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

Follow this and additional works at: http://repository.upenn.edu/cis_papers

Part of the Computer Sciences Commons

Recommended Citation
Martin Hoffmann, Benjamin C. Pierce, and Daniel Wagner, 'Edit Lenses', , January 2012.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/677
For more information, please contact repository@pobox.upenn.edu.
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

ACM COPYRIGHT NOTICE. Copyright © 2012 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

doi>10.1145/2103656.2103715

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/677
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 each 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, \( f \in X \to Y \) (telling how to map an updated \( X \) state to an appropriate \( Y \) state) and \( g \in Y \to X \). 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., \( f \in X \times Y \to Y \)—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 \( X \), 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 syntactic 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
decides to add a new composer, Monteverdi, at the end of the list. This change is described by the edit script \( \text{ins}(3); \text{mod}(3, ("Monteverdi", "1567-1643")) \). 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): \( \text{ins}(3); \text{mod}(3, ("Monteverdi", "1")) \). Note that the translated mod command overwrites the name component but leaves the country component with its default value, “?country?”. This is the best we 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 one 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 \( \ell^* \), where \( \ast \) is a “list mapping” lens combinator and \( \ell \) is a lens for translating edits to a single record—i.e., \( \ell \) is a lens from \( X \times Y \) to \( X \times Z \). Moreover, \( \ell \) itself should be built as the product \( \ell_1 \times \ell_2 \) of a lens \( \ell_1 \in X \rightarrow Y \) that translates composer editors verbatim, while \( \ell_2 \) 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 \( \ast \). Its elements will have the form \( \text{ins}(i) \) where \( i \) is a position, \( \text{del}(i) \), \( \text{reorder}(i_1, \ldots, i_n) \) where \( i_1, \ldots, i_n \) is a permutation on positions (compactly represented, e.g. as a branching program), and \( \text{mod}(p, dv) \), 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 \( \partial X \) is the set of edits to composers and \( \partial Y \) is the set of edits to dates. Finally, both \( \partial X \) and \( \partial Y \) are sets of primitive “overwrite edits” that can completely replace one string with another, together with an identity edit \( I \) that does nothing at all; so \( \partial X \) can be just \{I\} + X (with I = \text{ins}()) and similarly for \( Y \) and \( Z \).

Our lens \( \ell^* \) will consist of two components—one for translating edits from the left side to the right, written \( (\ell^*)_R \), and...
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 $\{\text{Unid}\}$. The translation functions manipulate just the complements and the edits, which are also small compared to the size of the replicas.

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 $(M, \cdot, 1_M)$ of a set $M$, an associative binary operation $\cdot : M \times M \to M$, and a unit element $1_M \in M$—that is, with $\cdot$ and $1_M$ such that $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ and $1_M \cdot x = x = x \cdot 1_M$.

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

The unit element represents a “change nothing” edit. Multiplication of edits corresponds to propagating updates across edits in 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 $\text{Unit} \amalg X$. Combining two edits in this monoid simply drops the second (unless the first is the unit): $\text{inl}() \cdot e = e$ and $\text{inr}(x) \cdot e = \text{inr}(x)$.

These equations allow this edit language to represent an arbitrarily long sequence of updates using a single element of $X$ (and, $\text{en passant}$, 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 $\circ \in M \times X \to X$ satisfying two laws: $1 \circ x = x$ and $(m \cdot n) \circ x = m \circ (n \circ x)$.

As with monoid multiplication, we often elide the monoid action symbol, writing $m \cdot x$ for $m \circ x$. 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

---

(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.

$\partial(X \times Y)^* \rightarrow \partial(X \times Z)^*$, and another for transporting edits from right to left, written $(F). \in \in \partial(X \times Z)^* \rightarrow \partial(X \times Y)^*$.

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 $\text{inl}/\text{inr}$ 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 $\in$ function is given just an insertion edit for the homogenous 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 $p.\in \in \partial((X + Y)^*) \times C \rightarrow \partial(X^* \times Y^*) \times C$ and $p.\in \in \partial(X^* \times Y^*) \times C \rightarrow \partial((X + Y)^*) \times C$. Part (c) demonstrates the use (and update) of the complement when translating the insertion.

Note that the complement stores just the $\text{inl}/\text{inr}$ tags, not the actual names of the authors and composers in the left-hand list. In

---

$^2$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 $\text{put}$ function of the state-based symmetric lenses in [7] and the $\text{put}$ function of the state-based asymmetric lenses in [3, 5].
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 \(<X, \text{init}_X, \partial X, \odot X>\) comprising a set \(X\), an element \(\text{init}_X \in X\), a monoid \(\partial X\), and a monoid action \(\odot X\) of \(\partial X\) on \(X\).

