Chapter 1

Related Work

1.1 Graph-based delta lenses

There is a closely related line of work focused on designing edit-based lenses which begins with much the same motivation our work does [12, 13, 20, 32]. They arrive at a slightly different point in the design space compared to us, with a primary difference being their treatment of edits. For them, edits are typed—with edit type $x \to x'$ classifying edits that can be applied to value x and result in value x'—and edit application is total. Before we investigate their definitions of asymmetric and symmetric delta lens, let us review their model of edits in detail. We will begin with a few standard definitions to put some notation in place. Whenever possible, we will pun notation between graphs and categories; after all, a graph with suitable extra structure *is* a category.

1.1.1 Definition: A *graph* G is a quadruple $\langle G_0, G_1, \mathsf{dom}, \mathsf{cod}\rangle$ consisting of a set of nodes G_0, a set of edges G_1, and two functions $\mathsf{dom}, \mathsf{cod} \in G_1 \to G_0$ giving the domain and codomain of each edge. We will write $e : v \to v'$ as shorthand for the assertion that $e \in G_1$, that $\mathsf{dom}(e) = v$, and that $\mathsf{cod}(e) = v'$. If the directionality of the edge is uninteresting, we will write $e : v \mathrel{—\!\!—} v'$ to mean either $e : v \to v'$ or $e : v' \to v$.

Below, we will use graphs to model edits: nodes of the graph will correspond to repository states, and an edge $\mathrm{d}x : x \to x'$ will correspond to an edit $\mathrm{d}x$ which, when applied to state x, results in state x'. As in our development, it is natural to impose a little bit of structure on edges, such as the existence of a "do-nothing" edit and the ability to combine two edits into one. We introduce these restrictions separately so that we may talk about lenses between edit models with only some of this structure. We will also introduce a constraint that says that no matter which two repository states you choose, there is some edit between them, which may be an important practical consideration but does not seem to affect the theory

significantly one way or another. For the discussion of symmetric delta lenses, we will also want to consider edits which can be "undone".

1.1.2 Definition: Given function $f \in X \to Y$, we say x and x' are *equivalent under* f, denoted $x \approx_f x'$, when $f(x) = f(x')$.

It is easy to see that $\approx_f$ is an equivalence relation for any f.

1.1.3 Definition: A graph G is *reflexive* if it comes equipped with a function $id_G \in G_0 \to G_1$ which chooses a distinguished self loop $id_{G,v} : v \to v$ for each node v. By abuse of notation, we will write id_v instead of $id_{G,v}$ when there can be no confusion about which graph is meant.

1.1.4 Definition: A graph G is *connected* if for each $v, v' \in G_0$ there exists an edge $e : v \to v'$.

1.1.5 Definition: A reflexive graph G is *involutive* if it comes equipped with a function ${}^\smile \in G_1 \to G_1$ which associates with each edge $e : v \to v'$ an opposing edge $e^\smile : v' \to v$. It is required to be an involution (so that $e^{\smile\smile} = e$) and to respect the reflexive structure of the graph (so that $id_v{}^\smile = id_v$).

The delta lens frameworks discussed here are based on two edit models: their asymmetric lenses are based on a connected category model of edits, and their symmetric lenses are based on a connected involutive graph model of edits.

In both cases, there is an underlying graph, and in particular this means that each edit must uniquely identify the state that it can be applied to along with the state it produces. At least naively, this requirement seems to be in conflict with our goal of representing edits with objects significantly smaller than the repository states. Many of our edit modules exploit the ability to reuse edits as modifications to many different repository states. Nevertheless, totality of edit application is a nice feature. One can view the two approaches as two extremes, with on one end graphs with a single node representing all possible repository states and on the other end graphs with many nodes where each node represents a single repository state. There may be a middle ground in which graph nodes each represent many possible repository states; the hope then would be that one could keep the benefit of a total edit application function while reusing single edits on many different states. For example, for list edits, one might consider a graph with one node for each possible length of list. Then one would have, for example, deletion edges $\mathsf{del} : m \to n$ when $m < n$; such an edge must store marginally more information than our edit module did (the domain and codomain length rather than a single number telling their difference), but the set of repositories to which it applies is much more clearly delimited. Attempting to recast the edit modules and lenses proposed above in this light would be an interesting area for future work.

