# Symmetric Lenses

Martin Hofmann
Ludwig-Maximilians-Universität

Benjamin Pierce
University of Pennsylvania

Daniel Wagner
University of Pennsylvania

## Abstract

*Lenses*—bidirectional transformations between pairs of connected structures—have been extensively studied and are beginning to find their way into industrial practice. However, some aspects of their foundations remain poorly understood. In particular, most previous work has focused on the special case of *asymmetric lenses*, where one of the structures is taken as primary and the other is thought of as a projection, or view. A few studies have considered symmetric variants, where each structure contains information not present in the other, but these all lack the basic operation of *composition*. Moreover, while many domain-specific languages based on lenses have been designed, lenses have not been thoroughly studied from a more fundamental algebraic perspective.

We offer two contributions to the theory of lenses. First, we present a new symmetric formulation, based on *complements*, an old idea from the database literature. This formulation generalizes the familiar structure of asymmetric lenses, and it admits a good notion of composition. Second, we explore the algebraic structure of the space of symmetric lenses. We present generalizations of a number of known constructions on asymmetric lenses and settle some longstanding questions about their properties—in particular, we prove the existence of (symmetric monoidal) tensor products and sums and the *non*-existence of full categorical products or sums in the category of symmetric lenses. We then show how the methods of universal algebra can be applied to build *iterator lenses* for structured data such as lists and trees, yielding lenses for operations like mapping, filtering, and concatenation from first principles. Finally, we investigate an even more general technique for constructing mapping combinators, based on the theory of *containers*.

## 1. Introduction

The electronic world is rife with partially synchronized data—replicated structures that are not identical but that share some common parts, and where the shared parts need to be kept up to date as the structures change. Examples include databases and materialized views, in-memory and on-disk representations of heap structures, connected components of user interfaces, and models representing different aspects of the same software system.

In current practice, the propagation of changes between connected structures is mostly handled by *ad hoc* methods: given a pair of structures $X$ and $Y$, we write one transformation that maps changes to $X$ into changes to $Y$ and a separate transformation that maps $Y$ changes to $X$ changes. When the structures involved are complex, managing such pairs of transformations manually can be a maintenance nightmare.

This has led to a burgeoning interest in *bidirectional programming languages*, in which every expression denotes a related pair of transformations. A great variety of bidirectional languages have been proposed (see [8, 13] for recent surveys), and these ideas are beginning to see commercial application, e.g., in RedHat's system administration tool, Augeas [21].

One particularly well-studied class of bidirectional programming languages is the framework of *lenses* introduced in [11]. Prior work on lenses and lens-like structures has mostly been carried out in specific domains—designing combinators for lenses that work over strings [5, 7, 12], trees [11, 17, 20, 25], relations [6], graphs [16], or software models [9, 10, 15, 26, 27, 30]. By contrast, our aim in this paper is to advance the *foundations* of lenses in two significant respects.

First, we show that lenses can be generalized from their usual asymmetric presentation—where one of the structures is always "smaller"—to a fully *symmetric* version where each of the two structures may contain information that is not present in the other (Section 2). This generalization is significantly more expressive than any previously known: although symmetric variants of lenses have been studied [10, 23, 28], they all lack a notion of sequential composition of lenses, a significant technical and practical limitation (see Section 10). As we will see, the extra structure that we need to support composition is nontrivial; in particular, constructions involving symmetric lenses need to be proved correct modulo a notion of *behavioral equivalence* (Section 3).

Second, we undertake a systematic investigation of the *algebraic structure* of the space of lenses, using the concepts of elementary category theory as guiding and organizing principles. Our presentation is self contained, but readers may find some prior familiarity with basic concepts of category theory helpful. Most proofs are omitted for brevity; they can be found in a long version of the paper, available from the second author's web page.

We begin this algebraic investigation with some simple generic constructions on symmetric lenses: composition, dualization, terminal lenses, simple bijections, etc. (Section 4). We then settle some basic questions about products and sums (Sections 5 and 6). In particular, it was previously known that asymmetric lenses admit constructions intuitively corresponding to pairing and projection [7] and another construction that is intuitively like a sum [11]. However, these constructions were not very well understood; in particular, it was not known whether the pairing and projection operations formed a full categorical product, while the injection arrows from $X$ to $X + Y$ and from $Y$ to $X + Y$ were not definable at all in the asymmetric setting. We prove that the category of symmetric lenses does *not* have full categorical products or sums, but does have "symmetric monoidal" structures with many of the useful properties of products and sums.

Next, we consider how to build lenses over more complex data structures such as lists and trees (Section 7). We first observe that the standard construction of algebraic datatypes can be lifted straightforwardly from the category of sets to the category of lenses. For example, from the definition of lists as the least solution of the equation $L(X) \simeq \mathit{Unit} + X \times L(X)$ we obtain a lens connecting the set $L(X)$ with the set $\mathit{Unit} + X \times L(X)$; the two directions of this lens correspond to the *unfold* and *fold* operations on lists. Moreover, the familiar notion of *initial algebra* also generalizes to lenses, giving us powerful iterators that allow

for a modular definition of many symmetric lenses on lists and trees—e.g., mapping a symmetric lens over a list, filtering, reversing, concatenating, and translating between lists and trees.

Finally, we briefly investigate an even more general technique for constructing "mapping lenses" that apply the action of a given sublens to all the elements of some data structure (Section 8). This technique applies not only to algebraic data structures but to an arbitrary *container* in the sense of Abbot, Altenkirch, and Ghani [2]. This extends the variety of list and tree mapping combinators that we can construct from first principles to include non-inductive datatypes such as labeled dags and graphs.

We carry out these investigations in the richer space of symmetric lenses, but many of the results and techniques should also apply to the special case of asymmetric lenses. Indeed, we can show (Section 9) that asymmetric lenses form a subcategory of symmetric ones in a natural way: every asymmetric lens can be embedded in a symmetric lens, and many of the algebraic operators on symmetric lenses specialize to known constructions on asymmetric lenses. Conversely, a symmetric lens can be factored into a "back to back" assembly of two asymmetric ones.

Sections 10 and 11 discuss related and future work.

## 2. Fundamental Definitions

***Asymmetric Lenses*** To set the stage, let's review the standard definition of asymmetric lenses. (Other definitions can be given, featuring both weaker and stronger laws, but this version is perhaps the most widely accepted. We discuss some alternatives in Section 10.) Suppose $X$ is some set of source structures (say, the possible states of a database) and $Y$ a set of target structures (views of the database). An asymmetric lens from $X$ to $Y$ comprises two functions:

$$\begin{array}{rcl} get & \in & X \to Y \\ put & \in & Y \times X \to X \end{array}$$

The $get$ component is the forward transformation, a total function from $X$ to $Y$. The $put$ component takes an old $X$ and a modified $Y$ and yields a correspondingly modified $X$. Furthermore, every lens must obey two "round-tripping" laws for every $x \in X$ and $y \in Y$:

$$put\ (get\ x)\ x = x \qquad \text{(GETPUT)}$$

$$get\ (put\ y\ x) = y \qquad \text{(PUTGET)}$$

It is also useful to be able to create an element of $x$ given just an element of $y$, with no "original $x$" to put it into; in order to handle this in a uniform way, we assume that each lens also comes equipped with a function $create \in Y \to X$ and one more axiom:

$$get\ (create\ y) = y \qquad \text{(CREATEGET)}$$

***Complements*** The key step toward symmetric lenses is the notion of *complements*. The idea, which dates back to a famous paper in the database literature on the view update problem [4] and was adapted to lenses in [5] (and, for a slightly different definition, [22]), is quite simple. If we think of the $get$ component of a lens as a sort of projection function, then there is another projection from $X$ into some set $C$ that keeps all the information discarded by $get$. Equivalently, we can think of $get$ as returning two results—an element of $Y$ and an element of $C$—that together contain all the information needed to reconstitute the original element of $X$. The $put$ function doesn't need a whole $x \in X$ to recombine with some updated $y \in Y$, either— it can just take the complement $c \in C$ generated from $x$ by the $get$, since this will contain all the information that is missing from $y$. Moreover, instead of a separate $create$ function, we can simply pick a distinguished element $missing \in C$ and define $create(y)$ as $put(y, missing)$.

Formally, an *asymmetric lens with complement* mapping between $X$ and $Y$ consists of a set $C$, a distinguished element $missing \in C$, and two functions

$$\begin{array}{rcl} get & \in & X \to Y \times C \\ put & \in & Y \times C \to X \end{array}$$

obeying the following laws for every $x \in X$, $y \in Y$, and $c \in C$:[1]

$$\frac{get\ x = (y, c)}{put\ (y, c) = x} \qquad \text{(GETPUT)}$$

$$\frac{get\ (put\ (y, c)) = (b', c')}{b' = y} \qquad \text{(PUTGET)}$$

Note that the type is just "lens from $X$ to $Y$": the set $C$ is an internal component, not part of the externally visible type. In type-theoretic notation, we could write $Lens(X, Y) = \exists C.\ \{missing : C,\ get : X \to Y \times C,\ put : Y \times C \to X\}$.

***Symmetric Lenses*** Now we can symmetrize. First, instead of having only $get$ return a complement, we make $put$ return a complement, too, and we take this complement as a second argument to $get$. So we have $get \in X \times C_Y \to Y \times C_X$ and $put \in Y \times C_X \to X \times C_Y$. Intuitively, $C_X$ is the "information from $X$ that is discarded by $get$," and $C_Y$ is the "information from $Y$ that is discarded by $put$." Next, we observe that we can, without loss of generality, use the same set $C$ as the complement in both directions. So now we have $get \in X \times C \to Y \times C$ and $put \in Y \times C \to X \times C$. Intuitively, we can think of the combined complement $C$ as $C_X \times C_Y$—that is, each complement contains some "private information from $X$" and some "private information from $Y$"; by convention, the $get$ function reads the $C_Y$ part and writes the $C_X$ part, while the $put$ reads the $C_X$ part and writes the $C_Y$ part. Lastly, now that everything is symmetric, the $get$ / $put$ distinction is not helpful, so we rename the functions to $putr$ and $putl$. This brings us to our core definition.

