	Alignment	Symmetry	Performance	Syntax
asymmetric delta lenses	✓			✓
symmetric delta lenses	✓	✓		
comma category lenses	✓			
group-based lenses	✓			
matching lenses	✓			✓
annotation-based lenses	✓			✓
constraint maintainers	✓	✓		✓
symmetric lenses		✓		✓
edit lenses	✓	✓	✓	✓

Table 1.1: Feature coverage for various approaches to bidirectional programming

bidirectional transformation problem, as the design of bidirectional building blocks can be separated from the process of gluing the blocks together into particular useful transformations.

The term "lens" is a broad term encompassing a large family of related language-based approaches to the bidirectional transformation problem. In §1.1, we will introduce one of the earliest language-based approaches, asymmetric lenses, as a way to ground our ongoing discussion of the features that make bidirectional programming attractive and practical. We will identify four key challenges in lens design: alignment (the ability to represent data evolution precisely, §1.2), symmetry (no more restrictions on one piece of connected data than on the other, §1.3), performance (handling data in an incremental way, §1.4), and syntax (the existence of example transformations, §1.5). Table 1.1 compares extant lens frameworks with respect to these four key features, with a focus on frameworks which aim at dealing with the problem of alignment. We will discuss this figure in depth in Chapter 5; for now, it suffices to observe that the edit lens framework as described in this document is the first framework to support all four features.

1.1 Asymmetric Lenses

One well-studied approach to bidirectional programming is the framework of *asymmetric*, *state-based* lenses. A thorough review of this work is available elsewhere [17], so we will give only a brief introduction to the core concepts. Suppose there are two repositories; one repository stores a piece of data represented by an element of the set

dence between two repositories by propagating edits in both directions". A full-blown synchronization tool would also include, at a minimum, some mechanism for dealing with conflicts between disconnected edits to the two structures, which is outside the scope of this document. Note, though, that we will go beyond most existing synchronization tools in allowing the repositories to be structured differently and to share only a part of their information.

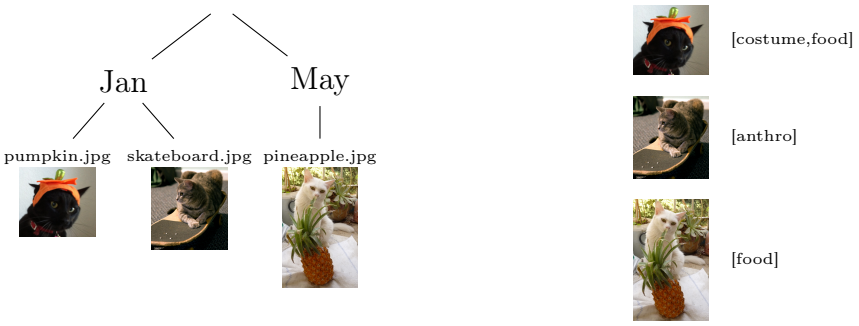


Figure 1.7: A whimsical symmetric synchronization scenario

bidirectional programs. To see why, we will introduce a bidirectional transformation which is most naturally modeled using composition.

The whimsical situation shown in Figure 1.7 involves a web server, which must keep a file system storing pictures of cats synchronized with a user-modifiable web page (modeled here as a list of cat pictures with descriptive tags).[5] One natural approach to implementing this transformation is pictured in Figure 1.8. First, we separately implement two constraint maintainers: a *flatten* maintainer that flattens trees to lists by extracting the leaves, and a *relabel* maintainer that describes the connection between a single leaf in our original tree and a single list entry in our final list. We would then like to run these maintainers back-to-back; that is, we would like a sequential composition operator $-;-$ with a typing rule like:

$$\frac{k \in A \overset{c}{\leftrightarrow} B \qquad \ell \in B \overset{c}{\leftrightarrow} C}{k;\ell \in A \overset{c}{\leftrightarrow} C}$$

Unfortunately, implementing this combinator is not possible: we must design the $(k;\ell).get \in A \times C \to C$ component using the components $k.get$, $k.put$, $\ell.get$, and $\ell.put$, all of which require a B as input. It is certainly possible to build a constraint maintainer which has the desired behavior wholesale, but this involves writing both constraint maintenance functions and proving that they are consistent with each other—the exact task we set out to avoid by designing a language. Alternately, one can step a little bit outside the constraint maintainer framework by keeping a copy of the "intermediate" repository around somewhere and running the constraint maintainers in sequence on each update. Making this choice, however, leads one to immediately ask how to model such maintainer chains and what behavioral guarantees one can expect!