The connection between modules and edit graphs can be made precise as follows. To pass from a module X to a graph $\mathbf{Gr}(X)$, let $\mathbf{Gr}(X)_0 = |X|$ be the set of nodes and $\mathbf{Gr}(X)_1 = \{m : x \to y \mid mx = y\}$ be the set of edges (so that $\mathsf{dom}(m : x \to y) = x$ and $\mathsf{cod}(m : x \to y) = y$, hence $(m : x \to y) : x \to y$). The graph can be made reflexive by defining $id_x = \mathbf{1} : x \to x$; if we further define the composition $(m : x \to y); (m' : y \to z) = m'm : x \to z$, the monoid action laws guarantee that we can regard the graph as a category. Now let us see how to pass from a category G to a module $\mathbf{Mod}(G)$. (An arbitrary reflexive graph may be turned into a category: for the arrows between nodes v and v', use the set of paths from v to v' that do not have any *id* edges[1]; for the composition, use path concatenation; and for the identities, use empty paths.) Let the values $|\mathbf{Mod}(G)| = G_0$ be the set of nodes, and edits $\partial\mathbf{Mod}(G) = G_1^\star/\sim_;$ be the set of paths quotiented by the congruence relation that identifies factorable paths $\langle f; g\rangle$ with the factoring $\langle g, f\rangle$. In other words, the edits of $\mathbf{Mod}(G)$ are sequences of edges $\langle g_1, \ldots, g_n\rangle$ which are *not* well-typed: $\mathsf{cod}(g_{i+1}) \neq \mathsf{dom}(g_i)$ for each i. (One can further identify all lists of length more than one—that is, lists that have an internal typing error—with a single fail edit. The result is still a module, but the lens lifting we perform below would not result in a lens: the monoid homomorphism laws may require some ill-typed compositions in the source to be translated to well-typed compositions in the view.) The empty list serves as the identity. Edit application is generated by the equation $\mathrm{d}x \odot_g x = x'$ when $\mathrm{d}x : x \to x'$ (and undefined otherwise). One may choose any object to play the role of *init* (so that there are as many ways to turn a category into a module as there are objects in the category). Passing from a category to a module and back adjoins a fresh identity edit to each object, but otherwise leaves the category unchanged. On the other hand, passing from a module to a category and back may produce a significantly more verbose edit language, even after accounting for the many ways to represent internal failure: each edit m in the source module induces a collection of edits $\{\langle m : x \to y\rangle \mid mx = y\}$ in the target module.

The involutive graph model of edits demands the existence of undo edits, something we did not consider carefully in the edit lens framework above. A suitable module-based analog of the typed involution would be to require each module to include an untyped involution $^\smile$ such that $(\mathrm{d}x^\smile\ \mathrm{d}x) \odot x = x$ whenever $\mathrm{d}x \odot x$ is defined. (Thus $\mathrm{d}x^\smile\ \mathrm{d}x$ is a restricted identity: not necessarily equal to $\mathbf{1}$, but behaves like it for some subset of the values being edited.) Many of the modules and module combinators we have defined above can be equipped with this structure. A notable few that cannot include edit operations which actually delete information, such as the sum module's switch edits and the list module's del edits. These edits would need to be enriched or restricted to include the information being deleted; for example, one could modify the action associated with del edits to only succeed when the list elements being deleted were *init* (so that edits which wish to delete an

[1]Equivalently, the set of paths quotiented by the smallest congruence relation containing the equation that ensures that *id* edges are the unit for composition: $\langle id\rangle = \langle\rangle$.

element must first modify it to being *init* with mod edits), and one could enrich the switch edits with an edit that returns the value to a tagged *init* before switching sides of the sum. Thus in general it seems that requiring an "undo" ability can require mildly larger edit operations.

With these considerations about edits in mind, let us discuss how to generalize asymmetric, state-based lenses first to asymmetric delta lenses and then to symmetric delta lenses.

1.1.1 Asymmetric

Now, let us take the edit model above and see how to enrich asymmetric lenses to take edit information rather than states. As with the state-based version, we will assume that there is strictly more information in the source category S than in the view category V. This means that in the *get* direction, it seems natural to assume that each source edit $\mathrm{d}s : s \to s'$ uniquely determines a view edit $\mathrm{d}v : v \to v'$ by "throwing away" the extra information. It is also quite natural to require this *get* transformation to respect the category structure in S: that is, we should expect $get(id) = id$ and $get(\mathrm{d}s; \mathrm{d}s') = get(\mathrm{d}s); get(\mathrm{d}s')$. Together, these say that *get* is a functor from S to V.

As with all asymmetric frameworks, the *put* direction is a bit more delicate, because it needs to restore missing information. Suppose we have a view edit $\mathrm{d}v : v \to v'$ and wish to produce a source edit. It seems natural to wish that the source edit we produce respects typing in the sense that if we produce $\mathrm{d}s : s \to s'$, then s is in the preimage of v and s' is in the preimage of v'. But this is still very unconstrained; in particular, since there really is a particular s_0 which is currently in synch with v, we really want to produce an edit $\mathrm{d}s$ whose domain is s_0—that is, there is simply not enough information available in a view edit to produce a reasonable translation function *put*. So we cannot translate directly from a view edit to a source edit; however, the key insight of this line of work is that we can translate from a view edit to a *family* of source edits indexed by the source that is currently synchronized with the view.

In detail, given an object mapping $get_0 \in S_0 \to V_0$, we can construct the preimage category S/get_0 as follows. The objects of S/get_0 are the equivalence classes of objects of S under $\approx_{get_0}$. The arrows $f : [s_i] \to [s_o]$ are the functions which take an element $s \in [s_i]$ and produce an arrow $f(s) : s \to s'$ for some $s' \in [s_o]$. The identity arrow is the one which associates to each s the arrow id_s; the composition is defined by $(f; g)(s) = f(s); g(\mathsf{cod}(f(s)))$. When *get* is a graph morphism, we will write S/get as shorthand for S/get_0. With this category in hand, we are ready to define asymmetric delta lenses.

1.1.6 Definition: A *graph morphism* $f \in S \to V$ between graphs S and V is a pair $\langle f_0, f_1 \rangle$ of mappings $f_0 \in S_0 \to V_0$ and $f_1 \in S_1 \to V_1$ such that $f_1(e) : f_0(\mathsf{dom}(e)) \to f_0(\mathsf{cod}(e))$.

1.1.7 Definition: A *semifunctor* $f \in S \to V$ between reflexive graphs is a graph morphism for which $f_1(id_s) = id_{f_0(s)}$.

