1-1-2012

Edit Lenses

Martin Hoffmann
Ludwig-Maximilians-Universitat

Benjamin C. Pierce
University of Pennsylvania, bcpierce@cis.upenn.edu

Daniel Wagner
University of Pennsylvania

Recommended Citation

Martin Hoffmann, Benjamin C. Pierce, and Daniel Wagner, "Edit Lenses", . January 2012.

Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. **Edit Lenses**. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania*, January 2012.

doi>10.1145/2103656.2103715

Edit Lenses

Abstract
A lens is a bidirectional transformation between a pair of connected data structures, capable of translating an edit on one structure into an appropriate edit on the other. Many varieties of lenses have been studied, but none, to date, has offered a satisfactory treatment of how edits are represented. Many foundational accounts [5, 7] only consider edits of the form "overwrite the whole structure," leading to poor behavior in many situations by failing to track the associations between corresponding parts of the structures when elements are inserted and deleted in ordered lists, for example. Other theories of lenses do maintain these associations, either by annotating the structures themselves with change information [6, 15] or using auxiliary data structures [2, 4], but every extant theory assumes that the entire original source structure is part of the information passed to the lens. We offer a general theory of edit lenses, which work with descriptions of changes to structures, rather than with the structures themselves. We identify a simple notion of "editable structure"—a set of states plus a monoid of edits with a partial monoid action on the states—and construct a semantic space of lenses between such structures, with natural laws governing their behavior. We show how a range of constructions from earlier papers on "statebased" lenses can be carried out in this space, including composition, products, sums, list operations, etc. Further, we show how to construct edit lenses for arbitrary containers in the sense of Abbott, Altenkirch, and Ghani [1]. Finally, we show that edit lenses refine a well-known formulation of state-based lenses [7], in the sense that every state-based lens gives rise to an edit lens over structures with a simple overwrite-only edit language, and conversely every edit lens on such structures gives rise to a state-based lens.

Disciplines
Computer Sciences

Comments
Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. **Edit Lenses**. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania*, January 2012.

doi>10.1145/2103656.2103715

Edit Lenses

Martin Hofmann
Ludwig-Maximilians-Universität

Benjamin Pierce
University of Pennsylvania

Daniel Wagner
University of Pennsylvania

Abstract

A *lens* is a bidirectional transformation between a pair of connected data structures, capable of translating an edit on one structure into an appropriate edit on the other. Many varieties of lenses have been studied, but none, to date, has offered a satisfactory treatment of how edits are *represented*. Many foundational accounts [5, 7] only consider edits of the form "overwrite the whole structure," leading to poor behavior in many situations by failing to track the associations between corresponding parts of the structures when elements are inserted and deleted in ordered lists, for example. Other theories of lenses do maintain these associations, either by annotating the structures themselves with change information [6, 15] or using auxiliary data structures [2, 4], but every extant theory assumes that the entire original source structure is part of the information passed to the lens.

We offer a general theory of *edit lenses*, which work with descriptions of changes to structures, rather than with the structures themselves. We identify a simple notion of "editable structure"—a set of states plus a monoid of edits with a partial monoid action on the states—and construct a semantic space of lenses between such structures, with natural laws governing their behavior. We show how a range of constructions from earlier papers on "state-based" lenses can be carried out in this space, including composition, products, sums, list operations, etc. Further, we show how to construct edit lenses for arbitrary *containers* in the sense of Abbott, Altenkirch, and Ghani [1]. Finally, we show that edit lenses refine a well-known formulation of state-based lenses [7], in the sense that every state-based lens gives rise to an edit lens over structures with a simple overwrite-only edit language, and conversely every edit lens on such structures gives rise to a state-based lens.

1. Introduction

Recent years have seen growing interest in *bidirectional programming languages*—domain-specific languages where a program describes how to maintain a connection between data structures of two different shapes, translating updates to one structure into appropriate updates to the other. The core concepts of bidirectional programming have roots in early work on the database view update problem (see [5] for a survey); more recently, they have been explored in diverse areas including model-driven software development [13], data synchronization [5], user interfaces [10], and Unix system configuration management [9].

The meaning of a bidirectional program connecting a set X to a set Y—often called a *lens* from X to Y—is intuitively a pair of transformations, one mapping X updates to Y updates and the other mapping Y updates to X updates, subject to some behavioral laws specifying how the two transformations fit together. Technically, this intuition can be realized in numerous ways. A naive definition is to say that a lens from X to Y is just a pair of functions, $f \in X \rightarrow Y$ (telling how to map an updated X state to an appropriate Y state) and $g \in Y \rightarrow X$. But this is too simple: if the lens laws impose the reasonable requirement that f and g should "round trip," then our bidirectional programs will only denote bijections—an important but limited special case. To allow for situations where each structure can contain a mixture of information that is shared with the other and information that is not, something more than just the updated structure must be given as input to the transformations.

Different variants of lenses differ as to what this "something more" should be. We might, for example, give the transformation from X to Y both a new X and an old Y—i.e., $f \in X \times Y \rightarrow Y$—with the intention that f should weave together the "shared" information from the new X with the "local" information from the old Y to produce a new Y. Or instead of a whole Y, we might pass f some smaller structure (a *complement*) representing just the information that is needed to build an updated Y out of an updated X. Or perhaps one of these plus some additional information about the *alignment* of the updated information (e.g., "a new element was inserted at the beginning of this list, so the second element in the new X corresponds to the first element in the old Y"), either in the form of an auxiliary data structure or perhaps somehow embedded as annotations in the updated X itself.

What all these variants have in common is that the inputs to a lens always include the *whole* updated state. This leaves an unfortunate gap between the theory and practical realizations, which generally represent updates in some simpler, more compact form that only describes what has changed in a possibly large structure.

In this paper, we offer the first foundational treatment of *edit lenses*—lenses that operate directly on edits, rather than on whole structures. Our theory of edit lenses is built from simple and familiar algebraic structures (§3). It supports a wide range of fundamental syntactic constructions—composition, products, sums, list operations, etc.—allowing us to construct lenses for complex data structures together with appropriate representations for edits in a compositional fashion (§4). Indeed, the theory includes a general account of how to construct "mapping" lenses for a wide class of *container* data structures [1] such as lists and trees (§5). This rich set of syntax constructors should form a suitable basis for the design of new bidirectional languages, for example in the style of Boomerang [2]. Our theory can support a wide variety of *edit languages*. We mostly concentrate on the simplest form, where compound edits are freely generated from some set of atomic edits; §6 considers the extension to richer languages with additional algebraic laws. Finally, our theory generalizes and refines the state-based *symmetric lenses* of Hofmann, Pierce, and Wagner [7] in a precise sense (§7). The paper

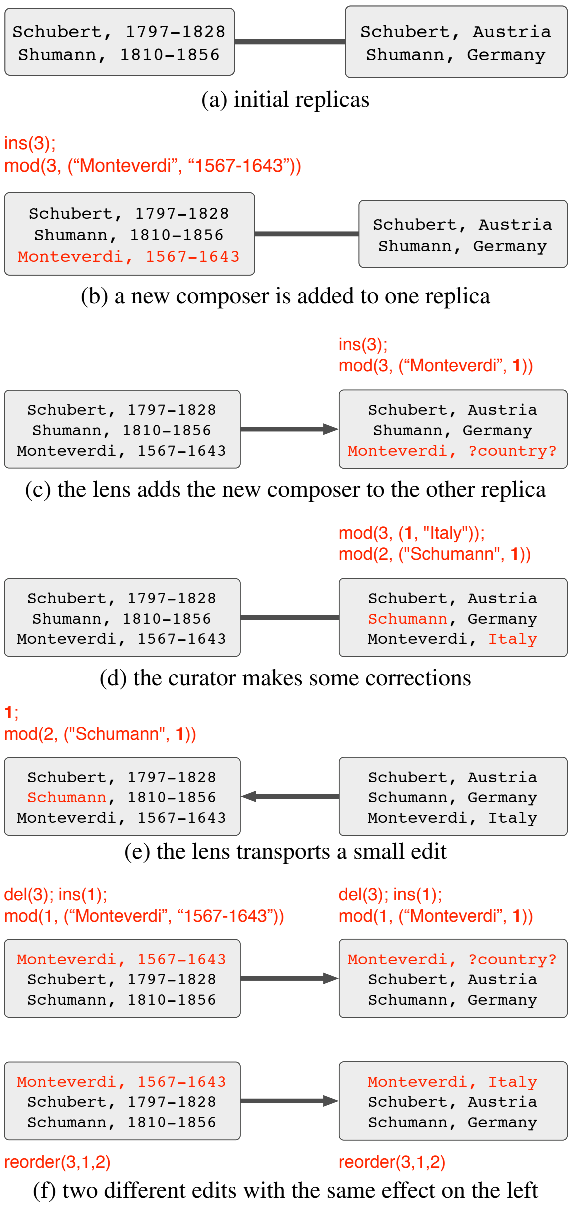


Figure 1. A simple (complement-less) edit lens in action.

ends with a discussion of related work (§8) and some remarks on future directions (§9).

2. Overview

Before diving into technicalities, let's take a brief tour of the main ideas via some examples. Figure 1 demonstrates a simple use of edit lenses to synchronize two replicas.[1] In part (a), we see the initial replicas, which are in a synchronized state. On the left, the replica is a list of records describing composers' birth and death years; on the right, a list of records describing the same composers' countries of origin. In part (b), the user interacting with the left-hand replica decides to add a new composer, Monteverdi, at the end of the list. This change is described by the edit script ins(3); mod(3, ("Monteverdi", "1567-1643")). The script says to first *insert* a dummy record at index three, then *modify* this record by replacing the left field with "Monteverdi" and replacing the right field with "1567-1643". (One could of course imagine other edit languages where the insertion would be done in one step. We represent it this way because this is closer to how our generic "container mapping" combinator in §5 will do things.) The lens connecting the two replicas now converts this edit script into a corresponding edit script that adds Monteverdi to the right-hand replica, shown in part (c): ins(3); mod(3, ("Monteverdi", **1**)). Note that the translated mod command overwrites the name component but leaves the country component with its default value, "?country?." This is the best it can do, since the edit was in the left-hand replica, which doesn't mention countries. Later, an eagle-eyed editor notices the missing country information and fills it in, at the same time correcting a spelling error in Schumann's name, as shown in (d). In part (e), we see that the lens discards the country information when translating the edit from right to left, but propagates the spelling correction.

Of course, a particular new replica state can potentially be achieved by many different edits, and these edits may be translated differently. Consider part (f) of Figure 1, where the left-hand replica ends up with a row for Monteverdi at the beginning of the list, instead of at the end. Two edit scripts that achieve this effect are shown. The upper script deletes the old Monteverdi record and inserts a brand new one (which happens to have the same data) at the top; the lower script rearranges the order of the list. The translation of the upper edit leaves Monteverdi with a default country, while the lower edit is translated to a rearrangement, preserving all the information associated with Monteverdi.

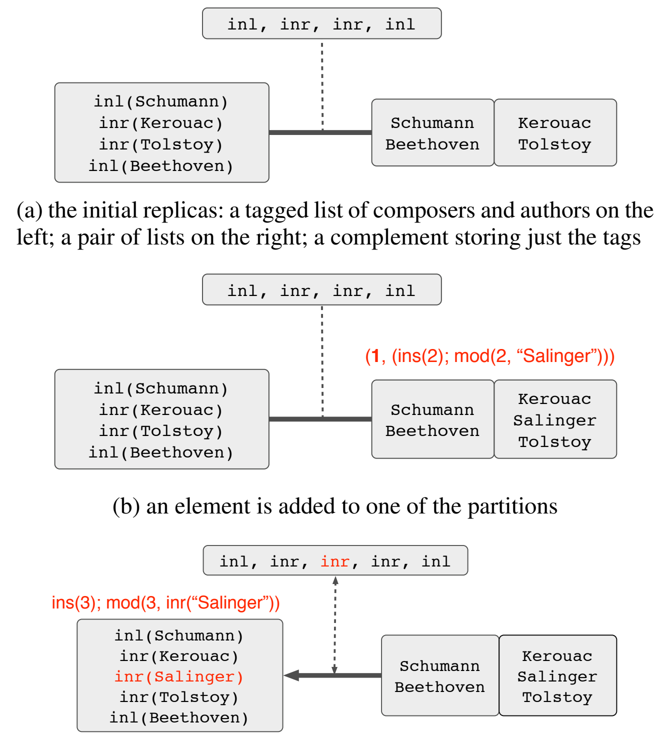
We do not address the question of where these edits come from or who decides, in cases like part (f), which of several possible edits is intended. As argued in [2], answers to these questions will tend to be intertwined with the specifics of particular editing and/or diffing tools and will tend to be messy, heuristic, and domain-specific—unpromising material for a foundational theory. Rather, our aim is to construct a theory that shows how edits, however generated, can be translated between replicas of different shapes.

Abstractly, the lens we are discussing maps between structures of the form $(X \times Y)^*$ and ones of the form $(X \times Z)^*$, where X is the set of composer names, Y the set of date strings, and Z the set of countries. We want to build it compositionally—that is, the whole lens should have the form ℓ^*, where $*$ is a "list mapping" lens combinator and ℓ is a lens for translating edits to a single record—i.e., ℓ is a lens from $X \times Y$ to $X \times Z$. Moreover, ℓ itself should be built as the product $\ell_1 \times \ell_2$ of a lens $\ell_1 \in X \to X$ that translates composer edits verbatim, while ℓ_2 is a "disconnect" lens that maps every edit on either side to a trivial identity edit on the other side.

