Symmetric Lenses

(Full version)

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, 15] for recent surveys), and these ideas are beginning to see commercial application, e.g., in RedHat's system administration tool, Augeas [24].

One particularly well-studied class of bidirectional programming languages is the framework of *lenses* introduced in [12]. 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, 13, 14], trees [12, 20, 23, 28], relations [6], graphs [19], or software models [9–11, 17, 18, 29, 30, 33]. 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 [11, 26, 31], 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.

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 [12]. 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 $L(X) \simeq Unit + X \times L(X)$ we obtain a lens connecting the set $L(X)$ with the set $Unit + X \times L(X)$; 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:

$$get \ \in \ X \to Y$$
$$put \ \in \ Y \times X \to X$$

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, [25]), 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

$$get \ \in \ X \to Y \times C$$
$$put \ \in \ Y \times C \to X$$

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.

$$get \ \in \ X \times C_Y \to Y \times C_X$$
$$put \ \in \ Y \times C_X \to X \times C_Y$$

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. (This seeming tweak is actually critical: it is what allows us to define composition of symmetric lenses.)

$$get \ \in \ X \times C \to Y \times C$$
$$put \ \in \ Y \times C \to X \times C$$

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

$$putr \ \in \ X \times C \to Y \times C$$
$$putl \ \in \ Y \times C \to X \times C$$

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 ℓ, etc.

---

[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 $put'(y, f) = f(y)$. Going the other way, if $(get, put, missing)$ are the components of an asymmetric lens with complement, we can define a traditional lens by $get'(x) = \mathsf{fst}(get(x))$ and $put'(y, x) = put(y, \mathsf{snd}(get(x)))$ and $create(y) = put(y, missing)$.
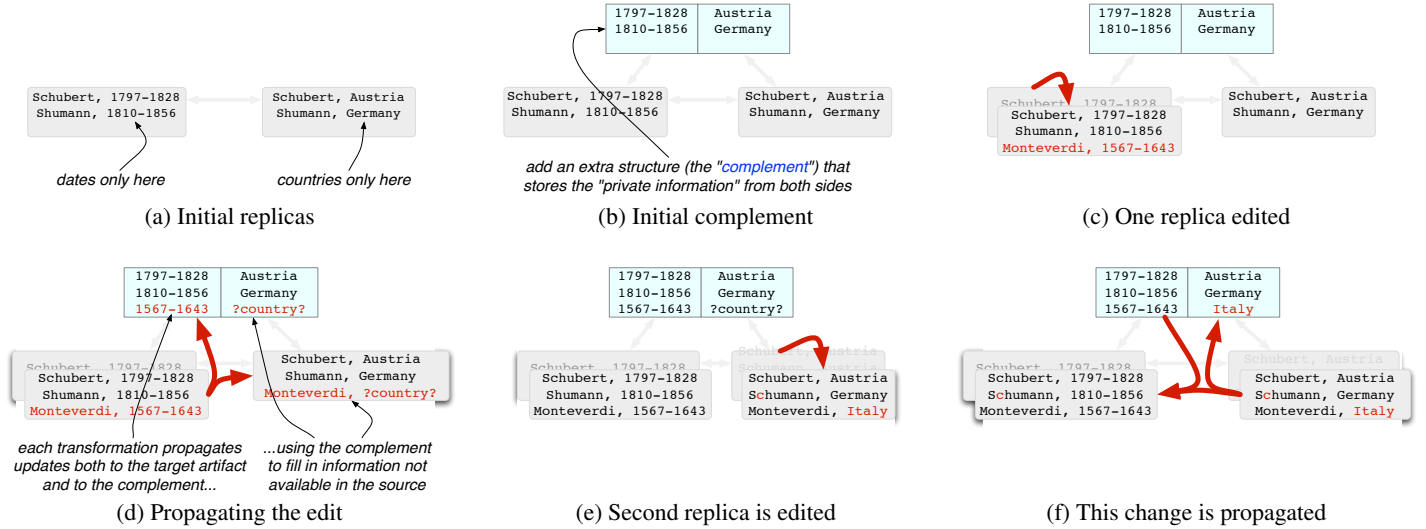
Figure 1. Behavior of a symmetric lens

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.

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 *put*s—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:

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

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

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.

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

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

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.

Put-Put Laws

2.2 Lemma: The following "put the same thing twice" laws follow from the ones we have:

$$
\frac{putr(x, c) = (y, c')}{putr(x, c') = (y, c')} \tag{PUTR2}
$$

$$
\frac{putl(y, c) = (x, c')}{putl(y, c') = (x, c')} \tag{PUTL2}
$$

We could consider generalizing these to say that putting an arbitrary pair of values, one after the other, is the same as doing just the second *put* into the first complement:

$$
\frac{putr(x, c) = (\_, c')}{putr(x', c') = putr(x', c)} \tag{STRONG-PUTPUTR*}
$$

$$\frac{putl(y,c) = (\_, c')}{putl(y', c') = putl(y', c)} \qquad \text{(STRONG-PUTPUTL}^*\text{)}$$

But these laws are very strong—probably too strong to be useful (the * annotations in their names are a reminder that we do *not* adopt them). The reason is that they demand that the effect of every update is completely undoable—not only the effect on the other replica, but also the effect of the first update on the complement must be completely forgotten if we make a second update. In particular, neither of the list-mapping lenses in Section 6 satisfy these laws.

A weaker version of these laws, constraining the output but not the effect on the complement, may be more interesting:

$$\frac{\begin{array}{c} putr(x,c) = (\_, c') \\ putr(x', c) = (y, \_) \\ putr(x', c') = (y', \_) \end{array}}{y = y'} \qquad \text{(WEAK-PUTPUTR}^*\text{)}$$

$$\frac{\begin{array}{c} putl(y,c) = (\_, c') \\ putl(y', c) = (x, \_) \\ putl(y', c') = (x', \_) \end{array}}{x = x'} \qquad \text{(WEAK-PUTPUTL}^*\text{)}$$

We do not choose to adopt these laws here because they are not satisfied by the "forgetful" variants of our summing and list mapping lenses. However, the forgetful variants are only mainly interesting because of their close connection to analogous asymmetric lenses; in practice, the "retentive" variants seem more useful, and these do satisfy the weak PUTPUT laws.

Alignment   One important *non*-goal of the present paper is dealing with the (important) issue of *alignment* [5, 7]. We consider 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 sets 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:

$$\frac{\begin{array}{c} (c_f, c_g) \in R \\ f(x, c_f) = (y_f, c_f') \\ g(x, c_g) = (y_g, c_g') \end{array}}{y_f = y_g \wedge (c_f', c_g') \in R}$$

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

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

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

The following is straightforward.