1.1.8 Definition: An *asymmetric delta lens* $\ell \in S \overset{\Delta}{\underset{a}{\leftrightarrows}} V$ between connected categories S and V is a pair $\langle get, put \rangle$ of graph morphisms $get \in S \to V$ and $put \in V \to S/get$. The lens is *well behaved* (respectively, *very well behaved*) if get and put are semifunctors (resp. functors) and satisfy the behavioral law:

$$get(put(\mathrm{d}v)(s)) = \mathrm{d}v \qquad \text{(ADPutGet)}$$

We will abbreviate "well behaved asymmetric delta lens" to "wbad lens" and "very well behaved asymmetric delta lens" to "vwbad lens".

The suggested behavioral law enforces the intuition given above that all of the information available in the view edits is available in source edits, too. It turns out that the obvious definitions for identity and composition lenses satisfy the behavioral law and induce a category whose objects are connected categories and arrows are vwbad lenses. As in our discussion above relating edit lenses to symmetric lenses, one can connect wbad lenses to asymmetric state-based lenses by adjoining an operation to compute the difference between two states. The paper goes on to show that vwbad lenses only violate the controversial PutPut law if their differencing operation violates a similar DiffDiff law—that is, failure is never due to incorrect edit propagation, only incorrect edit discovery.

One can construct an edit lens out of a vwbad lens as follows. The complement set will be source objects adjoined with a fresh fail value, so that the partial edit application can be extended to a total one with explicit failure:

$$\begin{array}{rr} \mathrm{d}s \odot_t \mathsf{inl}\ s = \mathsf{inl}\ \mathrm{d}s \odot s & \mathrm{d}s \odot s{\downarrow} \\ \mathrm{d}s \odot_t s = \mathsf{inr}\ \mathsf{fail} & \text{otherwise} \end{array}$$

Then the lens construction goes as follows.

$$\frac{\ell \in S \overset{\Delta}{\underset{a}{\leftrightarrows}} V \qquad \ell \text{ is a vwbad lens} \qquad s \in S_0}{\mathsf{symm}_s(\ell) \in \mathbf{Mod}(S) \overset{\Delta}{\leftrightarrows} \mathbf{Mod}(V)}$$

$$\begin{array}{lcl} C & = & S_0 \uplus \{\mathsf{fail}\} \\ \mathit{missing} & = & \mathsf{inl}\ s \\ K & = & \{(s, \mathsf{inl}\ s, \ell.get_0(s)) \mid s \in S_0\} \\ \Rrightarrow_g(\mathrm{d}s, s) & = & (\ell.get_1(\mathrm{d}s), \mathrm{d}s \odot_t s) \\ \Lleftarrow_g(\mathrm{d}v, \mathsf{inl}\ s) & = & \mathsf{let}\ \mathrm{d}s = \ell.put_1(\mathrm{d}v)(s)\ \mathsf{in}\ (\mathrm{d}s, \mathrm{d}s \odot_t \mathsf{inl}\ s) \\ & & \text{when } \mathsf{dom}(\mathrm{d}v) = \ell.get_0(s) \\ \Lleftarrow_g(\mathrm{d}v, s) & = & (\mathbf{1}, \mathsf{inr}\ \mathsf{fail}) \text{ otherwise} \end{array}$$

The proof that this is well-defined and forms a lens is tedious but straightforward. It relies critically on $\ell.get$ and $\ell.put$ respecting arrow composition and on the object part of the roundtrip law, but not that $\ell.get$ and $\ell.put$ respect identities (because we introduce a fresh identity) or that they roundtrip on arrows (as our formalism does not have an analogous law). We conjecture that a construction similar to the one used to decompose symmetric lenses into a pair of asymmetric, state-based lenses can be used to decompose edit lenses into a pair of vwbad lenses.

Later work proposes a concrete edit model and a collection of wbad lenses and combinators [32]. In addition to the above definition of wbad lenses, they give a proposal for a framework of *horizontal delta lenses* which are more convenient to implement but whose behavioral guarantees are less intuitive. Horizontal delta lenses are nevertheless suitably constrained so that they can be converted into wbad lenses as necessary. Their data model is based on containers, with edits containing (in part) an injective relation between the positions of the old and new pieces of data. In addition to many constructions similar to ours, they also discuss fold and unfold operations for containers that are built from fixpoints of regular higher-order functors. These give rise to significant complications in handling the full range of edits; they discuss how to handle insertions and deletions of nodes, but do not discuss reordering. Unlike our development, all repositories are homogeneous containers; in particular, their tensor product analog restricts the contained values in the two parts of the tuple to have identical types.

1.1.2 Symmetric

Diskin et al. also spend some effort considering what machinery is needed to support transformations between domains that each have missing information—that is, symmetric transformations [13]. As we observed in our symmetric lens development, passing from asymmetric to symmetric lenses is cleanest if one introduces a complement—some extra information about how the values in the two repositories correspond. Their development similarly allows for extra information, with a little bit of extra notational complexity arising from the pervasive use of typing: edits are typed via a category, as discussed above, and complements are also typed, as we discuss now. Because complements are typed, we will need a notion of when the types of an edit and a complement match. We give two such notions below: one for complements that match before an edit is applied, and one for complements that match after.

