. Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Reflections on monadic lenses. In: Lindley, S., McBride, C., Trinder, P.W., Sannella, D. (eds.) A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 9600, pp. 1–31. Springer (2016). https://doi.org/10.1007/978-3-319-30936-1_1, https://doi.org/10.1007/978-3-319-30936-1_1
2. Ahman, D., Uustalu, T.: Taking updates seriously. In: Eramo, R., Johnson, M. (eds.) Proceedings of the 6th International Workshop on Bidirectional Transformations co-located with The European Joint Conferences on Theory and Practice of Software, BX@ETAPS 2017, Uppsala, Sweden, April 29, 2017. CEUR Workshop Proceedings, vol. 1827, pp. 59–73. CEUR-WS.org (2017), https://ceur-ws.org/Vol-1827/paper11.pdf
3. Almeida, P.S., Shoker, A., Baquero, C.: Delta state replicated data types. J. Parallel Distributed Comput. **111**, 162–173 (2018). https://doi.org/10.1016/J.JPDC.2017.08.003, https://doi.org/10.1016/j.jpdc.2017.08.003
4. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. **6**(4), 557–575 (1981). https://doi.org/10.1145/319628.319634, https://doi.org/10.1145/319628.319634
5. Barbosa, D.M.J., Cretin, J., Foster, N., Greenberg, M., Pierce, B.C.: Matching lenses: alignment and view update. In: Hudak, P., Weirich, S. (eds.) Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010. pp. 193–204. ACM (2010). https://doi.org/10.1145/1863543.1863572, https://doi.org/10.1145/1863543.1863572
6. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 407–419. ACM (2008). https://doi.org/10.1145/1328438.1328487, https://doi.org/10.1145/1328438.1328487
7. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: Vansummeren, S. (ed.) Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA. pp. 338–347. ACM (2006). https://doi.org/10.1145/1142351.1142399, https://doi.org/10.1145/1142351.1142399
8. Boisseau, G.: String diagrams for optics. In: Ariola, Z.M. (ed.) 5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference). LIPIcs, vol. 167, pp. 17:1–17:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPICS.FSCD.2020.17, https://doi.org/10.4230/LIPIcs.FSCD.2020.17

9. Chin, W.: Towards an automated tupling strategy. In: Schmidt, D.A. (ed.) Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93, Copenhagen, Denmark, June 14-16, 1993. pp. 119–132. ACM (1993). https://doi.org/10.1145/154630.154643, https://doi.org/10.1145/154630.154643
10. Cunha, J., Fernandes, J.P., Mendes, J., Pacheco, H., Saraiva, J.: Bidirectional transformation of model-driven spreadsheets. In: Hu, Z., de Lara, J. (eds.) Theory and Practice of Model Transformations - 5th International Conference, ICMT@TOOLS 2012, Prague, Czech Republic, May 28-29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7307, pp. 105–120. Springer (2012). https://doi.org/10.1007/978-3-642-30476-7_7, https://doi.org/10.1007/978-3-642-30476-7_7
11. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. J. Object Technol. **10**, 6: 1–25 (2011). https://doi.org/10.5381/JOT.2011.10.1.A6, https://doi.org/10.5381/jot.2011.10.1.a6
12. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. pp. 233–246. ACM (2005). https://doi.org/10.1145/1040305.1040325, https://doi.org/10.1145/1040305.1040325
13. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3) (2007)
14. Foster, J.N., Pilkiewicz, A., Pierce, B.C.: Quotient lenses. In: Hook, J., Thiemann, P. (eds.) Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008. pp. 383–396. ACM (2008). https://doi.org/10.1145/1411204.1411257, https://doi.org/10.1145/1411204.1411257