So an ideal model would capture the behavior of "sequencing", retain a symmetric presentation, and allow each repository to retain information not available in the other.

[5]Pictures used with permission [4, 11, 53].

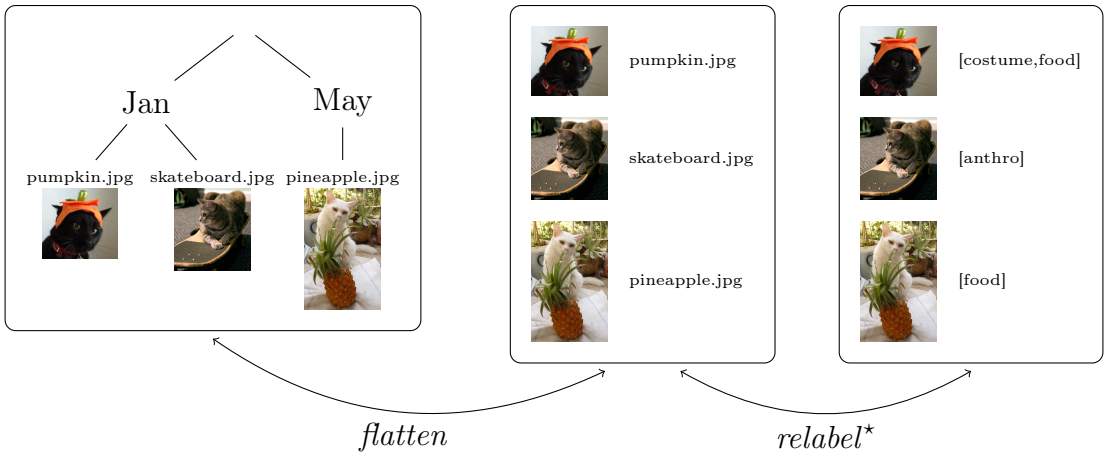


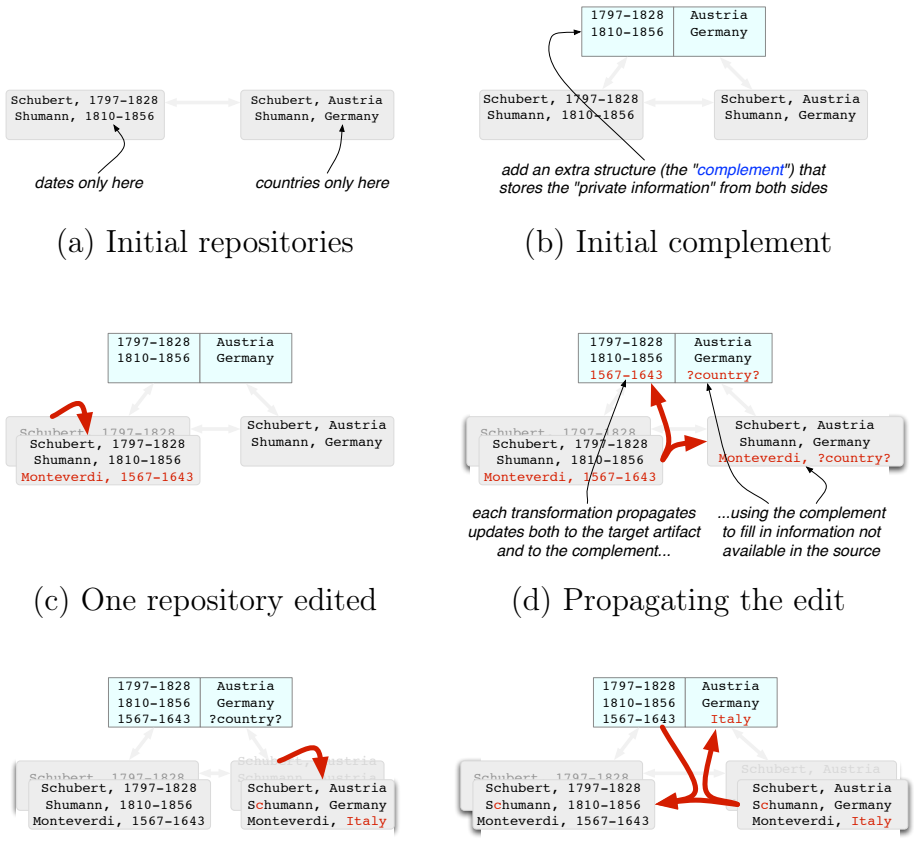
Figure 1.8: Adding an intermediate structure can improve modularity