If \(X\) is a module, we refer to its first component by either \(|X|\) or \(X\), 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 \(\Rightarrow\) and \(\Leftarrow\) 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:

\[
\begin{align*}
\ell \cdot h &= (1_X, e) \\
(h(m, e) &= (n, c') h(m', e) = (n', c'')
\end{align*}
\]

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 \(\text{init}_X\) and \(\text{init}_Y\), respectively, and the lens is initialized with complement \(\ell.\text{init}\). The users then perform actions and propagate them across the lens. An action consists of producing an edit \(dx\) (or \(dy\)), applying it to one’s current replica \(x\) (resp. \(y\)), putting the edit through the lens to obtain an edit \(dy\) (resp. \(dx\)), and asking the user on the other side to apply \(dy\) (or \(dx\)) 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 \(\text{init}_C\), two stateful monoid homomorphisms \(\Rightarrow \in \partial X \times C \to \partial Y \times C\) and \(\Leftarrow \in \partial Y \times C \to \partial X \times C\), and a ternary consistency relation \(K \subseteq \langle X \times \partial X \times Y \rangle\) such that

- \((\text{init}_X, \text{init}_Y) \in K\);
- if \((x, c, y) \in K\) and \(dx\) is defined and \(\Rightarrow (dx, c) = (dy, c')\), then \(dy\) is also defined and \((dx, c', dy) \in K\);
- if \((x, c, y) \in K\) and \(dy\) is defined and \(\Leftarrow (dy, c) = (dx, c')\), then \(dx\) is also defined and \((dx, c', dy) \in K\).

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 \(\ell.\text{run} \in (\partial X + \partial Y)^* \to X \times \ell.\text{C} \times Y\) is defined by:

- \(\ell.\text{run} (\varepsilon) = (\text{init}_X, \ell.\text{init}, \text{init}_Y)\)
- \(\ell.\text{run}(w) = (x_0, c, y_0)\) \(\Rightarrow (dx_1, c) = (dy_1, c_1)\)
- \(\ell.\text{run}(\text{init}(dx_1)w) = (dx_1, x_0, c_1, dy_1)\)
- \(\ell.\text{run}(w) = (x_0, c, y_0)\) \(\Leftarrow (dy_1, c) = (dx_1, c_1)\)
- \(\ell.\text{run}(\text{init}(dy_1)w) = (dx_1, x_0, c_1, dy_1)\)

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

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 \(\Rightarrow\) and \(\Leftarrow\). 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 \(\Rightarrow\) and \(\Leftarrow\).
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 $w$,

- if $k.run(w)$ 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 $dx x$ is defined and $\ell.\Rightarrow(dx, c) = (dy, \_)$ and $k.\Rightarrow(dx, d) = (dy', \_)$ then $dy = dy'$, and the analogous property for $\Leftarrow$.

(Nota 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) $(initX, k, initX, \ell, initY) \in S$; (2) if $(x, c, y) \in S$ and $dx x$ is defined, then if $(dy_1, c') = k.\Rightarrow(dx, c)$ and $(dy_2, d') = \ell.\Rightarrow(dx, d)$, then $dy_1 = dy_2$ and $(dx, c', d', dy, y) \in S$; and (3) analogously for $\Leftarrow$.

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.

\[
\begin{align*}
    id_X \in X \leftrightarrow X \\
    C &= \text{Unit} \\
    K &= \{ (x, c) \mid x \in X \} \\
    \Rightarrow(dx, c) &= (dx, c) \\
    \Leftarrow(dx, c) &= (dx, c)
\end{align*}
\]

Here and below, we elide the definition of the $init$ component when $C = \text{Unit} = \{ () \}$, 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 $\ell$ 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 $\Rightarrow$ and $\Leftarrow$ operations.

\[
\begin{align*}
    k \in X \leftrightarrow Y & \quad \ell \in Y \leftrightarrow Z \\
    k; \ell \in X \leftrightarrow Z
\end{align*}
\]

\[
\begin{align*}
    C &= k.C \times \ell.C \\
    init &= (k.init, \ell.init) \\
    K &= \{ (x, (c, c_\ell)) \mid y, (x, c, z) \in k.K \land (y, c_\ell, z) \in \ell.K \} \\
    \Rightarrow(dx, (c, c_\ell)) &= \text{let} (dy, c'_{\ell}) = k.\Rightarrow(dx, c) \text{ in } \text{let } (dz, c'_{\ell}) = \ell.\Rightarrow(dx, c) \text{ in } (dz, (c'_{\ell}, c'_k)) \\
    \Leftarrow(dx, (c, c_\ell)) &= \text{let } (dy, c'_{\ell}) = \ell.\Leftarrow(dx, c) \text{ in } \text{let } (dz, c'_{\ell}) = k.\Leftarrow(dx, c) \text{ in } (dz, (c'_{k}, c'_\ell))
\end{align*}
\]

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 $(k;\ell; m) = k; (\ell; m)$.