15. Goldstein, H., Frohlich, S., Wang, M., Pierce, B.C.: Reflecting on random generation. Proc. ACM Program. Lang. **7**(ICFP), 322–355 (2023). https://doi.org/10.1145/3607842, https://doi.org/10.1145/3607842
16. Hegner, S.J.: Foundations of canonical update support for closed database views. In: Abiteboul, S., Kanellakis, P.C. (eds.) ICDT'90, Third International Conference on Database Theory, Paris, France, December 12-14, 1990, Proceedings. Lecture Notes in Computer Science, vol. 470, pp. 422–436. Springer (1990). https://doi.org/10.1007/3-540-53507-1_93, https://doi.org/10.1007/3-540-53507-1_93
17. Hegner, S.J.: An order-based theory of updates for closed database views. Ann. Math. Artif. Intell. **40**(1-2), 63–125 (2004). https://doi.org/10.1023/A:1026158013113, https://doi.org/10.1023/A:1026158013113
18. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing graph transformations. In: Hudak, P., Weirich, S. (eds.) Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010. pp. 205–216. ACM (2010). https://doi.org/10.1145/1863543.1863573, https://doi.org/10.1145/1863543.1863573
19. Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 371–384. ACM (2011). https://doi.org/10.1145/1926385.1926428, https://doi.org/10.1145/1926385.1926428

20. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. pp. 495–508. ACM (2012). https://doi.org/10.1145/2103656.2103715, https://doi.org/10.1145/2103656.2103715
21. Horn, R., Perera, R., Cheney, J.: Incremental relational lenses. Proc. ACM Program. Lang. **2**(ICFP), 74:1–74:30 (2018). https://doi.org/10.1145/3236769, https://doi.org/10.1145/3236769
22. Hu, Z., Iwasaki, H., Takeichi, M., Takano, A.: Tupling calculation eliminates multiple data traversals. In: Peyton Jones, S.L., Tofte, M., Berman, A.M. (eds.) Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997. pp. 164–175. ACM (1997). https://doi.org/10.1145/258948.258964, https://doi.org/10.1145/258948.258964
23. Hu, Z., Ko, H.: Principles and practice of bidirectional programming in BiGUL. In: Gibbons, J., Stevens, P. (eds.) Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures. Lecture Notes in Computer Science, vol. 9715, pp. 100–150. Springer (2016). https://doi.org/10.1007/978-3-319-79108-1_4, https://doi.org/10.1007/978-3-319-79108-1_4
24. Hu, Z., Mu, S., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. In: Heintze, N., Sestoft, P. (eds.) Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2004, Verona, Italy, August 24-25, 2004. pp. 178–189. ACM (2004). https://doi.org/10.1145/1014007.1014025, https://doi.org/10.1145/1014007.1014025
25. Keller, A.M.: Comments on Bancilhon and Spyratos' "update semantics and relational views". ACM Trans. Database Syst. **12**(3), 521–523 (1987). https://doi.org/10.1145/27629.214296, https://doi.org/10.1145/27629.214296
26. Ko, H., Zan, T., Hu, Z.: BiGUL: a formally verified core language for putback-based bidirectional programming. In: Erwig, M., Rompf, T. (eds.) Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 61–72. ACM (2016). https://doi.org/10.1145/2847538.2847544, http://doi.acm.org/10.1145/2847538.2847544
27. van Laarhoven, T.: Cps based functional references. blog post: https://www.twanvl.nl/blog/haskell/cps-functional-references (2009), visited 2024-10-08
28. Macedo, N., Pacheco, H., Sousa, N.R., Cunha, A.: Bidirectional spreadsheet formulas. In: Fleming, S.D., Fish, A., Scaffidi, C. (eds.) IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014. pp. 161–168. IEEE Computer Society (2014). https://doi.org/10.1109/VLHCC.2014.6883041, https://doi.org/10.1109/VLHCC.2014.6883041
29. Matsuda, K., Frohlich, S., Wang, M., Wu, N.: Embedding by unembedding. Proc. ACM Program. Lang. **7**(ICFP), 1–47 (2023). https://doi.org/10.1145/3607830, https://doi.org/10.1145/3607830
30. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007. pp. 47–58. ACM (2007). https://doi.org/10.1145/1291151.1291162, https://doi.org/10.1145/1291151.1291162

31. Matsuda, K., Nguyen, M., Wang, M.: Lenses for partially-specified states (extended version) (2026), https://arxiv.org/abs/2601.04573
32. Matsuda, K., Wang, M.: FliPpr: A prettier invertible printing system. In: Felleisen, M., Gardner, P. (eds.) ESOP. Lecture Notes in Computer Science, vol. 7792, pp. 101–120. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_6, https://doi.org/10.1007/978-3-642-37036-6_6
33. Matsuda, K., Wang, M.: Applicative bidirectional programming with lenses. In: Fisher, K., Reppy, J.H. (eds.) ICFP. pp. 62–74. ACM (2015). https://doi.org/10.1145/2784731.2784750, http://doi.acm.org/10.1145/2784731.2784750
34. Matsuda, K., Wang, M.: "Bidirectionalization for free" for monomorphic transformations. Sci. Comput. Program. **111**, 79–109 (2015). https://doi.org/10.1016/j.scico.2014.07.008, https://doi.org/10.1016/j.scico.2014.07.008
35. Matsuda, K., Wang, M.: Applicative bidirectional programming: Mixing lenses and semantic bidirectionalization. J. Funct. Program. **28**, e15 (2018). https://doi.org/10.1017/S0956796818000096, https://doi.org/10.1017/S0956796818000096
36. Matsuda, K., Wang, M.: Embedding invertible languages with binders: a case of the FliPpr language. In: Wu, N. (ed.) Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018. pp. 158–171. ACM (2018). https://doi.org/10.1145/3242744.3242758, https://doi.org/10.1145/3242744.3242758
37. Matsuda, K., Wang, M.: HOBiT: Programming lenses without using lens combinators. In: Ahmed, A. (ed.) ESOP. Lecture Notes in Computer Science, vol. 10801, pp. 31–59. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_2, https://doi.org/10.1007/978-3-319-89884-1_2
38. Mayer, M., Kuncak, V., Chugh, R.: Bidirectional evaluation with direct manipulation. Proc. ACM Program. Lang. **2**(OOPSLA), 127:1–127:28 (2018). https://doi.org/10.1145/3276497, https://doi.org/10.1145/3276497
39. Meertens, L.: Designing constraint maintainers for user interaction (1998), available on: https://www.kestrel.edu/people/meertens/pub/dcm.pdf
40. Mu, S., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: Chin, W. (ed.) Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings. Lecture Notes in Computer Science, vol. 3302, pp. 2–20. Springer (2004). https://doi.org/10.1007/978-3-540-30477-7_2, https://doi.org/10.1007/978-3-540-30477-7_2
41. O'Connor, R.: Functor is to lens as applicative is to biplate: Introducing multiplate. CoRR **abs/1103.2841** (2011), http://arxiv.org/abs/1103.2841, accepted in WGP '11, but not included in its proceedings
42. Pacheco, H., Cunha, A.: Generic point-free lenses. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) Mathematics of Program Construction, 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6120, pp. 331–352. Springer (2010). https://doi.org/10.1007/978-3-642-13321-3_19, https://doi.org/10.1007/978-3-642-13321-3_19
43. Pickering, M., Gibbons, J., Wu, N.: Profunctor optics: Modular data accessors. Art Sci. Eng. Program. **1**(2), 7 (2017). https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2017/1/7, https://doi.org/10.22152/programming-journal.org/2017/1/7
44. Riley, M.: Categories of optics (2018), https://arxiv.org/abs/1809.00738
45. Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France,

October 10-12, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6976, pp. 386–400. Springer (2011). https://doi.org/10.1007/978-3-642-24550-3_29, https://doi.org/10.1007/978-3-642-24550-3_29