1.4 Performance

Real-world synchronization tools inevitably address a third concern: performance. Typical repositories are large objects; consequently, there can be significant time or memory costs associated with processing the data in a repository or transmitting a repository across a network. For existing file system synchronization tools like rsync, DropBox, and Unison; collaborative document editing tools like Apache Wave and Google Docs; and revision control systems like CVS, SVN, darcs, mercurial, and git [5, 8, 12, 15, 18, 19, 42, 43, 45, 49], network speed is a significant bottleneck. For these tools, where little computation on the repositories themselves is required, the relatively simple *delta compression* technique, which involves noting what has changed since a previous run of the tool, provides a serious network transmission performance boost. For bidirectional transformations, where repositories must be not just copied but transformed, computation time or memory usage may also be concerns. Extending the use of delta compression to address processing speed and memory requirements involves, in part, showing that the computations of interest can be performed solely by inspecting the deltas—that is, without decompressing and traversing the original repositories. Since it seems likely that a practical tool will need to avoid incurring high resource usage, a theory that faithfully models a successful tool should therefore model not just repository states but also repository edits, edit transformations, and the connection between edits and repository states.

1.5 Syntax

One of the outstanding features of the body of asymmetric, state-based lens work and its closest variants is the devotion to retaining a large collection of lenses and



(a) Initial repositories

(b) Initial complement

(c) One repository edited

(d) Propagating the edit

(e) Second repository is edited

(f) This change is propagated

Figure 2.1: Behavior of a symmetric lens

the *put* reads the C_X part and writes the C_Y part. Lastly, now that everything is symmetric, the *get* / *put* distinction is not helpful, so we rename the functions to *putr* and *putl*. This brings us to our core definition.

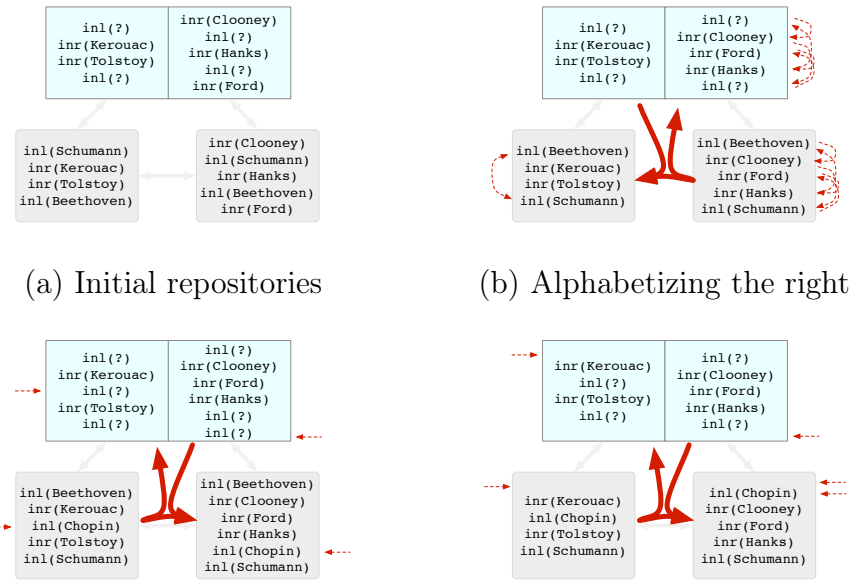
2.1.1 Definition [Symmetric lens]: A lens ℓ from X to Y (written $\ell \in X \leftrightarrow Y$) has three parts: a set of complements C, a distinguished element $missing \in C$, and two functions

$$\begin{aligned} putr &\in X \times C \to Y \times C \\ putl &\in Y \times C \to X \times C \end{aligned}$$

satisfying the following round-tripping laws:

$$\frac{putr(x,c) = (y,c')}{putl(y,c') = (x,c')} \qquad \text{(PutRL)}$$

$$\frac{putl(y,c) = (x,c')}{putr(x,c') = (y,c')} \qquad \text{(PutLR)}$$



(a) Initial repositories (b) Alphabetizing the right

(c) Inserting Chopin on the left (d) Deleting Beethoven from the left

Figure 2.2: Synchronizing lists of sums

Abstractly, to achieve this behavior we need to define a lens *comp* between $(X+Y)^\star$ and $(X+Z)^\star$. To do this, it is convenient to first define a lens that connects $(X+Y)^\star$ and $X^\star \times Y^\star$; call this lens *partition*. The complement of the *partition* is a list of booleans telling whether the corresponding element of the left list is an X or a Y. The *putr* function is fairly simple: we separate the $(X+Y)$ list into X and Y lists by checking the tag of each element, and set the complement to exactly match the tags. For example:

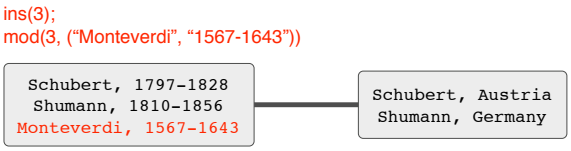
$$putr(\langle \mathsf{inl}\ a, \mathsf{inl}\ b, \mathsf{inr}\ 1\rangle, c) = ((\langle a, b\rangle, \langle 1\rangle), \langle \mathsf{false}, \mathsf{false}, \mathsf{true}\rangle)$$
$$putr(\langle \mathsf{inl}\ a, \mathsf{inr}\ 1, \mathsf{inl}\ b\rangle, c) = ((\langle a, b\rangle, \langle 1\rangle), \langle \mathsf{false}, \mathsf{true}, \mathsf{false}\rangle)$$

These examples demonstrate that *putr* ignores the complement entirely, fabricating a completely new one from its input. The *putl* function, on the other hand, relies entirely on the complement for its ordering information. When there are extra entries (not accounted for by the complement), it adds them at the end. Consider taking the output of the second *putr* above and adding c to the X list and 2 to the Y list:

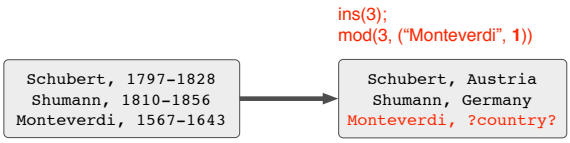
$$\begin{aligned} &putl((\langle a, b, c\rangle, \langle 1, 2\rangle), \langle \mathsf{false}, \mathsf{true}, \mathsf{false}\rangle) = \\ &\quad (\langle \mathsf{inl}\ a, \mathsf{inr}\ 1, \mathsf{inl}\ b, \mathsf{inl}\ c, \mathsf{inr}\ 2\rangle, \\ &\quad \langle \mathsf{false}, \mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{true}\rangle) \end{aligned}$$

The *putl* fills in as much of the beginning of the list as it can, using the complement to indicate whether to draw elements from $X^\star$ or from $Y^\star$. (How the remaining X and Y elements are interleaved is a free choice, not specified by the lens laws, since this case only arises when we are *not* in a round-tripping situation. The strategy shown

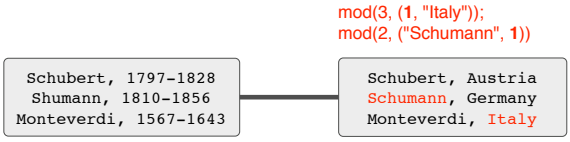
(a) initial repositories



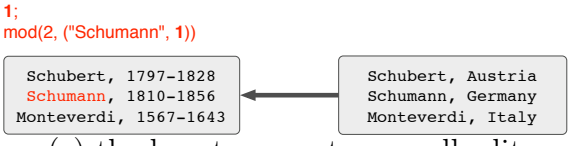
(b) a new composer is added to one repository