1.1.9 Definition: Given graphs G and H, we define the *domain-* and *codomain-coincident* edge pairings as follows:

$$G \times_{\mathsf{dom}} H = \{(e_g, e_h) \mid e_g : v \to v' \in G_1 \wedge e_h : v \multimap v'' \in H_1\}$$
$$G \times_{\mathsf{cod}} H = \{(e_g, e_h) \mid e_g : v \to v' \in G_1 \wedge e_h : v' \multimap v'' \in H_1\}$$

1.1.10 Definition: A *symmetric delta lens* ℓ connecting connected categories X and Y, written $\ell \in X \overset{\Delta}{\underset{s}{\leftrightarrows}} Y$, consists of:

- a bipartite graph R whose two parts are X_0 and Y_0 (the edges of R are called *correspondence relations*),
- a function $\mathsf{fPpg} \in X \times_{\mathsf{dom}} R \to Y_1 \times R_1$, and
- a function $\mathsf{bPpg} \in Y \times_{\mathsf{dom}} R \to X_1 \times R_1$.

We will write fPpg_1 and fPpg_2 (and similarly for bPpg) for the Y_1 and R parts of fPpg's output, respectively.

The preconditions for fPpg above stating that the edit and correspondence relations are domain-coincident is somewhat similar to our precondition requiring an edit which applies cleanly. The bipartite graph R plays a similar role to our consistency relations: if there is an edge $r : x \text{ —— } y$, then we can think of r as being a (typed) complement that is consistent with states x and y.

A major contribution of this line of research is an exploration of behavioral guarantees that reasonable symmetric delta lenses might offer. The obvious laws are too strong; but the insight of this development is that if we take the obvious laws and replace equalities by a slightly coarser equivalence relation, we get laws that are much more plausible. The core of the problem is that equality on X edits distinguishes between edits that modify information not available in Y; we would prefer a relation that compares only the parts of the edit that affect the shared information. At first it seems difficult to define "shared information" formally, but lenses are exactly transformations that define what information is shared; so the relation is parameterized by a lens.

1.1.11 Definition: Given symmetric delta lens $\ell \in X \overset{\Delta}{\underset{s}{\leftrightarrows}} Y$ and a correspondence relation $r : x \text{ —— } y$ for ℓ, we define equivalence relations on edits to x and y, respectively:

$$\sim_{\ell/r} \;=\; \approx_{\lambda \mathrm{d}x.\ \ell.\mathsf{fPpg}_1(\mathrm{d}x,r)}$$
$$\ell/r\sim \;=\; \approx_{\lambda \mathrm{d}y.\ \ell.\mathsf{bPpg}_1(\mathrm{d}y,r)}$$

When the lens is understood from context, we will write $\mathrm{d}x \sim_r \mathrm{d}x'$ instead of $\mathrm{d}x \sim_{\ell/r} \mathrm{d}x'$ (and similarly for $\mathrm{d}y \;{}_r\!\sim \mathrm{d}y'$).

Armed with this notation, they propose several possible restrictions that one could place on symmetric delta lenses. The first two restrictions are analogous to ones discussed in our work above. Like our demand that applicable edits get translated to applicable edits that restore consistency, rule SDWellTyped below demands that the edits and correspondence relations involved in an invocation of

fPpg form a well-typed square. They also demand that the propagation functions preserve the self-loop structure of the edit graphs via the SDID rule.

$$\frac{\mathsf{fPpg}(\mathrm{d}x, r) = (\mathrm{d}y, r')}{\begin{array}{c} r : \mathsf{dom}(\mathrm{d}x) \text{ —} \mathsf{dom}(\mathrm{d}y) \\ r' : \mathsf{cod}(\mathrm{d}x) \text{ —} \mathsf{cod}(\mathrm{d}y) \end{array}} \qquad \text{(SDWELLTYPED)}$$

$$\frac{r : x \text{ —} y}{\mathsf{fPpg}(id_x, r) = (id_y, r)} \qquad \text{(SDID)}$$

The edit graphs have another kind of structure given by the $^\smile$ undo operation. One might hope that this structure is preserved in a similar way; for example, a rule like SDFUNDO-STRONG* seems reasonable at first blush.

$$\frac{\mathsf{fPpg}(\mathrm{d}x, r) = (\mathrm{d}y, r')}{\mathsf{fPpg}(\mathrm{d}x^\smile, r') = (\mathrm{d}y^\smile, r)} \qquad \text{(SDFUNDO-STRONG*)}$$

Unfortunately, this rule is very restrictive. Suppose the Y side of the lens were to store some information not available in the X side, and propagating $\mathrm{d}x$ produces a $\mathrm{d}y$ that deletes some of that information. Then this information could not be restored from the information in $\mathrm{d}x^\smile$.[2] One way to weaken this law to something more plausible would be to demand that we output something that behaves like $\mathrm{d}y^\smile$ on the shared information; that is, by weakening the equality in the conclusion to our coarser equivalence relation from above:

$$\frac{\mathsf{fPpg}(\mathrm{d}x, r) = (\mathrm{d}y, r')}{\mathsf{fPpg}_1(\mathrm{d}x^\smile, r') \; {}_{r'}{\sim}\, \mathrm{d}y^\smile} \qquad \text{(SDFUNDO)}$$

The fourth and final behavioral law proposed demands that the edit propagation functions be near inverses: that is, if we propagate $\mathrm{d}x$ to $\mathrm{d}y$, then the corresponding edit determined by the other propagation function should be $\mathrm{d}x$. As stated, this law is again too strong, because some of the modifications described by $\mathrm{d}x$ are to unshared data, and hence are not available in $\mathrm{d}y$ during re-propagation. As before, we can make the rule more reasonable by weakening from equality to equivalence:

$$\mathsf{bPpg}_1(\mathsf{fPpg}_1(\mathrm{d}x, r), r) \sim_r \mathrm{d}x \qquad \text{(SDINVERTIBLE)}$$

This behavioral law is called a roundtrip law in their development, but that name is a little misleading, as the update to the y value and the updated correspondence relation are discarded before applying the bPpg function. (It is as if two separate people happened to take flights that crossed paths in the middle, rather than a single person taking a round trip.) We will instead call this law a *triple-trip* law—for the two trips evident in the law plus one trip hidden by the equivalence relation.