Another simple lens combinator is dualization: for each lens $\ell : X \leftrightarrow Y$, we can construct its dual, $\ell^\dagger \in Y \leftrightarrow X$, by swapping $\Rightarrow$ and $\Leftarrow$.

For the next definition, observe that the set $\text{Unit}$ 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 $\text{Unit}_x$. When context clearly calls for a module, we will abbreviate $\text{Unit}_x \in \text{Unit}$ to simply $\text{Unit}$.

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

\[
\begin{align*}
    \text{term}_X \in X \leftrightarrow \text{Unit} \\
    C &= \text{Unit} \\
    K &= X \times \text{Unit} \times \text{Unit} \\
    \Rightarrow(dx, c) &= (\text{Unit}, c) \\
    \Leftarrow(dx, c) &= (\text{Unit}, c)
\end{align*}
\]

The disconnect lens that we saw in §2 can be built from $\text{term}$. The term lens is also unique (up to equivalence): the implementation of $\Rightarrow$ is forced by the size of its range monoid $\text{Unit}$, and the implementation of $\Leftarrow$ 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 $f : X \rightarrow Y$ and a monoid homomorphism $h \in \partial X \rightarrow \partial Y$ such that $f(\text{init}_X) = \text{init}_Y$ and $f(dx x) = h(dx)f(x)$. There is an identity $(\lambda x, x, \lambda dx, dx)$ for every module, and the point-wise composition of module homomorphisms is also a homomorphism, so modules form a category. If module homomorphisms $(e, g) \in X \leftrightarrow Y$ and $(f, h) \in Y \rightarrow X$ 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:

\[
\begin{align*}
    (f, h) \in X \rightarrow Y & \quad (f, h) \text{ is inverse to } (f^{-1}, h^{-1}) \\
    iso_{X \rightarrow Y} & \in X \leftrightarrow Y
\end{align*}
\]

\[
\begin{align*}
    C &= \text{Unit} \\
    K &= \{ (x, c) \mid x \in X \} \\
    \Rightarrow(dx, c) &= (h(dx), c) \\
    \Leftarrow(dy, c) &= (h^{-1}(dy), c)
\end{align*}
\]
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 combiners 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. §3 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 $\varepsilon$ 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 $\cdot$. Languages with nontrivial laws.

Given a set $G$, we write $G^*$ for the set of finite sequences of elements of $G$. We write $\varepsilon$ 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 $g_1 \ldots g_n$, we also use $g$ to refer to the entire sequence. The notation $|g|$ means the length of a sequence: $|g| = n$. It is easy to show that $G^*$ together with sequence concatenation and $\varepsilon$ 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 $f \in G \to M$ on generators, the monoid homomorphism $f \in G^* \to M$ is defined by $f(\varepsilon) = 1$ and $f(g_1 \cdot \ldots \cdot g_n) = f(g_1) \cdot \ldots \cdot f(g_n)$. Similarly, given a stateful function $f \in G \times C \to M \times C$, we can define a stateful monoid homomorphism $f \in G^* \times C \to M \times C$ by setting $f(\varepsilon, c) = (1, c)$ and $f(g_1 \cdot \ldots \cdot g_n, c) = (f(g_1, c) \cdot \ldots \cdot f(g_n, c), c)$.

**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.

$$G_{X,Y}^\circ = \{ \text{left}(dx) \mid dx \in \partial X \} \sqcup \{ \text{right}(dy) \mid dy \in \partial Y \}$$

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

- $\text{left}(dx) \cdot g \cdot (x, y) = (dx \cdot x, y)$
- $\text{right}(dy) \cdot g \cdot (x, y) = (x, dy \cdot y)$

The full module is then given by $X \otimes Y = \langle [X] \times [Y], \langle \text{init}_X, \text{init}_Y \rangle, (G_{X,Y}^\circ)^*, \cdot \rangle$.