3.3 Lemma: Lens equivalence is an equivalence relation.

3.4 Definition: Given a lens $\ell \in X \leftrightarrow Y$, define a *put object* for ℓ to be a member of $X + Y$. Define a function *apply* taking a lens, an element of that lens' complement set, and a list of put objects as follows (using ML-like syntax):

$$\begin{aligned} apply(\ell, c, \mathsf{inl}\ x{:}puts) &= \mathsf{let}\ (y, c') = \ell.putr(x, c)\ \mathsf{in} \\ &\quad \mathsf{inr}\ y{:}apply(\ell, c', puts) \\ apply(\ell, c, \mathsf{inr}\ y{:}puts) &= \mathsf{let}\ (x, c') = \ell.putl(y, c)\ \mathsf{in} \\ &\quad \mathsf{inl}\ x{:}apply(\ell, c', puts) \\ apply(\ell, c, \langle\rangle) &= \langle\rangle \end{aligned}$$

3.5 Definition [Observational equivalence]: Lenses $k, \ell \in X \leftrightarrow Y$ are *observationally equivalent* (written $k \approx \ell$) if, for every sequence of put objects $P \in (X + Y)^\star$ we have

$$apply(k, k.missing, P) = apply(\ell, \ell.missing, P).$$

3.6 Theorem: $k \approx \ell$ iff $k \equiv \ell$.

Proof: ($\Longleftarrow$) Suppose that $k \equiv \ell$ via relation R. For all sequences of put objects P, and for elements $c \in k.C$ and $d \in k.C$ such that $(c, d) \in R$, we have $apply(k, c, P) = apply(\ell, d, P)$. This follows by induction on the length of P from the definition of *apply*. Thus, $k \approx \ell$ follows by specialization to $c = k.missing$ and $d = \ell.missing$.

($\Longrightarrow$) Now suppose $k \approx \ell$. To show $k \equiv \ell$, define $R \subseteq k.C \times \ell.C$ by $R = \{(c, d) \mid apply(k, c, P) = apply(\ell, d, P)$ for all $P\}$. By assumption, we have $(k.missing, \ell.missing) \in R$.

Now suppose that $(c, d) \in R$ and that $k.putr(x, c) = (y, c')$ and $\ell.putr(x, d) = (y', d')$. Applying the assumption $(c, d) \in R$ to the length-one sequence $P = \langle \mathsf{inl}\ (x) \rangle$ shows $y = y'$. To show $(c', d') \in R$ let P be an arbitrary sequence of put objects and define $P' = \mathsf{inl}\ (x){:}P$. The assumption $(c, d) \in R$ gives $apply(k, c, P') = apply(\ell, d, P')$, hence in particular $apply(k, c', P) = apply(\ell, d', P)$, thus $(c', d') \in R$. We have thus shown that $k.putr \sim_R \ell.putr$. Analogously, we show that $k.putl \sim_R \ell.putl$, and it follows that $k \equiv \ell$ via relation R. $\square$

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 () its only element.

$$\frac{}{id_X \in X \leftrightarrow X}$$

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

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. The proof is a straightforward calculation.

4.2 Definition [Lens composition]:

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

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

Verifying that this forms a lens is straightforward.

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.

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

4.4 Definition [Swizzle]: The following function on relations will be convenient in several proofs:

$$R_1 \times R_2 = \{((c_1,c_2),(c_1',c_2')) \mid (c_1,c_1') \in R_1 \wedge (c_2,c_2') \in R_2\}$$

Proof: If R_k is a simulation witnessing $k \equiv k'$ and R_ℓ witnesses $\ell \equiv \ell'$ then it is straightforward to verify that $R_k \times R_\ell$ witnesses $k; \ell \equiv k'; \ell'$. $\qquad \square$

4.5 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.)

Proof: We define a witnessing simulation relation R by

$$(c_1,(c_2,c_3)) \; R \; ((c_1,c_2),c_3)$$

whenever $c_1 \in j.C$ and $c_2 \in k.C$ and $c_3 \in \ell.C$. $\qquad \square$

4.6 Lemma [Identity arrows]: The identity arrow is a left and right identity for composition:

$$id_X; \ell \equiv \ell; id_Y \equiv \ell$$

Proof: For left identity we use the simulation relation R given by $((),c_2)Rc_2$ whenever $c_2 \in \ell.C$. The verification is direct; right identity is symmetric. $\qquad \square$

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 $[k];[\ell] = [k;\ell]$.

4.7 Proposition [Bijective lenses]: Every bijective function gives rise to a lens:

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

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

(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.)

We omit the obvious verification that bij_f is a lens.

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.

4.8 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 $bij_f; bij_g = bij_{f;g}$.

Proof: It is easy to see that $bij_{id_X} = id_X$. To show that $bij_f; bij_g = bij_{f;g}$, we use the complete relation $R \in (Unit \times Unit) \times Unit$ together with the identity $(f;g)^{-1} = g^{-1}; f^{-1}$. $\qquad \square$

Since functors preserve isomorphisms it follows that bijective lenses are isomorphisms in the category of lenses. However, not every isomorphism in LENS is of that form.

Consider the following counterexample. Define the set $Trit = \{-1,0,1\}$ and the function $f \in Trit \times Trit \to Trit$ which returns its arguments if they are equal and the third possible value if they are not:

c	x	$f(c,x)$
-1	-1	-1
-1	0	1
-1	1	0
0	-1	1
0	0	0
0	1	-1
1	-1	0
1	0	-1
1	1	1

For any particular c, the partial application $f(c)$ is a bijection and an involution. Thus, we can define the following lens, which is its own inverse but is not equivalent to any bijective lens:

$$\boxed{\begin{array}{l} \quad strange \in Trit \leftrightarrow Trit \\ \hline C \qquad\qquad = \; Unit + Trit \\ missing \qquad = \; \mathsf{inl} \; () \\ putr(x, \mathsf{inl} \; ()) \;=\; (x, \mathsf{inr} \; x) \\ putr(x, \mathsf{inr} \; c) \;=\; (f(c,x), \mathsf{inr} \; c) \\ putl(x, \mathsf{inl} \; ()) \;=\; (x, \mathsf{inr} \; x) \\ putl(x, \mathsf{inr} \; c) \;=\; (f(c,x), \mathsf{inr} \; c) \end{array}}$$

We can show, however, that the $putr$ and $putl$ functions of any invertible lens induce a bijection between the two replicas for any pair of reachable complements. More precisely:

4.9 Lemma: Suppose we have lenses $k \in X \leftrightarrow Y$ and $\ell \in Y \leftrightarrow X$ such that $k; \ell \equiv id_X$ and $\ell; k \equiv id_Y$. Then there is a relation $R \in k.C \times \ell.C$ satisfying the following conditions:

$$(k; \ell).missing \in R \tag{1}$$

$$\frac{(k; \ell).putr(x, c) = (x', c') \qquad c \in R}{x' = x \wedge c' \in R} \tag{2}$$

$$\frac{(k; \ell).putl(x, c) = (x', c') \qquad c \in R}{x' = x \wedge c' \in R} \tag{3}$$

$$\frac{(\ell; k).putr(y, c) = (y', c') \qquad \gamma^\times(c) \in R}{y' = y \wedge \gamma^\times(c') \in R} \tag{4}$$

$$\frac{(\ell; k).putl(y, c) = (y', c') \qquad \gamma^\times(c) \in R}{y' = y \wedge \gamma^\times(c') \in R} \tag{5}$$

Here, the function $\gamma^\times$ is the symmetry in SET, namely $\gamma^\times((x,y)) = (y, x)$.