The line of research goes on to describe other theoretical frameworks with interfaces closer to what an end-user programmer might want to implement that can

[2] One could imagine storing just enough information in the correspondence relations to allow undoing one operation. Perhaps this could be made to work, but it is unlikely this would scale well in situations where there are composite edits (and hence composite undos).

give rise to symmetric delta lenses. In particular, they describe a framework they call *consistency maintainers* [13] which include explicit alignment and consistency-restoration phases as well as explore conditions under which a triple-graph grammar can be used to produce a law-abiding lens [20]. No concrete instantiations are given for any of the three frameworks. We have found that undertaking this endeavor is a valuable crucible in which to test prospective frameworks, as the design of a lens language makes a mismatch between behavioral laws and actual behavior much more clear. (Just as a good framework helps to spot potential implementation bugs, an implementation helps point out potential framework bugs.) In particular, sequential composition—in our experience, a crucial tool for building practical lenses—is not considered, and the ensuing need for a notion of lens equivalence is not addressed.

The proposed SDInvertible and SDFUndo laws seem on the surface to be quite natural restrictions. Our development does not have analogous laws, and it seems that including them would necessitate a stronger equational theory for many of the modules proposed above. Exploring the consequences of these laws could be an interesting avenue for future work on edit lenses.

1.2 Algebraic rephrasing

There is a line of work on algebraic foundations for delta lenses that arrives at a model very similar to the edit lens framework described above [37]. They consider, as we do, edit monoids together with edit translation morphisms and (total) edit application actions. One significant difference is that they consider generalizing asymmetric rather than symmetric lenses, adopting correspondingly modified behavioral laws. In particular, in their setting, a delta lens is a *lens-like split short exact sequence*. Below we discuss each of these restrictions in right-to-left order. We begin with two standard definitions to establish some notation.

1.2.1 Definition: The *image* of a function $f \in X \to Y$ is the set of elements $\mathrm{im}(f) \subset Y$ that f can output:

$$\mathrm{im}(f) = \{f(x) \mid x \in X\}$$

1.2.2 Definition: The *kernel* of a monoid homomorphism f, denoted $\ker(f)$, is the preimage of $\mathbf{1}$:

$$\ker(f) = \{x \mid f(x) = \mathbf{1}\}$$

1.2.3 Definition: An *exact sequence* is a sequence $\langle f_1, \ldots, f_n \rangle$ of monoid morphisms with compatible domains and codomains, that is,

$$X_0 \xrightarrow{f_1} X_1 \xrightarrow{f_2} X_2 \xrightarrow{f_3} \cdots \xrightarrow{f_{n-2}} X_{n-2} \xrightarrow{f_{n-2}} X_{n-1} \xrightarrow{f_n} X_n$$

and such that $\mathrm{im}(f_i) = \ker(f_{i+1})$ for each i.

1.2.4 Definition: An exact sequence is *short* if it has four morphisms and starts and ends at $\partial\mathit{Unit}$:

$$\partial\mathit{Unit} \xrightarrow{\quad i \quad} K \xrightarrow{\quad k \quad} \partial S \xrightarrow{\quad f \quad} \partial V \xrightarrow{\quad s \quad} \partial\mathit{Unit}$$

We will say *around* to mean the third element of a sequence, as in, "$\langle i, k, f, s\rangle$ is a short exact sequence around f."

Before we define what split and lens-like mean, let us consider when an edit translation homomorphism $f \in \partial S \to \partial V$ may be extended to a short exact sequence. The homomorphism $s \in \partial V \to \partial\mathit{Unit}$ must be the constantly-$\mathbf{1}$ function (there are no other functions with that type), so that its kernel is $\ker(s) = \partial V$. Hence the restriction $\mathrm{im}(f) = \ker(s)$ that arises from extending the sequence to the right says that f must be surjective. On the other hand, the sequence may always be extended to the left by choosing K to be the submonoid $\ker(f)$ and k to be the inclusion function. (The homomorphism i is completely determined by the homomorphism laws once we have chosen a monoid K: it must map the sole input element $\mathbf{1}_{\partial\mathit{Unit}}$ to $\mathbf{1}_K$.) Other choices for K are possible—for example, by adding a fresh generator to K that k maps to any non-trivial element of ∂S—but we will not be interested in this ability below.

In lens terms, one should think of f as being an edit-lens analog of the asymmetric lens framework's *get* function. Giving a short exact sequence amounts to identifying an edit translation function $f \in \partial S \to \partial V$ that is compatible with the monoid structure on edits and such that each V-edit has at least one analogous S-edit.

1.2.5 Definition: A short exact sequence around $f \in \partial S \to \partial V$ is said to *split* if there is a homomorphism $g \in \partial V \to \partial S$ such that $g; f = id_{\partial V}$.

We are guaranteed that there is a *function* g by the fact that f is surjective, but not guaranteed that any such function is a monoid homomorphism. If we do have such a homomorphism g that splits the sequence, then in lens terms we should consider that g to be an edit-lens analog of the *put* function. Then $g; f = id_{\partial V}$ says that all the information available in V-edits are also available in S-edits, an analogous restriction to the one on state-based lenses that says that all the information available in the view repository is available in the source repository.