46. Stevens, P.: A landscape of bidirectional model transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE. Lecture Notes in Computer Science, vol. 5235, pp. 408–424. Springer (2008). https://doi.org/10.1007/978-3-540-88643-3_10, https://doi.org/10.1007/978-3-540-88643-3_10
47. Takeichi, M.: Configuring bidirectional programs with functions. Presented at IFL 2009: International Symposium/Workshop on Implementation and Application of Functional Languages (2009), available from: https://takeichimasato.net/attachments/XFun0.pdf
48. Voigtländer, J.: Bidirectionalization for free! (pearl). In: Shao, Z., Pierce, B.C. (eds.) Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. pp. 165–176. ACM (2009). https://doi.org/10.1145/1480881.1480904, https://doi.org/10.1145/1480881.1480904
49. Voigtländer, J., Hu, Z., Matsuda, K., Wang, M.: Combining syntactic and semantic bidirectionalization. In: Hudak, P., Weirich, S. (eds.) Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010. pp. 181–192. ACM (2010). https://doi.org/10.1145/1863543.1863571, https://doi.org/10.1145/1863543.1863571
50. Voigtländer, J., Hu, Z., Matsuda, K., Wang, M.: Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. J. Funct. Program. **23**(5), 515–551 (2013). https://doi.org/10.1017/S0956796813000130, https://doi.org/10.1017/S0956796813000130
51. Williams, J., Gordon, A.D.: Where-provenance for bidirectional editing in spreadsheets. J. Comput. Lang. **73**, 101155 (2022). https://doi.org/10.1016/J.COLA.2022.101155, https://doi.org/10.1016/j.cola.2022.101155
52. Xie, R., Schrijvers, T., Hu, Z.: Biparsers: Exact printing for data synchronisation. Proc. ACM Program. Lang. **9**(POPL), 2205–2231 (2025). https://doi.org/10.1145/3704910, https://doi.org/10.1145/3704910
53. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA. pp. 164–173. ACM (2007). https://doi.org/10.1145/1321631.1321657, https://doi.org/10.1145/1321631.1321657
54. Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., Montrieux, L.: Maintaining invariant traceability through bidirectional transformations. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. pp. 540–550. IEEE Computer Society (2012). https://doi.org/10.1109/ICSE.2012.6227162, https://doi.org/10.1109/ICSE.2012.6227162
55. Zhang, X., Hu, Z.: Towards bidirectional live programming for incomplete programs. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022. pp. 2154–2164. ACM (2022). https://doi.org/10.1145/3510003.3510195, https://doi.org/10.1145/3510003.3510195
56. Zhang, X., Xie, R., Guo, G., He, X., Zan, T., Hu, Z.: Fusing direct manipulations into functional programs. Proc. ACM Program. Lang. **8**(POPL), 1211–1238 (2024). https://doi.org/10.1145/3632883, https://doi.org/10.1145/3632883

57. Zhu, Z., Ko, H., Martins, P., Saraiva, J., Hu, Z.: BiYacc: Roll your parser and reflective printer into one. In: Cunha, A., Kindler, E. (eds.) Proceedings of the 4th International Workshop on Bidirectional Transformations co-located with Software Technologies: Applications and Foundations, STAF 2015, L'Aquila, Italy, July 24, 2015. CEUR Workshop Proceedings, vol. 1396, pp. 43–50. CEUR-WS.org (2015), https://ceur-ws.org/Vol-1396/p43-zhu.pdf
58. Zhu, Z., Zhang, Y., Ko, H., Martins, P., Saraiva, J., Hu, Z.: Parsing and reflective printing, bidirectionally. In: van der Storm, T., Balland, E., Varró, D. (eds.) Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016. pp. 2–14. ACM (2016). https://doi.org/10.1145/2997364, http://dl.acm.org/citation.cfm?id=2997369