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

### The Sum Lens

\[
\begin{array}{c}
k \in X \leftrightarrow Z & \ell \in Y \leftrightarrow W \\
k \otimes \ell \in X \times Y \leftrightarrow W
\end{array}
\]

<table>
<thead>
<tr>
<th>$C$</th>
<th>$\ell \in Y \leftrightarrow W$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{init}$</td>
<td>$\text{init}(k, \text{init})$</td>
</tr>
<tr>
<td>$K$</td>
<td>${(\text{init}(x), \text{init}(c), \text{init}(y)) \mid (x, c, y) \in k.K}$</td>
</tr>
<tr>
<td>$\vdash_g(\text{left}(dx), (c_k, c_{\ell}))$</td>
<td>$\text{let}\left(\text{dz}, (c'<em>k, c'</em>{\ell})\right) = k.\vdash(\text{dx}, c_k)$ in $\text{left}(\text{dz}, (c'<em>k, c'</em>{\ell}))$</td>
</tr>
<tr>
<td>$\vdash_g(\text{right}(dy), (c_k, c_{\ell}))$</td>
<td>$\text{let}\left(\text{dy}, (c'<em>{\ell})\right) = \ell.\vdash(\text{dy}, c</em>{\ell})$ in $\text{right}(\text{dy}, (c'_{\ell}))$</td>
</tr>
</tbody>
</table>

$\models$ similarly

**Figure 3:** The sum lens

### 4.1 Theorem:

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

$$\{(x, y), ((c, d), (c', d')), ((c, c'), (d, d')), (x', y') \mid \exists (x', y'), (x, x', x''), (y, y', y''), (d'') \in k.K \land (x, x', x'') \in k'.K \land (y, y', y'') \in \ell.K \land (y', y'') \in \ell'.K\}$$

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.

**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 $\partial X$ and $\partial Y$, we can describe the generators for the free monoid of
4.2 Theorem: When and are lenses, so is .

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

\[
(\text{init}_X \otimes \ell, \text{init}_Y \otimes \ell) \in K,
\]

since \((\text{init}_X, \text{init}_Y) \in K\). So it remains to show that and preserve this relation. We need only consider the case where we begin with an arbitrary consistent triple \((\text{init}_X, \text{init}_Y, \text{init}_Y) \in K\) and \(d \in X \otimes Z\) for which \(\text{init}_X\). The cases where the triple is of the form \((\text{init}_X, \text{init}_Y, \text{init}_Y) \in K\) are similar, swapping \(k\) and \(f\) in some places; the cases where

We are considering a \(d \in Y \otimes W\) are similar, but use \(\otimes\) instead of \(\otimes\) everywhere. Since \(\text{init}_X(d)\) is defined, there are three forms of \(d\) to consider: \(\text{switch}_L(d), \text{switch}_R(d),\) and \(\text{stay}_L(d)\). Here is the most interesting case:

\[
\begin{align*}
\text{Case } d & = \text{switch}_L(d) \\
\text{We define } & (d, c') = k.\text{putr}(d, k.\text{init}) \quad \text{and } (x', y') = (d.x, d.y) \\
\text{since } & (\text{init}_X, k.\text{init}, \text{init}_Y) \in K.
\end{align*}
\]

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 \(\text{ins}(i)\) and \(\text{del}(i)\) now specifies how many elements to insert or delete.

\[
\begin{align*}
\text{mod}(p, dx) \circ \text{ins}(i) & = x_1 \ldots x_{i-1} (dx_{x_i} \ldots dx_{x_n}) \\
\text{ins}(i) & \circ \text{mod}(p, dx) = x_1 \ldots x_i \cdot \text{init}_X \ldots \text{init}_Y \\
\text{del}(i) & \circ \text{mod}(p, dx) = x_1 \ldots x_{i-1} \\
\text{reorder}(f) & \circ \text{mod}(p, dx) = x_f(0) \ldots x_f(n) \\
\text{fail} & \circ \text{mod}(p, dx) = x_1 \ldots x_n
\end{align*}
\]

We take \(\text{mod}(p, dx) \circ \text{ins}(i)\) to be undefined when \(p > |x|\), and similarly take \(\text{del}(i) \circ \text{ins}(i)\) to be undefined when \(i > |x|\). The list module is then \(X^* = \langle |X|^*, \varepsilon, G_X^*, \circ \rangle\).