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

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

satisfying the following round-tripping laws:

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

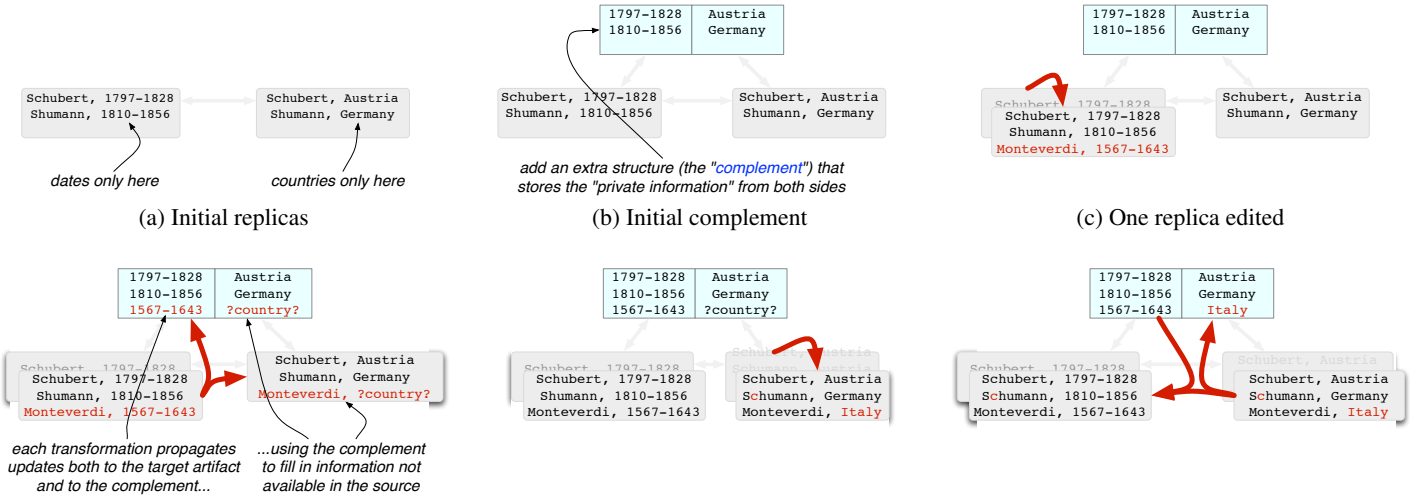
$$\frac{putl(y, c) = (x, c')}{putr(x, c') = (y, c')} \qquad \text{(PUTLR)}$$

When several lenses are under discussion, we use record notation to identify their parts, writing $\ell.C$ for the complement set of $\ell$, etc.

The force of the PUTRL and PUTLR laws is to establish some "consistent" or "steady-state" triples $(x, y, c)$, for which $put$s of $x$ from the left or $y$ from the right will have no effect. In general, a $put$ of a new $x'$ from the left entails finding a $y'$ and a $c'$ that restore consistency. We will use the equation $putr(x, c) = (y, c)$ to characterize the steady states.

One can imagine other laws. In particular, the long version of the paper considers symmetric forms of "PUTPUT" laws, which specify that two $put$ operations in a row should have the same effect as the second one alone. As with asymmetric lenses, these laws appear too strong to be desirable in practice.

[1] We can convert back and forth between the two presentations; in particular, if $(get, put, create)$ are the components of a traditional lens, then we define a canonical complement by $C = \{f \in Y \to X \mid \forall y.\ get(f(y)) = y\}$. We then define the components $missing'$, $get'$, and $put'$ of an asymmetric lens with complement as $missing' = create$ and $get'(x) = (get(x), \lambda y.put(y, x))$ and $put'(y, f) = f(y)$.




(a) Initial replicas (b) Initial complement (c) One replica edited

(d) Propagating the edit (e) Second replica is edited (f) This change is propagated

**Figure 1.** Behavior of a symmetric lens

***Examples*** Figure 1 illustrates the use of a symmetric lens. The structures in this example are lists of textual records describing composers. The partially synchronized records (a) have a name and two dates on the left and a name and a country on the right. The complement (b) contains all the information that is discarded by both *puts*—all the dates from the left-hand structure and all the countries from the right-hand structure. (It can be viewed as a pair of lists of strings, or equivalently as a list of pairs of strings; the way we build list lenses later actually corresponds to the latter.) If we add a new record to the left hand structure (c) and use the *putr* operation to propagate it through the lens (d), we copy the shared information (the new name) directly from left to right, store the private information (the new dates) in the complement, and use a default string to fill in both the private information on the right and the corresponding right-hand part of the complement. If we now update the right-hand structure to fill in the missing information and correct a typo in one of the other names (e), then a *putl* operation will propagate the edited country to the complement, propagate the edited name to the other structure, and use the complement to restore the dates for all three composers.

Viewed a little more abstractly, the connection between the information about a single composer in the two tables is a lens from $X \times Y$ to $Y \times Z$, with complement $X \times Z$—let's call it $e$. Its *putr* component updates the $X$ part of the complement and uses the $Z$ part (together with the $Y$ from its input) to build its output; the *putl* component does the opposite. Then the top-level lens in Figure 1—let's call it $e^\star$—abstractly has type $(X \times Y)^\star \leftrightarrow (Y \times Z)^\star$ and can be thought of as the "lifting" of $e$ from elements to lists.

There are several plausible implementations of $e^\star$, giving rise to slightly different behaviors when list elements are added and removed—i.e., when the input and complement arguments to *putr* or *putl* are lists of different lengths. One possibility is to take $e^\star.C = (e.C)^\star$ and either truncate the complement list if it is longer or pad it (with $e.missing$) if it is shorter. For example, taking $X = \{a, b, c, \ldots\}$, $Y = \{1, 2, 3, \ldots\}$, $Z = \{A, B, C, \ldots\}$, and $e.missing = (z, Z)$, we have:

$$\begin{array}{ll} & putr([(a,1)],\ [(m,M),(n,N)]) \\ = & putr([(a,1)],\ [(m,M)]) \\ = & ([(1,M)],\ [(a,M)]) \end{array}$$

$$\begin{array}{ll} & putr([(a,1),(b,2)],\ [(a,M)]) \\ = & putr([(a,1),(b,2)],\ [(a,M),(z,Z)]) \\ = & ([(1,M),(2,Z)],\ [(a,M),(b,Z)]) \end{array}$$

Notice that, after the first *putr*, the information in the second component of the complement (containing the value $N$) is lost. Even though the second *putr* restores the second element of the list, the value $N$ is gone forever; what's left is the default value $Z$.

A slightly sneakier—and arguably better behaved—possibility is to keep an *infinite* list of complements. Whenever we do a *put*, we use (and update) a prefix of the complement list of the same length as the current value being *put*, but we keep the infinite tail so that, later, we have values to use when the list being *put* is longer.

$$\begin{array}{ll} & putr([(a,1)],\ [(m,M),(n,N),(z,Z),(z,Z),\ldots]) \\ = & ([(1,M)],\ [(a,M),(n,N),(z,Z),(z,Z),\ldots]) \end{array}$$

$$\begin{array}{ll} & putr([(a,1),(b,2)],\ [(a,M),(n,N),(z,Z),(z,Z),\ldots]) \\ = & ([(1,M),(2,N)],\ [(a,M),(b,N),(z,Z),\ldots]) \end{array}$$

We call the first form the *forgetful* list mapping lens and the second the *retentive* list mapping lens. We will see, later, that the difference between these two precisely boils down to a difference in the behavior of the lens-summing operator $\oplus$ in the specification $e^\star \simeq id_{Unit} \oplus (e \otimes e^\star)$ of the list mapping lens.

***Alignment*** One important *non*-goal of the present paper is dealing with the (critical) issue of *alignment* [5, 7]. We consider here only the simple case of lenses that work "positionally." For example, the lens $e^\star$ in the example will always use $e$ to propagate changes between the first element of $x$ and the first element of $y$, between the second element of $x$ and the second of $y$, and so on. This amounts to assuming that the lists are edited either by editing individual elements in-place or by adding or deleting elements at the end of the list; if an actual edit inserts an element at the head of one of the lists, positional alignment will produce surprising (and probably distressing) results. We see two avenues for incorporating richer notions of alignment: either we can generalize the mechanisms of *matching lenses* [5] to the setting of symmetric lenses, or we can refine the whole framework of symmetric lenses with a notion of *delta propagation* (see Section 11).

## 3. Equivalence

Since each lens carries its own complement—and since this need not be the same as the complement of another lens with the same domain and codomain—we need to define what it means for two lenses to be indistinguishable (in the sense that no user could ever tell the difference between them by observing just the $X$ and $Y$

parts of their outputs). We will use this relation pervasively in what follows: indeed, most of the laws we would like our constructions to validate—even things as basic as associativity of composition—will not hold "on the nose," but only up to equivalence.

**3.1 Definition:** Given $X, Y, C_f, C_g$ and a relation $R \in C_f \times C_g$, we say that functions $f \in X \times C_f \to Y \times C_f$ and $g \in X \times C_g \to Y \times C_g$ are *$R$-similar*, written $f \sim_R g$, if they take inputs with $R$-related complements to equal outputs with $R$-related complements:

$$\frac{(c_f, c_g) \in R \quad f(x, c_f) = (y_f, c'_f) \quad g(x, c_g) = (y_g, c'_g)}{y_f = y_g \wedge (c'_f, c'_g) \in R}$$

**3.2 Definition [Lens equivalence]:** Two lenses $k$ and $\ell$ are *equivalent* (written $k \equiv \ell$) if there is a relation $R \in k.C \times \ell.C$ with

1. $(k.missing, \ell.missing) \in R$
2. $k.putr \sim_R \ell.putr$
3. $k.putl \sim_R \ell.putl$.