Proof: We get an R_1 that satisfies 1-3 from the fact that $k; \ell \equiv id_X$, and we get an R_2 that satisfies 1, 4, and 5 from the fact that $\ell; k \equiv id_Y$. Then we can define $R = R_1 \cap R_2$. There are four conditions to check, but we will consider only one of them here, as the others are very similar:

$$\frac{(k; \ell).putr(x, c) = (x', c') \qquad c \in R}{c' \in R_2}$$

Now $c \in R$ means there are c_k, c_ℓ such that $c_k \; R_1 \; c_\ell$ and $c_k \; R_2 \; c_\ell$. We can define

$$\begin{aligned} (y, c_k') &= k.putr(x, c_k) \\ (x', c_\ell') &= \ell.putr(y, c_\ell). \end{aligned}$$

Since R_1 satisfies 2, we know $x' = x$, that is, we know

$$\begin{aligned} \ell.putr(y, c_\ell) &= (x, c_\ell') \\ k.putr(x, c_k) &= (y, c_k'). \end{aligned}$$

Now the fact that R_2 satisfies 4 above tells us that $c_k' \; R_2 \; c_\ell'$, that is, $c' \in R_2$. $\qquad\square$

4.10 Corollary: Consider the functions f and g which give the value-only part of a lens' puts:

$$\begin{aligned} f_{\ell,c_\ell}(x) &= \mathsf{fst}(\ell.putr(x, c_\ell)) \\ g_{\ell,c_\ell}(x) &= \mathsf{fst}(\ell.putl(x, c_\ell)) \end{aligned}$$

If $c_k \; R \; c_\ell$ (using the R given by the previous lemma), then f_{k,c_k}, f_{ℓ,c_ℓ}, g_{k,c_k}, and g_{ℓ,c_ℓ} are all bijections.

Proof: We show that they are injective first. Choose x_1, x_2 such that $f_{k,c_k}(x_1) = f_{k,c_k}(x_2)$; then

$$\begin{aligned} x_1 &= f_{\ell,c_\ell}(f_{k,c_k}(x_1)) && \text{by 2} \\ &= f_{\ell,c_\ell}(f_{k,c_k}(x_2)) && \text{by assumption} \\ &= x_2 && \text{by 2} \end{aligned}$$

The other functions can similarly be shown to be injective. But now since $f_{k,k.missing} \in X \to Y$ and $f_{\ell,\ell.missing} \in Y \to X$ are both injective, we know $|X| = |Y|$ and hence that each of the functions is also bijective. $\qquad\square$

4.11 Definition [Dual of a lens]:

$$\boxed{\begin{array}{c} \dfrac{\ell \in X \leftrightarrow Y}{\ell^{op} \in Y \leftrightarrow X} \\[2mm] \hline \begin{array}{ll} C & = \; \ell.C \\ missing & = \; \ell.missing \\ putr(y, c) & = \; \ell.putl(y, c) \\ putl(x, c) & = \; \ell.putr(x, c) \end{array} \end{array}}$$

The verifications of well-definedness as a lens and on equivalence classes is straightforward.

It is easy to see that $(-)^{op}$ is involutive—that is, that $(\ell^{op})^{op} = \ell$ for every ℓ—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.

The following is direct.

4.12 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 $[\ell^{op}]$.

4.13 Corollary: The category LENS is self dual, i.e., equivalent to its own opposite. (Note that this does not mean that each arrow is its own inverse!)

Proof: This is because the morphism part of $(-)^{op}$ is bijective. $\square$

4.14 Definition [Terminal lens]:

$$\boxed{\begin{array}{c} \dfrac{x \in X}{term_x \in X \leftrightarrow Unit} \\[2mm] \hline \begin{array}{ll} C & = \; X \\ missing & = \; x \\ putr(x', c) & = \; ((), x') \\ putl((), c) & = \; (c, c) \end{array} \end{array}}$$

The verification is straightforward.

4.15 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 $k \equiv term_x$.

Proof: The behavior of k is uniquely defined by the given data: $putl$ must return x the first time and echo the last $putr$ henceforth. Formally, we may define a simulation relation and return x the first time. We use the simulation relation

$$R = \{(c, y) \mid \mathsf{fst}(k.putl((), c)) = y\}$$

The verifications are straightforward. $\qquad\square$

4.16 Definition [Disconnect lens]:

$$\boxed{\begin{array}{c} \dfrac{x \in X \qquad y \in Y}{disconnect_{xy} \in X \leftrightarrow Y} \\[2mm] \hline disconnect_{xy} \;=\; term_x; term_y^{op} \end{array}}$$

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.

(Note that we do not need an explicit proof that $disconnect$ is a lens: this follows from the fact that $term$ is a lens and $(-)^{op}$ and ; construct lenses from lenses.)

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. 4.15 this shows that $Unit \times Unit$ is a one-element set. This, on the other hand, contradicts the symmetric nature of $lens$.

In more detail: Assume, for a contradiction, that LENS does have products, and let W be the product of $Unit$ with itself. The two projections are maps into $Unit$. By Proposition 4.15 there is exactly one lens from $Unit$ to $Unit$. By uniqueness of pairing we can then conclude that there is exactly one map from $Unit$ to W. Now for each $w \in W$ the lens $(term_w)^{op}$ is such a map, whence W must be a singleton set, and we can without loss of generality assume $W = Unit$. But now consider the pairing of $term_0$ and $term_1$ from $\{0, 1\}$ to $Unit$. Their pairing is a lens from $\{0, 1\}$ to $W = Unit$, hence itself of the form $term_x$ for some $x \in \{0, 1\}$. But each of these violate the naturality laws. $\square$

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 $\pi_1, \pi_2, diag$

$$\pi_{1,X,Y} \in X \otimes Y \to X$$
$$\pi_{2,X,Y} \in X \otimes Y \to Y$$
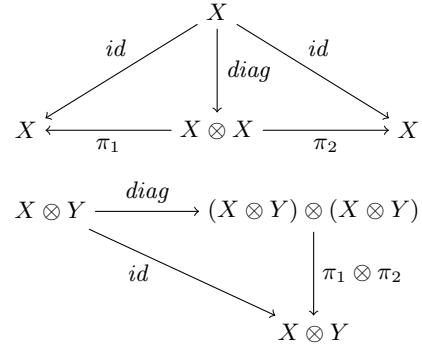$$diag_X \in X \to X \otimes X$$

such that (suppressing subscripts to reduce clutter)

$$(f \otimes g); \pi_1 = \pi_1; f \qquad (1)$$
$$(f \otimes g); \pi_2 = \pi_2; g \qquad (2)$$
$$diag; (f \otimes f) = f; diag \qquad (3)$$

for all arrows f and g. Moreover, the following diagrams must commute, in the sense that composite arrows with the same endpoints represent equal arrows:

The former diagram says that the result of applying $diag$ is an element whose components are both equal to the original. The latter diagram says that the application of $diag$ results in independent copies of the original.

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 make the additional diagrams commute.

5.2 Definition [Tensor product lens]:

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

$$
\begin{aligned}
C &= k.C \times \ell.C \\
missing &= (k.missing, \ell.missing) \\
putr((x, y), (c_k, c_\ell)) &= \text{let } (z, c_k') = k.putr(x, c_k) \text{ in} \\
& \quad \text{let } (w, c_\ell') = \ell.putr(y, c_\ell) \text{ in} \\
& \quad ((z, w), (c_k', c_\ell')) \\
putl((z, w), (c_k, c_\ell)) &= \text{let } (x, c_k') = k.putl(z, c_k) \text{ in} \\
& \quad \text{let } (y, c_\ell') = \ell.putl(w, c_\ell) \text{ in} \\
& \quad ((x, y), (c_k', c_\ell'))
\end{aligned}
$$

The verification that this forms a lens is straightforward.