Mapping lens The list mapping lens \(\ell^*\) uses \(\ell\) to translate mod edits from \(X\) to \(Y\) and vice versa (Figure 4). Other kinds of edits (ins, del, and reorder) are carried across unchanged. The notation \(c[p \mapsto c']\) in the rule for mod edits means “the list that is just like \(c\) except that the element in position \(p\) is replaced by \(c'\)”.

Figure 4: The list mapping lens
\[
\tagof{\text{inl}}(x) = L \quad \tagof{\text{inr}}(y) = R
\]
\[
\text{map}_f(c) = \varepsilon \quad \text{cycle}_p(n)(m) = \begin{cases} 
  p & \text{if } p < m = n \\
  m+1 & \text{if } p < m < n \\
  m & \text{otherwise}
\end{cases}
\]
\[
\text{leqts}(c) = \varepsilon \quad \text{revers}(c_1 \cdots c_n) = c_n \cdots c_1
\]
\[
\text{leqts}(\text{inl}(x) w) = x \text{ leqts}(w) \quad \text{leqts}(\text{inr}(y) w) = y \text{ leqts}(w)
\]
\[
\text{rights}(\text{inl}(x) w) = \text{rights}(w) \quad \text{rights}(\text{inr}(y) w) = y \text{ rights}(w)
\]
\[
\text{leqts}(\text{inr}(y) w) = \text{leqts}(w) \quad \text{rights}(\text{inr}(y) w) = y \text{ rights}(w)
\]
\[
\text{tag}(L, dx) = \text{left}(dx) \quad \text{out}(\text{inl}(x)) = x \\
\text{tag}(R, dy) = \text{right}(dy) \quad \text{out}(\text{inr}(y)) = y
\]
\[
\text{count}(p, c) = (1, 1) \quad \text{count}(1, c) = (1, 1)
\]
\[
\text{count}(p, c_1 \cdots c_n) = \text{let} \ (n_L, n_R) = \text{count}(p-1, c_2 \cdots c_n) \in \begin{cases} 
  (n_L + 1, n_R) & c = L \\
  (n_L, n_R + 1) & c = R
\end{cases}
\]

**Figure 5:** The \(\text{partition}\) lens

**Figure 6:** Supplementary functions for \(\text{partition}\)
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, $\text{cycle}_5(5)$ looks like this:

$$
\begin{array}{cccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\text{cycle}_5(5)(p) & 1 & 2 & 3 & 4 & 5 \\
\end{array}
$$

So, we can delete position $p$ by first reordering with $\text{reorder}(\text{cycle}_p)$, then deleting one element with $\text{del}(1)$. The $\text{del}(1)$ macro encapsulates this pattern; there is a similar pattern for inserting a new element at position $p$ encapsulated by $\text{ins}(p)$. 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 $\text{right}(\text{mod}(1,dn)) \left(\text{right}(\text{ins}(1)) \left(\text{left}(\text{del}(2))\right)\right)$. 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 ins, 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 Dvořák. 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):

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

The edit itself is then $\text{left}(\text{reorder}(f))$. Our job is now to compute some $f'$ for which $\text{reorder}(f')$ swaps $\text{inl}(\text{Schumann})$ and $\text{inl}(\text{Dvořák})$ 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:

$$
\begin{array}{c|cccccc}
\text{left} & 1 & 2 & 3 & 4 & 5 \\
\text{right} & \text{inl}(1) & \text{inl}(2) & \text{inr}(1) & \text{inr}(2) & \text{inl}(3) \\
\end{array}
$$

We can use $f(3)$ as a permutation on the inl elements, defining $g(\text{inl}(p)) = \text{inl}(f(3)(p))$ and $g(\text{inr}(p)) = \text{inr}(p)$. 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 $\text{iso}^{-1}$. Expanding the table above with these translations yields:

$$
\begin{array}{c|cccccc}
\text{left} & 1 & 2 & 3 & 4 & 5 \\
\text{iso}(\text{left}) & \text{inl}(1) & \text{inl}(2) & \text{inr}(1) & \text{inr}(2) & \text{inl}(3) \\
g(\text{iso}(\text{left})) & \text{inl}(3) & \text{inl}(2) & \text{inr}(1) & \text{inr}(2) & \text{inl}(1) \\
\text{iso}^{-1}(g(\text{iso}(\text{left}))) & 5 & 2 & 3 & 4 & 1 \\
\end{array}
$$

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

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

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 skipped 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 $P(i) = \{p \mid p \in \text{live}(i)\}$. Conversely, given a container type in the sense of [1], we can define $P = \{(i, p) \mid p \in P(i)\}$ and 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 $(I, P, \text{live})$ 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 $\text{live} : I \to P(P)$ which tells for each shape which positions belong to it.