We write $X \iff Y$ for the set of equivalence classes of lenses from $X$ to $Y$. When $\ell$ is a lens, we write $[\ell]$ for the equivalence class of $\ell$ (that is, $\ell \in X \leftrightarrow Y$ iff $[\ell] \in X \iff Y$). Where no confusion results, we abuse notation and drop these brackets, using $\ell$ for both a lens and its equivalence class.

We show in the long version that this definition of lens equivalence coincides with a more "observational" definition where two lenses are equivalent iff they always give the same sequence of outputs when presented with the same sequence of inputs, starting with a $missing$ complement.

## 4. Basic Constructions

With the basic definitions in hand, we can now begin defining lenses. We begin in this section with several relatively simple constructions.

**4.1 Definition [Identity lens]:** Let $Unit$ be a distinguished singleton set and $()$ is its only element.

$$id_X \in X \leftrightarrow X$$

$$\begin{array}{lcl} C & = & Unit \\ missing & = & () \\ putr(x, ()) & = & (x, ()) \\ putl(x, ()) & = & (x, ()) \end{array}$$

To check that this definition is well formed, we must show that the components defined in the lower box satisfy the round-trip laws implied by the upper box. This proof and analogous ones for later lens definitions are given in full in the long version.

**4.2 Definition [Lens composition]:**

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

$$\begin{array}{lcl} C & = & k.C \times \ell.C \\ missing & = & (k.missing, \ell.missing) \\ putr(x, (c_k, c_\ell)) & = & \mathsf{let}\ (y, c'_k) = k.putr(x, c_k)\ \mathsf{in} \\ & & \quad \mathsf{let}\ (z, c'_\ell) = \ell.putr(y, c_\ell)\ \mathsf{in} \\ & & \quad (z, (c'_k, c'_\ell)) \\ putl(z, (c_k, c_\ell)) & = & \mathsf{let}\ (y, c'_\ell) = \ell.putl(z, c_\ell)\ \mathsf{in} \\ & & \quad \mathsf{let}\ (x, c'_k) = k.putl(y, c_k)\ \mathsf{in} \\ & & \quad (x, (c'_k, c'_\ell)) \end{array}$$

This definition specifies what it means to compose two individual lenses. To show that this definition lifts to equivalence classes of lenses, we need to check the following congruence property. Again, the proof of this property and similar lemmas for the operators on lenses that we define below can be found in the long version.

**4.3 Lemma [Composition preserves equivalence]:** If $k \equiv k'$ and $\ell \equiv \ell'$, then $k; \ell \equiv k'; \ell'$.

**4.4 Lemma [Associativity of composition]:** $j; (k; \ell) \equiv (j; k); \ell$. (The equivalence is crucial here: $j; (k; \ell)$ and $(j; k); \ell$ are not the same lens because their complements are structured differently.)

**4.5 Lemma [Identity arrows]:** The identity arrow is a left and right identity for composition: $id_X; \ell \equiv \ell; id_Y \equiv \ell$.

Thus symmetric lenses form a category, LENS, with sets as objects and equivalence classes of lenses as arrows. The identity arrow for a set $X$ is $[id_X]$. Composition is $[k]; [\ell] = [k; \ell]$.

**4.6 Proposition [Bijective lenses]:**

$$\frac{f \in X \to Y \qquad f \text{ bijective}}{bij_f \in X \leftrightarrow Y}$$

$$\begin{array}{lcl} C & = & Unit \\ missing & = & () \\ putr(x, ()) & = & (f(x), ()) \\ putl(y, ()) & = & (f^{-1}(y), ()) \end{array}$$

(If we were designing *syntax* for a bidirectional language, we might not want to include $bij$, since we would then need to offer programmers some notation for writing down bijections in such a way that we can verify that they *are* bijections and derive their inverses. However, even if it doesn't appear in the surface syntax, we will see several places where $bij$ is useful in talking about the algebraic theory of symmetric lenses.)

This transformation (and several others) respect much of the structure available in our category. Formally, $bij$ is a functor. Recall that a a *covariant* (respectively, *contravariant*) *functor* between categories $\mathcal{C}$ and $\mathcal{D}$ is a pair of maps—one from objects of $\mathcal{C}$ to objects of $\mathcal{D}$ and the other from arrows of $\mathcal{C}$ to arrows of $\mathcal{D}$—that preserve typing, identities, and composition:

- The image of any arrow $f : X \to Y$ in $\mathcal{C}$ has the type $F(f) : F(X) \to F(Y)$ (respectively, $F(f) : F(Y) \to F(X)$) in $\mathcal{D}$.
- For every object $X$ in $\mathcal{C}$, we have $F(id_X) = id_{F(X)}$ in $\mathcal{D}$.
- If $f; g = h$ in $\mathcal{C}$, then $F(f); F(g) = F(h)$ (respectively, $F(g); F(f) = F(h)$) in $\mathcal{D}$.

Covariant functors are simply called functors. When it can be inferred from the arrow mapping, the object mapping is often elided.

**4.7 Lemma:** The $bij$ operator forms a functor from the category ISO, whose objects are sets and whose arrows are isomorphic functions, to LENS—that is, $bij_{id_X} = id_X$ and $bij_f; bij_g = bij_{f;g}$.

**4.8 Definition [Dual of a lens]:**

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

$$\begin{array}{lcl} C & = & \ell.C \\ missing & = & \ell.missing \\ putr(y, c) & = & \ell.putl(y, c) \\ putl(x, c) & = & \ell.putr(x, c) \end{array}$$

It is easy to see that $(-)^{op}$ is involutive—that is, that $(\ell^{op})^{op} = \ell$ for every $\ell$—and that $bij_{f^{-1}} = bij_f^{op}$ for any bijective $f$. Recalling that an endofunctor is a functor whose source and target categories are identical, we can also show the following lemma.

**4.9 Lemma:** The $(-)^{op}$ operation can be lifted to a contravariant endofunctor on the category LENS mapping each object to itself and each arrow $[\ell]$ to $[\ell^{op}]$.

**4.10 Corollary:** The category LENS is self dual. (Note that this does not mean that each arrow is its own inverse!)

**4.11 Definition [Terminal lens]:**

$$\frac{x \in X}{term_x \in X \leftrightarrow Unit}$$

$$\begin{array}{lcl} C & = & X \\ missing & = & x \\ putr(x', c) & = & ((), x') \\ putl((), c) & = & (c, c) \end{array}$$

**4.12 Proposition [Uniqueness of terminal lens]:** Lenses with the same type as a terminal lens are equivalent to a terminal lens. More precisely, suppose $k \in X \leftrightarrow Unit$ and $k.putl((), k.missing) = (x, c)$. Then $k \equiv term_x$.

**4.13 Definition [Disconnect lens]:**

$$\frac{x \in X \qquad y \in Y}{disconnect_{xy} \in X \leftrightarrow Y}$$

$$disconnect_{xy} = term_x; term_y^{op}$$

The disconnect lens does not synchronize its two sides at all. The complement, $disconnect.C$, is $X \times Y$; inputs are squirreled away into one side of the complement, and outputs are retrieved from the other side of the complement.

## 5. Products

A few more notions from elementary category theory will be useful at this point for giving us ideas about what sorts of properties to look for and for structuring the discussion of which of these properties hold and which fail for lenses.

The *categorical product* of two objects $X$ and $Y$ is an object $X \times Y$ and arrows $\pi_1 : X \times Y \to X$ and $\pi_2 : X \times Y \to Y$ such that for any two arrows $f : Z \to X$ and $g : Z \to Y$ there is a unique arrow $\langle f, g\rangle : Z \to X \times Y$—the *pairing* of $f$ and $g$—satisfying $\langle f, g\rangle; \pi_1 = f$ and $\langle f, g\rangle; \pi_2 = g$. It is well known that, if a categorical product exists at all, it is unique up to isomorphism. If a category $\mathcal{C}$ has a product for each pair of objects, we say that $\mathcal{C}$ has products.

**5.1 Theorem:** LENS does not have products.

**Proof:** Uniqueness of pairing shows that there is exactly one lens from $Unit$ to $Unit \times Unit$ (whatever this may be). Combined with Prop. 4.12 this shows that $Unit \times Unit$ is a one-element set. This, on the other hand, contradicts the symmetric nature of $lens$. (A more detailed proof appears in the full paper.) □

However, LENS *does* have a similar (but weaker) structure: a *tensor product*—i.e., an associative, two-argument functor. For any two objects $X$ and $Y$, we have an object $X \otimes Y$, and for any two arrows $f : A \to X$ and $g : B \to Y$, an arrow $f \otimes g : A \otimes B \to X \otimes Y$ such that $(f_1; f_2) \otimes (g_1; g_2) = (f_1 \otimes g_1); (f_2 \otimes g_2)$ and $id_X \otimes id_Y = id_{X \otimes Y}$. Furthermore, for any three objects $X, Y, Z$ there is a natural isomorphism $\alpha_{X,Y,Z} : (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z)$ satisfying certain coherence conditions (which specify that all ways of re-associating a quadruple are equal).