In analogous fashion, the edit languages for the top-level structures will be constructed compositionally. The set of edits for structures of the form $(X \times Y)^*$, written $\partial((X \times Y)^*)$, will be defined together with the list constructor $*$. Its elements will have the form $\mathsf{ins}(i)$ where i is a position, $\mathsf{del}(i)$, $\mathsf{reorder}(i_1, \ldots, i_n)$ where $i_1, \ldots, i_n$ is a permutation on positions (compactly represented, e.g. as a branching program), and $\mathsf{mod}(p, dv)$, where $dv \in \partial(X \times Y)$ is an edit for $X \times Y$ structures. Pair edits $dv \in \partial(X \times Y)$ have the form $\partial X \times \partial Y$, where ∂X is the set of edits to composers and ∂Y is the set of edits to dates. Finally, both ∂X and ∂Y are sets of primitive "overwrite edits" that completely replace one string with another, together with an identity edit $\mathbf{1}$ that does nothing at all; so ∂X can be just $\{()\} + X$ (with $\mathbf{1} = \mathsf{inl}(())$) and similarly for Y and Z.

Our lens ℓ^* will consist of two components—one for transporting edits from the left side to the right, written $(\ell^*).\!\Rrightarrow \in$

[1] We use the word "synchronize" informally to mean simply "maintain a correspondence between two replicas 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 paper.



(a) the initial replicas: 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 2. A lens with complement.

$\partial(X \times Y)^* \to \partial(X \times Z)^*$,[2] and another for transporting edits from right to left, written $(\ell^*).\Leftarrow \in \partial(X \times Z)^* \to \partial(X \times Y)^*$.

We sometimes need lenses to have a little more structure than this simple example suggests. To see why, consider defining a *partitioning* lens p between the sets $\partial((X+Y)^*)$ and $\partial(X^* \times Y^*)$. Figure 2 demonstrates the behavior of this lens. In part (a), we show the original replicas: on the left, a single list that intermingles authors and composers (with $\mathsf{inl}/\mathsf{inr}$ tags showing which is which), and on the right a pair of homogeneous (untagged) lists, one for authors and one for composers. Now consider an edit, as in (b), that inserts a new element somewhere in the author list on the right. It is clear that we should transport this into an insertion on the left replica, but where, exactly, should we insert it? If the $\Leftarrow$ function is given just an insertion edit for the homogeneous author list and nothing else, there is no way it can translate this edit into a sensible position in the combined list on the left, since it doesn't know how the lists of authors and composers are interleaved on the left.

The solution is to store a small list, called a *complement*, off to the side, recording the *tags* (inl or inr) from the original, intermingled list, and pass this list as an extra argument to translation. We then enrich the types of the edit translation functions to accept a complement and return a new complement, so that $p.\Rightarrow \in \partial((X+Y)^*) \times C \to \partial(X^* \times Y^*) \times C$ and $p.\Leftarrow \in \partial(X^* \times Y^*) \times C \to \partial((X+Y)^*) \times C$. Part (c) demonstrates the use (and update) of the complement when translating the insertion.

Note that the complement stores just the $\mathsf{inl}/\mathsf{inr}$ tags, not the actual names of the authors and composers in the left-hand list. In general, the information stored in C will be much smaller than the information in the replicas; indeed, our earlier example illustrates the common case in which C is the trivial single-element set Unit. The translation functions manipulate just the complements and the edits, which are also small compared to the size of the replicas.

[2] The symbol $\Rightarrow$ is pronounced "put an edit through the lens from left to right," or just "put right." It is the edit-analog of the putr function of the state-based symmetric lenses in [7] and the put function of the state-based asymmetric lenses in [3, 5].

3. Edit Lenses

A key design decision in our formulation of edit lenses is to separate the *description* of edits from the *action* of applying an edit to a state. This separation is captured by the standard mathematical notions of *monoid* and *monoid action*.

3.1 Definition: A *monoid* is a triple $\langle M, \cdot_M, \mathbf{1}_M\rangle$ of a set M, an associative binary operation $\cdot_M \in M \times M \to M$, and a unit element $\mathbf{1}_M \in M$ — that is, with $\cdot_M$ and $\mathbf{1}_M$ such that $x \cdot_M (y \cdot_M z) = (x \cdot_M y) \cdot_M z$ and $\mathbf{1}_M \cdot_M x = x = x \cdot_M \mathbf{1}_M$.

When no confusion results, we use M to denote both the set and the monoid, drop subscripts from $\cdot$ and $\mathbf{1}$, and write mn for $m \cdot n$.

The unit element represents a "change nothing" edit. Multiplication of edits corresponds to packaging up multiple edits into a single one representing their combined effects.

Modeling edits as monoid elements gives us great flexibility in concrete representations. The simplest edit language is a free monoid whose elements are just words over some set of primitive edits and whose multiplication is concatenation. However, it may be useful to put more structure on edits, either (a) to allow more compact representations or (b) to capture the intuition that edits to different parts of a structure do not interfere with each other and can thus be applied in any order. We will see an example of (b) in §6. For a simple example of (a), recall from §2 that, for every set X, we can form an *overwrite* monoid where the edits are just the elements of X together with a fresh unit element—i.e., edits can be represented as elements of the disjoint union $\mathit{Unit} + X$. Combining two edits in this monoid simply drops the second (unless the first is the unit): $\mathsf{inl}(()) \cdot e = e$ and $\mathsf{inr}(x) \cdot e = \mathsf{inr}(x)$. These equations allow this edit language to represent an arbitrarily long sequence of updates using a single element of X (and, *en passant*, to recover state-based lenses as a special case of edit lenses).The monoid framework can also accommodate more abstract notions of edit. For example, the set of all total functions from a set X to itself forms a monoid, where the multiplication operation is function composition. This is essentially the form of edits considered by Stevens [14]. We mostly focus on the simple case where edit languages are free monoids. §6 considers how additional laws can be added to the product and sum lens constructions.

3.2 Definition: Given a monoid M and a set X, a *monoid action* on M and X is a partial function $\odot \in M \times X \rightharpoonup X$ satisfying two laws: $\mathbf{1} \odot x = x$ and $(m \cdot n) \odot x = m \odot (n \odot x)$.

As with monoid multiplication, we often elide the monoid action symbol, writing mx for $m \odot x$. In standard mathematical terminology, a monoid action in our sense might instead be called a "partial monoid action," but since we always work with partial actions we find it convenient to drop the qualifier.

A bit of discussion of partiality is in order. Multiplication of edits is a total operation: given two descriptions of edits, we can always find a description of the composite actions of doing both in sequence. On the other hand, *applying* an edit to a particular state may sometimes fail. This means we need to work with expressions and equations involving partial operations. As usual, any term that contains an undefined application of an operation to operands is undefined—there is no way of "catching" undefinedness. An equation between possibly undefined terms (e.g., as in the definition above) means that if either side is defined then so is the other, and their values are equal (Kleene equality).

Why deal with failure explicitly, rather than keeping edit application total and simply defining our monoid actions so that applying an edit in a state where it is not appropriate yields the same state again (or perhaps some other state)? One reason is that it seems natural to directly address the fact that some edits are not applicable in some states, and to have a canonical outcome in all such cases. A more technical reason is that, when we work with monoids with nontrivial equations, making inapplicable edits behave like the identity is actually wrong.[3]

However, although the framework allows for the possibility of edits failing, we still want to know that the edits produced by our lenses will never actually fail when applied to replica states arising in practice. This requirement, corresponding to the *totality* property of previous presentations of lenses [5], is formalized in Theorem 3.7. In general, we adopt the design principle that partiality should be kept to a minimum; this simplifies the definitions.

It is convenient to bundle a particular choice of monoid and monoid action, plus an initial element, into a single structure:

3.3 Definition: A *module* is a tuple $\langle X,\ init_X,\ \partial X,\ \odot_X\rangle$ comprising a set X, an element $init_X \in X$, a monoid ∂X, and a monoid action $\odot_X$ of ∂X on X.

If X is a module, we refer to its first component by either $|X|$ or just X, and to its last component by $\odot$ or simple juxtaposition.

We will use modules to represent the structures connected by lenses. Before coming to the definition of lenses, however, we need one last ingredient: the notion of a *stateful homomorphism* between monoids. As we saw in §2, there are situations where the information in an edit may be insufficient to determine how it should be translated—we may need to know something more about how the two structures correspond. The exact nature of the extra information needed varies according to the lens. To give lenses a place to store such auxiliary information, we follow [7] and allow the edit-transforming components of a lens (the $\Rrightarrow$ and $\Lleftarrow$ functions) to take a *complement* as an extra input and return an updated complement as an extra output.

3.4 Definition: Given monoids M and N and a *complement set* C, a *stateful monoid homomorphism* from M to N over C is a function $h \in M \times C \to N \times C$ satisfying two laws:

$$\frac{}{h(\mathbf{1}_M, c) = (\mathbf{1}_N, c)}$$

$$\frac{h(m,c) = (n,c') \qquad h(m',c') = (n',c'')}{h(m' \cdot_M m, c) = (n' \cdot_N n, c'')}$$

These are basically just the standard monoid homomorphism laws, except that h is given access to some internal state $c \in C$ that it uses (and updates) when mapping from M to N; in the second law, we must thread the state c' produced by the first h into the second use of h, and we demand that both the result and the effect on the state should be the same whether we send a composite element $m' \cdot m$ through h all at once or in two pieces.

The intended usage of an edit lens is as follows. There are two users, one holding an element of X the other one an element of Y, both referred to hereafter as *replicas*. Initially, they hold $init_X$ and $init_Y$, respectively, and the lens is initialized with complement $\ell.init$. The users then perform actions and propagate them across the lens. An action consists of producing an edit $\mathrm{d}x$ (or $\mathrm{d}y$), applying it to one's current replica x (resp. y), putting the edit through the lens to obtain an edit $\mathrm{d}y$ (resp. $\mathrm{d}x$), and asking the user on the other side to apply $\mathrm{d}y$ ($\mathrm{d}x$) to their replica. In the process, the internal state c of the lens is updated to reflect the new correspondence between the two replicas. We further assume there is some *consistency* relation K between X, Y, and C, which describes the "synchronized states" of the replicas and complement. This gives us a natural way to state the totality requirement discussed above: if we start in a consistent state, make a successful edit (one that does not fail at the initiating side), and put it through the lens, the resulting edit is guaranteed (a) to be applicable on the receiving side and (b) to lead again to a consistent state. We make no guarantees about edits that fail at the initiating side: these should not be put through the lens.

3.5 Definition: A *symmetric edit lens* between modules X and Y consists of a complement set C, a distinguished element $init \in C$, two stateful monoid homomorphisms $\Rrightarrow\ \in \partial X \times C \to \partial Y \times C$ and $\Lleftarrow\ \in \partial Y \times C \to \partial X \times C$, and a ternary *consistency relation* $K \subseteq |X| \times C \times |Y|$ such that

- $(init_X, init, init_Y) \in K$;
- if $(x, c, y) \in K$ and $\mathrm{d}x\ x$ is defined and $\Rrightarrow(\mathrm{d}x, c) = (\mathrm{d}y, c')$, then $\mathrm{d}y\ y$ is also defined and $(\mathrm{d}x\ x, c', \mathrm{d}y\ y) \in K$;
- if $(x, c, y) \in K$ and $\mathrm{d}y\ y$ is defined and $\Lleftarrow(\mathrm{d}y, c) = (\mathrm{d}x, c')$, then $\mathrm{d}x\ x$ is also defined and $(\mathrm{d}x\ x, c', \mathrm{d}y\ y) \in K$.

Since symmetric edit lenses are the main topic of this paper, we will generally write "edit lens" or just "lens" for these, deploying additional adjectives to talk about other variants such as the state-based symmetric lenses of [7].

The intuition about K's role in guaranteeing totality can be formalized as follows.

3.6 Definition: Let $\ell \in X \leftrightarrow Y$ be a lens. A *dialogue* is a sequence of edits—a word in $(\partial X + \partial Y)^*$. The partial function $\ell.run \in (\partial X + \partial Y)^* \rightharpoonup X \times \ell.C \times Y$ is defined by:

$$\frac{}{\ell.run(\varepsilon) = (init_X, \ell.init, init_Y)}$$

$$\frac{\ell.run(w) = (x_0, c, y_0) \qquad \ell.{\Rrightarrow}(\mathrm{d}x_1, c) = (\mathrm{d}y_1, c_1)}{\ell.run(\mathsf{inl}(\mathrm{d}x_1)w) = (\mathrm{d}x_1\ x_0, c_1, \mathrm{d}y_1\ y_0)}$$

$$\frac{\ell.run(w) = (x_0, c, y_0) \qquad \ell.{\Lleftarrow}(\mathrm{d}y_1, c) = (\mathrm{d}x_1, c_1)}{\ell.run(\mathsf{inr}(\mathrm{d}y_1)w) = (\mathrm{d}x_1\ x_0, c_1, \mathrm{d}y_1\ y_0)}$$

3.7 Theorem: Let w be a dialogue and suppose that $\ell.run(w) = (x, c, y)$—in particular, all the edits in w succeed. Let $\mathrm{d}x \in \partial X$ be an edit with $\mathrm{d}x\ x$ defined. If $(\mathrm{d}y, c') = \ell.{\Rrightarrow}(\mathrm{d}x, c)$ then $\mathrm{d}y\ y$ is also defined. An analogous statement holds for $\Lleftarrow$.

Beyond its role in guaranteeing totality, the consistency relation in a lens plays two important roles. First, it is a sanity check on the behavior of $\Rrightarrow$ and $\Lleftarrow$. Second, if we project away the middle component, we can present it to programmers as documentation of the synchronized states of the two replicas—i.e., as a partial *specification* of $\Rrightarrow$ and $\Lleftarrow$.