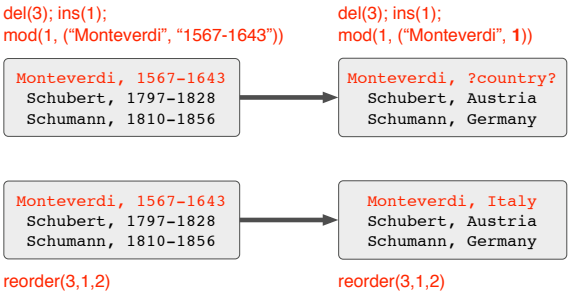
(c) the lens adds the new composer to the other repository



(d) the curator makes some corrections

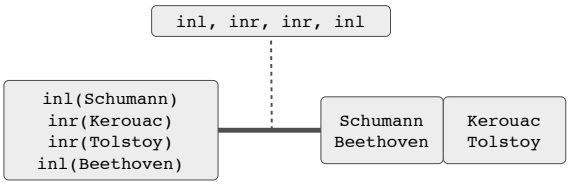


(e) the lens transports a small edit

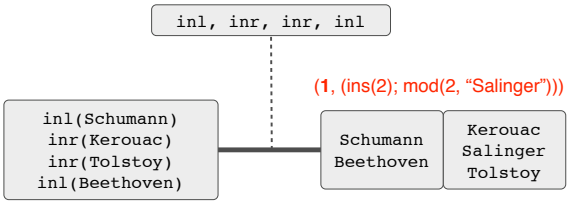


(f) two different edits with the same effect on the left

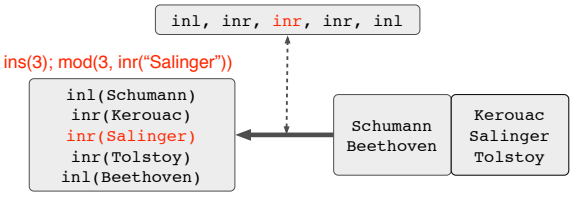
Figure 3.1: A simple (complement-less) edit lens in action.



(a) the initial repositories: a tagged list of composers and authors on the left; a pair of lists on the right; a complement storing just the tags

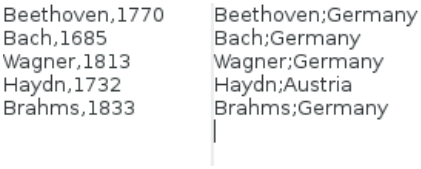


(b) an element is added to one of the partitions



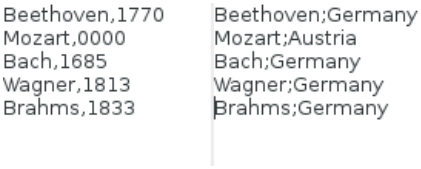
(c) the complement tells how to translate the index

Figure 3.2: A lens with complement.

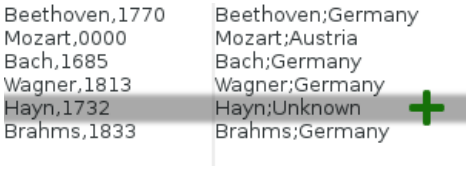


(a) An initial pair of databases in two text editing panes.

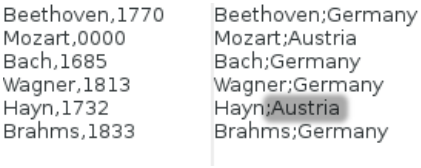
(b) Inserting Mozart on the right introduces default data on the left.



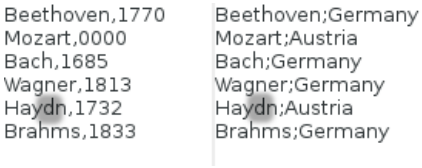
(c) Deleting Haydn from one side is reflected to the other automatically.



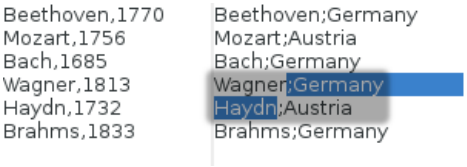
(d) A default country is used for the new row on the right.



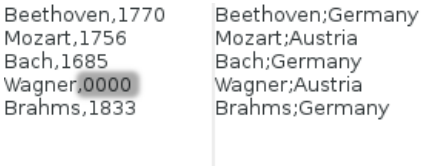
(e) Correcting Haydn's country has no effect on the birth year list...



(f) ...but correcting the spelling of Haydn in either list corrects both.



(g) More bizarre edits, like this deletion that spans records...