A categorical product is always a tensor product (by defining $f \otimes g = \langle \pi_1; f, \pi_2; g\rangle$), and conversely a tensor product is a categorical product if there are natural transformations $\pi_1, \pi_2, diag$

$$\begin{array}{c} \pi_{1,X,Y} \in X \otimes Y \to X \\ \pi_{2,X,Y} \in X \otimes Y \to Y \\ diag_X \in X \to X \otimes X \end{array}$$

such that (suppressing subscripts to reduce clutter)

$$\begin{array}{rcll} (f \otimes g); \pi_1 & = & \pi_1; f & (1) \\ (f \otimes g); \pi_2 & = & \pi_2; g & (2) \\ diag; (f \otimes f) & = & f; diag & (3) \\ diag; \pi_1 & = & id & (4) \\ diag; \pi_2 & = & id & (5) \\ diag; (\pi_1 \otimes \pi_2) & = & id & (6) \end{array}$$

for all arrows $f$ and $g$. Building a categorical product from a tensor product is not the most familiar presentation, but it can be shown to be equivalent (see Proposition 13 in [3], for example).

In the category LENS, we can build a tensor product and can also build projection lenses with reasonable behaviors. However, these projections are not quite natural transformations—laws 1 and 2 above hold only with an additional indexing constraint for particular $f$ and $g$. More seriously, while it seems we can define some reasonable natural transformations with the type of $diag$ (that is, lenses satisfying law 3), none of them satisfy the final three laws.

**5.2 Definition [Tensor product lens]:**

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

$$\begin{array}{lcl} C & = & k.C \times \ell.C \\ missing & = & (k.missing, \ell.missing) \\ putr((x, y), (c_k, c_\ell)) & = & \mathsf{let}\ (z, c'_k) = k.putr(x, c_k)\ \mathsf{in} \\ & & \mathsf{let}\ (w, c'_\ell) = \ell.putr(y, c_\ell)\ \mathsf{in} \\ & & ((z, w), (c'_k, c'_\ell)) \\ putl((z, w), (c_k, c_\ell)) & = & \mathsf{let}\ (x, c'_k) = k.putl(z, c_k)\ \mathsf{in} \\ & & \mathsf{let}\ (y, c'_\ell) = \ell.putl(w, c_\ell)\ \mathsf{in} \\ & & ((x, y), (c'_k, c'_\ell)) \end{array}$$

**5.3 Lemma [Product bijection]:** For bijections $f$ and $g$,

$$bij_f \otimes bij_g \equiv bij_{f \times g}.$$

In fact, the particular tensor product defined above is very well behaved: it induces a *symmetric monoidal category*—i.e., a category with a unit object 1 and the following natural isomorphisms:

$$\begin{array}{rcl} \alpha_{X,Y,Z} & : & (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z) \\ \lambda_X & : & 1 \otimes X \to X \\ \rho_X & : & X \otimes 1 \to X \\ \gamma_{X,Y} & : & X \otimes Y \to Y \otimes X \end{array}$$

These are known as the *associator*, *left-unitor*, *right-unitor*, and *symmetry*, respectively. In addition to the equations implied by these being natural isomorphisms, they must also satisfy some coherence conditions (given in the full version).

**5.4 Proposition [LENS,⊗ is a symmetric monoidal category]:** In the category SET, the Cartesian product is a bifunctor with $Unit$ as unit, and gives rise to a symmetric monoidal category. Let

$\alpha^\times, \lambda^\times, \rho^\times, \gamma^\times$ be associator, left-unitor, right-unitor, and symmetry natural isomorphisms. Then the $\otimes$ bifunctor also gives rise to a symmetric monoidal category of lenses, with $Unit$ as unit and $\alpha^\otimes = bij \circ \alpha^\times$, $\lambda^\otimes = bij \circ \lambda^\times$, $\rho^\otimes = bij \circ \rho^\times$, and $\gamma^\otimes = bij \circ \gamma^\times$ as associator, left-unitor, right-unitor, and symmetry, respectively.

Knowing that LENS is a symmetric monoidal category is useful for several reasons. First, it tells us that, even though it is not quite a full-blown product, the tensor construction is algebraically quite well behaved. Second, it justifies a convenient intuition where lenses built from multiple tensors are pictured as graphical "wiring diagrams," and suggests a possible syntax for lenses that shuffle product components (which we briefly discuss in Section 11).

**5.5 Definition [Projection lenses]:** In LENS, the projection is parametrized by an extra element to return when executing a $putl$ with a $missing$ complement.

$$\frac{y \in Y}{\pi_{1y} \in X \times Y \leftrightarrow X}$$

$$\pi_{1y} \;=\; (id_X \otimes term_y); \rho_X$$

The other projection is defined similarly.

The extra parameter to the projection prevents full naturality from holding (and therefore prevents this from being a categorical product), but the following "indexed" version of the naturality law does hold.

**5.6 Lemma [Naturality of projections]:** Suppose $k \in X_k \leftrightarrow Y_k$ and $\ell \in X_\ell \leftrightarrow Y_\ell$ and choose some initial value $y_i \in Y_\ell$. Define $(x_i, c_i) = \ell.putl(y_i, \ell.missing)$. Then $(k \otimes \ell); \pi_{1y_i} \equiv \pi_{1x_i}; k$.

The most serious problem, though, is that there is no diagonal. There are, of course, lenses with the *type* we need for $diag$—for example, $disconnect$! Or, more usefully, the lens that coalesces the copies of $X$ whenever possible, preferring the left one when it cannot coalesce (this is essentially the *merge* lens from [11])

$$diag \in X \to X \times X$$

$$\begin{array}{lcl} C & = & Unit + X \\ missing & = & \mathsf{inl}\ () \\ putr(x, \mathsf{inl}\ ()) & = & ((x,x), \mathsf{inl}\ ()) \\ putr(x, \mathsf{inr}\ x') & = & ((x,x'), eq(x,x')) \\ putl((x,x'),c) & = & (x, eq(x,x')) \end{array}$$

where here the $eq$ function tests its arguments for equality:

$$eq(x,x') = \begin{cases} \mathsf{inl}\ () & x = x' \\ \mathsf{inr}\ x' & x \neq x' \end{cases}$$

## 6. Sums and Lists

The status of sums has been even more mysterious than that of products. In particular, the *injection arrows* from $A$ to $A + B$ and $B$ to $A + B$ do not even make sense in the asymmetric setting; as functions, they are not surjective, so they cannot satisfy PUTGET.

A *categorical sum* of two objects $X$ and $Y$ is an object $X + Y$ and arrows $inl : X \to X + Y$ and $inr : Y \to X + Y$ such that for any two arrows $f : X \to Z$ and $g : Y \to Z$ there is a unique arrow $[f, g] : X + Y \to Z$—the *choice* of $f$ or $g$—satisfying $inl; [f, g] = f$ and $inr; [f, g] = g$. As with products, if a sum exists, it is unique up to isomorphism.

Since products and sums are dual, Corollary 4.10 and Theorem 5.1 imply that LENS does not have sums. Nevertheless, we do have a tensor whose object part is a set-theoretic sum—in fact, there are at least two different ones that are worth discussing—and we can define useful associated structures, including a choice operation on lenses. As with products, a tensor can be extended to a sum by providing injection and *co-diagonal* natural transformations satisfying a family of equations, but these constructions are even farther away from being categorical sums than what we saw with products.

The two tensors, which we called *retentive* and *forgetful* in Section 2, differ in how they handle the complement when faced with a situation where the new value being $put$ is from a different branch of the sum than the last one that was $put$. The retentive sum keeps complements for *both* sublenses in its own complement and switches between them as needed. The forgetful sum keeps only one complement, corresponding to whichever branch was last $put$. If the next $put$ switches sides, the complement is replaced with $missing$. We give just the retentive one here, since it is both more useful and a bit more complicated; the forgetful one can be found in the long version.

**6.1 Definition [Retentive tensor sum lens]:**

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

$$\begin{array}{lcl} C & = & k.C \times \ell.C \\ missing & = & (k.missing, \ell.missing) \\ putr(\mathsf{inl}\ x, (c_k, c_\ell)) & = & \mathsf{let}\ (z, c_k') = k.putr(x, c_k)\ \mathsf{in} \\ & & (\mathsf{inl}\ z, (c_k', c_\ell)) \\ putr(\mathsf{inr}\ y, (c_k, c_\ell)) & = & \mathsf{let}\ (w, c_\ell') = \ell.putr(y, c_\ell)\ \mathsf{in} \\ & & (\mathsf{inr}\ w, (c_k, c_\ell')) \\ putl(\mathsf{inl}\ z, (c_k, c_\ell)) & = & \mathsf{let}\ (x, c_k') = k.putl(z, c_k)\ \mathsf{in} \\ & & (\mathsf{inl}\ x, (c_k', c_\ell)) \\ putl(\mathsf{inr}\ w, (c_k, c_\ell)) & = & \mathsf{let}\ (y, c_\ell') = \ell.putl(y, c_\ell)\ \mathsf{in} \\ & & (\mathsf{inr}\ y, (c_k, c_\ell')) \end{array}$$

**6.2 Lemma [Sum bijection]:** For bijections $f$ and $g$,

$$bij_f \oplus bij_g \equiv bij_{f+g}$$

**6.3 Proposition [LENS,$\oplus$ is a symmetric monoidal category]:** In SET, the disjoint union gives rise to a symmetric monoidal category with $\emptyset$ as unit. Let $\alpha^+, \lambda^+, \rho^+, \gamma^+$ be associator, left-unitor, right-unitor, and symmetry natural isomorphisms. Then the $\oplus$ and $\oplus^f$ bifunctors each give rise to a symmetric monoidal category of lenses with $\emptyset$ as unit and $\alpha^\oplus = bij \circ \alpha^+$, $\lambda^\oplus = bij \circ \lambda^+$, $\rho^\oplus = bij \circ \rho^+$, and $\gamma^\oplus = bij \circ \gamma^+$ as associator, left-unitor, right-unitor, and symmetry, respectively.