[3] Here is a slightly contrived example. Suppose that the set of states is natural numbers and that edits have the form $(x \mapsto y)$, where the intended interpretation is that, if the current state is x, then the edit yields state y. It is reasonable to impose the equation $(y \mapsto z) \cdot (x \mapsto y) = (x \mapsto z)$, allowing us to represent sequences of edits in a compact form. But now consider what happens when we apply the edit $(5 \mapsto 7) \cdot (3 \mapsto 5)$ to the state 5. The second monoid action law demands that $((5 \mapsto 7) \cdot (3 \mapsto 5)) \odot 5 = (5 \mapsto 7) \odot ((3 \mapsto 5) \odot 5)$, which, by the equation we imposed, is the same as $(3 \mapsto 7) \odot 5 = (5 \mapsto 7) \odot ((3 \mapsto 5) \odot 5)$. But the left-hand side is equal to 5 (since the edit $(3 \mapsto 7)$ does not apply to the state 5), while the right-hand side is equal to 7 (since the first edit, $(3 \mapsto 5)$, is inapplicable to the state 5, so it behaves like the identity and returns 5 from which $(5 \mapsto 7)$ takes us to 7), so the action law is violated.

One technical issue arising from the definition of edit lenses is that the hidden complements cause many important laws—like associativity of composition—to hold only up to *behavioral equivalence*. This phenomenon was also observed in [7, §3] for the case of symmetric state-based lenses, and the appropriate behavioral equivalence for edit lenses is a natural refinement of the one used there (taking the consistency relations into account).

3.8 Definition [Lens equivalence]: Two lenses $k, \ell : X \leftrightarrow Y$ are *equivalent* (written $k \equiv \ell$) if, for all dialogues w,

- $k.run(w)$ is defined iff $\ell.run(w)$ is defined;
- if $k.run(w) = (x, c, y)$ and $\ell.run(w) = (x', d, y')$, then $x = x'$ and $y = y'$; and
- if $k.run(w) = (x, c, y)$ and $\ell.run(w) = (x', d, y')$ and $\mathrm{d}x\ x$ is defined and $\ell.\Rrightarrow(\mathrm{d}x, c) = (\mathrm{d}y, \_)$ and $k.\Rrightarrow(\mathrm{d}x, d) = (\mathrm{d}y', \_)$ then $\mathrm{d}y = \mathrm{d}y'$, and the analogous property for $\Lleftarrow$.

(Note that the second clause is actually implied by the third.)

Since the complements of the two lenses in question may not even have the same type, it does not make sense to require that they be equal. Instead, the equivalence hides the complements, relying on the observable effects of the lens actions. However, by finding a relationship between the complements, we can prove lens equivalence with a bisimulation-style proof principle:

3.9 Theorem: Lenses $k, \ell : X \leftrightarrow Y$ are equivalent iff there exists a relation $S \subseteq X \times k.C \times \ell.C \times Y$ such that (1) $(init_X, k.init, \ell.init, init_Y) \in S$; (2) if $(x, c, d, y) \in S$ and $\mathrm{d}x\ x$ is defined, then if $(\mathrm{d}y_1, c') = k.\Rrightarrow(\mathrm{d}x, c)$ and $(\mathrm{d}y_2, d') = \ell.\Rrightarrow(\mathrm{d}x, d)$, then $\mathrm{d}y_1 = \mathrm{d}y_2$ and $(\mathrm{d}x\ x, c', d', \mathrm{d}y_1\ y) \in S$; and (3) analogously for $\Lleftarrow$.

4. Edit Lens Combinators

We have proposed a semantic space of edit lenses and justified its design. But the proof of the pudding is in the syntax—in whether we can actually build primitive lenses and lens combinators that live in this semantic space and that do useful things.

Generic Constructions As a first baby step, here is an identity lens that connects identical structures and maps edits by passing them through unchanged.

$$id_X \in X \leftrightarrow X$$

$$\begin{array}{lcl} C & = & Unit \\ K & = & \{(x, (), x) \mid x \in X\} \\ \Rrightarrow(\mathrm{d}x, ()) & = & (\mathrm{d}x, ()) \\ \Lleftarrow(\mathrm{d}x, ()) & = & (\mathrm{d}x, ()) \end{array}$$

Here and below, we elide the definition of the *init* component when $C = Unit = \{()\}$, since it can only be one thing.

In lens definitions like this one, the upper box serves both as a typing rule and as the implicit statement of a theorem saying that the functions in the box below it inhabit the appropriate types and satisfy the corresponding lens laws. For lens combinators, the definition also makes an implicit statement about compatibility with lens equivalence. For brevity, and because they are generally straightforward, we usually elide these theorems.

Now for a more interesting case: Given lenses k and ℓ connecting X to Y and Y to Z, we can build a composite lens $k;\ell$ that connects X directly to Z. Note how the complement of the composite lens includes a complement from each of the components, and how these complements are threaded through the $\Rrightarrow$ and $\Lleftarrow$ operations.

$$\frac{k \in X \leftrightarrow Y \qquad \ell \in Y \leftrightarrow Z}{k;\ell \in X \leftrightarrow Z}$$

$$\begin{array}{lcl} C & = & k.C \times \ell.C \\ init & = & (k.init, \ell.init) \\ K & = & \{(x, (c_k, c_\ell), z) \mid \\ & & \quad \exists y.\ (x, c_k, y) \in k.K \\ & & \quad \wedge (y, c_\ell, z) \in \ell.K\ \} \\ \Rrightarrow(\mathrm{d}x, (c_k, c_\ell)) & = & \mathsf{let}\ (\mathrm{d}y, c_k') = k.\Rrightarrow(\mathrm{d}x, c_k)\ \mathsf{in} \\ & & \mathsf{let}\ (\mathrm{d}z, c_\ell') = \ell.\Rrightarrow(\mathrm{d}y, c_\ell)\ \mathsf{in} \\ & & (\mathrm{d}z, (c_k', c_\ell')) \\ \Lleftarrow(\mathrm{d}z, (c_k, c_\ell)) & = & \mathsf{let}\ (\mathrm{d}y, c_\ell') = \ell.\Lleftarrow(\mathrm{d}z, c_\ell)\ \mathsf{in} \\ & & \mathsf{let}\ (\mathrm{d}x, c_k') = k.\Lleftarrow(\mathrm{d}y, c_k)\ \mathsf{in} \\ & & (\mathrm{d}x, (c_k', c_\ell')) \end{array}$$

As might be expected, composition of lenses is associative, and the identity lens is a unit for composition. However, as mentioned above, we need to be a little careful: it is not quite the case that $(k;\ell);m = k;(\ell;m)$—in particular they have different complements. Instead, what we can show is that $(k;\ell);m \equiv k;(\ell;m)$.

Another simple lens combinator is dualization: for each lens $\ell \in X \leftrightarrow Y$, we can construct its dual, $\ell^{op} \in Y \leftrightarrow X$, by swapping $\Rrightarrow$ and $\Lleftarrow$.

For the next definition, observe that the set $Unit$ gives rise to a trivial monoid structure and, for any given set X and element $x \in X$, a trivial module with initial element x, which we write $Unit_{x \in X}$. When context clearly calls for a module, we will abbreviate $Unit_{() \in Unit}$ to simply $Unit$.

Now, for each module X, there is a *terminal lens* that connects X to the trivial $Unit$ module by throwing away all edits.

$$term_X \in X \leftrightarrow Unit$$

$$\begin{array}{lcl} C & = & Unit \\ K & = & X \times Unit \times Unit \\ \Rrightarrow(\mathrm{d}x, ()) & = & (\mathbf{1}, ()) \\ \Lleftarrow(\mathbf{1}, ()) & = & (\mathbf{1}, ()) \end{array}$$

The $disconnect$ lens that we saw in §2 can be built from $term$. The $term$ lens is also unique (up to equivalence): the implementation of $\Rrightarrow$ is forced by the size of its range monoid $Unit$, and the implementation of $\Lleftarrow$ is forced by the homomorphism laws.

There is a trivial lens between any two isomorphic modules. Formally, a *module homomorphism* (f, h) between modules X and Y is a function $f \in X \to Y$ and a monoid homomorphism $h \in \partial X \to \partial Y$ such that $f(init_X) = init_Y$ and $f(\mathrm{d}x\, x) = h(\mathrm{d}x)\, f(x)$. There is an identity $(\lambda x.\, x, \lambda \mathrm{d}x.\, \mathrm{d}x)$ for every module, and the point-wise composition of module homomorphisms is also a homomorphism, so modules form a category. If module homomorphisms $(e, g) \in X \to Y$ and $(f, h) \in Y \to X$ satisfy $(e, g);(f, h) = id_X$ and $(f, h);(e, g) = id_Y$, then (e, g) is an *isomorphism* and (f, h) is *inverse* to (e, g). Now:

$$\frac{(f, h) \in X \to Y \qquad (f, h) \text{ is inverse to } (f^{-1}, h^{-1})}{iso_{(f,h)} \in X \leftrightarrow Y}$$

$$\begin{array}{lcl} C & = & Unit \\ K & = & \{(x, (), f(x)) \mid x \in X\} \\ \Rrightarrow(\mathrm{d}x, ()) & = & (h(\mathrm{d}x), ()) \\ \Lleftarrow(\mathrm{d}y, ()) & = & (h^{-1}(\mathrm{d}y), ()) \end{array}$$

The fact that this always defines a lens, plus a couple of other easy facts, amounts to saying that there is a functor from the category of module isomorphisms to the category of edit lenses.

Generators for free monoids For writing practical lenses, we want not only generic combinators like the ones presented above, but also more specific lenses for structured data such as products, sums, and lists. We show in the rest of this section how to define simple versions of these constructors whose associated edit monoids are freely generated. §5 shows how to generalize the list mapping lens to other forms of containers, and §6 discusses edit languages with nontrivial laws.

Given a set G, we write G^* for the set of finite sequences of elements of G. We write ε for the empty sequence and g to denote both a generator element and the single-element sequence containing such an element. Sequence concatenation is denoted by juxtaposition; when discussing a sequence $g_1 \cdots g_n$, we also use g to refer to the entire sequence. The notation $|g|$ means the length of a sequence: $|g_1 \cdots g_n| = n$. It is easy to show that G^* together with sequence concatenation and ε forms a monoid.

It is often convenient to specify the behavior of a monoid homomorphism by giving its output on each generator. Given a function $f_g \in G \to M$ on generators, the monoid homomorphism $f \in G^* \to M$ is defined by $f(\varepsilon) = \mathbf{1}$ and $f(g_1 \cdots g_n) = f_g(g_1)f(g_2 \cdots g_n)$. Similarly, given a stateful function $f_g \in G \times C \to M \times C$, we can define a stateful monoid homomorphism $f \in G^* \times C \to M \times C$ by setting $f(\varepsilon, c) = (\mathbf{1}, c)$ and

$$\begin{aligned} f(g_1 \cdots g_n, c) = {} & \mathsf{let}\ (m', c') = f(g_2 \cdots g_n, c)\ \mathsf{in} \\ & \mathsf{let}\ (m'', c'') = f_g(g_1, c')\ \mathsf{in} \\ & (m''m', c''). \end{aligned}$$

Tensor Product Given modules X and Y, a primitive edit to a pair in $|X| \times |Y|$ is either an edit to the X part or an edit to the Y part.

$$G^{\otimes}_{X,Y} = \{\mathsf{left}(\mathrm{d}x) \mid \mathrm{d}x \in \partial X\} \cup \{\mathsf{right}(\mathrm{d}y) \mid \mathrm{d}y \in \partial Y\}$$

We can turn these generators into a module by giving specifying a monoid action for the free monoid $(G^{\otimes}_{X,Y})^*$:

$$\begin{aligned} \mathsf{left}(\mathrm{d}x) \odot_g (x, y) &= (\mathrm{d}x\ x, y) \\ \mathsf{right}(\mathrm{d}y) \odot_g (x, y) &= (x, \mathrm{d}y\ y) \end{aligned}$$

The full module is then given by

$$X \otimes Y = \left\langle |X| \times |Y|, (\mathit{init}_X, \mathit{init}_Y), (G^{\otimes}_{X,Y})^*, \odot \right\rangle.$$

Now we can build a lens that "runs two sub-lenses in parallel" on the components of a product module. The $\Rrightarrow$ and $\Lleftarrow$ functions are defined via stateful monoid homomorphism specifications.