(h) ...reset the alignment, but only for the affected records.

Figure 4.1: A demonstration use of the prototype, using the composers lens

Chapter 5

Related Work

Recent years have seen a large body of work on bidirectional transformations in general, and frameworks based on the language-based approach embodied by lenses in particular. A great deal of the work is motivated by the apparent difficulty of the alignment problem discussed in §1.2. Table 5.1 gives a summary of the most closely related work in the area. The first four columns indicate whether the approach addresses alignment, symmetry, performance, and syntax concerns, while the final columns gives a pointer to a section with more in-depth discussion of the approach.

Asymmetric delta lenses and group-based lenses are extensions of asymmetric, state-based lenses which replace all or most of the repository data that a lens consumes or produces with edit information, instead. Symmetric delta lenses extend asymmetric delta lenses with some significant additional machinery for handling complement information, very similarly to the way our symmetric lenses generalize asymmetric, state-based lenses. The two variants of delta lenses are predicated on a model of edits which includes information about the repositories themselves; this makes it difficult to guarantee that the lenses are not traversing the repositories and causing performance problems. Additionally, the body of work on symmetric delta lenses does not

	Align.	Symm.	Perf.	Synt.	Disc.
asymmetric delta lenses	✓			✓	§5.1.1
symmetric delta lenses	✓	✓			§5.1.2
comma category lenses	✓				§5.2
group-based lenses	✓				§5.3
matching lenses	✓			✓	§5.4
annotation-based lenses	✓			✓	§5.5
constraint maintainers	✓	✓		✓	§5.6
symmetric lenses		✓		✓	Chap. 2
edit lenses	✓	✓	✓	✓	Chap. 3

Table 5.1: Feature coverage for various alternatives to edit lenses

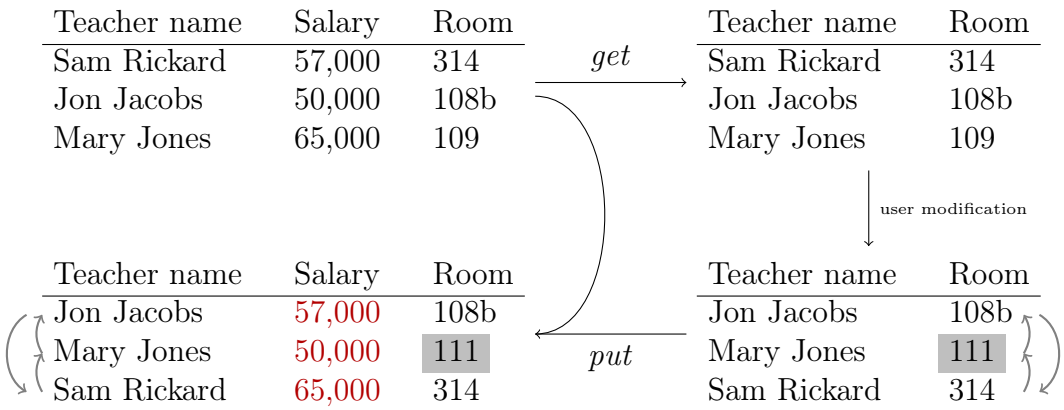


Figure 5.2: An easily fixed misalignment

5.4 Matching lenses

Dictionary lenses [9] and their sequel, matching lenses [7], are also motivated by the alignment problems discussed above. We will consider a variation of our motivating example from Chapter 1 which showcases a particularly annoying example of bad alignment—annoying both because it is a common scenario and because it seems especially clear how to get the right answer. Figure 5.2 shows again the bad behavior of positional alignment. Gray annotations mark changes with respect to a previous version of a given repository. The salary column of the updated source repository is marked in red because it has been misaligned with the updated view: the names have been shuffled, but the salaries have not.

The observation of dictionary lenses is that the teacher names in the view repository act somewhat like a key: the reordering that the user did can be recovered by comparing the order of names before and after the modification. Experience with lens programming shows that the existence of a key is fairly common, so merely giving the programmer the ability to specify which parts of the data correspond to keys can improve the *put* behavior in a wide range of applications. However, there is an unfortunate behavioral regression: with positional alignment, changing a key is handled gracefully, but with a dictionary lens, a changed key results in a loss of any associated information. Figure 5.3 gives an example of a dictionary lens resetting a salary that a plain lens would preserve. The observation here is that simple key equality is too strict. Matching lenses relax this restriction; they parameterize lenses by an alignment strategy—which can do arbitrary computation—that computes how chunks of the old and new copies of the repository correspond. Several heuristics that satisfy the interface of an alignment strategy are given, for example, for computing the least-cost alignment according to some function that computes the cost of a single-chunk change.