The final condition placed on this variant of delta lenses is that they be lens-like. Thus far, all the conditions have been purely in terms of edits; this final pair of properties connect the world of edits and the world of states. This is similar to the edit lens law that requires $\Rrightarrow$ and $\Lleftarrow$ to respect a consistency relation on states.

1.2.6 Definition: A monoid action $\odot \in \partial X \times |X| \to |X|$ is *transitive* if for all $x, x' \in |X|$ there is $\mathrm{d}x \in \partial X$ such that $\mathrm{d}x \odot x = x'$. We will say a module is transitive when its action is.

1.2.7 Definition: A short exact sequence around $f \in \partial S \to \partial V$ split by g is *lens-like* if it comes equipped with transitive, total modules for ∂S and ∂V such that two conditions hold:

$$\frac{\mathrm{d}s\ \mathit{init}_S = \mathrm{d}s'\ \mathit{init}_S}{f(\mathrm{d}s)\ \mathit{init}_V = f(\mathrm{d}s')\ \mathit{init}_V} \tag{LL1}$$

$$\frac{\mathrm{d}v\ f(\mathrm{d}s)\ \mathit{init}_V = f(\mathrm{d}s)\ \mathit{init}_V}{g(\mathrm{d}v)\ \mathrm{d}s\ \mathit{init}_S = \mathrm{d}s\ \mathit{init}_S} \tag{LL2}$$

Rule LL1 amounts to saying that f is (part of) a module homomorphism (not just a monoid homomorphism). Rule LL2 is a bit more subtle, but is motivated by this rephrasing of the state-based asymmetric lens framework's GetPut law:

$$\frac{\mathit{get}(s) = v}{\mathit{put}(v, s) = s} \tag{GetPutAlt}$$

Rule GetPutAlt says, roughly, "if the view v has not changed since the last synchronization, then the source s should not change, either." Similarly, LL2 says, roughly, "if the edit we are about to translate does not change the view $f(\mathrm{d}s)\ \mathit{init}_V$, then the edit we output should not change the source $\mathrm{d}s\ \mathit{init}_S$."

That work goes on to explore the properties of this kind of delta lens. One can take a lens-like sequence around $f \in \partial S \to \partial V$ equipped with a diffing operation $\mathit{dif} \in |S| \times |V| \to \partial V$ (satisfying the obvious sanity condition) and produce an asymmetric, state-based lens. Additionally, there is a close relationship between demanding the existence of inverse edits—that is, working with edit groups rather than edit monoids—and the PutPut asymmetric lens law:

$$\mathit{put}(v, \mathit{put}(v', s)) = \mathit{put}(v, s) \tag{PutPut}$$

They show that one can define suitably restricted submonoids $\partial V \subset V \to V$ and $\partial S \subset S \to S$ and lift PutPut-abiding asymmetric lenses into a lens-like sequence on groups. Furthermore, the two translations agree with each other: converting a lens to a lens-like sequence and back is the identity transformation, regardless of the choice of *dif* operation in the latter transformation.

The primary difference between their work and ours is that they consider only asymmetric situations. However, they also consider many fundamentally different restrictions than the current development does, even after accounting for the different setting. For example, they propose a law requiring that when g splits a sequence around f we additionally have $g; f = \mathit{id}$. Since f is surjective, this is the same as demanding $f; g; f = f$, akin to Diskin's proposed triple-trip law discussed above. As mentioned in that discussion, it is not a law that we have considered

Teacher name	Salary
Sam Rickard	57,000
Jon Jacobs	50,000
Mary Jones	65,000

(a) HR's view

Teacher name
Sam Rickard
Jon Jacobs
Mary Jones

(b) A secretary's view

Figure 1.1: A school's staff list, as seen by HR and by the principal's secretary

carefully; but it seems we may be able to achieve something similar in many of the lenses we defined by introducing appropriate equalities to our edit monoids for structured data. The paper also spends some time discussing the ramifications of demanding an edit group rather than an edit monoid. We have not explored this restriction deeply, but some cursory investigations suggest that including enough information to undo each operation may be at odds with the size benefits promised above. Another restriction they have throughout their development is that their edit application actions are invariably total. We believe that partiality of these actions is an important real-world consideration. Treating it carefully allows us to distinguish between error conditions and edits which successfully do nothing, and to give a guarantee that our lenses do not spuriously turn a succesful nothing into an error condition.

They also treat backwards-compatibility with asymmetric, state-based lenses very seriously, which gives rise to their lens-like restrictions. Their rule LL1 stating that the *get* direction is a module homomorphism can be seen as saying that edit translation is consistent with state translation. Our demand that the edit translations preserve a consistency relation can be seen as a generalization of this. On the other hand, their rule LL2—necessary to ensure that their delta lenses behave like state-based lenses regardless of *dif* operation—seems quite strong. The goal appears to be to preserve the state-based behavior that changing nothing on one side changes nothing on the other; however, it is our view that demanding that the distinguished do-nothing edit from one module be translated to the distinguished do-nothing edit from the other module already captures this intuition. There are edits which appear to do nothing to a given view but which nevertheless have semantic content, and should therefore be allowed to be distinguished by a lens. Consider the example of Figure **??** again, reproduced here as Figure 1.1. An edit which deletes the last element of the secretary's view, then inserts a fresh element with value "Mary Jones", apparently does nothing to the current view. Nevertheless, it seems quite natural[3] for the translation of this edit to reset the salary associated with "Mary Jones" to a default value; a rule like LL2 would prevent lenses from having this kind of nuanced behavior.