$$\frac{k \in X \leftrightarrow Z \qquad \ell \in Y \leftrightarrow W}{k \otimes \ell \in X \otimes Y \leftrightarrow Z \otimes W}$$

$$\begin{array}{lcl} C & = & k.C \times \ell.C \\ \mathit{init} & = & (k.\mathit{init}, \ell.\mathit{init}) \\ K & = & \{\ ((x,z),(c_k,c_\ell),(y,w)) \mid \\ & & \quad (x, c_k, y) \in k.K \\ & & \quad \wedge (z, c_\ell, w) \in \ell.K\ \} \\ \Rrightarrow_g(\mathsf{left}(\mathrm{d}x), (c_k, c_\ell)) & = & \mathsf{let}\ (\mathrm{d}z, c_k') = k.\Rrightarrow(\mathrm{d}x, c_k)\ \mathsf{in} \\ & & (\mathsf{left}(\mathrm{d}z), (c_k', c_\ell)) \\ \Rrightarrow_g(\mathsf{right}(\mathrm{d}y), (c_k, c_\ell)) & = & \mathsf{let}\ (\mathrm{d}w, c_\ell') = \ell.\Rrightarrow(\mathrm{d}y, c_\ell)\ \mathsf{in} \\ & & (\mathsf{right}(\mathrm{d}w), (c_k, c_\ell')) \\ \Lleftarrow_g \text{ similarly} & & \end{array}$$

$$\frac{k \in X \leftrightarrow Y \qquad \ell \in Z \leftrightarrow W}{k \oplus \ell \in X \oplus Z \leftrightarrow Y \oplus W}$$

$$\begin{array}{lcl} C & = & k.C + \ell.C \\ \mathit{init} & = & \mathsf{inl}(k.\mathit{init}) \\ K & = & \{(\mathsf{inl}(x), \mathsf{inl}(c), \mathsf{inl}(y)) \\ & & \quad \mid (x, c, y) \in k.K\} \\ & \cup & \{(\mathsf{inr}(z), \mathsf{inr}(c), \mathsf{inr}(w)) \\ & & \quad \mid (z, c, w) \in \ell.K\} \\ c_k & = & k.\mathit{init} \\ c_\ell & = & \ell.\mathit{init} \\ \Rrightarrow_g(\mathsf{switch}_{LL}(\mathrm{d}x), \mathsf{inl}(c)) & = & \mathsf{let}\ (\mathrm{d}y, c') = k.\Rrightarrow(\mathrm{d}x, c_k) \\ & & \mathsf{in}\ (\mathsf{switch}_{LL}(\mathrm{d}y), \mathsf{inl}(c')) \\ \Rrightarrow_g(\mathsf{switch}_{RL}(\mathrm{d}x), \mathsf{inr}(c)) & = & \mathsf{let}\ (\mathrm{d}y, c') = k.\Rrightarrow(\mathrm{d}x, c_k) \\ & & \mathsf{in}\ (\mathsf{switch}_{RL}(\mathrm{d}y), \mathsf{inl}(c')) \\ \Rrightarrow_g(\mathsf{switch}_{LR}(\mathrm{d}z), \mathsf{inl}(c)) & = & \mathsf{let}\ (\mathrm{d}w, c') = \ell.\Rrightarrow(\mathrm{d}z, c_\ell) \\ & & \mathsf{in}\ (\mathsf{switch}_{LR}(\mathrm{d}w), \mathsf{inr}(c')) \\ \Rrightarrow_g(\mathsf{switch}_{RR}(\mathrm{d}z), \mathsf{inr}(c)) & = & \mathsf{let}\ (\mathrm{d}w, c') = \ell.\Rrightarrow(\mathrm{d}z, c_\ell) \\ & & \mathsf{in}\ (\mathsf{switch}_{RR}(\mathrm{d}w), \mathsf{inr}(c')) \\ \Rrightarrow_g(\mathsf{stay}_L(\mathrm{d}x), \mathsf{inl}(c)) & = & \mathsf{let}\ (\mathrm{d}y, c') = k.\Rrightarrow(\mathrm{d}x, c) \\ & & \mathsf{in}\ (\mathsf{stay}_L(\mathrm{d}y), \mathsf{inl}(c')) \\ \Rrightarrow_g(\mathsf{stay}_R(\mathrm{d}z), \mathsf{inr}(c)) & = & \mathsf{let}\ (\mathrm{d}w, c') = \ell.\Rrightarrow(\mathrm{d}z, c) \\ & & \mathsf{in}\ (\mathsf{stay}_R(\mathrm{d}w), \mathsf{inr}(c')) \\ \Rrightarrow_g(e, c) & = & (\mathsf{fail}, c) \text{ in all other cases} \\ \Lleftarrow_g & & \text{is analogous} \end{array}$$

Figure 3: The sum lens

4.1 Theorem:

- $k \otimes \ell$ is indeed a lens.
- If $k \equiv k'$ and $\ell \equiv \ell'$, then $k \otimes \ell \equiv k' \otimes \ell'$.
- $\mathit{id} \otimes \mathit{id} \equiv \mathit{id}$.
- $(k \otimes \ell); (k' \otimes \ell') \equiv (k; k') \otimes (\ell; \ell')$.
- $((k \otimes \ell) \otimes m); \mathit{iso}_{\mathit{assoc}} \equiv \ell \otimes (k \otimes m)$, where assoc is the isomorphism between $(X \otimes Y) \otimes Z$ and $X \otimes (Y \otimes Z)$ for all X, Y, Z.
- $(k \otimes \ell); \mathit{iso}_{\mathit{swap}} \equiv \ell \otimes k$, where swap is the isomorphism between $X \times Y$ and $Y \times X$.

Proof: For the first statement (being a good lens), first note that preservation of monoid multiplication is immediate since $\partial(X \otimes Y)$ is free. It remains to show that the consistency relation of $k \otimes \ell$ is preserved and guarantees definedness. This is direct from the definition and the assumption that k and ℓ are lenses.

The remaining statements are direct consequences of the definitions, together with Theorem 3.9; for example, the third equivalence can be witnessed by the simulation relation

$$\begin{aligned} \{((x,y), ((c,d),(c',d')), ((c,c'),(d,d')), (x'',y'')) \mid \\ \exists (x',y').\ (x,c,x') \in k.K \wedge (x',c',x'') \in k'.K \\ \wedge (y,d,y') \in \ell.K \wedge (y',d',y'') \in \ell'.K\}. \end{aligned} \qquad \square$$

As in [7], the tensor construction is not quite a full categorical product, because duplicating information does not give rise to a well-behaved lens—there is no lens with type $X \leftrightarrow X \otimes X$ that satisfies all the equivalences a lens programmer would want.

However, tensor product does yield various symmetric monoidal categories of edit lenses; for lack of space we omit the details.

Sum We now present one way (not the only one—see footnote 4) of constructing a sum module and a sum lens. Given sets of edits ∂X and ∂Y, we can describe the generators for the free monoid of edits to a sum by:

$$\begin{array}{rcl} G^{\oplus}_{X,Y} & = & \{\mathsf{switch}_{iL}(\mathrm{d}x) \mid i \in \{L,R\}, \mathrm{d}x \in \partial X\} \\ & \cup & \{\mathsf{switch}_{iR}(\mathrm{d}y) \mid i \in \{L,R\}, \mathrm{d}y \in \partial Y\} \\ & \cup & \{\mathsf{stay}_L(\mathrm{d}x) \mid \mathrm{d}x \in \partial X\} \cup \{\mathsf{stay}_R(\mathrm{d}y) \mid \mathrm{d}y \in \partial Y\} \\ & \cup & \{\mathsf{fail}\} \end{array}$$

The idea is that edits to a sum can either change just the content or change the tag (and therefore necessarily also the content, which is superseded by the given new content). That is, we want the "atoms" of the edit language to express the operations of editing content and switching sides. This gives us the switch_{LR}, switch_{RL}, and stay edits. For present purposes, we could leave it at this and define the monoid of edits to be the free monoid over just these generators. However, in Section 6 we will introduce a more compact representation that allows multiple edits to be combined into one, and this representation will give rise to the other two switch operations; for example, switch_{LL} represents a switch_{LR} followed by a switch_{RL}. To avoid having two similar but subtly different definitions, we include these edits here in the basic generators as well. Finally, we introduce an always-failing edit to represent sequences of edits that are internally inconsistent—e.g., a switch to the left side followed by an attempt to apply an edit which stays on the right side. These intuitions are formalized in the application function:

$$\begin{array}{rcl} \mathsf{switch}_{LL}(\mathrm{d}x) \odot_g \mathsf{inl}(x) & = & \mathsf{inl}(\mathrm{d}x\; \mathit{init}_X) \\ \mathsf{switch}_{LR}(\mathrm{d}y) \odot_g \mathsf{inl}(x) & = & \mathsf{inr}(\mathrm{d}y\; \mathit{init}_Y) \\ \mathsf{switch}_{RL}(\mathrm{d}x) \odot_g \mathsf{inr}(y) & = & \mathsf{inl}(\mathrm{d}x\; \mathit{init}_X) \\ \mathsf{switch}_{RR}(\mathrm{d}y) \odot_g \mathsf{inr}(y) & = & \mathsf{inr}(\mathrm{d}y\; \mathit{init}_Y) \\ \mathsf{stay}_L(\mathrm{d}x) \odot_g \mathsf{inl}(x) & = & \mathsf{inl}(\mathrm{d}x\; x) \\ \mathsf{stay}_R(\mathrm{d}y) \odot_g \mathsf{inr}(y) & = & \mathsf{inr}(\mathrm{d}y\; y) \\ e \odot_g v & & \text{undefined in all other cases} \end{array}$$

We then define the sum of modules X and Y as

$$X \oplus Y = \left\langle |X| + |Y|, \mathsf{inl}(\mathit{init}_X), (G^{\oplus}_{X,Y})^*, \odot \right\rangle.$$

We now wish to give a lens combinator $k \oplus \ell$ that runs lens k on the parts of edits that apply to inl values and ℓ on the parts of edits that apply to inr values.[4] Figure 3 shows the full definition.

4.2 Theorem: When k and ℓ are lenses, so is $k \oplus \ell$.

Proof: The homomorphism laws are again trivial. We must show that the consistency relation K is maintained. We have

$$\begin{array}{rl} & (\mathit{init}_{X\oplus Z}, \mathit{init}, \mathit{init}_{Y\oplus W}) \\ = & (\mathsf{inl}(\mathit{init}_X), \mathsf{inl}(k.\mathit{init}), \mathsf{inl}(\mathit{init}_Y)) \in K, \end{array}$$

since $(\mathit{init}_X, k.\mathit{init}, \mathit{init}_Y) \in k.K$. So it remains to show that that $\Rrightarrow$ and $\Lleftarrow$ preserve this relation. We need only consider the case where we begin with an arbitrary consistent triple $(\mathsf{inl}(x), \mathsf{inl}(c), \mathsf{inl}(y)) \in K$ and $\mathrm{d}v \in X \oplus Z$ for which $\mathrm{d}v\,\mathsf{inl}(x)$ is defined. (The cases where the triple is of the form $(\mathsf{inr}(x), \mathsf{inr}(c), \mathsf{inr}(y)) \in K$ are similar, swapping k and ℓ in some places; the cases where we are considering a $\mathrm{d}v \in Y \oplus W$ are similar, but use $\Lleftarrow$ instead of $\Rrightarrow$ everywhere.) Since $\mathrm{d}v\,\mathsf{inl}(x)$ is defined, there are three forms of $\mathrm{d}v$ to consider: $\mathsf{switch}_{LL}(\mathrm{d}x)$, $\mathsf{switch}_{LR}(\mathrm{d}z)$, and $\mathsf{stay}_L(\mathrm{d}x)$. Here is the most interesting case:

Case $\mathrm{d}v = \mathsf{switch}_{LL}(\mathrm{d}x)$: We define $(\mathrm{d}y, c') = k.\mathit{putr}(\mathrm{d}x, k.\mathit{init})$ and $(x', y') = (\mathrm{d}x\; \mathit{init}_X, \mathrm{d}y\; \mathit{init}_Y)$. Since k is a lens, we know $(\mathit{init}_X, k.\mathit{init}, \mathit{init}_Y) \in k.K$ and therefore that $(x', c', y') \in k.K$. This means $(\mathsf{inl}(x'), \mathsf{inl}(c'), \mathsf{inl}(y')) \in K$. Since $(k \oplus \ell).\Rrightarrow(\mathrm{d}v, \mathsf{inl}(c)) = (\mathsf{switch}_{LL}(\mathrm{d}y), \mathsf{inl}(c'))$ and $\mathrm{d}v\,\mathsf{inl}(x) = \mathsf{inl}(x')$ and $\mathsf{switch}_{LL}(\mathrm{d}y)\,\mathsf{inl}(y) = \mathsf{inl}(y')$, this shows that K is preserved in this case. $\square$

Like the tensor product, this lens combinator is a bifunctor: $id \oplus id \equiv id$ and $(k \oplus \ell); (k' \oplus \ell') \equiv (k; k') \oplus (\ell; \ell')$.

List module Next, let us consider lists. Given a module X, we define the basic edits for lists over $|X|$ to include in-place modifications, insertions, deletions, and reorderings:

$$\begin{array}{rl} G^{\mathrm{list}}_X = & \{\mathsf{mod}(p, \mathrm{d}x) \mid p \in \mathbb{N}^+, \mathrm{d}x \in \partial X\} \\ \cup & \{\mathsf{ins}(i) \mid i \in \mathbb{N}\} \;\cup\; \{\mathsf{del}(i) \mid i \in \mathbb{N}\} \\ \cup & \{\mathsf{reorder}(f) \mid \forall i \in \mathbb{N}. f(i) \text{ permutes } \{1, \ldots, i\}\} \\ \cup & \{\mathsf{fail}\} \end{array}$$

For compatibility with the generalization to arbitrary containers in §5, we slightly change the behavior of these operations from what we saw in §2. Insertions and deletions are now always performed at the end of the list; to insert in the middle of the list, you first insert at the end, then reorder the list. The argument i to $\mathsf{ins}(i)$ and $\mathsf{del}(i)$ now specifies how *many* elements to insert or delete.

$$\begin{array}{rcl} \mathsf{mod}(p, \mathrm{d}x) \odot_g x_1 \cdots x_n & = & x_1 \cdots x_{p-1}\; (\mathrm{d}x\; x_p)\; x_{p+1} \cdots x_n \\ \mathsf{ins}(i) \odot_g x_1 \cdots x_n & = & x_1 \cdots x_n \underbrace{\mathit{init}_X \cdots \mathit{init}_X}_{i \text{ times}} \\ \mathsf{del}(i) \odot_g x_1 \cdots x_n & = & x_1 \cdots x_{n-i} \\ \mathsf{reorder}(f) \odot_g x_1 \cdots x_n & = & x_{f(n)(1)} \cdots x_{f(n)(n)} \\ \mathsf{fail} \odot_g x_1 \cdots x_n & & \text{undefined} \end{array}$$

We take $\mathsf{mod}(p, \mathrm{d}x) \odot_g x$ to be undefined when $p > |x|$, and similarly take $\mathsf{del}(i) \odot_g x$ to be undefined when $i > |x|$. The list module is then $X^* = \langle |X|^*, \varepsilon, (G^{\mathrm{list}}_X)^*, \odot \rangle$.