Proof of preservation of equivalence: If R_k is a witness that $k \equiv k'$ and R_ℓ is a witness that $\ell \equiv \ell'$, then $R_k \times R_\ell$ witnesses $k \otimes \ell \equiv k' \otimes \ell'$. $\square$

5.3 Lemma: The tensor product operation on lenses induces a bifunctor on the category LENS, that is,

$$id_X \otimes id_Y \equiv id_{X \times Y}, \text{ and}$$
$$(k_1; \ell_1) \otimes (k_2; \ell_2) \equiv (k_1 \otimes k_2); (\ell_1 \otimes \ell_2).$$

Proof of functoriality: The complete relation $R \in (Unit \times Unit) \times Unit$ witnesses the former equivalence.

The latter has a slightly more intricate (but hardly more interesting) witness relation:

$$((c_{k_1}, c_{\ell_1}), (c_{k_2}, c_{\ell_2})) \; R \; ((c_{k_1}, c_{k_2}), (c_{\ell_1}, c_{\ell_2}))$$

That is, one state is related to another precisely when it is a rearrangement of the component states. It is clear that this relates the $missing$ states of each lens, and the $putr$ and $putl$ components do identical computations (albeit in a different order), so they are related by $\sim_R$ as necessary. $\square$

5.4 Lemma [Product bijection]: For bijections f and g,

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

Proof: Write $k = bij_f \otimes bij_g$ and $\ell = bij_{f \times g}$. The total relation $R \in (Unit \times Unit) \times Unit$ is a witness. It's clear that $k.missing\ R\ \ell.missing$, so let's show that the puts are similar. Since all complements are related, this reduces to showing that equal input values yield equal output values.

$$
\begin{aligned}
k.putr((x,y),((),())) &= \quad \text{let } (x',c_1) = bij_f.putr(x,()) \text{ in} \\
&\qquad \text{let } (y',c_2) = bij_g.putr(y,()) \text{ in} \\
&\qquad ((x',y'),(c_1,c_2)) \\
&= \quad ((f(x),g(y)),((),())) \\
\ell.putr((x,y),()) &= \quad ((f(x),g(y)),())
\end{aligned}
$$

The $putl$ direction is similar. $\qquad\square$

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:

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

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 the coherence equations:

$$
\begin{aligned}
\alpha ; \alpha &= (\alpha \otimes id); \alpha; (id \otimes \alpha) \\
\rho \otimes id &= \alpha; (id \otimes \lambda) \\
\alpha; \gamma; \alpha &= (\gamma \otimes id_1); \alpha; (id_1 \otimes \gamma) \\
\alpha^{-1}; \gamma; \alpha^{-1} &= (id_1 \otimes \gamma); \alpha^{-1}; (\gamma \otimes id_1) \\
\gamma; \gamma &= id
\end{aligned}
$$

5.5 Proposition [LENS,$\otimes$ 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).

Proof: We know $\alpha^\otimes$, $\lambda^\otimes$, $\rho^\otimes$, and $\gamma^\otimes$ are all isomorphisms because every bijection lens is an isomorphism. Showing that they are natural is a straightforward calculation. The five coherence conditions follow from coherence in SET, functoriality of bij, and Lemma 5.4. $\qquad\square$

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

$$
\boxed{
\begin{array}{c}
\dfrac{y \in Y}{\pi_{1y} \in X \times Y \leftrightarrow X} \\[1.5em]
\hline
\pi_{1y} = (id_X \otimes term_y); \rho_X
\end{array}
}
$$

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.

5.7 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 $(k \otimes \ell); \pi_{1y_i} \equiv \pi_{1x_i}; k$.

Proof: We show that the following diagram commutes:

$$
\begin{array}{ccc}
X_k \times X_\ell & \xrightarrow{\;\; k \otimes \ell \;\;} & Y_k \times Y_\ell \\
{\scriptstyle id_{X_k} \otimes term_{x_i}} \downarrow & & \downarrow {\scriptstyle id_{Y_k} \otimes term_{y_i}} \\
X_k \times Unit & \xrightarrow{\; k \otimes id_{Unit} \;} & Y_k \times Unit \\
{\scriptstyle \rho_{X_k}} \downarrow & & \downarrow {\scriptstyle \rho_{Y_k}} \\
X_k & \xrightarrow{\;\; k \;\;} & Y_k
\end{array}
$$

To show that the top square commutes, we invoke functoriality of $\otimes$ and the property of identities; all that remains is to show that

$$\ell; term_{y_i} \equiv term_{x_i}$$

which follows from the uniqueness of terminal lenses and the definition of x_i. The bottom square commutes because ρ is a natural isomorphism. $\qquad\square$

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 [12])

$$
\boxed{diag \in X \to X \times X}
$$

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

where here the eq function tests its arguments for equality:

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

This assumes that X possesses a decidable equality, a reasonable assumption for the applications of lenses that we know about.

Proof of well-formedness:
PUTRL:

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

$\square$

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.

Before we study the question for LENS, let us formally define a sum. 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 4.13 and Theorem 5.1 implies 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. But these constructions are even farther away from being categorical sums than what we saw with products.

As with products, a tensor can be extended to a sum by providing three natural transformations—this time written inl, inr, and $codiag$; that is, for each pair of objects X and Y, there must be arrows

$$
\begin{aligned}
inl_{X,Y} &\in X \to X \oplus Y \\
inr_{X,Y} &\in Y \to X \oplus Y \\
codiag_X &\in X \oplus X \to X
\end{aligned}
$$

such that

$$
\begin{aligned}
inl; (f \oplus g) &= f; inl \\
inr; (f \oplus g) &= g; inr \\
(f \oplus f); codiag &= codiag; f
\end{aligned}
$$

and making the following diagrams commute:

$$
\begin{array}{ccc}
X \xrightarrow{inl} & X \oplus X & \xleftarrow{inr} X \\
\end{array}
$$

(diagram with id, $codiag$, id to X)

(diagram with $X \oplus Y$, id, $inl \oplus inr$, $codiag$, $(X \oplus Y) \oplus (X \oplus Y)$, $X \oplus Y$)

These diagrams are identical to the product diagrams, with the exception that the arrows point in the opposite directions (that is, the sum diagrams are the dual of the product diagrams).

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*.