The types of these natural isomorphisms are:

$$\begin{array}{rcl} \alpha^\oplus_{X,Y,Z} & \in & (X + Y) + Z \leftrightarrow X + (Y + Z) \\ \lambda^\oplus_X & \in & \emptyset + X \leftrightarrow X \\ \rho^\oplus_X & \in & X + \emptyset \leftrightarrow X \\ \gamma^\oplus_{X,Y} & \in & X + Y \leftrightarrow Y + X \end{array}$$

Unlike the product unit, there are no interesting lenses with the sum's unit, so this cannot be used to define the injection lenses. We have to do it by hand.

**6.4 Definition [Injection lenses]:**

$$\frac{x \in X}{inl_x \in X \leftrightarrow X + Y}$$

$$\begin{array}{lcl} C & = & X \times (Unit + Y) \\ missing & = & (x, \mathsf{inl}\ ()) \\ putr(x, (x', \mathsf{inl}\ ())) & = & (\mathsf{inl}\ x, (x, \mathsf{inl}\ ())) \\ putr(x, (x', \mathsf{inr}\ y)) & = & (\mathsf{inr}\ y, (x, \mathsf{inr}\ y)) \\ putl(\mathsf{inl}\ x, c) & = & (x, (x, \mathsf{inl}\ ())) \\ putl(\mathsf{inr}\ y, (x, c)) & = & (x, (x, \mathsf{inr}\ y)) \end{array}$$

We also define $inr_y = inl_y; \gamma^{\oplus}_{Y,X}$.

**6.5 Proposition:** The injection lenses are not natural.

As with products, where we have a useful lens of type $X \leftrightarrow X \times X$ that is nevertheless not a diagonal lens, we can craft a useful conditional lens of type $X + X \leftrightarrow X$ that is nevertheless not a codiagonal lens. In fact, we define a more general lens $union \in X + Y \leftrightarrow X \cup Y$. Occasionally, a value that is both an $X$ and a $Y$ may be put to the left across one of these union lenses. In this situation, the lens may legitimately choose either an $\mathsf{inr}$ tag or an $\mathsf{inl}$ tag. The $union$ lens uses the most recent unambiguous put to break the tie. (In the long version, we also define a variant that looks back to the last tagged value that was put to the right that was in both sets.)

**6.6 Definition [Union lens]:**

$$union_{XY} \in X + Y \leftrightarrow X \cup Y$$

$$\begin{array}{lcl} C & = & Bool \\ missing & = & \mathsf{false} \\ putr(\mathsf{inl}\ x, c) & = & (x, \mathsf{false}) \\ putr(\mathsf{inr}\ y, c) & = & (y, \mathsf{true}) \\ putl(xy, c) & & \\ \quad = \begin{cases} (\mathsf{inl}\ xy, \mathsf{false}) & xy \notin Y \vee (xy \in X \wedge \neg c) \\ (\mathsf{inr}\ xy, \mathsf{true}) & xy \notin X \vee (xy \in Y \wedge c) \end{cases} & \end{array}$$

This definition is not symmetric in $X$ and $Y$, because $putl$ prefers to return an $\mathsf{inl}$ value if there have been no tie breakers yet. Because of this preference, $union$ cannot be used to construct a true codiagonal. However, there are two useful related constructions:

**6.7 Definition [Switch lens]:**

$$switch_X \in X + X \leftrightarrow X$$

$$switch_X = union_{XX}$$

**6.8 Definition [Case lens]:**

$$\frac{f \in X \leftrightarrow Z \qquad g \in Y \leftrightarrow Z}{case_{f,g} \in X \oplus Y \leftrightarrow Z}$$

$$case_{f,g} = (f \oplus g); switch_X$$

***Lists*** We can also define a variety of lenses operating on lists. We'll just consider mapping here, because in the next section we're going to see how to obtain this and a whole variety of other functions on lists as instances of a powerful generic theorem, but it is useful to see one concrete instance first!

Write $X^\star$ for the set of lists with elements from the set $X$. Write $\langle\rangle$ for the empty list and $x{:}xs$ for the list with head $x$ and tail $xs$. Write $X^\omega$ for the set of infinite lists over $X$. When $x \in X$ and $ss \in X^\omega$, write $x{:}ss \in X^\omega$ for the infinite list with head $x$ and tail $ss$. Write $x^\omega \in X^\omega$ for the infinite list of $x$'s.

**6.9 Definition [Retentive list mapping lens]:**

$$\frac{\ell \in X \leftrightarrow Y}{map(\ell) \in X^\star \leftrightarrow Y^\star}$$

$$\begin{array}{lcl} C & = & (\ell.C)^\omega \\ missing & = & (\ell.missing)^\omega \\ putr(x, c) & = & \mathsf{let}\ \langle x_1, \ldots, x_m \rangle = x\ \mathsf{in} \\ & & \mathsf{let}\ \langle c_1, \ldots \rangle = c\ \mathsf{in} \\ & & \mathsf{let}\ (y_i, c'_i) = \ell.putr(x_i, c_i)\ \mathsf{in} \\ & & (\langle y_1, \ldots, y_m \rangle, \langle c'_1, \ldots, c'_m, c_{m+1}, \ldots \rangle) \\ putl & & \text{(similar)} \end{array}$$

As we saw in Section 2, there is also a forgetful variant of the list mapping lens. Indeed, this is the one that corresponds to the known list mapping operator on asymmetric lenses [7, 11].

## 7. Iterators

In functional programming, mapping functionals are usually seen as instances of more general "fold patterns," or defined by general recursion. In this section, we investigate to what extent this path can be followed in the world of symmetric lenses.

Allowing general recursive definitions for symmetric lenses may be possible, but in general, complements change when unfolding a recursive definition; this means that the structure of the complement of the recursively defined function would itself have to be given by some kind of fixpoint construction. Preliminary investigation suggests that this is possible, but it would considerably clutter the development—on top of the general inconvenience of having to deal with partiality.

Therefore, we choose a different path. We identify a "fold" combinator for lists, reminiscent of the view of lists as initial algebras. We show that several important lenses on lists—including mapping—can be defined with the help of a fold, and that, due to the self-duality of lenses, folds can be composed back-to-back to yield general recursive patterns in the style of *hylomorphisms* [24]. We also discuss iteration patterns on trees and argue that the methodology carries over to other polynomial inductive datatypes.

### 7.1 Lists

Let $f \in Unit + (X \times X^\star) \to X^\star$ be the bijection between "unfolded" lists and lists that takes $\mathsf{inl}\ ()$ to $\langle\rangle$ and $\mathsf{inr}\ (x, xs)$ to $x{:}xs$. Note that $bij_f \in Unit + (X \times X^\star) \Longleftrightarrow X^\star$ is then a bijective arrow in the category LENS.

**7.1.1 Definition:** An *$X$-list algebra* on a set $Z$ is an arrow $\ell \in Unit + (X \times Z) \Longleftrightarrow Z$ and a function $w \in Z \to \mathbb{N}$ such that $\ell.putl(z, c) = (inr(x, z'), c')$ implies $w(z') < w(z)$. We write $T^\star_X$ for the functor that sends any lens $k$ to $id_{Unit} \oplus (id_X \otimes k)$.

**7.1.2 Theorem:** For $X$-list algebra $\ell$ on $Z$, there is a unique arrow $It(\ell) \in X^\star \Longleftrightarrow Z$ such that the following diagram commutes:

$$\begin{array}{ccc} T^\star_X(X^\star) & \xrightarrow{bij_f} & X^\star \\ {\scriptstyle T^\star_X(It(\ell))}\downarrow & & \downarrow{\scriptstyle It(\ell)} \\ T^\star_X(Z) & \xrightarrow[\ell]{} & Z \end{array}$$

**Proof sketch:** We choose the complement of $It(\ell)$ to be $\ell.C^\omega$, so that the complements of the two arms of the commuting square are isomorphic. We then take commutativity of the diagram as a suggestion for a recursive definition of both $putr$ and $putl$. The length of the list in the case of $putr$ and the weight function in the case of $putl$ are used as ranking functions establishing totality of the recursive definitions. One must then prove, again by induction on these ranking functions, that the square does indeed commute. □

In the terminology of universal algebra, an algebra for a functor $F$ from some category to itself is simply an object $Z$ and an arrow $F(Z) \to Z$. A homomorphism between $F$-algebras $(Z, f)$ and $(Z', f')$ is a morphism $u \in Z \to Z'$ such that $f; u = F(u); f'$. The $F$-algebras thus form a category themselves. An initial $F$-algebra is an initial object in that category (an initial object has exactly one arrow to each other object, and is unique up to isomorphism). $F$-algebras can be used to model a wide variety of inductive datatypes, including lists and various kinds of trees [29]. Using this terminology, Theorem 7.1.2 says that $bij_f$ is an initial object in the subcategory consisting of those $T^\star_X$-algebras for which a weight function $w$ is available.

Let us consider some concrete instances of the theorem. First, if $k \in X \iff Y$ is a lens, then we can form an $X$-list algebra $\ell$ on $Y^\star$ by composing $id_{Unit} \oplus (k \otimes id_{Y^\star}) \in Unit + (X \times Y^\star) \iff Unit + (Y \times Y^\star)$ with $bij_f \in Unit + (Y \times Y^\star) \iff Y^\star$. A suitable weight function is given by $w(ys) = length(ys)$. The induced lens $It(\ell) \in X^\star \iff Y^\star$ is the lens analog of the familiar list mapping function. In fact, substituting the lens $e \in X \times Y \iff Y \times Z$ (from the introduction) for $k$ in the above diagram, we find that $It(\ell)$ is the sneakier variant of the lens $e^\star$.