Mapping lens The list mapping lens ℓ^* uses ℓ to translate mod edits from X to Y and vice versa (Figure 4). Other kinds of edits (ins, del, and $\mathsf{reorder}$) are carried across unchanged. The notation $c[p \mapsto c'_p]$ in the rule for mod edits means "the list that is just like c except that the element in position p is replaced by c'_p." When translating non-modification edits, we update the

$$\frac{\ell \in X \leftrightarrow Y}{\ell^* \in X^* \leftrightarrow Y^*}$$

$$\begin{array}{lcl} C & = & \ell.C^* \\ \mathit{init} & = & \varepsilon \\ K & = & \{(x, c, y) \mid |x| = |c| = |y| \wedge \\ & & \quad \forall 1{\le}p{\le}|x|.\ (x_p, c_p, y_p) \in \ell.K\} \\ \Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), c) & = & \mathsf{let}\ (\mathrm{d}y, c'_p) = \ell.{\Rrightarrow}(\mathrm{d}x, c_p)\ \mathsf{in} \\ & & (\mathsf{mod}(p, \mathrm{d}y), c[p \mapsto c'_p])) \\ & & \quad \text{when } p \le n \\ \Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), c) & = & (\mathsf{fail}, c) \text{ when } p > n \\ \Rrightarrow_g(\mathsf{fail}, c) & = & (\mathsf{fail}, c) \\ \Rrightarrow_g(\mathrm{d}x, c) & = & (\mathrm{d}x, \mathrm{d}x\; c) \text{ in all other cases} \\ \Lleftarrow & & \text{similar} \end{array}$$

Figure 4: The list mapping lens

[4] In [7], there is some discussion regarding "forgetful" and "retentive" sum lenses, with the distinction revolving around what to do with the complement when an edit switches between sides of the sum. For state-based lenses, lenses on recursive structures like lists were given in terms of lenses on the non-recursive structure, and the retentive sum lens gave rise to a retentive list mapping lens whereas the forgetful sum lens gave rise to a forgetful list mapping lens. The poor alignment strategies given in that paper were mediated somewhat by the retentive map's ability to use complements from previous versions of a list, making retentive sums somewhat more attractive than forgetful ones. In this presentation, however, the mapping lens has much better alignment information, so we eschew the more complicated retentive lenses in favor of simpler forgetful versions.

$$partition \in (X \oplus Y)^* \leftrightarrow X^* \otimes Y^*$$

$$\begin{array}{lll}
C & = & \{L, R\}^* \\
init & = & \varepsilon \\
K & = & \{(z, \mathsf{map}_{\mathsf{tagof}}(z), (\mathsf{lefts}(z), \mathsf{rights}(z))) \mid z \in (|X| + |Y|)^*\}
\end{array}$$

$$\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}v), c) = (\mathsf{fail}, c) \text{ when } p > |c| \tag{1}$$

$$\Rrightarrow_g(\mathsf{mod}(p, \varepsilon), c) = (\varepsilon, c) \text{ when } 1 \le p \le |c| \tag{2}$$

$$\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}v\mathrm{d}vs), c) = (d'\,d, c'') \text{ where } \begin{array}{ll} 1 < n & (d, c') = \Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}vs), c) \\ 1 \le p \le |c| & (d', c'') = \Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}v), c') \end{array} \tag{3}$$

$$\Rrightarrow_g(\mathsf{mod}(p, \mathsf{switch}_{jk}(\mathrm{d}v)), c) = (d_2 d_1 d_0, c[p \mapsto k]), \text{ where } \begin{array}{ll} (p_L, p_R) = \mathsf{count}(p, c) & d_0 = \mathsf{map}_{\lambda d.\,\mathsf{tag}(j,d)}(\mathsf{del}'(p_j)) \\ d_2 = \mathsf{tag}(k, \mathsf{mod}(p_k, \mathrm{d}v)) & d_1 = \mathsf{map}_{\lambda d.\,\mathsf{tag}(k,d)}(\mathsf{ins}'(p_k)) \end{array} \tag{4}$$

$$\Rrightarrow_g(\mathsf{mod}(p, \mathsf{stay}_j(\mathrm{d}v)), c) = (\mathsf{tag}(j, \mathsf{mod}(p_j, \mathrm{d}v)), c), \text{ where } (p_L, p_R) = \mathsf{count}(p, c) \tag{5}$$

$$\Rrightarrow_g(\mathsf{mod}(p, \mathsf{fail}), c) = (\mathsf{fail}, c) \tag{6}$$

$$\Rrightarrow_g(\mathsf{ins}(i), c) = (\mathsf{left}(\mathsf{ins}(i)), \mathsf{ins}(i)\ c) \tag{7}$$

$$\Rrightarrow_g(\mathsf{del}(i), c) = (d_1 d_0, \mathsf{del}(i)\ c), \text{ where } \begin{array}{ll} c' = \mathsf{reverse}(c) & d_0 = \mathsf{left}(\mathsf{del}(n_L{-}1)) \\ (n_L, n_R) = \mathsf{count}(i{+}1, c') & d_1 = \mathsf{right}(\mathsf{del}(n_R{-}1)) \end{array} \tag{8}$$

$$\Rrightarrow_g(\mathsf{reorder}(f), c) = (d_L d_R, c'), \text{ where } \begin{array}{ll} h = \mathsf{iso}(c) & c' = \mathsf{reorder}(f)\ c \\ h' = \mathsf{iso}(c') & (n_L, n_R) = \mathsf{count}(|c|, c) \\ h'' = h'^{-1}; f(|c|); h & f_k(n \ne n_k) = \lambda p.\, p \\ d_L = \mathsf{left}(\mathsf{reorder}(f_L)) & f_L(n_L) = \mathsf{inl}; h''; \mathsf{out} \\ d_R = \mathsf{right}(\mathsf{reorder}(f_R)) & f_R(n_R) = \mathsf{inr}; h''; \mathsf{out} \end{array} \tag{9}$$

$$\Rrightarrow_g(\mathsf{fail}, c) = (\mathsf{fail}, c) \tag{10}$$

$$\Lleftarrow_g(\varepsilon, c) = (\varepsilon, c) \tag{11}$$

$$\Lleftarrow_g(\mathrm{d}v\mathrm{d}vs, c) = (d'd, c'') \text{ when } n > 1, \text{ where } (d, c') = \Lleftarrow_g(\mathrm{d}vs, c) \quad (d', c'') = \Lleftarrow_g(\mathrm{d}v, c') \tag{12}$$

$$\Lleftarrow_g(\mathsf{left}(\mathsf{mod}(p, \mathrm{d}x)), c) = (\mathsf{stay}_L(\mathsf{mod}(p', \mathrm{d}x)), c), \text{ where } p' = \mathsf{iso}(c)^{-1}(\mathsf{inl}(p)) \tag{13}$$

$$\Lleftarrow_g(\mathsf{left}(\mathsf{reorder}(f)), c) = (\mathsf{reorder}(f'), c), \text{ where } \begin{array}{ll} g(\mathsf{inr}(p)) = \mathsf{inr}(p) & f'(n \ne |c|) = \lambda p.\, p \\ g(\mathsf{inl}(p)) = \mathsf{inl}(f(n_L)(p)) & f'(|c|) = h; g; h^{-1} \\ (n_L, n_R) = \mathsf{count}(|c|, c) & h = \mathsf{iso}(c) \end{array} \tag{14}$$

$$\Lleftarrow_g(\mathsf{left}(\mathsf{ins}(i)), c) = (\mathsf{ins}(i), \mathsf{ins}(i)\ c) \tag{15}$$

$$\Lleftarrow_g(\mathsf{left}(\mathsf{del}(0)), c) = (\varepsilon, c) \tag{16}$$

$$\Lleftarrow_g(\mathsf{left}(\mathsf{del}(i)), c) = (d''\ \mathsf{del}'(p), c''), \text{ where } \begin{array}{ll} h = \mathsf{iso}(c) & (n_L, n_R) = \mathsf{count}(|c|, c) \\ p = h^{-1}(\mathsf{inl}(n_L)) & (d'', c'') = \Lleftarrow_g(d', c') \\ c' = \mathsf{del}'(p)\ c & d' = \mathsf{left}(\mathsf{del}(i{-}1)) \end{array} \text{ when } 0 < i \le n_L{+}1 \tag{17}$$

$$\Lleftarrow_g(\mathsf{left}(\mathsf{del}(i)), c) = (\mathsf{fail}, c) \text{ otherwise} \tag{18}$$

$$\Lleftarrow_g(\mathsf{left}(\mathsf{fail}), c) = (\mathsf{fail}, c) \tag{19}$$

$$\Lleftarrow_g(\mathsf{right}(\mathrm{d}y), c) \quad \text{similar}$$

Figure 5: The *partition* lens

$$\begin{array}{ll}
\mathsf{tagof}(\mathsf{inl}(x)) = L & \mathsf{map}_f(\varepsilon) = \varepsilon \\
\mathsf{tagof}(\mathsf{inr}(y)) = R & \mathsf{map}_f(cw) = f(c)\,\mathsf{map}_f(w) \\
\mathsf{lefts}(\varepsilon) = \varepsilon & \mathsf{rights}(\varepsilon) = \varepsilon \\
\mathsf{lefts}(\mathsf{inl}(x)w) = x\,\mathsf{lefts}(w) & \mathsf{rights}(\mathsf{inl}(x)w) = \mathsf{rights}(w) \\
\mathsf{lefts}(\mathsf{inr}(y)w) = \mathsf{lefts}(w) & \mathsf{rights}(\mathsf{inr}(y)w) = y\,\mathsf{rights}(w) \\
\mathsf{tag}(L, \mathrm{d}x) = \mathsf{left}(\mathrm{d}x) & \mathsf{out}(\mathsf{inl}(x)) = x \\
\mathsf{tag}(R, \mathrm{d}y) = \mathsf{right}(\mathrm{d}y) & \mathsf{out}(\mathsf{inr}(y)) = y \\
\mathsf{count}(p, \varepsilon) = (1, 1) & \mathsf{count}(1, c) = (1, 1)
\end{array}$$

$$\mathsf{count}(p, c_1 \cdots c_n) = \mathsf{let}\ (n_L, n_R) = \mathsf{count}(p{-}1, c_2 \cdots c_n)\ \mathsf{in} \begin{cases} (n_L + 1, n_R) & c = L \\ (n_L, n_R + 1) & c = R \end{cases}$$

$$\mathsf{cycle}_p(n)(m) = \begin{cases} p & p < m = n \\ m + 1 & p \le m < n \\ m & \text{otherwise} \end{cases}$$

$$\mathsf{reverse}(c_1 \cdots c_n) = c_n c_{n-1} \cdots c_1$$

$$\mathsf{del}'(p) = \mathsf{del}(1)\,\mathsf{reorder}(\mathsf{cycle}_p)$$

$$\mathsf{ins}'(p) = \mathsf{reorder}(\lambda n.\ \mathsf{cycle}_p(n)^{-1})\,\mathsf{ins}(1)$$

$$\mathsf{iso}(c) = \lambda p.\ \mathsf{let}\ (n_L, n_R) = \mathsf{count}(p, c)\ \mathsf{in} \begin{cases} \mathsf{inl}(n_L) & c_p = L \\ \mathsf{inr}(n_R) & c_p = R \end{cases}$$

Figure 6: Supplementary functions for *partition*

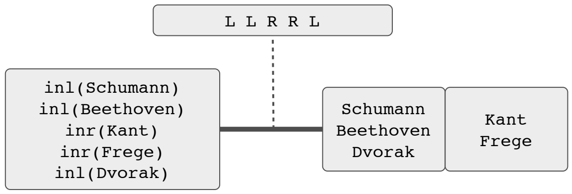


Figure 7. A consistent triple for the partition lens.

complement in a way almost identical to the way the two replicas are updated; to reflect this similarity, we use edit application from the $Unit^*_{\ell.init \in \ell.C}$ module to define the new complement.

Partition lens Figures 5 and 6 give the definition of a list partitioning lens that (as we saw in §2) separates a list of tagged elements into those tagged inl and those tagged inr. We write fail to stand for $\mathsf{left}(\mathsf{fail})\mathsf{right}(\mathsf{fail})$ when defining $\Rrightarrow_g$. Additionally, as with the mapping lens, we consider the complement to belong to a module; this time, to the module $Unit^*_{L \in \{L,R\}}$.

These figures may be a bit intimidating at first, but there is nothing very deep going on—just some everyday functional programming over lists. To illustrate how it all works, let's consider a few example invocations of the $partition$ lens. Each of them begins with the consistent triple illustrated in Figure 7. Note that only the middle part—the complement—is actually available to the partition lens as it runs: its other input is just an edit.

As a warm-up, consider a simple edit: changing Dvorak's name to Dvořák (with correct diacritics) in the left repository. The edit describing this has the form $\mathsf{mod}(5, \mathsf{stay}_L(\mathrm{d}n))$, where $\mathrm{d}n$ describes the string edit to the name. To translate this edit, we first need to translate the index 5 to an index into the list of composers in the right-hand repository (line 5 in Figure 5). We can do this by simply counting how many composers appear up to and including Dvorak, that is, how many L values appear in the complement list up to index 5—in this case, 3. We then wrap this index up, along with the $\mathrm{d}n$ edit, in a new edit of the form $\mathsf{left}(\mathsf{mod}(3, \mathrm{d}n))$; the complement need not change because we have not changed the structure of the lists. This pattern—count to translate the index, then re-tag the edit appropriately—can be generalized to all modifications that stay on the same side of the sum; the count and tag functions defined in Figure 6 implement these two steps.