6.1 Definition [Retentive tensor sum lens]:

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

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

Proof of well-formedness: This is a straightforward, but tedious case analysis. $\square$

Proof of preservation of equivalence: Suppose $k \equiv k'$ and $\ell \equiv \ell'$, as witnessed by relations R_k and R_ℓ, respectively. Then $R_k \times R_\ell$ witnesses the equivalence $k \oplus \ell \equiv k' \oplus \ell'$. $\square$

6.2 Lemma: The tensor sum operation on lenses induces a bifunctor on LENS.

Proof of functoriality: The total relation $R \in (Unit \times Unit) \times Unit$ is a witness that $id_X \oplus id_Y \equiv id_{X+Y}$. For composition, the obvious isomorphism between complements witnesses the equivalence $(k;\ell) \oplus (k';\ell') \equiv (k \oplus k'); (\ell \oplus \ell')$, namely:

$$
\frac{c_k \in k.C \qquad c'_k \in k'.C \qquad c_\ell \in \ell.C \qquad c'_\ell \in \ell'.C}{(((c_k, c_\ell), (c'_k, c'_\ell)), ((c_k, c'_k), (c_\ell, c'_\ell))) \in R}
$$

$\square$

6.3 Definition [Forgetful tensor sum]:

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

$$
\begin{aligned}
C &= k.C + \ell.C \\
missing &= \mathsf{inl}\,k.missing \\
putr(\mathsf{inl}\,x, \mathsf{inl}\,c_k) &= \mathsf{let}\,(z, c'_k) = k.putr(x, c_k)\,\mathsf{in} \\
& \quad (z, \mathsf{inl}\,c'_k) \\
putr(\mathsf{inl}\,x, \mathsf{inr}\,c_\ell) &= \mathsf{let}\,(z, c_k) = k.putr(x, k.missing)\,\mathsf{in} \\
& \quad (z, \mathsf{inl}\,c_k) \\
putr(\mathsf{inr}\,y, \mathsf{inl}\,c_k) &= \mathsf{let}\,(w, c_\ell) = \ell.putr(y, \ell.missing)\,\mathsf{in} \\
& \quad (w, \mathsf{inr}\,c_\ell) \\
putr(\mathsf{inr}\,y, \mathsf{inr}\,c_\ell) &= \mathsf{let}\,(w, c'_\ell) = k.putr(y, c_\ell)\,\mathsf{in} \\
& \quad (w, \mathsf{inr}\,c'_\ell) \\
\\
putl \text{ is similar}
\end{aligned}
$$

Proof of well-formedness: Straightforward case analysis. □

Proof of preservation of equivalence: If R witnesses $k \equiv k'$ and S witnesses $\ell \equiv \ell'$ then $k \oplus^f \ell \equiv k' \oplus^f \ell'$ may be witnessed by

$$\{(\mathsf{inl}\,(c), \mathsf{inl}\,(c')) \mid cRc'\} \cup \{(\mathsf{inr}\,(c), \mathsf{inr}\,(c')) \mid cSc'\}$$

The verification is straightforward. □

Proof of functoriality: By straightforward case analysis. □

6.4 Lemma [Sum bijection]: For bijections f and g,

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

Proof: Write $k = bij_f \oplus bij_g$, $k^f = bij_f \oplus^f bij_g$, and $\ell = bij_{f \times g}$. The total relation $R \in (Unit \times Unit) \times Unit$ is a witness that $k \equiv \ell$ and the total relation $R^f \in (Unit + Unit) \times Unit$ is a witness that $k^f \equiv \ell$. It's clear that $k.missing\ R\ \ell.missing$ and $k^f.missing\ R^f\ \ell.missing$, so let's show that the puts are similar. Since all complements are related, this reduces to showing that equal input values yield equal output values.

$$
\begin{aligned}
k.putr(\mathsf{inl}\ x, ((), ())) &= \text{let } (z, c_k) = bij_f.putr(x, ()) \text{ in} \\
&\qquad (\mathsf{inl}\ z, (c_k, ())) \\
&= \text{let } (z, c_k) = (f(x), ()) \text{ in} \\
&\qquad (\mathsf{inl}\ z, (c_k, ())) \\
&= (\mathsf{inl}\ f(x), ((), ())) \\
k.putr(\mathsf{inr}\ y, ((), ())) &= (\mathsf{inr}\ g(y), ((), ())) \\
k^f.putr(\mathsf{inl}\ x, c) &= \text{let } (z, c_k) = bij_f.putr(x, ()) \text{ in} \\
&\qquad (\mathsf{inl}\ z, \mathsf{inl}\ c_k) \\
&= \text{let } (z, c_k) = (f(x), ()) \text{ in} \\
&\qquad (\mathsf{inl}\ z, \mathsf{inl}\ c_k) \\
&= (\mathsf{inl}\ f(x), \mathsf{inl}\ ()) \\
k^f.putr(\mathsf{inr}\ y, c) &= (\mathsf{inr}\ g(y), \mathsf{inr}\ ()) \\
\ell.putr(\mathsf{inl}\ x, ()) &= ((f + g)(\mathsf{inl}\ x), ()) \\
&= (\mathsf{inl}\ f(x), ()) \\
\ell.putr(\mathsf{inr}\ y, ()) &= (\mathsf{inr}\ g(y), ())
\end{aligned}
$$

The $putl$ direction is similar. □

6.5 Proposition [LENS,$\oplus$, $\oplus^f$ are symmetric monoidal categories]: 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:

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

Proof: We know $\alpha^\oplus$, $\lambda^\oplus$, $\rho^\oplus$, and $\gamma^\oplus$ are all isomorphisms because every bijection lens is an isomorphism. Showing that they are natural is a straightforward calculation. The only subtlety comes in showing that $(k \oplus^f \ell); \gamma^\oplus \equiv \gamma^\oplus; (\ell \oplus^f k)$. We must be careful to include the $missing$ complements in the relation; the following

relation will do:

$$
\begin{aligned}
R \quad = \quad &\{(\mathsf{inl}\ c, \mathsf{inr}\ c) \mid c \in k.C\} \cup \\
&\{(\mathsf{inr}\ c, \mathsf{inl}\ c) \mid c \in \ell.C\} \cup \\
&\{(\mathsf{inl}\ k.missing, \mathsf{inl}\ \ell.missing)\}
\end{aligned}
$$

The five coherence conditions follow from coherence in SET, functoriality of bij, and Lemma 6.4. □

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.

6.6 Definition [Injection lenses]:

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

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

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

Proof of well-formedness: Straightforward case analysis. □

6.7 Proposition: The injection lenses are not natural.

Proof: We first define a lens that counts the number of changes it sees in the $putr$ direction, and allows puts of non-numbers to be overridden in the $putl$ direction:

$$\frac{x \in X}{count_x \in X \leftrightarrow Unit + \mathbb{N}}$$

$$
\begin{aligned}
C &= X \times Bool \times \mathbb{N} \\
missing &= (x, \mathsf{true}, 0) \\
putr(x, (x', b, n)) &= \\
&\begin{cases}
(\mathsf{inl}\ (), (x, b, n)) & x = x' \wedge \neg b \\
(\mathsf{inr}\ n, (x, b, n)) & x = x' \wedge b \\
(\mathsf{inr}\ (n+1), (x, \mathsf{true}, n)) & x \neq x'
\end{cases} \\
putl(\mathsf{inl}\ (), (x, b, n)) &= (x, (x, \mathsf{false}, n)) \\
putl(\mathsf{inr}\ n, (x, b, n')) &= (x, (x, \mathsf{true}, n))
\end{aligned}
$$

We now contrast the lens $inl_b; (count_{b'} \oplus id_{Unit})$ with $count_{b'}; inl_n$ (where b and b' are arbitrary $Bool$ values and n is an arbitrary $Unit + \mathbb{N}$ value). Consider the put object

$\langle \mathsf{inl}\ \mathsf{true}, \mathsf{inr}\ (\mathsf{inr}\ ()), \mathsf{inl}\ \mathsf{false}, \mathsf{inr}\ (\mathsf{inl}\ (\mathsf{inl}\ ())), \mathsf{inl}\ \mathsf{true}, \mathsf{inl}\ \mathsf{false}\rangle$

The first two values in the put object are simply initializing the lens: we first put true to the right, getting an inl object out on the right from both lenses, then put back an inr object, switching sides.