Matching lenses give a concrete way to separate alignment discovery from update

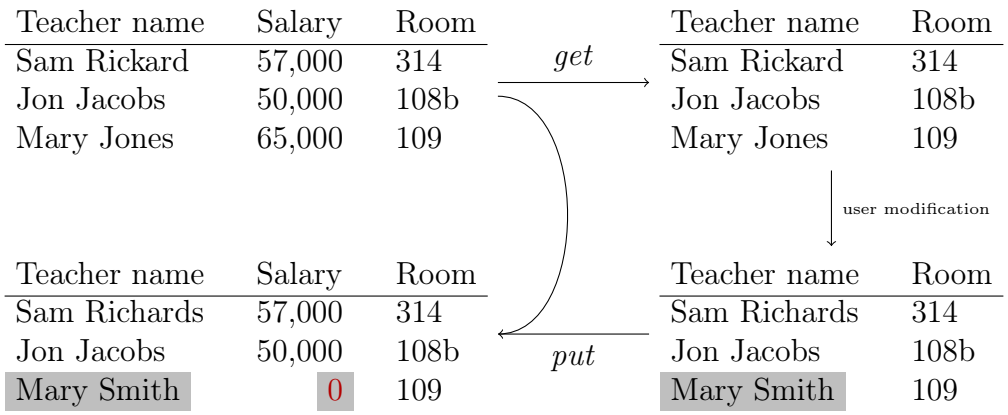


Figure 5.3: With dictionary lenses, changing a key causes information loss

propagation, and propose several promising discovery heuristics. There is also an implementation available for a string-based data model.

The basic model of matching lenses formalizes a framework for container mapping and restructuring lenses: the structure of the source and view containers need not be identical, but there must be an identical set of positions (and the connection between the positions in the source and the positions in the view must be the trivial one—that is, no reordering). They show how to extend the basic model to allow the contained values to have different types, to allow reordering, and to allow the contained values to themselves be containers. The framework of the basic model of matching lenses is already complicated; by the time it is extended in this way, the machinery is quite baroque. By comparison, the basic formalism of edit lenses can be summarized quite compactly, and is nevertheless flexible enough to accommodate all the extensions proposed. Additionally, edit lenses support a more flexible array of container operations, and in particular may be used to define lenses between structures with differing numbers of holes.

5.5 Annotation-based delta lenses

A weakness of our approach is that the lack of categorical products indicates that we cannot duplicate information during our transformations. For some applications, this is a critical feature, and allowing this requires different fundamentals. Work on annotation-based delta lenses addresses this need [24, 25, 40]. Their foundations are fundamentally symmetric; however, the way they propose using it is essentially asymmetric. Besides that, there are two key differences between their development and ours. First, their data model is ordered, node-labeled trees, and rather than separating edits from the data, they merge them: edits are represented by annotating the trees with insertion, deletion, and modification markings. Reordering is not con-

	Alignment	Symmetry	Performance	Syntax
asymm. δ	explicit alignments	not a goal	edits include repositories	via alternate framework
symm. δ	edits	yes, but equiv. not explored	edits include repositories	alternate frameworks not instantiated
comma category	inclusion morphisms	no	edits include repositories	bespoke examples
group-based	edits	no	possibly, but unexplored	not a goal
matching	mapping from holes to holes	no	repository and alignment information both processed	variants of most AS-lens combinators
annotated	insertion, deletion, modification markers	no	alignment information includes repository	includes $diag \in X \leftrightarrow X \times X$
const. maint.	uninterpreted edits	yes; does not require equiv.	no; all edits relative to *init*	many primitives, but no composition
symm. state	very bad	yes; requires equivalence	no	mostly domain agnostic
edit lenses	edits	yes; requires equivalence	small edits support incremental operation	most standard lenses, and container map

Table 6.1: Feature coverage for various lens frameworks, including the two proposed by our work (green means satisfies the objective, red indicates some shortcomings)