The left-to-right translation of other in-place modifications, insertions, and deletions and the right-to-left translation of in-place modifications, insertions, and deletions to either list are built from the same primitives, using count to translate indices and re-tagging edits with tag. In a few cases, we use some edit "macros": since insertions and deletions always happen at the end of a list, we write del' and ins' for edits that do some shuffling to ensure that the inserted or deleted element moves to the appropriate position.

Perhaps the most interesting of these is an in-place modification to the left repository that switches sides of a sum (line 4). For example, suppose we want to replace Beethoven with Plato. The edit to do this has the form $\mathsf{mod}(2, \mathsf{switch}_{LR}(\mathrm{d}n))$—that is, at position 2, switch from an inl to an inr. Here, the translated edit must do *four* things: delete Beethoven from the left list, insert a new element into the right list, re-tag $\mathrm{d}n$ so that it changes the new element to Plato, and finally fix up the complement to match the new interleaving. As before, we can use count to translate the position 2 in the interleaved list into a position in the left list in the right replica. But then we hit a minor snag: deletions only occur at the end of a list. The solution is to first reorder the list, so that Beethoven appears at the end, then delete one element. Figure 6 defines the cycle function, which constructs permutations to do this reordering. The function $\mathsf{cycle}_p(n)$ permutes lists of size n by moving position p to the end of the list, and shifting all the other elements after p down one to fill in the resulting hole. For example, $\mathsf{cycle}_2(5)$ looks like this:

p	1	2	3	4	5
$\mathsf{cycle}_2(5)(p)$	1	3	4	5	2

So, we can delete position p by first reordering with $\mathsf{reorder}(\mathsf{cycle}_p)$, then deleting one element with $\mathsf{del}(1)$. The $\mathsf{del}'(p)$ macro encapsulates this pattern; there is a similar pattern for inserting a new element at position p encapsulated by $\mathsf{ins}'(p)$. Finally, since position 2 in the interleaved list corresponds to positions 2 and 1 in the left and right non-interleaved lists, respectively, the final edit can be written as $\mathsf{right}(\mathsf{mod}(1, \mathrm{d}n))\,\mathsf{right}(\mathsf{ins}'(1))\,\mathsf{left}(\mathsf{del}'(2))$. To fix up the complement, we can simply set the flag at position p to match the new tag: in our case, position 2 is now an inr, so we should set $c_2 = R$.

The most delicate cases involve translating reorderings. Consider an edit to the right repository that swaps Schumann and Dvorak. One way to write this edit is in terms of a function that swaps indices one and three for lists of size at least three (and does nothing on lists of size smaller than three):

$$f(n)(p) = \begin{cases} 4 - p & n \geq 3 \wedge p \in \{1,3\} \\ p & n < 3 \vee p \notin \{1,3\} \end{cases}$$

The edit itself is then $\mathsf{left}(\mathsf{reorder}(f))$. Our job is now to compute some f' for which $\mathsf{reorder}(f')$ swaps inl(Schumann) and inl(Dvorak) in the left repository (line 14). There is one wrinkle: f and f' are parameterized by the length of the lists they permute. Translating f naively would therefore seem to require a way for f' to *guess* the number of composers in lists whose lengths do not match that of the complement. Fortunately, f' need only behave correctly for exactly those lists that are consistent with the current complement, for which our "guess" about how many composers there are is guaranteed to be accurate. So we need only construct a single permutation (and use, say, the identity permutation for all inconsistent list lengths). We use the count function to construct this permutation. It is convenient to derive an isomorphism between positions in the left repository and positions tagged by which list they are indexing into in the right repository; the iso function shows how to use count to do this. In our example, the resulting isomorphism looks like this:

left	1	2	3	4	5
right	$\mathsf{inl}(1)$	$\mathsf{inl}(2)$	$\mathsf{inr}(1)$	$\mathsf{inr}(2)$	$\mathsf{inl}(3)$

We can use $f(3)$ as a permutation on the inl elements, defining $g(\mathsf{inl}(p)) = \mathsf{inl}(f(3)(p))$ and $g(\mathsf{inr}(p)) = \mathsf{inr}(p)$. Then, to find out where position p in the left repository should come from, we can simply translate p into an index into the right repository using iso, apply g to find out where that index came from, and translate back into the left repository using iso^{-1}. Expanding the table above with these translations yields:

left	1	2	3	4	5
$\mathsf{iso}(\mathsf{left})$	$\mathsf{inl}(1)$	$\mathsf{inl}(2)$	$\mathsf{inr}(1)$	$\mathsf{inr}(2)$	$\mathsf{inl}(3)$
$g(\mathsf{iso}(\mathsf{left}))$	$\mathsf{inl}(3)$	$\mathsf{inl}(2)$	$\mathsf{inr}(1)$	$\mathsf{inr}(2)$	$\mathsf{inl}(1)$
$\mathsf{iso}^{-1}(g(\mathsf{iso}(\mathsf{left})))$	5	2	3	4	1

This swaps indices 1 and 5, so our final f' looks like:

$$f'(n)(p) = \begin{cases} 6 - p & n = 5 \wedge p \in \{1,5\} \\ p & n \neq 5 \vee p \notin \{1,5\} \end{cases}$$

Translating a reordering of the left repository follows a similar path (line 9): restrict the reordering to lists consistent with the current complement, then compose the permutation with isomorphisms between the indices in the two repositories. There is one subtlety here: a reordering of the list in the left repository may shuffle which positions are inl's and which are inr's. As a result, we must take care to construct *two* separate position isomorphisms: one for "before" the reordering, and one for "after."

5. Containers

The list mapping lens from the previous section can be generalized to a much larger set of structures, called *containers*, that also includes trees, labeled graphs, etc. We will also provide a general construction for "reorganization lenses" between *different* container types (over the same type of entries). Together with composition and tensor product, this will provide a set of building blocks for constructing many useful lenses. The reorganization lenses also furnish further examples of lenses with nontrivial complements. (Only a small part of §6 depends on this material; it can safely be skimmed on a first reading.)

Containers were first proposed by Abbott, Altenkirch, and Ghani [1]. The idea is that a container type specifies a set I of shapes and, for each shape i, a set of positions $P(i)$. A container with entries in X and belonging to such a container type comprises a shape i and a function $f : P(i) \to X$. For example, lists are containers whose shapes are the natural numbers and for which $P(i) = \{0, \ldots, i-1\}$, whereas binary trees are containers whose shapes are prefix-closed subsets of $\{0, 1\}^*$ (access paths) and where $P(i) = i$ itself. Even labeled graphs can be modeled using unlabeled graphs as shapes. One can further generalize the framework to allow the types of entries to depend on their position, but for the sake of simplicity we will not do so here.

In the present context, containers are useful because they allow for the definition of a rich edit language, allowing the insertion and deletion of positions, modification of particular entries, and reorganizations such as tree rotations. We can then define lenses for containers that propagate these general edit operations.

In the case of state-based symmetric lenses [7], it has been observed that lens iterators akin to "fold left" for inductive data structures also permit the definition of powerful (state-based) lenses. In the edit-based framework iterators are less convenient because it is unclear how edits in an arbitrary module should be propagated to, say, list edits in such a way that the rich edit structure available for lists is meaningfully exploited. (Of course, it is possible to propagate everything to a "rebuild from scratch" edit, thus aping the state-based case.)

In the following we slightly deviate from the presentation of containers from [1, 7] in that we do not allow the set of positions to vary with the shapes. We rather have a universal set of positions P and a predicate live that delineates a subset of P for each shape i. We can then obtain a container type in the original sense by putting $P(i) = \{p \mid p \in \mathsf{live}(i)\}$. Conversely, given a container type in the sense of [1], we can define $P = \{(i, p) \mid p \in P(i)\}$ and $\mathsf{live}(i) = \{(i, p) \mid p \in P\}$. Furthermore, as we already did in [7], we require a *partially-ordered* set of shapes I and ask that live be monotone. Formulating this in the original setting would require a coherent family of transition functions $P(i) \to P(i')$ when $i \leq i'$, which is more cumbersome. Another advantage of the present formulation of container types is that it lends itself more easily to an implementation in a programming language without dependent types.

5.1 Definition: A *container type* is a triple $\langle I, P, \mathsf{live}\rangle$ comprising (1) a *module* I of *shapes* whose underlying set is partially ordered (but whose action need not be monotone); (2) a set P of *positions*; and (3) a *liveness predicate* in the form of a monotone function $\mathsf{live} \in I \to \mathcal{P}(P)$ which tells for each shape which positions belong to it.

If $T = \langle I, P, \mathsf{live}\rangle$ is a container type and X is a set, we can form the set $T(X)$ of containers of type T with entries from X by setting $T(X) = \sum_{i \in I} \mathsf{live}(i) \to X$. Thus a container of type T and entries from X comprises a shape i and, for every position that is live at i—i.e. every element of $\mathsf{live}(i)$—an entry taken from X.

Our aim is now to explain how the mapping $X \mapsto T(X)$ lifts to a functor on the category of lenses—i.e., for each module X, how to construct a module $T(X)$ whose underlying set of states is the set of containers $T(|X|)$, and for each lens $\ell \in X \leftrightarrow Y$, how to construct a "container mapping lens" $T(\ell) \in T(X) \leftrightarrow T(Y)$. We will see that this mapping is well defined on equivalence classes of lenses and respect identities and composition. We begin by defining a module structure on containers.

5.2 Definition: Let $T = \langle I, P, \mathsf{live}\rangle$ be a container type. An edit $\mathrm{d}i \in \partial I$ is an *insertion* if $\mathrm{d}i\ i \geq i$ whenever defined. It is a *deletion* if $\mathrm{d}i\ i \leq i$ whenever defined. It is a *rearrangement* if $|\mathsf{live}(\mathrm{d}i\ i)| = |\mathsf{live}(i)|$ (same cardinality) whenever defined. We only employ edits from these three categories as ingredients of container edits; any other edits in the module will remain unused. This division of container edits into "pure" insertions, deletions, and rearrangements facilitates the later definition of lenses operating on such edits.

5.3 Definition: If $\langle I, P, \mathsf{live}\rangle$ is a container type, $\mathrm{d}i \in \partial I$, and $f \in I \to P \to P$, then we say f is *consistent* with $\mathrm{d}i$ if, whenever $\mathrm{d}i\ i$ is defined, $f(i)$ restricted to $\mathsf{live}(i)$ is a bijection to $\mathsf{live}(\mathrm{d}i\ i)$.

A typical insertion could be the addition of a node to a binary tree, a typical deletion the removal of some node, and a typical rearrangement the rotation of a binary tree about some node.

5.4 Definition [Container edits]: Given container T and module X we define edits for $T(|X|)$ as follows (we give some intuition after Definition 5.5):

$$\begin{array}{ll} & \{\mathsf{fail}\} \\ \cup & \{\mathsf{mod}(p, \mathrm{d}x) \mid p \in P, \mathrm{d}x \in \partial X\} \\ \cup & \{\mathsf{ins}(\mathrm{d}i) \quad\ \mid \mathrm{d}i \text{ an insertion}\} \\ \cup & \{\mathsf{del}(\mathrm{d}i) \quad\ \mid \mathrm{d}i \text{ a deletion}\} \\ \cup & \{\mathsf{rearr}(\mathrm{d}i, f) \mid f \text{ consistent with } \mathrm{d}i\} \end{array}$$

In the last case, often either $\mathrm{d}i$ will only be defined for very few i or f will have a generic definition, so the representation of a rearrangement edit does not have to be large.

5.5 Definition [Edit application]: The application of an edit to a container (i, f) is defined as follows:

$\mathsf{fail}\ (i, f)$ is always undefined
$\mathsf{mod}(p, \mathrm{d}x)\ (i, f) = (i, f[p \mapsto \mathrm{d}x\ f(p)])$ when $p \in \mathsf{live}(i)$
$\mathsf{ins}(\mathrm{d}i)\ (i, f) = (\mathrm{d}i\ i, f')$
  where $f'(p) = \text{if } p \in \mathsf{live}(i) \text{ then } f(p) \text{ else } init_X$

$\mathsf{del}(\mathrm{d}i)\ (i, f) = (\mathrm{d}i\ i, f{\restriction}\mathsf{live}(\mathrm{d}i\ i))$
$\mathsf{rearr}(\mathrm{d}i, f)\ (i, g) = (\mathrm{d}i\ i, g')$
  where $g'(p) = g(f(i)(p))$

The $\mathsf{mod}(p, \mathrm{d}x)$ edit modifies the contents of position p according to $\mathrm{d}x$. If that position is absent the edit fails. The shape of the resulting container is unchanged. The $\mathsf{ins}(\mathrm{d}i)$ edit alters the shape by $\mathrm{d}i$, growing the set of positions in the process (since $\mathrm{d}i\ i \geq i$). The new positions are filled with $init_X$. The $\mathsf{del}(\mathrm{d}i)$ edit works similarly, but the set of positions may shrink; the contents of deleted positions are discarded. The fail edit never applies and will be returned *pro forma* by some container lenses if the input edit does not match the current complement.