The next put of false to the right is where the problem really arises. For the $count_{b'}; inl_n$ lens, the counting lens first registers the change from true to false, then its output gets thrown away. On the other hand, in the $inl_b; (count_{b'} \oplus id_{Unit})$ lens, the false gets thrown away before the counting lens can see it, so the complement in the counting lens doesn't get updated.

The remainder of the puts simply manifest this problem by switching the sum back to the counting side, and getting an output from the counting lenses; one will give a higher count than the other.

The proof for inr is symmetric. □

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 inr tag or an inl tag. Below, we propose two lenses that break this tie in different ways. The $union$ lens uses the most recent unambiguous put to break the tie. The $union'$ lens, on the other hand, looks back to the last tagged value that was put to the right that was in both sets.

6.8 Definition [Union lens]:

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

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

Proof of well-formedness:

PutRL:

$$
\begin{aligned}
putl(putr(\mathsf{inr}\ x, c)) &= putl(x, \mathsf{false}) \\
&= (\mathsf{inr}\ x, \mathsf{false}) \\
putl(putr(\mathsf{inl}\ y, c)) &= putl(y, \mathsf{true}) \\
&= (\mathsf{inl}\ y, \mathsf{true})
\end{aligned}
$$

PutLR: There are six cases to consider, corresponding to which of the sets X, Y, and $X \cap Y$ our value is a member of and to whether the complement is true or false.

$$
\begin{aligned}
putr(putl(xy, \mathsf{false})) &= putr(\mathsf{inl}\ xy, \mathsf{false}) \\
&= (xy, \mathsf{false}) \\
putr(putl(x, \mathsf{false})) &= putr(\mathsf{inl}\ x, \mathsf{false}) \\
&= (x, \mathsf{false}) \\
putr(putl(y, \mathsf{false})) &= putr(\mathsf{inr}\ y, \mathsf{true}) \\
&= (y, \mathsf{true})
\end{aligned}
$$

The cases for when the complement is true are symmetric. □

6.9 Definition [Another union lens]: Given two sets X and Y, let's define a few bijections:

$$
\begin{aligned}
f &\in X \to X \setminus Y + X \cap Y \\
g &\in Y \to X \cap Y + Y \setminus X \\
h &\in X \setminus Y + X \cap Y + Y \setminus X \to X \cup Y \\
f(x) &= \begin{cases} \mathsf{inl}\ x & x \notin Y \\ \mathsf{inr}\ x & x \in Y \end{cases} \\
g(y) &= \begin{cases} \mathsf{inl}\ y & y \in X \\ \mathsf{inr}\ y & y \notin X \end{cases} \\
h(\mathsf{inl}\ x) &= x \\
h(\mathsf{inr}\ (\mathsf{inl}\ xy)) &= xy \\
h(\mathsf{inr}\ (\mathsf{inr}\ y)) &= y
\end{aligned}
$$

$$union'_{XY} \in X + Y \leftrightarrow X \cup Y$$

$$
\begin{aligned}
union'_{XY} = \ &bij_{(f+g);\alpha+}; \\
&(id_X \oplus (union_{X \cap Y, X \cap Y} \oplus id_Y)); \\
&bij_h
\end{aligned}
$$

These definitions are not symmetric in X and Y, because $putl$ prefers to return an inl value if there have been no tie breakers yet. Because of this preference, neither $union$ nor $union'$ can be used to construct a true codiagonal. However, there are two useful related constructions:

6.10 Definition [Switch lens]:

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

$$switch_X = union_{XX}$$

We've used $union$ rather than $union'$ in this definition, but it actually doesn't matter: the two lenses' tie-breaking methods are equivalent when $X = Y$:

6.11 Lemma:

$$union_{XX} \equiv union'_{XX}$$

Proof: The relation that equates the states of the two $union$ lenses is a witness: $R = \{(b, (((), (b, ())), ())) \mid b \in Bool\}$. □

6.12 Definition [Retentive case lens]:

$$\frac{k \in X \leftrightarrow Z \qquad \ell \in Y \leftrightarrow Z}{case_{k,\ell} \in X + Y \leftrightarrow Z}$$

$$case_{k,\ell} = (k \oplus \ell); switch_X$$

6.13 Definition [Forgetful case lens]:

$$\frac{k \in X \leftrightarrow Z \qquad \ell \in Y \leftrightarrow Z}{case^f_{k,\ell} \in X + Y \leftrightarrow Z}$$

$$case^f_{k,\ell} = (k \oplus^f \ell); switch_X$$

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^ω 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.

6.14 Definition [Retentive list mapping lens]:

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

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

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, 12].

6.15 Definition [Forgetful list mapping lens]:

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

$$
\begin{aligned}
C &= \ell.C^{\star} \\
missing &= \langle \rangle \\
putr(x,c) &= \mathsf{let}\ \langle x_1, \ldots, x_m \rangle = x\ \mathsf{in} \\
&\quad \mathsf{let}\ \langle c_1, \ldots, c_n \rangle = c\ \mathsf{in} \\
&\quad \mathsf{let}\ \langle c_{n+1}, \ldots \rangle = (\ell.missing)^{\omega}\ \mathsf{in} \\
&\quad \mathsf{let}\ (y_i, c_i') = \ell.putr(x_i, c_i)\ \mathsf{in} \\
&\quad (\langle y_1, \ldots, y_m \rangle, \langle c_1', \ldots, c_m' \rangle) \\
putl &\quad \text{(similar)}
\end{aligned}
$$

Rather than proving that these two forms of list mapping are lenses, preserve equivalence, induce functors, and so on, we show that these properties hold for a generalization of their construction in the next section.

We can make the relationship between the retentive sum and map lenses and the forgetful sum and map lenses precise; the following two diagrams commute:

$$
\begin{array}{ccc}
Unit + X \times X^{\star} & \xrightarrow{\ bij\ } & X^{\star} \\
{\scriptstyle id_{Unit} \oplus (\ell \otimes map(\ell))}\downarrow & & \downarrow{\scriptstyle map(\ell)} \\
Unit + Y \times Y^{\star} & \xrightarrow{\ bij\ } & Y^{\star}
\end{array}
$$

$$
\begin{array}{ccc}
Unit + X \times X^{\star} & \xrightarrow{\ bij\ } & X^{\star} \\
{\scriptstyle id_{Unit} \oplus^f (\ell \otimes map^f(\ell))}\downarrow & & \downarrow{\scriptstyle map^f(\ell)} \\
Unit + Y \times Y^{\star} & \xrightarrow{\ bij\ } & Y^{\star}
\end{array}
$$

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, of course, the mapping combinator—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* [27].

We also discuss iteration patterns on trees and argue that the methodology carries over to other polynomial inductive datatypes.