If $T = (I, P, \text{live})$ 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 $T(X) = \sum_{i \in I} \text{live}(i) \to X$. 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 $\text{live}(i)$—an entry taken from $X$.

Our aim is now to explain how the mapping $X \to T(X)$ lifts to a function 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 $T = (I, P, \text{live})$ be a container type. An edit $d_i \in \partial I$ is an insertion if $d_i i \geq i$ whenever defined. It is a deletion if $d_i i \leq i$ whenever defined. It is a rearrangement if $\text{live}(d_i i) \neq \text{live}(i)$ (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 $(I, P, \text{live})$ is a container type, $d_i \in \partial I$, and $f \in I \to P \to P$, then we say $f$ is consistent with $d_i$ if, whenever $d_i$ is defined, $f(i)$ restricted to live$(i)$ is a bijection to live$(d_i i)$. 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):

\[
\begin{align*}
\text{fail} & \\
\cup \{\text{mod}(p, dx) \mid p \in P, dx \in \partial X\} & \\
\cup \{\text{ins}(d) \mid d \text{ an insertion}\} & \\
\cup \{\text{del}(d) \mid d \text{ a deletion}\} & \\
\cup \{\text{rearr}(d, f) \mid f \text{ consistent with } d\}
\end{align*}
\]

In the last case, often either $d_i$ 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:

\[
\begin{align*}
\text{fail} (i, f) & \text{ is always undefined} \\
\text{mod}(p, dx) (i, f) & = (i, f[p \mapsto dx f(p)]) \text{ when } p \in \text{live}(i) \\
\text{ins}(d) (i, f) & = (d i, f') \\
del(d) (i, f) & = (d i, f\mid_\text{live}(d i)) \\
\text{rearr}(d, f) (i, g) & = (d i, g') \\
\text{where } g'(p) & = g(f(i)(p))
\end{align*}
\]

The mod($p, dx$) edit modifies the contents of position $p$ according to $dx$. If that position is absent the edit fails. The shape of the resulting container is unchanged. The ins($d$) edit alters the shape by $d_i$, growing the set of positions in the process (since $d_i i \geq i$). The new positions are filled with init$X$. The del($d$) 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 rearr($d, f$) 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
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 another 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 equal laws, in particular with a view to compact normal forms of container edits.

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

5.6 Example: For any module \( X \) we can construct the list module \( X^* \) as a particular container type \( (I, P, \text{live}) \) 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 \( \text{live}(n) = \{0, \ldots, n - 1\} \).

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

In Figure 8 we define the mapping lens turning \( \text{T}(\cdot) \) into an endofunctor on the category of lenses. We note that this is only the second lens to have a nontrivial complement (after the functor on the category of lenses. We note that this is only the basic edits defined above. In Section 6 we discuss the possibility of making such as a particular container type \( (I, P, \text{live}) \) that \( I \) 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 bijection correspondence, there is e.g. in the case of list reversal—no fixed order that, say, “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.

We also require that \( \ell \) 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 \( f_1, f_2, f_3 \) must be chosen in such a way that the container edits in which they appear are well-formed (this is possible since \( d_{i'} \) is an insertion, deletion, or restructuring as appropriate) and such that the following three constraints are satisfied: in each case \( i, i', \ell \), etc., refer to the current values from above and \( p \in \text{live}(\ell i') \) is an arbitrary position.

\[
\begin{align*}
\ell f_1(i, i') & = f_1(i, i') \quad f_2(i, i', i') \quad f_3(i, i') \\
\ell f_1(i, i') & = f_1(i, i') \quad f_2(i, i', i') \quad f_3(i, i') \\
\ell f_1(i, i') & = f_1(i, i') \quad f_2(i, i', i') \quad f_3(i, i')
\end{align*}
\]

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 \( \ell \) is an isomorphism lens, but we refrain from formulating details.

The bijection \( f_1 \) contains a little more choice, namely the behavior on the \( T \) positions in \( f_{1,i'}(\ell i', \text{live}(\ell i'), \text{live}(\ell (i'))) \). Fortunately, they all contain \( \text{init}_X \) so that the choice does not affect the resulting state after application of the edit.