The $\mathsf{rearr}(\mathrm{d}i, f)$ edit, finally, changes the shape of a container but neither adds nor removes entries. As already mentioned, a typical example is the left-rotation of a binary tree about the root. This rotation applies whenever the root has two grandchildren to the left and a child to the right. For this example, one may worry

$$\frac{\ell \in X \leftrightarrow Y \qquad T = \langle I, P, \mathsf{live}\rangle \text{ a container type}}{T(\ell) \in T(X) \leftrightarrow T(Y)}$$

$$\begin{array}{lcl}
C & = & T(\ell.C) \\
\mathit{init} & = & (\mathit{init}_I, \lambda p.\ \ell.\mathit{init}) \\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), (i, f)) & = & (\mathsf{mod}(p, \mathrm{d}y), (i, f')) \\
\multicolumn{3}{l}{\quad\text{when } p \in \mathsf{live}(i) \text{ and where}} \\
\multicolumn{3}{l}{\quad f' = f[p{\mapsto}c'], (\mathrm{d}y, c') = \ell.{\Rrightarrow}(\mathrm{d}x, f(p))} \\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), (i, f)) & = & (\mathsf{fail}, (i, f)) \text{ if } p \notin \mathsf{live}(i) \\
\Rrightarrow_g(\mathsf{ins}(\mathrm{d}i), (i, g)) & = & (\mathsf{ins}(\mathrm{d}i), \\
& & (\mathrm{d}i\ i, g[p{\mapsto}\ell.\mathit{init}])) \\
& & \text{when } \mathrm{d}i\ i \text{ is defined} \\
\Rrightarrow_g(\mathsf{del}(\mathrm{d}i), (i, g)) & = & (\mathsf{del}(\mathrm{d}i), (\mathrm{d}i\ i, g{\restriction}\mathsf{live}(\mathrm{d}i\ i))) \\
& & \text{when } \mathrm{d}i\ i \text{ is defined} \\
\Rrightarrow_g(\mathsf{rearr}(\mathrm{d}i, h), (i, g)) & = & (\mathsf{rearr}(\mathrm{d}i, h), \\
& & (\mathrm{d}i\ i, \lambda p.g(h(i)(p)))) \\
& & \text{when } \mathrm{d}i\ i \text{ is defined} \\
\Rrightarrow_g(\mathrm{d}z, c) & = & (\mathsf{fail}, c) \text{ in all other cases} \\
\Lleftarrow_g(-,-) & = & \text{analogous} \\
K & = & \{((i, f), (i, g), (i, f')) \mid i \in I \\
& & \wedge (f(p), g(p), f'(p)) \in \ell.K\}
\end{array}$$

Figure 8: Generic container-mapping lens

about the size of f, since it affects many positions; however, it can be serialized to a small, three line if-then-else. That we do not, at this point, provide edits that copy the contents of some position into other positions; their investigation is left for future work.

We define the monoid $\partial T(X)$ as the free monoid generated by the basic edits defined above. In Section 6 we discuss the possibility of imposing equational laws, in particular with a view to compact normal forms of container edits.

Setting $\mathit{init}_{T(X)} = (\mathit{init}_I, \lambda p.\mathit{init}_X)$ when $T = \langle I, P, \mathsf{live}\rangle$ completes the definition of the module $T(X)$.

5.6 Example: For any module X we can construe the list module X^* as a particular container type $\langle I, P, \mathsf{live}\rangle$ where $I = \mathbb{N}$ with ∂I generated by $i \in \mathbb{Z}$ with $i \odot n = \max(i + n, 0)$. Furthermore, $P = \mathbb{N}$ and $\mathsf{live}(n) = \{0, \dots, n-1\}$.

Then all list edits arise as specific container edits, however, the generic formulation of container edits also includes some esoteric edits, such as $\mathsf{ins}(10 \cdot (-10))$ which brings a list to minimum length 10 by appending default elements if needed.

In Figure 8 we define the mapping lens turning $T(-)$ into an endofunctor on the category of lenses. We note that this is only the second lens to have a nontrivial complement (after *partition*).

Given that this definition looks complex at first we state and prove explicitly that it is indeed a lens.

5.7 Theorem: If $T = \langle I, P, \mathsf{live}\rangle$ is a container and ℓ is a lens so is $T(\ell)$. Moreover, $T(-)$ respects lens equivalence and preserves the identity lens and composition of lenses (up to equivalence), and thus defines a functor on the category of lenses.

We can also define a restructuring lens between containers of different container type but with the same type of entries, i.e. between $T(X)$ and $T'(X)$ where $T = \langle I, P, \mathsf{live}\rangle$ and $T' = \langle I', P', \mathsf{live}'\rangle$. For this to be possible, we need a lens ℓ between I and I' and for any triple $(i, c, i') \in \ell.K$ a bijection $f_{i,c,i'} \in \mathsf{live}'(i') \simeq \mathsf{live}(i)$. The complement of this lens consists of those triples (i, c, i'), and thus "knows" at any time which bijection links the positions at either end.

One typical instance of this kind of lens is list reversal; another is a lens between trees and lists which ensures that the list entries agree with the tree entries according to some fixed order, e.g. in-order or breadth first. Although the live positions of the containers to be synchronized are in bijective correspondence, there is—e.g. in the case of list reversal—no fixed edit that, say, a "modify the second position" edit is mapped to. Indeed, the restructuring lens we are about to construct can be seen as a kind of state-indexed isomorphism, but the full scaffolding of edit lenses is needed to make such a notion precise.

$$\frac{\begin{array}{c} T = \langle I, P, \mathsf{live}\rangle \text{ a container type} \\ T' = \langle I', P', \mathsf{live}'\rangle \text{ a container type} \\ \ell \in I \leftrightarrow I' \end{array}}{[T, T'](\ell) \in T(X) \leftrightarrow T'(X)}$$

$$\begin{array}{lcl}
C & = & \ell.K \\
\mathit{init} & = & (\mathit{init}_I, \ell.\mathit{init}, \mathit{init}_{I'}) \\
\multicolumn{3}{l}{K = \{((i, f), (i, c, i'), (i', f'))} \\
\multicolumn{3}{l}{\qquad \mid (i, c, i') \in \ell.K \wedge \forall p{\in}\mathsf{live}'(i').f(f_{i,c,i'}(p)) = f'(p)\}} \\
\Rrightarrow_g(\mathsf{fail}, x) & = & (\mathsf{fail}, x) \\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), (i, c, i')) & = & (\mathsf{mod}(f^{-1}_{i,c,i'}(p), \mathrm{d}x), (i, c, i') \\
& & \text{when } p \in \mathsf{live}(i) \\
\Rrightarrow_g(\mathsf{ins}(\mathrm{d}i), (i, c, i')) & = & (\mathsf{rearr}(\mathbf{1}, f_i)\mathsf{ins}(\mathrm{d}i'), \\
& & (\mathrm{d}i\ i, c', \mathrm{d}i'\ i')) \\
\Rrightarrow_g(\mathsf{del}(\mathrm{d}i), (i, c, i')) & = & (\mathsf{rearr}(\mathbf{1}, f_d)\mathsf{del}(\mathrm{d}i'), \\
& & (\mathrm{d}i\ i, c', \mathrm{d}i'\ i')) \\
\Rrightarrow_g(\mathsf{rearr}(\mathrm{d}i, f), (i, c, i')) & = & (\mathsf{rearr}(\mathrm{d}i', f_r), \\
& & (\mathrm{d}i\ i, c', \mathrm{d}i'\ i')) \\
& & \text{see below for } f_i, f_d, f_r \\
\text{in the last three clauses:} & & (\mathrm{d}i', c') = \ell.{\Rrightarrow}(\mathrm{d}i, c) \\
\Rrightarrow_g(\mathrm{d}c, (i, c, i')) & = & \mathsf{fail} \text{ in all other cases} \\
\Lleftarrow_g(-,-) & = & \text{analogous}
\end{array}$$

Figure 9: Container restructuring lens

We also require that ℓ maps insertions to insertions, deletions to deletions, and rearrangements to rearrangements. Note that this is well-defined on equivalence classes of lenses.

Given these data, we define the restructuring lens in Figure 9, with a few supplementary definitions below. The families of bijections f_i, f_d, f_r must be chosen in such a way that the container edits in which they appear are well-formed (this is possible since $\mathrm{d}i'$ is an insertion, deletion, or restructuring as appropriate) and such that the following three constraints are satisfied: in each case i, i', etc., refer to the current values from above and $p \in \mathsf{live}'(\mathrm{d}i'\ i')$ is an arbitrary position.

$$\begin{array}{l}
f_i(\mathrm{d}i'\ i')(p) = f^{-1}_{i,c,i'}(f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p)) \\
\qquad\qquad\qquad \text{when } f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p) \in \mathsf{live}(i) \\
f_d(\mathrm{d}i'\ i')(p) = f^{-1}_{i,c,i'}(f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p)) \\
f_r(\mathrm{d}i'\ i')(p) = f^{-1}_{i,c,i'}(f(i)(f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p)))
\end{array}$$

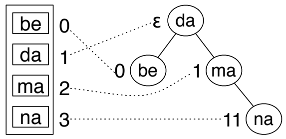
The propagated edits are supposed to be applied to a container of the current shape i', so these arbitrary decisions do not really matter; nevertheless it would be nice if we could be a bit more uniform. This is indeed possible in the case where ℓ is an isomorphism lens, but we refrain from formulating details.

The bijection f_i contains a little more choice, namely the behavior on the T' positions in $f^{-1}_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(\mathsf{live}(\mathrm{d}i\ i) \setminus \mathsf{live}(i))$. Fortunately, they all contain init_X so that the choice does not affect the resulting state after application of the edit.

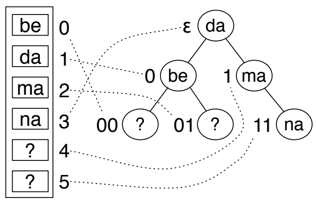
We illustrate the propagation of an $\mathsf{ins}(\mathrm{d}i)$ edit in the particular case where we are synchronizing a tree with the list formed by its in-order traversal. Thus, $I = \mathbb{N}$; $P = \mathbb{N}$; $\mathsf{live}(i) = \{p \mid p < i\}$ and I' comprises prefix closed subsets of $\{0,1\}^*$; $P' = \{0,1\}^*$;

$\mathsf{live}'(i') = i'$. The monoid ∂I has increment and decrement operations; the monoid $\partial I'$ has operations for adding and removing nodes in leaf positions and also for rotating tree shapes.

The lens $\ell \in I \leftrightarrow I'$ does not know anything about the intended application; it has a trivial complement $Unit$ and merely maintains the constraint that the list shape and the tree shape have the same number of positions. It has some freedom how it translates list edits; e.g., it might add and remove tree nodes at the left.

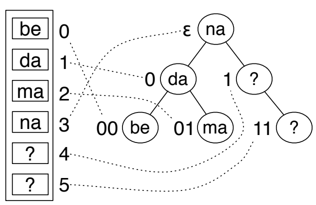


The family of bijections $f_{i,c,i'}$ models the in-order correspondence; thus, for example if $i = 4$ and $i' = \{\varepsilon, 0, 1, 11\}$ the bijection would be as shown above. (For illustration we also indicate possible X-contents of the positions.) Formally, we have $f_{i,c,i'} = \{(0,0), (1,\varepsilon), (2,1), (3,11)\}$.



Now suppose that $\mathrm{d}i\; i = i+2$ and that $\mathrm{d}i'$ (the result of $\mathrm{d}i$ propagated through ℓ) installs two children at the leftmost node. In our in-order application we then have $f_{\mathrm{d}i\; i, c', \mathrm{d}i'\; i'} = \{(0,00), (1,0), (2,01), (3,\varepsilon), (4,1), (5,11)\}$ and after applying both $\mathsf{ins}(\mathrm{d}i)$ and $\mathsf{ins}(\mathrm{d}i')$ we are in the as-yet-inconsistent situation depicted above.

To restore consistency we also apply $\mathsf{rearr}(1, f_r)$ where $f_r(i') = \{(00,0), (0,\varepsilon), (01,1), (\varepsilon,11), (1,00), (11,01)\}$. We could also have chosen $f_r(i') = \{\ldots, (1,01), (11,01)\}$; this is precisely the additional freedom of choice. Of course $f_r(i'')$ for $i'' \neq i'$ is also completely unconstrained. After applying $\mathsf{rearr}(1, f_r)$ we end up with the desired consistent state.



6. Adding Monoid Laws

The edit languages accompanying the constructions in the previous two sections were all freely generated. This was a good place to begin as it is relatively easy to understand, but, as discussed in §3, there are good reasons for investigating richer languages. This section takes a first step in this direction by showing how to equip the product and sum combinators with more interesting edits.

Given modules X and Y, there is a standard definition of *module product* motivated by the intuition that an edit to an $|X| \times |Y|$ value is a pair of an edit to the $|X|$ part and an edit to the $|Y|$ part. The monoid multiplication goes pointwise, and one can define an edit application that goes pointwise as well.

$$X \otimes Y = \langle |X| \times |Y|, (init_X, init_Y), \partial X \otimes \partial Y, \odot_{X \otimes Y} \rangle$$
$$\mathbf{1}_{M \otimes N} = (\mathbf{1}_M, \mathbf{1}_N)$$
$$(m, n) \cdot_{M \otimes N} (m', n') = (m\, m', n\, n')$$
$$(\mathrm{d}x, \mathrm{d}y) \odot_{X \otimes Y} (x, y) = (\mathrm{d}x\, x, \mathrm{d}y\, y)$$

One might wonder whether the standard definition has any connection to the definition we give earlier. One way to bridge the gap is to add equational laws to the free monoid.[5] The equations below demand that left and right be monoid homomorphisms, and that they commute:

$$\mathsf{left}(\mathbf{1}) = \varepsilon$$
$$\mathsf{left}(\mathrm{d}x)\mathsf{left}(\mathrm{d}x') = \mathsf{left}(\mathrm{d}x\mathrm{d}x')$$
$$\mathsf{right}(\mathbf{1}) = \varepsilon$$
$$\mathsf{right}(\mathrm{d}y)\mathsf{right}(\mathrm{d}y') = \mathsf{right}(\mathrm{d}y\mathrm{d}y')$$
$$\mathsf{left}(\mathrm{d}x)\mathsf{right}(\mathrm{d}y) = \mathsf{right}(\mathrm{d}y)\mathsf{left}(\mathrm{d}x)$$

It is not hard to show that the free monoid subject to the above equations is isomorphic to the natural monoid product.