Second, suppose that $X = X_1 + X_2$ and let $Z$ be $X_1^\star \times X_2^\star$. Writing $X_i^+$ for $X_i \times X_i^\star$, we can define isomorphisms

$$\begin{array}{rcl} f & \in & (X_1 + X_2) \times X_1^\star \times X_2^\star \\ & \to & (X_1^+ + X_2^+) + (X_1^+ \times X_2^+ + X_1^+ \times X_2^+) \\ g & \in & Unit + ((X_1^+ + X_2^+) + X_1^+ \times X_2^+) \\ & \to & X_1^\star \times X_2^\star \end{array}$$

by distributing the sum and unfolding the list type for $f$ and by factoring the polynomial and folding the list type for $g$. Then we can create

$$\ell \in Unit + ((X_1 + X_2) \times Z) \leftrightarrow Z$$

$$\begin{array}{rcl} \ell & = & (id_{Unit} \oplus bij_f); \\ & & (id_{Unit} \oplus (id_{X_1^+ + X_2^+} \oplus switch_{X_1^+ \times X_2^+})); \\ & & bij_g \end{array}$$

A suitable weight function for $\ell$ is given by $w((xs_1, xs_2)) = length(xs_1) + length(xs_2)$. The lens $It(\ell) \in (X_1 + X_2)^\star \iff X_1^\star \times X_2^\star$ that we obtain from iteration partitions the input list in one direction and uses a stream of booleans from the state to put them back in the right order in the other direction. Composing it with a projection yields a filter lens. (Alternatively, the filter lens could be obtained directly by iterating a slightly trickier $\ell$.)

**7.1.3 Corollary:** Suppose $k^{op}$ is an $X$-list algebra on $W$ and $\ell$ is an $X$-list algebra on $Z$. Then there is a lens $Hy(k, \ell) \in W \iff Z$ such that the following diagram commutes:

$$\begin{array}{ccc} T^\star_X(W) & \xleftarrow{\ k\ } & W \\ {\scriptstyle T^\star_X(Hy(k,\ell))}\downarrow & & \downarrow{\scriptstyle Hy(k,\ell)} \\ T^\star_X(Z) & \xrightarrow[\ell]{} & Z \end{array}$$

**Proof:** Define $Hy(k, \ell)$ as the composition $It(k^{op})^{op}; It(\ell)$. □

One can think of $Hy(k, \ell)$ as a recursive definition of a lens. The lens $k$ tells whether a recursive call should be made, and if so, produces the argument for the recursive call and some auxiliary data. The lens $\ell$ then describes how the result is to be built from the result of the recursive call and the auxiliary data. This gives us a lens version of the hylomorphism pattern from functional programming [24]. Unfortunately, we were unable to prove or disprove the uniqueness of $Hy(k, \ell)$.

We have not formally studied the question of whether $It(\ell)$ is actually an initial algebra, i.e., whether it can be defined and is unique even in the absence of a weight function. However, this seems unlikely, because then it would apply to the case where $Z$ is the set of finite and infinite $X$ lists and $\ell$ the obvious bijective lens. The $putl$ component of $It(\ell)$ would then have to truncate an infinite list, which would presumably break the commuting square.

### 7.2 Other Datatypes

Analogs of Theorem 7.1.2 and Corollary 7.1.3 are available for a number of other functors, in particular those that are built up from variables by $+$ and $\times$. All of these can also be construed as containers (see Section 8), but the iterator and hylomorphism patterns provide more powerful operations for the construction of lenses than the mapping operation available for general containers. Moreover, the universal property of the iterator provides a modular proof method, allowing one to deduce equational laws which can be cumbersome to establish directly because of the definition of equality as behavioral equivalence. For instance, we can immediately deduce that list mapping is a functor. Containers, on the other hand, subsume datatypes such as labeled graphs that are not initial algebras.

***Parametrized lists*** The list iterator allows us to define a lens between $X^\star$ and some other set $Z$. In order to define a lens between $X^\star \times Y$ and $Z$ (think of $Y$ as modeling parameters) we cannot use Theorem 7.1.2 directly. In standard functional programming, a map from $X^\star \times Y$ to $Z$ is tantamount to a map from $X^\star$ to $Y \to Z$, so iteration with parameters is subsumed by the parameterless case. Unfortunately, LENS does not seem to have the function spaces required to play this trick.

Therefore, we introduce the functor $T^\star_{X,Y}(Z) = Y + X \times Z$ and notice that $T^\star_{X,Y}(X^\star \times Y) \simeq X^\star \times Y$. Just as before, an algebra for that functor is a lens $\ell \in T^\star_{X,Y}(Z) \leftrightarrow Z$ together with a function $w : Z \to \mathbb{N}$ such that $\ell.putl(z, c) = (\mathsf{inr}\ (x, z'), c')$ implies $w(z') < w(z)$.

As an example, let $Y = Z = X^\star$ and define

$$\ell \in X^\star + X \times X^\star \leftrightarrow X^\star$$

$$\begin{array}{lcl} C & = & Bool \\ missing & = & \mathsf{true} \\ \ell.putr(\mathsf{inl}\ xs, b) & = & (xs, \mathsf{true}) \\ \ell.putr(\mathsf{inr}\ (x, xs), b) & = & (x{:}xs, \mathsf{false}) \\ \ell.putl(\langle\rangle, b) & = & (\mathsf{inl}\ \langle\rangle, \mathsf{true}) \\ \ell.putl(x{:}xs, \mathsf{true}) & = & (\mathsf{inl}\ (x{:}xs), \mathsf{true}) \\ \ell.putl(x{:}xs, \mathsf{false}) & = & (\mathsf{inr}\ (x, xs), \mathsf{false}) \end{array}$$

Iteration yields a lens $X^\star \times X^\star \leftrightarrow X^\star$ that can be seen as a bidirectional version of list concatenation. The commuting square for the iterator corresponds to the familiar recursive definition of concatenation: $concat(\langle\rangle, ys) = ys$ and $concat(x{:}xs, ys) = x{:}concat(xs, ys)$. In the bidirectional case considered here the complement will automatically retain enough information as to allow splitting in the $putl$-direction.

We can use a version of Corollary 7.1.3 for parametrized lists in order to justify tail recursive constructions. Consider, for instance, the opposite of a $T^{\star}_{Unit,X^{\star}}$-algebra $k : X^{\star} \times X^{\star} \leftrightarrow X^{\star} + X^{\star} \times X^{\star}$ where

$$k.putr((acc, \langle\rangle), \mathsf{true}) = (\mathsf{inl}\ acc, \mathsf{true})$$
$$k.putr((acc, x{:}xs), \mathsf{true}) = (\mathsf{inr}\ (x{:}acc, xs), \mathsf{true})$$
$$k.putr((acc, xs), \mathsf{false}) = (\mathsf{inr}\ (acc, xs), \mathsf{false}).$$

Together with the $T^{\star}_{Unit,X^{\star}}$-algebra $switch_{X^{\star}} : X^{\star} + X^{\star} \leftrightarrow X^{\star}$ this furnishes a bidirectional version of the familiar tail recursive list reversal that sends $(acc, xs)$ to $xs^{rev}\, acc$.

***Trees*** For set $X$ let $Tree(X)$ be the set of binary $X$-labeled trees given inductively by $leaf \in Tree(X)$ and $x \in X, \ell \in Tree(X), r \in Tree(X) \Rightarrow node(x, \ell, r) \in Tree(X)$. Consider the endofunctor $T_X^{Tree}(Z) = Unit + X \times Z \times Z$. Let $c \in T_X^{Tree}(Tree(X)) \leftrightarrow Tree(X)$ denote the obvious bijective lens.

An $X$-tree algebra is a lens $\ell \in T_X^{Tree}(Z) \leftrightarrow Z$ and a function $w \in Z \to \mathbb{N}$ with the property that if $\ell.putl(z, c) = (\mathsf{inr}\ (x, z_l, z_r), c')$ then $w(z_l) < w(z)$ and $w(z_r) < w(z)$. The bijective lens $c$ is then the initial object in the category of $X$-tree algebras; that is, every $X$-tree algebra on $Z$ defines a unique lens in $Tree(X) \leftrightarrow Z$.

Consider, for example, the concatenation lens $concat : X^{\star} \times X^{\star} \leftrightarrow X^{\star}$. Let $concat' : Unit + X \times X^{\star} \times X^{\star} \leftrightarrow X^{\star}$ be the lens obtained from $concat$ by precomposing with the fold-isomorphism and the terminal lens $term_{\langle\rangle}$. Intuitively, this lens sends $\mathsf{inl}\ ()$ to $\langle\rangle$ and $x, xs, xs'$ to $x{:}xs@xs'$, using the complement to undo this operation properly. This lens forms an example of a tree algebra (with number of nodes as weight functions) and thus iteration furnishes a lens $Tree(X) \leftrightarrow X^{\star}$ which does a pre-order traversal, keeping enough information in the complement to rebuild a tree from a modified traversal.

The hylomorphism pattern can also be applied to trees, yielding the ability to define symmetric lenses by divide-and-conquer, i.e., by dispatching one call to two parallel recursive calls whose results are then appropriately merged.

## 8. Containers

The previous section suggests a construction for a variety of operations on datatypes built from polynomial functors. Narrowing the focus to the very common "map" operation, we can generalize still further, to any kind of *container functor* [1], i.e. a *normal functor* in the terminology of Hasegawa [14] or an *analytic functor* in the terminology of Joyal [19]. (These structures are also related to the *shapely types* of Jay and Cockett [18].)

**8.1 Definition [Container]:** A *container* consists of a set $I$ together with an $I$-indexed family of sets $B \in I \to Set$.