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

![Figure 8: Generic container-mapping lens](image8.png)

![Figure 9: Container restructuring lens](image9.png)
live\((i') = i'\). The monoid \(\partial I\) has increment and decrement operations; the monoid \(\partial I^2\) 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 \(\mathsf{Unit}\) 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 \(d i = i + 2\) and that \(d i'\) (the result of \(d i\) propagated through \(\ell\)) installs two children at the leftmost node. In our in-order application we then have \(f_{\partial I, \partial I', \partial I, \partial I'} = \{(0, 0), (1, 0), (2, 01), (3, \varepsilon), (4, 1), (5, 11)\}\) and after applying both \(\text{ins}(d i)\) and \(\text{ins}(d i')\) we are in the as-yet-inconsistent situation depicted above.

To restore consistency we also apply \(\text{rear}(1, f_{i})\) where \(f_{i}(i') = \{(00, 0), (0, \varepsilon), (01, 1), (\varepsilon, 11), (1, 00), (11, 01)\}\). We could also have chosen \(f_{i}(i') = \{(00, 0), (0, \varepsilon), (01, 1), (\varepsilon, 11), (1, 00), (11, 01)\}\); this is precisely the additional freedom of choice. Of course \(f_{i}(i'')\) for \(i'' \neq i'\) is also completely unconstrained. After applying \(\text{rear}(1, f_{i})\) 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.

\[
X \otimes Y = ([X] \times [Y'), \langle \text{init}_X, \text{init}_Y \rangle, \partial X \otimes \partial Y, \odot_{X \otimes Y})
\]

\[
1_{M \otimes N} = (1_M, 1_N)
\]

\[
(m, n)_{M \otimes N} (m', n') = (m m', n n')
\]

\[
(dx, dy) \odot_{X \otimes Y} (x, y) = (dx x, dy y)
\]

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:

\[
\begin{align*}
\text{left}(1) &= \varepsilon \\
\text{left}(dx) \text{left}(dx') &= \text{left}(dx dx') \\
\text{right}(1) &= \varepsilon \\
\text{right}(dy) \text{right}(dy') &= \text{right}(dy dy') \\
\text{left}(dx) \text{right}(dy) &= \text{right}(dy) \text{left}(dx)
\end{align*}
\]

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 equalational 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

\[
\text{left}(dx) \odot_g \text{right}(dy) \odot_g (x, y) = \text{right}(dy) \odot_g \text{left}(dx)_y \odot_g (x, y)
\]

Simple calculation shows that both sides are equal to \((dx x, dy y)\), so this law is respected; the rest follow similar lines.

Most importantly, we must check that the \(\Rightarrow\) and \(\Leftarrow\) 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:

\[
\text{left}(dx) \text{right}(dy) = \text{right}(dy) \text{left}(dx)
\]

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 \(\ell\) are lenses, then the \(\Rightarrow_g\) and \(\Leftarrow_g\) 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 \(X \otimes \mathsf{Unit}\) and \(X\). Similarly, let \(g\) be the obvious isomorphism between \(\mathsf{Unit} \otimes Y\) and \(Y\). We can then define \(\pi_1 = (id_X \otimes \text{term}_Y); \text{iso}_f\) and \(\pi_2 = (\text{term}_X \otimes id_Y); \text{iso}_g\). Thus, \(\pi_1\) first throws away any information in the right-hand part of a tuple with \(\text{term}_Y\), 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:

\[
\begin{align*}
\text{switch}_{ik}(m) \text{switch}_{ij}(m') &= \text{switch}_{ik}(m) \\
\text{switch}_{ij}(m) \text{stay}_{ij}(m') &= \text{switch}_{ij}(m) \\
\text{stay}_{ij}(m) \text{switch}_{ij}(m') &= \text{switch}_{ij}(mm') \\
\text{stay}_{ij}(m) \text{stay}_{ij}(m') &= \text{stay}_{ij}(mm') \\
d d' = \text{fail} & \quad \text{in all other cases}
\end{align*}
\]

\(^5\) 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 combiners, \( \text{switch}_L \) and \( \text{switch}_R \), 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 \( \text{switch}_L \) and \( \text{switch}_R \) forms, this would not be possible. In particular, we need to represent the fact that a \( \text{stay}_L \) edit followed by a \( \text{switch}_L \) edit fails when applied to a value tagged with \( \text{inr} \).

As with products, we must check that the remaining definitions are well-formed. In particular, it can be shown that, in the module if \( k \) and \( ℓ \) are lenses, then \( (k \oplus ℓ) \Rightarrow_g \) and \( (k \oplus ℓ), \lll_g \) respect the above equations.

Unfortunately, the \textit{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 insertions and deletions so that insertions and deletions in the above equations.