However, it is not obvious that the definitions relying on the free monoid product remain well defined after imposing the above equations. In particular, we must check that any monoid homomorphisms we defined respect these laws. For homomorphisms f specified via specification of f_g, it is enough to prove that, for each equational law $g = g'$, the specification respects the law—i.e., $f(g) = f(g')$.

For example, to check that we can create a well-defined tensor product module that includes the above equations, we must show that $\odot_g$ respects the equations. For the commutativity equation, we must show

$$\mathsf{left}(\mathrm{d}x) \odot_g \mathsf{right}(\mathrm{d}y) \odot_g (x,y) = \mathsf{right}(\mathrm{d}y) \odot_g \mathsf{left}(\mathrm{d}x)_g \odot_g (x,y).$$

Simple calculation shows that both sides are equal to $(\mathrm{d}x\; x, \mathrm{d}y; y)$, so this law is respected; the rest follow similar lines.

Most importantly, we must check that the $\Rrightarrow$ and $\Lleftarrow$ functions are still monoid homomorphisms; indeed, this check makes these equations interesting as a *specification*: in addition to the usual round-tripping laws we expect of state-based lenses, each non-trivial equation in a monoid presentation represents a behavioral limitation on lenses operating on that monoid. Take again the commutativity law:

$$\mathsf{left}(\mathrm{d}x)\ \mathsf{right}(\mathrm{d}y) = \mathsf{right}(\mathrm{d}y)\ \mathsf{left}(\mathrm{d}x)$$

The force of this law is that lenses operating on a monoid including this equation must ignore the interleaving of left and right edits: those two edits are treated independently by the lens.

6.1 Lemma: If k and ℓ are lenses, then the $\Rrightarrow_g$ and $\Lleftarrow_g$ functions defined above for $k \otimes \ell$ respect all of the above equations.

Adding the first four equations lets us create a projection lens out of smaller parts by observing that there are some new isomorphisms available. Let f be the isomorphism between $X \otimes Unit$ and X. Similarly, let g be the obvious isomorphism between $Unit \otimes Y$ and Y. We can then define $\pi_1 = (id_X \otimes term_Y); iso_f$ and $\pi_2 = (term_X \otimes id_Y); iso_g$. Thus, π_1 first throws away any information in the right-hand part of a tuple with $term_Y$, then collapses the (now degenerate) tuple with f.

We conjecture that these additional laws introduce enough isomorphisms that the tensor product gives rise to a symmetric monoidal category—that is, that tuples may be reordered and reassociated freely, provided the lens program acting on them is reordered and reassociated accordingly—but we have not explored this possibility fully.

We can perform a similar process for sum edits. We add the following equations:

$$\mathsf{switch}_{jk}(m)\ \mathsf{switch}_{ij}(m') = \mathsf{switch}_{ik}(m)$$
$$\mathsf{switch}_{ij}(m)\ \mathsf{stay}_i(m') = \mathsf{switch}_{ij}(m)$$
$$\mathsf{stay}_j(m)\ \mathsf{switch}_{ij}(m') = \mathsf{switch}_{ij}(mm')$$
$$\mathsf{stay}_i(m)\ \mathsf{stay}_i(m') = \mathsf{stay}_i(mm')$$
$$d\ d' = \mathsf{fail} \qquad \text{in all other cases}$$

[5] To make this formal, treat the equations as a relation between words in the free monoid; take the reflexive, symmetric, transitive, congruence closure of this relation; and quotient by the resulting equivalence relation.

This explains why we did not originally choose to have just two combinators, switch_L and switch_R, which would be interpreted as "switch to the left (respectively, right) side and reinitialize, no matter which side we are currently on." The idea of the above equations is that they allow us to collapse any sequence of edits down into a single one; if we only allowed ourselves switch_L and switch_R forms, this would not be possible. In particular, we need to represent the fact that a stay_L edit followed by a switch_i edit fails when applied to a value tagged with inr.

As with products, we must check that the remaining definitions are well-formed. In particular, it can be shown that, in the module defined above for sums, $\odot_g$ respects the above equations, and that, if k and ℓ are lenses, then $(k \oplus \ell).\Rrightarrow_g$ and $(k \oplus \ell).\Lleftarrow_g$ respect the above equations.

Unfortunately, the *partition* lens as given does *not* respect the above equations. It seems possible to enforce them by also imposing equations on list edits that coalesce adjacent $\mathsf{reorder}$ operations. We leave this to future work.

In a similar vein, we can impose equations on container edits—indeed, we need them, since we would like lists to form a special case of containers so that, possibly after *restructuring*, we can *partition* and reassemble containers, too. These equations would in particular allow us to coalesce adjacent reorderings and to reorder insertions and deletions with other edits so that insertions and deletions always come first. This would also give rise to a compact normal form of container edits. Again, we leave this to future work.

7. From State-Based to Edit Lenses and Back

In [7], we introduced a state-based framework for bidirectional transformations called *symmetric lenses*. We refer to them here as *state-based symmetric lenses*. Recall from [7] that a state-based symmetric lens ℓ between *sets* X and Y comprises a set of complements C, a distinguished element $\mathit{missing} \in C$, and two functions

$$\begin{array}{rcl} \mathit{putr} & \in & X \times C \to Y \times C \\ \mathit{putl} & \in & Y \times C \to X \times C \end{array}$$

satisfying the following round-tripping laws:

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

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

Equivalence of state-based symmetric lenses is defined through the existence of a simulating relation between the respective complement sets that relates the *missing* elements and is preserved by $\mathit{putl}, \mathit{putr}$. A characterization in terms of "dialogues" is also given. State-based symmetric lenses modulo equivalence form a category (they compose) and support a variety of constructions, in particular tensor product, sum, lists, trees, and container types.

Now, for any set X we have the monoid ∂X whose elements (edits) are lists of elements of X modulo the equality $xx = x$. An action of ∂X on X is defined by $\varepsilon x = x$ and $(xw)y = x$ where $x \in X, w \in X^*$. Note that this is well defined as $x(xy) = x = xy$. If, in addition, we have a distinguished element $x \in X$, we thus obtain a module denoted X_x where $|X_x| = X$ and $\mathit{init}_X = x$ and $\partial X_x = \partial X$.

Let ℓ be a state-based symmetric lens between X and Y along with elements $x \in X$ and $y \in Y$ satisfying $\ell.\mathit{putr}(x, \ell.\mathit{missing}) = (y, \ell.\mathit{missing})$. We then define a symmetric edit lens $\partial_{xy}\ell$ between the modules X_x and Y_y as follows: (1) $(\partial_{xy}\ell).C = \ell.C$; (2) $(\partial_{xy}\ell).\mathit{init} = \ell.\mathit{missing}$; (3) $(\partial_{xy}\ell).\Rrightarrow(\varepsilon, c) = (\varepsilon, c)$; (4) $(\partial_{xy}\ell).\Rrightarrow(xw, c) = (yv, c'')$ where $(\partial_{xy}\ell).\Rrightarrow(w, c) = (v, c')$ and $\ell.\mathit{putr}(x, c') = (y, c'')$; (5) analogous definitions for $\Lleftarrow$; and (6) $K = \{(x, c, y) \mid \ell.\mathit{putr}(x, c) = (y, c)\}$. $\partial_{xy}\ell$ is a symmetric edit lens and the passage from ℓ to $\partial\ell$ is compatible with the equivalences on symmetric lenses and symmetric edit lenses.

Let X be a module. A *differ* for X is a binary operation $\mathit{dif} \in X \times X \to \partial X$ satisfying $\mathit{dif}(x, x')x = x'$ and $\mathit{dif}(x, x) = \mathbf{1}$. Thus, a differ finds, for given states x, x', an edit operation $\mathrm{d}x$ such that $\mathrm{d}x\ x = x'$ and $\mathrm{d}x$ is "reasonable" at least in the sense that if $x = x'$ then the produced edit is minimal, namely $\mathbf{1}$. For example, the module X_x for set X and $x \in X$ admits the *canonical differ* given by $\mathit{dif}(x, x') = x'$ if $x \neq x'$ and $\mathit{dif}(x, x) = \varepsilon$, otherwise.

Given an edit lens ℓ between modules X and Y, both equipped with differs, we define a symmetric lens $|\ell|$ between $|X|$ and $|Y|$ by (1) $|\ell|.C = |X| \times \ell.C \times |Y|$; (2) $|\ell|.\mathit{init} = (\mathit{init}_X, \ell.\mathit{init}, \mathit{init}_Y)$; (3) $|\ell|.\mathit{putr}(x, (x_0, c, y_0)) = (\mathrm{d}y\ y_0,\ (x, c', \mathrm{d}y\ y_0))$ where $\mathrm{d}x = \mathit{dif}(x_0, x)$ and $(\mathrm{d}y, c') = \ell.\Rrightarrow(\mathrm{d}x, c)$; and (4) an analogous definition of $|\ell|.\mathit{putl}$ This defines a symmetric lens $|\ell|$ between $|X|$ and $|Y|$, and the passage $\ell \mapsto |\ell|$ is compatible with lens equivalence.

7.1 Theorem: Let X, Y be sets with distinguished elements x and y and equip the associated modules X_x and Y_y with their canonical differs. The constructions $|-|$ and ∂_{xy} then establish a one-to-one correspondence between equivalence classes of edit lenses between X_x and Y_y, on the one hand, and state-based lenses between X and Y, on the other.

We conjecture that this "isomorphism" between state-based and certain edit lenses is also compatible with various lens constructors, in particular tensor product and sum.

8. Related Work

The most closely related attempt at developing a theory of update propagation is [4] by Diskin et al. Their starting point is the observation (also discussed in [2]) that discovery of edits should be decoupled from their propagation. They thus propose a formalism, *sd-lenses*, for the propagation of edits across synchronized data structures, bearing some similarities with our edit lenses. The replicas, which we model as modules, are there modeled as categories (presented as reflexive graphs). Thus, for any two states x, x' there is a set of edits $X(x, x')$. An sd-lens then comprises two reflexive graphs X, Y and for any $x \in X$ and $y \in Y$ a set $C(x, y)$ of "correspondences" which roughly correspond to our complements. Forward and backward operations similar to our $\Lleftarrow$ and $\Rrightarrow$ then complete the picture. No concrete examples are given of sd-lenses, no composition, no notion of equivalence, and no combinators for constructing sd-lenses; the focus of the paper is rather on the discovery of suitable axioms, such as invertibility and undoability of edits, and a generalization of *hippocraticness* in the sense of Stevens [13]. They also develop a comparison with the state-based framework (cf. §7 above). In our opinion, the separation of edits and correspondences according to the states that they apply to or relate has two important disadvantages. First, in our examples, it is often the case that one and the same edit applies to more than one state and can be meaningfully propagated (and more compactly represented) as such. For example, while many of the container edits tend to only work for a particular shape, they are completely polymorphic in the contents of the container. Second, the fact that state sets are already categories suggests that a category of sd-lenses would be 2-categorical in flavor, entailing extra technical difficulties such as coherence conditions.

Meertens's seminal paper on *constraint maintainers* [10] discusses a form of containers for lists equipped with a notion of edits similar to our edit language for lists, but does not develop a general theory of edit-transforming constraint maintainers.

A long series of papers from the group at the University of Tokyo [6, 8, 11, 12, 15, etc.] deal with the alignment issue using an approach that might be characterized as a hybrid of state-based and edit-based. Lenses work with whole states, but these states are internally annotated with tags showing where edits have been applied—e.g., marking inserted or deleted elements of lists. Barbosa et al.'s *matching lenses* [2] offer another approach to dealing with issues of alignment in the framework of pure state-based lenses.

9. Conclusion

A prototype Haskell implementation of edit lenses is underway, as well as a demo showing how to construct GUIs connected by lenses. The main required extension to the theory presented here are extending the above constructions from algebraic data structures to strings, following Boomerang [3], and identifying good heuristics for converting unstructured string edits into structured edits of the form expected by the lenses above—a form of parsing and unparsing.

Containers offer a convenient abstraction on which to build generic lens combinators, as discussed in §5. To use these combinators in practice, we need to show how to instantiate the module of *shapes* for the kind of container we are interested in, as we did for lists. In the future, we would like to explore several other sorts of shapes; in particular, edit languages for graphs may be useful in model-driven development, while edits for relations are relevant to database applications.

Acknowledgments We are grateful to Nate Foster and Perdita Stevens for productive discussions of many points, to the members of the Penn PL Club for comments on an early draft, to the organizers and participants in the January 2011 Dagstuhl seminar on Bidirectional Transformations for creating a stimulating environment for work in this area, and to the POPL reviewers for their thoughtful suggestions. Our work has been supported by the National Science Foundation under grants 0534592, *Linguistic Foundations for XML View Update*, and 1017212, *Algebraic Foundations for Collaborative Data Sharing*.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, September 2010.

[7] 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.

[8] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional 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.

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

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

[11] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In *Seventh International Conference on Mathematics of Program Construction (MPC)*, 2004.

[12] 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.

[13] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS), Nashville, TN*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2007. ISBN 978-3-540-75208-0.

[14] 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.

[15] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *IEEE/ACM International Conference on Automated Software Engineering (ASE), Atlanta, GA*, pages 164–173, 2007.