Each container $(I, B)$ gives rise to an endofunctor $F_{I,B}$ on SET whose object part is defined by $F_{I,B}(X) = \sum_{i \in I} B(i) \to X$. For example, if $I = \mathbb{N}$ and $B(n) = \{0, 1, \ldots, n{-}1\}$, then $F_{I,B}(X)$ is $X^{\star}$ (up to isomorphism). Or, if $I = Tree(Unit)$ is the set of binary trees with trivial labels and $B(i)$ is the set of nodes of $i$, then $F_{I,B}(X)$ is the set of binary trees labeled with elements of $X$. In general, we can think of $I$ as a set of shapes and, for each shape $i \in I$, we can think of $B(i)$ as the set of "positions" in shape $i$. So an element $(i, f) \in F_{I,B}(X)$ consists of a shape $i$ and a function $f$ assigning an element $f(p) \in X$ to each position $p \in B(i)$. The morphism part of $F_{I,B}$ maps a function $u \in X \to Y$ to a function $F_{I,B}(u) \in F_{I,B}(X) \to F_{I,B}(Y)$ given by $(i, f) \mapsto (i, f; u)$.

Now, we would like to find a way to view a container as a functor on the category of lenses. In order to do this, we need a little extra structure.

**8.2 Definition:** A *container with ordered shapes* is a pair $(I, B)$ satisfying these conditions:

1. $I$ is a partial order with binary meets. We say $i$ is a *subshape* of $j$ whenever $i \le j$.
2. $B$ is a functor from $(I, \le)$ viewed as a category (with one object for each element and an arrow from $i$ to $j$ iff $i \le j$) into SET. When $B$ and $i$ are understood, we simply write $b|i'$ for $B(i \le i')(b)$ if $b \in B(i)$ and $i \le i'$.
3. If $i$ and $i'$ are both subshapes of a common shape $j$ and we have positions $b \in B(i)$ and $b' \in B(i')$ with $b|j = b'|j$, then there must be a unique $b_0 \in B(i \wedge i')$ such that $b = b_0|i$ and $b' = b_0|i'$. Thus such $b$ and $b'$ are really the same position. In other words, every diagram of the following form is a pullback:

$$\begin{array}{ccc} B(i \wedge i') & \xrightarrow{B(i \wedge i' \le i)} & B(i) \\ {\scriptstyle B(i \wedge i' \le i')}\downarrow & & \downarrow{\scriptstyle B(i \le j)} \\ B(i') & \xrightarrow[B(i' \le j)]{} & B(j) \end{array}$$

If $i \le j$, we can apply the instance of the pullback diagram where $i = i'$ and hence $i \wedge i' = i$ and deduce that $B(i \le j) \in B(i) \to B(j)$ is always injective.

For example, in the case of trees, we can define $t \le t'$ if every path from the root in $t$ is also a path from the root in $t'$. The morphism part of $B$ then embeds positions of a smaller tree canonically into positions of a bigger tree. The meet of two trees is the greatest common subtree starting from the root.

**8.3 Definition [Container mapping lens]:**

$$\frac{\ell \in X \leftrightarrow Y}{F_{I,B}(\ell) \in F_{I,B}(X) \leftrightarrow F_{I,B}(Y)}$$

$$\begin{array}{l} C = \\ \quad \{t \in \prod_{i \in I} B(i) \to \ell.(C) \mid \\ \quad\ \forall i, i'.\ i \le i' \supset \forall b{\in}B(i).\ t(i')(b|i') = t(i)(b)\} \\ missing(i)(b) \ = \ \ell.missing \\ putr((i, f), t) \ = \\ \quad \mathsf{let}\ f'(b) \quad = \ \mathsf{fst}(\ell.putr(f(b), t(i)(b)))\ \mathsf{in} \\ \quad \mathsf{let}\ t'(j)(b) \ = \\ \qquad \mathsf{if}\ \exists b_0 \in B(i \wedge j).\ b_0|j = b \\ \qquad \mathsf{then}\ \mathsf{snd}(\ell.putr(f(b_0|i), t(j)(b))) \quad \mathsf{in} \\ \qquad \mathsf{else}\ t(j)(b) \\ \quad ((i, f'), t') \\ putl \qquad\qquad \text{(similar)} \end{array}$$

(Experts will note that $C$ is the limit of the contravariant functor $i \mapsto (B(i) \to \ell.(C))$. Alternatively, we can construe $C$ as the function space $D \to \ell.(C)$ where $D$ is the colimit of the functor $B$. Concretely, $D$ is given by $\sum_{i \in I} B(i)$ modulo the equivalence relation $\sim$ generated by $(i, b) \sim (i', b')$ whenever $i \le i'$ and $b' = B(i \le i')(b)$.)

For the case of lists, this mapping lens coincides with the retentive map that we obtained from the iterator in Section 7. We believe it should also be possible to define a forgetful version where the complement is just $F_{I,B}(\ell.C)$.

## 9. Asymmetric Lenses as Symmetric Lenses

The final step in our investigation is to formalize the connection between symmetric lenses and the more familiar space of asym-

metric lenses and to show how known constructions in this space correspond to the constructions we have considered.

Write $X \overset{a}{\leftrightarrow} Y$ for the set of asymmetric lenses from $X$ to $Y$ (using the first presentation of asymmetric lenses from Section 2, with $get$, $put$, and $create$ components).

**9.1 Definition:** Every asymmetric lens can be embedded in a symmetric one.

$$\frac{\ell \in X \overset{a}{\leftrightarrow} Y}{\ell^{sym} \in X \leftrightarrow Y}$$

$$\begin{array}{lcl} C & = & \{f \in Y \to X \mid \forall y \in Y.\ \ell.get(f(y)) = y\} \\ missing & = & \ell.create \\ putr(x, f) & = & (\ell.get(x), f_x) \\ putl(y, f) & = & \mathsf{let}\ x = f(y)\ \mathsf{in}\ (x, f_x) \end{array}$$

(Here, $f_x(y)$ means $\ell.put(y, x)$.) Viewing $X$ as the source of an asymmetric lens (and therefore as having "more information" than $Y$), we can understand the definition of the complement here as being a value from $X$ stored as a closure over that value. The presentation is complicated slightly by the need to accommodate the situation where a complete $X$ does not yet exist—i.e. when defining $missing$—in which case we can use $create$ to fabricate an $X$ value out of a $Y$ value if necessary.

**9.2 Definition [Asymmetric lenses]:** Here are several useful asymmetric lenses, based on string lenses from [7]. (We give only their names and types here; full definitions appear in the long version.) We use $\ell_{X,Y}$ to stand for any asymmetric lens $\ell \in X \overset{a}{\leftrightarrow} Y$.

$$\begin{array}{rcl} copy_X & \in & X \overset{a}{\leftrightarrow} X \\ \ell_{X,Y}; \ell_{Y,Z} & \in & X \overset{a}{\leftrightarrow} Z \\ aconst_x & \in & X \overset{a}{\leftrightarrow} Unit \text{ whenever } x \in X \\ \ell_{X,Y} \cdot \ell_{Z,W} & \in & X \times Z \overset{a}{\leftrightarrow} Y \times W \\ \ell_{X,Y} | \ell_{Z,W} & \in & X + Z \overset{a}{\leftrightarrow} Y \cup W \\ \ell^{\star}_{X,Y} & \in & X^{\star} \overset{a}{\leftrightarrow} Y^{\star} \end{array}$$

**9.3 Theorem:** The symmetric embeddings of these lenses correspond nicely to definitions from earlier in this paper:

$$\begin{array}{rcll} copy_X^{sym} & \equiv & id_X & (7) \\ (k; \ell)^{sym} & \equiv & k^{sym}; \ell^{sym} & (8) \\ aconst_x^{sym} & \equiv & term_x & (9) \\ (k \cdot \ell)^{sym} & \equiv & k^{sym} \otimes \ell^{sym} & (10) \\ (k|\ell)^{sym} & \equiv & (k^{sym} \oplus^f \ell^{sym}); union & (11) \\ (\ell^{\star})^{sym} & \equiv & map^f(\ell^{sym}) & (12) \end{array}$$

The first two show that $(-)^{sym}$ is a functor. The $\oplus^f$ operator is the forgetful variant of $\oplus$; see the long version for the full definition.

This functor is not *full*—that is, there are some symmetric lenses which are not the image of any asymmetric lens. Injection lenses, for example, have no analog in the category of asymmetric lenses, and the composer lens illustrated in Figure 1 cannot be implemented as an asymmetric lens. However, we *can* characterize symmetric lenses in terms of asymmetric ones in a slightly more elaborate way.

**9.4 Theorem:** Given any arrow $\ell$ of LENS, there are asymmetric lenses $k_1, k_2$ such that $(k_1^{sym})^{op}; k_2^{sym} = \ell$.

To see this, we need to know how to "asymmetrize" a symmetric lens. We can view a symmetric lens as a pair of asymmetric lenses joined "tail to tail" whose common domain is consistent triples. For any lens $\ell \in X \leftrightarrow Y$, define $S_\ell = \{(x, y, c) \in X \times Y \times \ell.C \mid \ell.putr(x, c) = (y, c)\}$. Now define:

$$\frac{\ell \in X \leftrightarrow Y}{\ell_r^{asym} \in S_\ell \overset{a}{\leftrightarrow} X}$$

$$\begin{array}{lcl} get((x, y, c)) & = & x \\ put(x', (x, y, c)) & = & \mathsf{let}\ (y', c') = \ell.putr(x', c) \\ & & \mathsf{in}\ (x', y', c') \\ create(x) & = & \mathsf{let}\ (y, c) = \ell.putr(x, \ell.missing) \\ & & \mathsf{in}\ (x, y, c) \end{array}$$

The definition of $\ell_l^{asym} \in S_\ell \overset{a}{\leftrightarrow} Y$ is similar.

**Proof of 9.4:** Given $[\ell]$, choose $k_1 = \ell_r^{asym}$ and $k_2 = \ell_l^{asym}$. □

## 10. Related Work

There is now a large literature on lenses and related approaches to propagating updates between connected structures. We discuss only the most closely related work here; good general surveys of the area can be found in [8, 13]. Connections to the literature on *view update* in databases are surveyed in [11].