(6) \( K = \{(x, c, y) \mid ℓ.\text{putr}(x, c) = (y, c)\} \). \( \partial_yℓ \) is a symmetric edit lens and the passage from \( ℓ \) to \( ∂ℓ \) is compatible with the equivalences on symmetric lenses and symmetric edit lenses.

Let \( X \) be a module. A \textit{differ} for \( X \) is a binary operation \( \text{dif} \in X × X → ∂X \) satisfying \( \text{dif}(x, x′) = x′ = \text{dif}(x, x) = 1 \).

Thus, a differ finds, for given states \( x, x′ \), an edit operation \( dx \) such that \( dx \cdot x = x′ \) and \( dx \) is "reasonable" at least in the sense that if \( x = x′ \) then the produced edit is minimal, namely \( I \). For example, the module \( X_s \) for \( X \) and \( x, \in X \) admits the \textit{canonical differ} given by \( \text{dif}(x, x′) = x′ \) if \( x ≠ x′ \) and \( \text{dif}(x, x) = ε \), otherwise.

Given an edit lens \( ℓ \) between modules \( X \) and \( Y \), both equipped with differers, we define a symmetric lens \( |ℓ| \) between \([X] \) and \([Y]\) by (1) \( |ℓ|.C = [X] × ℓ.C × [Y] \); (2) \( |ℓ|.\text{init} = (\text{init}_X, ℓ.\text{init}, \text{init}_Y) \); (3) \( |ℓ|.\text{putr}(x, (x_0, c_0, y_0)) = (dy, (x, c, dy, y_0)) \) where \( dx = \text{dif}(x_0, x) \) and \( (c, c′) = ℓ.(dx, c) \); (4) an analogous definition of \( |ℓ|.\text{pull} \). This defines a symmetric lens \( |ℓ| \) between \([X] \) and \([Y]\) and the passage \( ℓ → |ℓ| \) is compatible with lens equivalence.

7. From State-Based to Edit Lenses and Back

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

\[
\begin{align*}
\text{putr} & \in X \times C → Y \times C \\
\text{pull} & \in Y \times C → X \times C
\end{align*}
\]

satisfying the following round-tripping laws:

\[
\begin{align*}
\text{putr}(x, c) & = (y, c′) \\
\text{pull}(y, c′) & = (x, c)
\end{align*}
\]

(\text{PutRL})

\[
\begin{align*}
\text{putr}(y, c′) & = (x, c′) \\
\text{pull}(x, c′) & = (y, c′)
\end{align*}
\]

(\text{PutLR})

Equivalence of state-based symmetric lenses is defined through the existence of a simulating relation between the respective complements \( C \). 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 (\textit{edit}s) are lists of elements of \( X \) modulo the equality \( xx = x \). An action of \( ∂X \) on \( X \) is defined as \( ex = x \) and \( (xwy) = y \) where \( x \in X, w \in X^* \). Note that this is well defined as \( x(yx) = x.y \).

If, in addition, we have a distinguished element \( x \in X \), we thus obtain a module denoted \( X \), where \( |X| = X \) and \( \text{init}_X = x \) and \( ∂X = ∂X \).

Let \( ℓ \) be a state-based symmetric lens between \( X \) and \( Y \) along with elements \( x \in X \) and \( y \in Y \) satisfying \( ℓ.\text{putr}(x, ℓ.\text{missing}) = (y, ℓ.\text{missing}) \). We then define a symmetric edit lens \( ∂_yℓ \) between the modules \( X \) and \( Y \) as follows: (1) \((∂_yℓ).C = ℓ.C; \)

(2) \((∂_yℓ).\text{init} = ℓ.\text{missing}; \)

(3) \((∂_yℓ).\text{pull}(x, c) = (x, c) \); (4) \((∂_yℓ).\text{putr}(x, c) = (y, c′) \); (5) analogous definitions for \( ∂xℓ \) and \( ℓ.\text{putr}(x, c′) = (y, c′) \).

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, \textit{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 then modeled as categories (presented as reflexive graphs). Thus, for any two states \( x, x′ \) there is a set of edits \( X(x, x′) \). An \textit{sd-lens} then comprises two reflexive graphs \( X, Y \) and for any \( x \in X \) and \( y \in Y \) a set \( |X(x, y)| \) of "correspondences" which roughly correspond to our complements. Forward and backward operations similar to our \( \lll \) and \( \ggg \) then complete the picture. No concrete examples are given of sd-lenses, no composition, no notion of equivalence, and no combinator 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 \textit{hippocratism} 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’ seminal paper on \textit{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