[3]Possibly even desirable—an obviously incorrect value is often preferable to a plausible incorrect value.

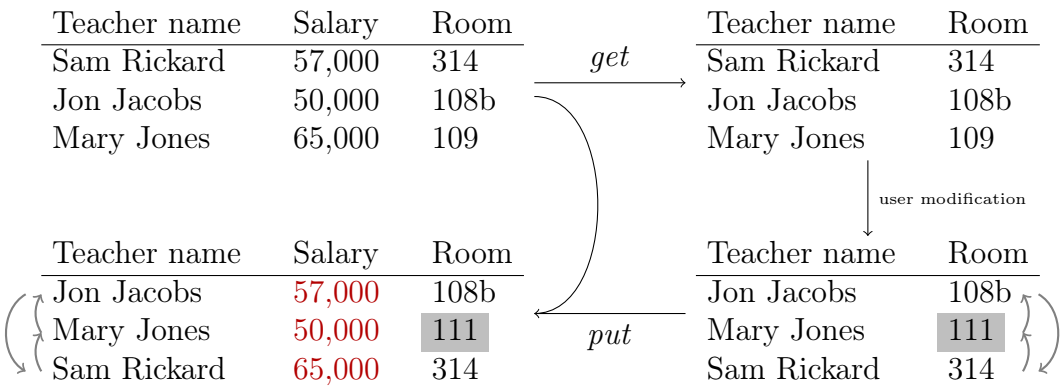

Figure 1.2: An easily fixed misalignment

Finally, our development includes significantly more effort instantiating the lens framework to particular lenses and lens combinators. We believe that this is good evidence that our behavioral restrictions are relaxed enough to accomodate important use cases; nevertheless, they were strict enough to prevent many genuinely undesirable behaviors in early proposals for these combinators (not documented here).

1.3 Matching lenses

Dictionary lenses [8] and their sequel, matching lenses [6], are also motivated by the alignment problems discussed above. We will consider a variation of our motivating example from Chapter **??** 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 1.2 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 1.3 gives an example of a dictionary lens resetting a salary that a plain lens would preserve. The observation here is that simple

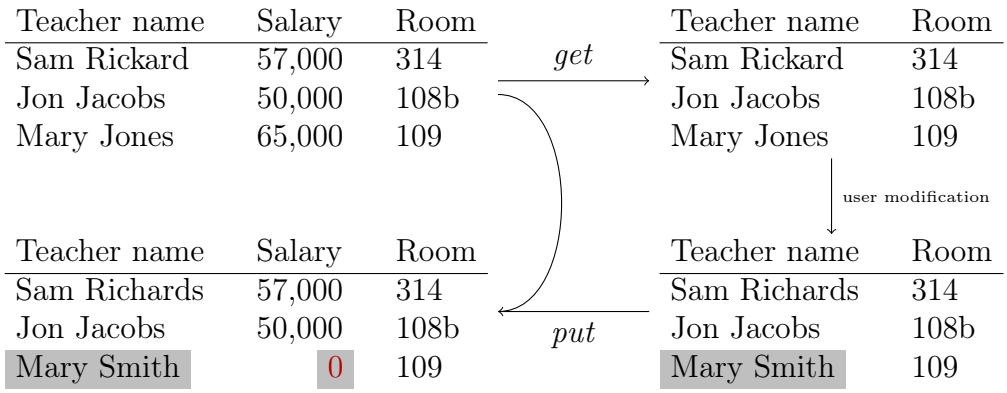

Figure 1.3: With dictionary lenses, changing a key causes information loss

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 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.

1.4 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 niche [22, 23, 31]. 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 considered at all; furthermore, annotated trees are always at least as big as the real tree they represent, so size issues are not addressed. Additionally, their behavioral laws govern how lenses treat annotated values. There is an erasure process to turn annotated trees into plain ones (by performing the respective insertions, deletions, and modifications), but no exploration of the interaction between erasure and the lens' behavioral laws. The second key difference is that allowing duplication requires them to significantly relax the behavioral laws: for example, if only one copy of some duplicated information is modified, one wishes a roundtrip of the transformations to modify the other copy analogously. The weakened laws allow this, but also allow many other apparently undesirable behaviors like ignoring all changes indiscriminately.

1.5 Constraint maintainers

Constraint maintainers are an early exploration of a symmetric framework for bidirectional transformations [29]. The framework is a very natural one, as mentioned in §**??**: given a relation $R \subset X \times Y$, a constraint maintainer is a pair of functions $\triangleright \in X \times Y \to Y$ and $\triangleleft \in X \times Y \to X$ for which $x \; R \; (x \triangleright y)$ and $(x \triangleleft y) \; R \; y$. One may optionally also require that related values remain unchanged, that is:

$$\frac{x \; R \; y}{x \triangleright y = y} \qquad\qquad \frac{x \; R \; y}{x \triangleleft y = x}$$

Some discussion of generalizing these behavioral laws to a Principle of Least Change is given; in any case, those maintainers which satisfy the property above can be lifted to symmetric lenses as follows.