7.1 Lists

Let $f \in Unit + (X \times X^{\star}) \rightarrow X^{\star}$ be the bijection between "unfolded" lists and lists that takes inl $()$ to $\langle \rangle$ and 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 $w \in Z \rightarrow \mathbb{N}$ such that $\ell.putl(z, c) = (\mathsf{inr}\ (x, z'), c')$ implies $w(z') < w(z)$. We write $T_X^{\star}$ for the functor that sends any lens k to $id_{Unit} \oplus (id_X \otimes k)$.

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

$$
\begin{array}{ccc}
T_X^{\star}(X^{\star}) & \xrightarrow{\ bij_f\ } & X^{\star} \\
{\scriptstyle T_X^{\star}(It(\ell))}\downarrow & & \downarrow{\scriptstyle It(\ell)} \\
T_X^{\star}(Z) & \xrightarrow{\quad \ell \quad} & Z
\end{array}
$$

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 $F(Z) \rightarrow Z$. A homomorphism between F-algebras (Z, f) and (Z', f') is a morphism $u \in Z \rightarrow Z'$ 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 [32]. Using this terminology, Theorem 7.1.2 says that bij_f is an initial object in the subcategory consisting of those $T_X^{\star}$-algebras for which a weight function w is available.

Before we give the proof, let us consider some concrete instances of the theorem. First, if $k \in X \Longleftrightarrow Y$ is a lens, then we can form an X-list algebra ℓ on $Y^{\star}$ by composing two lenses as follows:

$$
\begin{array}{c}
Unit + (X \times Y^{\star}) \\
\downarrow{\scriptstyle id_{Unit} \oplus (k \otimes id_{Y^{\star}})} \\
Unit + (Y \times Y^{\star}) \\
\downarrow{\scriptstyle bij_f} \\
Y^{\star}
\end{array}
$$

A suitable weight function is given by $w(ys) = length(ys)$. The induced lens $It(\ell) \in X^{\star} \Longleftrightarrow Y^{\star}$ is the lens analog of the familiar list mapping function. In fact, substituting the lens $e \in X \times Y \Longleftrightarrow Y \times Z$ (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

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

by distributing the sum and unfolding the list type for f and by factoring the polynomial and folding the list type for g.[2]

$$
\begin{aligned}
f(\mathsf{inl}\ x_1, xs_1, \langle\rangle) &= \mathsf{inl}\ (\mathsf{inl}\ (x_1, xs_1)) \\
f(\mathsf{inl}\ x_1, xs_1, x_2{:}xs_2) &= \mathsf{inr}\ (\mathsf{inl}\ ((x_1, xs_1), (x_2, xs_2))) \\
f(\mathsf{inr}\ x_2, \langle\rangle, xs_2) &= \mathsf{inl}\ (\mathsf{inr}\ (x_2, xs_2)) \\
f(\mathsf{inr}\ x_2, x_1{:}xs_1, xs_2) &= \mathsf{inr}\ (\mathsf{inr}\ ((x_1, xs_1), (x_2, xs_2)))
\end{aligned}
$$

$$
\begin{aligned}
g(\mathsf{inl}\ ()) &= (\langle\rangle, \langle\rangle) \\
g(\mathsf{inr}\ (\mathsf{inl}\ (\mathsf{inl}\ (x_1, xs_1)))) &= (x_1 : xs_1, \langle\rangle) \\
g(\mathsf{inr}\ (\mathsf{inl}\ (\mathsf{inr}\ (x_2, xs_2)))) &= (\langle\rangle, x_2 : xs_2) \\
g(\mathsf{inr}\ (\mathsf{inr}\ ((x_1, xs_1), (x_2, xs_2)))) &= (x_1 : xs_1, x_2 : xs_2)
\end{aligned}
$$

Then we can create

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

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

A suitable weight function for ℓ is given by $w((xs_1, xs_2)) = length(xs_1) + length(xs_2)$. The lens $It(\ell) \in (X_1 + X_2)^\star \Longleftrightarrow X_1^\star \times X_2^\star$ 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 ℓ.)

Proof of 7.1.2: We define the lens $It(\ell)$ explicitly.

$$
\boxed{
\cfrac{\ell \in T_X^\star(Z) \leftrightarrow Z \qquad \exists\ \text{suitable}\ w}{It(\ell) \in X^\star \leftrightarrow Z}
}
$$

$$
\boxed{
\begin{aligned}
It(\ell).C &= (\ell.C)^\omega \\
It(\ell).missing &= (\ell.missing)^\omega \\
It(\ell).putr(\langle\rangle, c{:}cs) &= \mathsf{let}\ (z, c') = \ell.putr(\mathsf{inl}\ (), c)\ \mathsf{in} \\
&\quad (z, c'{:}cs) \\
It(\ell).putr(x{:}xs, c{:}cs) &= \mathsf{let}\ (z, cs') = It(\ell).putr(xs, cs)\ \mathsf{in} \\
&\quad \mathsf{let}\ (z', c') = \ell.putr(\mathsf{inr}\ (x, z), c)\ \mathsf{in} \\
&\quad (z', c'{:}cs') \\
It(\ell).putl(z, c{:}cs) &= \mathsf{match}\ \ell.putl(z, c)\ \mathsf{with} \\
&\quad (\mathsf{inl}\ (), c') \to (\langle\rangle, c'{:}cs) \\
&\quad |\ (\mathsf{inr}\ (x, z'), c') \to \\
&\quad \mathsf{let}\ (xs, cs') = It(\ell).putl(z', cs)\ \mathsf{in} \\
&\quad (x{:}xs, c'{:}cs')
\end{aligned}
}
$$

Note that the first element of the complement list holds *both* the complement that is used when we do a *putr* of an empty list *and* the

[2] The bijections f and g can be written in terms of the associators, symmetries, unfolds, folds, and so forth that were already introduced, so the lenses bij_f and bij_g would not have to be defined "out of whole cloth" as they are here, but these definitions get bogged down in syntax without adding much value.

complement that is used for the first element when we do a *putr* of a non-empty list. Similarly, the second element of the complement list holds both the complement that is used at the end when we do a *putr* of a one-element empty list and the complement that is used for the second element when we do a *putr* of a two or more element list.

The recursive definition of $It(\ell).putr$ is clearly terminating because the first argument to the recursive call is always a shorter list; the recursive definition of $It(\ell).putl$ is terminating because the value of w is always smaller on the arguments to the recursive call. The round-trip laws are readily established by induction on xs and on $w(z)$, respectively. So this is indeed a lens.

Commutativity of the claimed diagram is a direct consequence of the defining equations (which have been crafted so as to make commutativity hold).