The first symmetric approach to update propagation was proposed by Meertens [23] and followed up by Stevens [26] and Diskin [10]. Meertens suggests modeling synchronization between two sets $X$ and $Y$ by a *consistency relation* $R \subseteq X \times Y$ and two *consistency maintainers* $\triangleleft : X \times Y \to X$ and $\triangleright : X \times Y \to Y$ such that $(x \triangleleft y)\ R\ y$ and $x\ R\ (x \triangleright y)$ always hold, and such that $x\ R\ y$ implies $x \triangleleft y = x$ and $x \triangleright y = y$.

The main advantage of symmetric lenses over consistency maintainers is their closure under composition. Indeed, all of the aforementioned authors note that, in general, consistency maintainers do not compose and view this as a drawback. Suppose that we have relations $R \subseteq X \times Y$ and $R' \subseteq Y \times Z$ maintained by $\triangleright, \triangleleft$ and $\triangleright', \triangleleft'$, resp. If we want to construct a maintainer for the composition $R; R'$, we face the problem that, given $x \in X$ and $z \in Z$, there is no canonical way of coming up with a $y \in Y$ that will allow us to use either of the existing maintainer functions. Concretely, Meertens gives the following counterexample. Let $X$ be the set of nonempty context free grammars over some alphabet, and let $Y$ be the set of words over that same alphabet. Let $R \subseteq X \times Y$ be given by $G\ R\ x \iff x \in L(G)$. It is easy to define computable maintainer functions making this relation a constraint maintainer. Composing this relation with its opposite yields an undecidable relation (namely, whether the intersection of two context-free grammars is nonempty), so there cannot be computable maintainer functions.

We can transform any constraint maintainer into a symmetric lens as follows: take the relation $R$ itself (viewed as a set of pairs) as the complement, and define $putl(x', (x, y)) = (x' \triangleright y, (x', x' \triangleright y))$ and similarly for $putr$. If we compose such a symmetric lens with its opposite we obtain $R \times R^{op}$ as the complement and, for example, $putr(x', ((x_1, y_1), (y_2, x_2))) = (x_2 \triangleleft (x' \triangleright y_1), ((x', x' \triangleright y_1), (x' \triangleright y_1, x_2 \triangleleft (x' \triangleright y_1))))$. For Meertens' counterexample, we would have complements of the form $((G_1, w_1), (w_2, G_2))$, with $w_1 \in L(G_1)$ and $w_2 \in L(G_2)$; "$putr$"-ing a new grammar $G_1'$ through the composed lens yields the complement $((G_1', w_1'), (w_1', G_2'))$, where $w_1'$ is $w_1$ if $w_1 \in L(G_1)$ and some default otherwise, and where $G_2' = G_2$ if $w_1' \in L(G_2)$ and $S \to w_1'$ (where $S$ is the start state) otherwise. Meertens recommends using a *chain* of consistency maintainers in such a situation to achieve a similar effect; however, the properties of such chains have not been explored.

For asymmetric lenses, a number of alternative laws have been explored. Some of these are weaker than ours; for example, a number of papers from a community of researchers based in Tokyo replace the PUTGET law with a somewhat looser PUTGETPUT law, permitting a broader range of useful behaviors for lenses that duplicate information. It would be interesting to see what kind of categorical structures arise from these choices. The proposal by Matsuda et al.[22] is particularly interesting because it also employs the idea of complements. Conversely, stronger laws can be imagined, such as the PUTPUT law discussed by Foster et al. [11].

A different foundation for defining lenses by recursion was explored by Foster et al. [11], using standard tools from domain theory to define monotonicity and continuity for lens combinators parametrized on other lenses. The main drawback of this approach is that the required (manual) proofs that such recursive lenses are total tend to be somewhat intricate. By contrast, we expect that our initial-algebra approach can be equipped with automatic proofs of totality (that is, choices of the weight function $w$) in many cases of interest.

## 11. Conclusions and Future Work

We have proposed the first notion of symmetric bidirectional transformations that supports composition. Composability opens up the study of symmetric bidirectional transformations from a category-theoretic perspective. We have explored the category of symmetric lenses, which is self-dual and has the category of bijections and that of asymmetric lenses each as full subcategories. We have surveyed the structure of this category and found it to admit tensor product structures that are the Cartesian product and disjoint union on objects. We have also investigated datatypes both inductively and as "containers" and found the category of symmetric lenses to support powerful mapping and folding constructs.

***Syntax*** Although we have focused here on semantic and algebraic foundations, many of our constructions have a straightforward syntactic realization. In particular, it is easy to give a string-transformation interpretation to all the constructions in Sections 4 to 6 (including lenses over lists); these could easily be used to build a symmetric version of Boomerang [7]. More interesting would be to eliminate Boomerang's built-in lists and instead obtain lenses over lists and other structures (mapping, reversing, flattening of lists, transforming trees into lists) solely by using the combinators derived from the category-theoretic structure we have exhibited. To accomplish this, two further fine points need to be considered. First, we would want an automatic way for discovering weight functions for iterators. We believe that a straightforward termination analysis based on unfolding (similar to the one built into Coq) could help, but the details remain to be checked. And second, we must invent a formal syntax for programming with containers. Surprisingly, the existing literature does not seem to contain such a proposal.

More speculatively, it is a well-known folklore result that symmetric monoidal categories are in 1-1 correspondence with wiring diagrams and with first-order linear lambda calculus. We would like to exploit this correspondence to design a lambda-calculus-like syntax for symmetric lenses and perhaps also a diagrammatic language. The linear lambda calculus has judgments of the form $x_1{:}A_1, \ldots, x_n{:}A_n \vdash t : A_0$, where $A_0, \ldots, A_n$ are sets or possibly syntactic type expressions and where $t$ is a linear term made up from basic lenses, lens combinators, and the variables $x_1, \ldots, x_n$. This could be taken as denoting a symmetric lens $A_1 \otimes \cdots \otimes A_n \leftrightarrow A_0$. For example, here is such a term for the lens $concat'$ from Section 7.2:

$$
\begin{aligned}
z{:}\mathit{Unit} \oplus A \otimes A^\star \otimes A^\star \vdash\ & \mathit{match}\ z\ \mathit{with} \\
& \mid inl() \mapsto term^{op}_{\langle\rangle} \\
& \mid inr(a, al, ar) \mapsto concat(a{:}al, ar)
\end{aligned}
$$

The interpretation of such a term in the category of lenses then takes care of the appropriate insertion of bijective lenses for regrouping and swapping tensor products.

***Complements as States*** One benefit of treating complements explicitly is that it opens the way to a *stateful* presentation of lenses. The idea is that the complement of a lens can be thought of as its local storage—the part of the heap that belongs to it. An obvious next step is that, instead of the lens components taking the local storage as an argument and returning an updated version as a result, they can just hang onto it themselves, internally, in mutable variables. The types of the $put$ operations then become just $\ell.putr \in A \to B$ and $\ell.putl \in B \to A$, where the $\to$ is now a "programming language function type," with the usual implicit treatment of the heap. This avoids destructing the given $C$ each time we propagate an update and rebuilding a new $C$ to yield as a result, improving the efficiency of the implementation. An additional improvement comes from the next potential extension.

***Alignment and Delta Lenses*** As we mentioned in Section 2, dealing correctly with alignment of structured information is crucial in practice. This issue has been extensively explored in the context of asymmetric lenses, and it seems it should be possible to adapt existing ideas such as *dictionary lenses* [7] and *matching lenses* [5] to symmetric lenses. An even better approach might be to change the fundamental nature of lenses so that, instead of working directly with entire *structures*, they work with *deltas*—descriptions of changes to the structures. These deltas can arise from simple positional judgments, as in this paper, from diff-like heuristics, from cues within the data itself, or perhaps even from user interaction—the lens itself doesn't need to know anything about this.

Many of our basic constructions can be adapted to deltas by taking the domain and codomain of a lens to be monoids (of edit operations) instead of sets, and then, for each lens construction, defining an appropriate edit monoid from the monoids of its components. For example, an edit for a pair lens is a pair of edits for the left- and right-hand sides of the pair. However, more thought is required to make this scheme really work: applying this idea naively leads to insufficiently expressive edit languages for structures like lists. In particular, we would like to see insertion and deletion as edit operations on lists (and rotations and the like for trees, etc.). Currently, we believe that containers are a promising framework for this endeavour.

***Acknowledgments*** We are grateful to Nate Foster for productive discussions of many points, especially about the category of lenses, and to both Nate and Alexandre Pilkiewicz for helpful comments on drafts of the paper. Our work is supported by the National Science Foundation under grant IIS-0534592 *Linguistic Foundations for XML View Update*.

## References

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

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

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

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

[5] Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: Alignment and view up-

date. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, September 2010.

[6] Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

[7] 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.

[8] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In Richard F. Paige, editor, *ICMT*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer, 2009. ISBN 978-3-642-02407-8.

[9] Z. Diskin, K. Czarnecki, and M. Antkiewicz. Model-versioning-in-the-large: algebraic foundations and the tile notation. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 7–12. IEEE Computer Society, 2009.

[10] Zinovy Diskin. Algebraic models for bidirectional model synchronization. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2008. ISBN 978-3-540-87874-2.

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

[12] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, Canada*, September 2008.

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

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

[15] S. Hidaka, Z. Hu, H. Kato, and K. Nakano. A compositional approach to bidirectional model transformation. In *New Ideas and Emerging Results Track of 31st International Conference on Software Engineering (ICSE 2009, NIER Track)*, 2009.

[16] 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.

[17] 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 to appear in *Higher Order and Symbolic Computation*, 2008.

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

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

[20] Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. Bidirectional interpretation of xquery. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM), Nice, France*, pages 21–30, New York, NY, USA, 2007.

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

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

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

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

[25] 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.

[26] 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.

[27] Perdita Stevens. A landscape of bidirectional model transformations. *Postproceedings of GTTSE*, 7, 2008.

[28] 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.

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

[30] 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.