$$\frac{\langle \triangleleft, \triangleright \rangle \text{ a maintainer for } R \subset X \times Y \qquad x \; R \; y}{\mathit{cmaint}(R, \triangleleft, \triangleright, x, y) \in X \leftrightarrow Y}$$

$$\begin{array}{lcl} C & = & R \\ \mathit{missing} & = & (x, y) \\ \mathit{putr}(x, (x', y)) & = & (x \triangleright y, (x, x \triangleright y)) \\ \mathit{putl}(y, (x, y')) & = & (x \triangleleft y, (x \triangleleft y, y)) \end{array}$$

There is a significant body of maintainers discussed there, with a notable exception being constraint maintainer composition. General maintainer composition is shown

to be uncomputable. A restricted composition in the case where one of the two maintainers corresponds to an asymmetric lens is given, and Meertens proposes the use of maintainer chains when this is insufficient, but the properties of these chains are not considered. There is some exploration of how to deal with alignment issues; the main idea they propose is to lift relations on values to relations on the edit sequences used to build values. One can then define constraint maintainers which inspect such edit sequences. However, little consideration is given to desirable behavioral laws with respect to these more fine-grained structures; furthermore, the incremental capability we expose from our edit lenses to allow for small updates is not explored.

Bibliography

[1] M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Foundations of Software Science and Computation Structures*, pages 23–38. Springer, 2003.

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005.

[3] Samson Abramsky and Nikos Tzevelekos. Introduction to categories and categorical logic. In Bob Coecke, editor, *New Structures for Physics*. Springer, 2010.

[4] Pets Adviser. Cat in Pumpkin Hat, 2012. URL `http://www.flickr.com/photos/petsadviser-pix/8126559828/`. Online; accessed 18-December-2013.

[5] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, December 1981.

[6] Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: Alignment and view update. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, September 2010.

[7] Brian Berliner et al. Cvs ii: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, volume 341, page 352, 1990.

[8] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*, January 2008.

[9] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 2007. To appear. Extended abstract in *Database Programming Languages (DBPL)* 2005.

[10] ChrisGampat. Cat thinks it's the next Tony Hawk, 2009. URL `http://www.flickr.com/photos/chrisgampat/3827798021/`. Online; accessed 18-December-2013.

[11] CollabNet. Apache Subversion. URL `http://subversion.apache.org/`. Online; accessed 18-December-2013.

[12] Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. *Journal of Object Technology*, 10: 6:1–25, 2011.

[13] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. Technical Report GSDLAB-TR 2011-05-03, University of Waterloo, May 2011.

[14] Dropbox. URL `https://www.dropbox.com/`. Online; accessed 18-December-2013.

[15] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007. ISSN 0164-0925. Extended abstract presented at *Principles of Programming Languages (POPL)*, 2005.

[16] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007. Extended abstract in *Principles of Programming Languages* (POPL), 2005.

[17] John Nathan Foster. *Bidirectional Programming Languages*. PhD thesis, University of Pennsylvania, December 2009.

[18] Git. URL `http://git-scm.com/`. Online; accessed 19-December-2013.

[19] R. Hasegawa. Two applications of analytic functors. *Theoretical Computer Science*, 272(1-2):113–175, 2002.

[20] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of model synchronization based on triple graph grammars. In *Model Driven Engineering Languages and Systems*, pages 668–682. Springer, 2011.

[21] Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas*, January 2011.

[22] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bi-directional transformations. In *Partial Evaluation and Program Manipulation (PEPM)*, pages 178–189, 2004. Extended version in *Higher Order and Symbolic Computation*, Volume 21, Issue 1-2, June 2008.

[23] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1–2), June 2008. Short version in PEPM '04.

[24] James Wayne Hunt and M Douglas McIlroy. *An algorithm for differential file comparison*. Bell Laboratories, 1976.

[25] C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In Donald Sannella, editor, *ESOP*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 1994. ISBN 3-540-57880-3.

[26] A. Joyal. Foncteurs analytiques et especes de structures. *Combinatoire énumérative*, pages 126–159, 1986.

[27] David Lutterkort. Augeas: A Linux configuration API, February 2007. Available from `http://augeas.net/`.

[28] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 47–58. ACM Press New York, NY, USA, 2007.

[29] Lambert Meertens. Designing constraint maintainers for user interaction, 1998. Manuscript.

[30] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[31] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, pages 2–20, November 2004.

[32] Hugo Pacheco, Alcino Cunha, and Zhenjiang Hu. Delta lenses over inductive types. In *First International Workshop on Bidirectional Transformations (BX)*, 2012.

[33] David Soria Parra, Arne Babenhauserheide, and Steve Losh. Mercurial SCM. URL `http://mercurial.selenic.com/`. Online; accessed 19-December-2013.

[34] Benjamin C Pierce and Jérôme Vouillon. What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer. Technical report, University of Pennsylvania, 2004.

[35] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. In *ACM Sigplan Notices*, volume 45, pages 1–12. ACM, 2010.

[36] David Roundy. Darcs – FrontPage. URL `http://darcs.net/`. Online; accessed 19-December-2013.

[37] Perdita Stevens. Towards an algebraic theory of bidirectional transformations. In *Graph Transformations: 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008, Proceedings*, page 1. Springer, 2008.

[38] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, 1996.

[39] Various. HackageDB: introduction, April 2012. URL `http://hackage.haskell.org/packages/hackage.html`.

[40] Varmo Vene. *Categorical Programming with Inductive and Coinductive Types*. PhD thesis, Universitatis Tartuensis, 2000.

[41] zeevveez. Cat Smells Pineapple, 2010. URL `http://www.flickr.com/photos/zeevveez/4906820567/`. Online; accessed 18-December-2013.