To show uniqueness, let $k \in X^\star \Longleftrightarrow Z$ be another lens for which the diagram commutes—i.e., such that:

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

Choose representatives of the equivalence classes k and ℓ—for convenience, call these representatives k and ℓ. Let $R \subseteq k.C \times (k.C \times \ell.C)$ be a simulation relation witnessing the commutativity of this diagram (recalling that equality of LENS-arrows means lens-equivalence of representatives). Notice that $k.C$ is the complement of (a representative of) the upper path through the diagram, and $k.C \times \ell.C$ is the complement of (a representative of) the lower path through the diagram. (Strictly speaking, the complements are $Unit \times k.C$ and $Unit \times Unit \times k.C \times \ell.C$; using these isomorphic forms reduces clutter.) Thus, the commutativity of the diagram means:

$$
(k.missing, (k.missing, \ell.missing)) \in R
$$

$$
\cfrac{\begin{array}{c} (d, (d', c)) \in R \\ k.putr(\langle\rangle, d) = (z, d_1) \qquad \ell.putr(\mathsf{inl}\ (), c) = (z', c_1) \end{array}}{(d_1, (d', c_1)) \in R\ \wedge\ z = z'}
$$

$$
\cfrac{\begin{array}{c} (d, (d', c)) \in R \qquad k.putr(x{:}xs, d) = (z, d_1) \\ k.putr(xs, d') = (z', d_1') \qquad \ell.putr(\mathsf{inr}\ (x, z'), c) = (z'', c_1) \end{array}}{(d_1, (d_1', c_1)) \in R\ \wedge\ z = z''}
$$

$$
\cfrac{\begin{array}{c} (d, (d', c)) \in R \\ k.putl(z, d) = (\langle\rangle, d_1) \end{array}}{\ell.putl(z, c) = (\mathsf{inl}\ (), c_1) \wedge (d_1, (d', c_1)) \in R}
$$

$$
\cfrac{(d, (d', c)) \in R \qquad k.putl(z, d) = (x{:}xs, d_1)}{\begin{array}{l} \ell.putl(z, c) = (\mathsf{inr}\ (x, z'), c_1) \\ \wedge\quad k.putl(z', d') = (xs, d_1') \\ \wedge\quad (d_1, (d_1', c_1)) \in R \end{array}}
$$

The variables c_1, z', d_1' in the last two rules are existentially quantified.

In order to show that $It(\ell) \equiv k$ we define a relation $S \subseteq It(\ell).C \times k.C$ inductively as follows:

$$
(It(\ell).missing, k.missing) \in S
$$

$$
\cfrac{(d, (d', c)) \in R \qquad (cs, d') \in S}{(c{:}cs, d) \in S}
$$

Notice that if $(c{:}cs, d) \in S$ by either one of the rules, then there exists d' such that $(d, (d', c)) \in R$ and $(cs, d') \in S$. In particular,

for the first rule, $c{:}cs = It(\ell).missing$ and we choose $d' = k.missing$.

It remains to show that S is compatible with $putl$ and $putr$. So assume that $(c{:}cs, d) \in S$, hence $(d, (d', c)) \in R$ and $(cs, d') \in S$ for some d'. We proceed by induction on $length(xs)$ in the $putr$ cases and by induction on $w(z)$ in the $putl$ cases.

Case for $putr$ of empty list: By definition, $It(\ell).putr(\langle\,\rangle, c{:}cs) = (z, c'{:}cs)$, where $(z, c') = \ell.putr(\mathsf{inl}\ (), c)$. Let $(z_1, d_1) = k.putr(\langle\,\rangle, d)$. Commutativity of the diagram then tells us that $(d_1, (d', c')) \in R$ and $z_1 = z$. Since $(cs, d') \in S$, we can conclude $(c'{:}cs, d_1) \in S$, as required.

Case for $putr$ of nonempty list: This time, the definition gives us $It(\ell).putr(x{:}xs, c{:}cs) = (z', c'{:}cs')$, where

$$\begin{aligned}(z, cs') &= It(\ell).putr(xs, cs)\\ (z', c') &= \ell.putr(\mathsf{inr}\ (x, z), c).\end{aligned}$$

Let

$$\begin{aligned}(z_1, d_1) &= k.putr(x{:}xs, d)\\ (z_2, d_2) &= k.putr(xs, d')\\ (z_3, c_3) &= \ell.putr(\mathsf{inr}\ (x, z_2), c).\end{aligned}$$

Inductively, we get $z_2 = z$ and $(cs', d_2) \in S$. Thus, $z_3 = z'$ and $c_3 = c'$. From commutativity we get $z_1 = z'$ and $(d_1, (d_2, c')) \in R$, so $(c'{:}cs', d_1) \in S$ and we are done.

Case where $It.putl$ on z returns the empty list: Suppose we have $It(\ell).putl(z, c{:}cs) = (\langle\,\rangle, c'{:}cs)$, where $(\mathsf{inl}\ (), c') = \ell.putl(z, c)$. Let $k.putr(z, d) = (xs, d_1)$. Commutativity of the diagram asserts that $(d_1, (c', d')) \in R$ and $xs = \langle\,\rangle$. Now, since $(cs, d') \in S$, we can conclude $(c'{:}cs, d_1) \in S$, as required.

Case where $It.putl$ on z returns a non-empty list: Suppose we have $It(\ell).putl(z, c{:}cs) = (x{:}xs, c'{:}cs')$, where $(\mathsf{inr}\ (x, z'), c') = \ell.putl(z, c)$ and $(xs, cs') = It(\ell).putl(z', cs)$. Since $\ell.putl(z, c)$ returns an inr we are in the situation of the fourth rule above and we have $k.putl(z, d) = (x{:}xs', d_1)$ for some xs' and d_1. Furthermore, we have $k.putl(z', d') = (xs', d_1')$ and $(d_1, (d_1', c_1)) \in R$. The induction hypothesis applied to z' in view of $w(z') < w(z)$ then yields $xs' = xs$ and also $(cs', d_1') \in S$. It then follows $(c'{:}cs', d_1) \in S$ and we are done. $\square$

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

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

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

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 ℓ 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 [27]. 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 ℓ 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 $Y{\rightarrow}Z$, 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 $T_{X,Y}^\star(Z) = Y + X \times Z$ and notice that $T_{X,Y}^\star(X^\star \times Y) \simeq X^\star \times Y$. Just as before, an algebra for that functor is a lens $\ell \in T_{X,Y}^\star(Z) \leftrightarrow Z$ 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

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

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

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 $T_{Unit,X^\star}^\star$-algebra $k : X^\star \times X^\star \leftrightarrow X^\star + X^\star \times X^\star$ where

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

Together with the $T_{Unit,X^\star}^\star$-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} given by $T_X^{Tree}(Z) = Unit + X \times Z \times Z$. 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 $w \in Z \rightarrow \mathbb{N}$ with the property that if $\ell.putl(z,c) = (\text{inr } (x, z_l, z_r), c')$ 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 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 [16] or an *analytic functor* in the terminology of Joyal [22]. (These structures are also related to the *shapely types* of Jay and Cockett [21].)

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

Each container (I, B) gives rise to an endofunctor $F_{I,B}$ on SET whose object part is defined by $F_{I,B}(X) = \sum_{i \in I} B(i) \rightarrow X$. 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 $u \in X \rightarrow Y$ to a function $F_{I,B}(u) \in F_{I,B}(X) \rightarrow F_{I,B}(Y)$ 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 $i \leq j$.
2. B is a functor from $(I, \leq)$ viewed as a category (with one object for each element and an arrow from i to j iff $i \leq j$) into SET. When B and i are understood, we simply write $b|i'$ for $B(i \leq i')(b)$ if $b \in B(i)$ and $i \leq i'$.
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:

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

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

For example, in the case of trees, we can define $t \leq t'$ 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]:

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

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

(Experts will note that C is the limit of the contravariant functor $i \mapsto (B(i) \rightarrow \ell.(C))$. Alternatively, we can construe C as the function space $D \rightarrow \ell.(C)$ 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 $i \leq i'$ and $b' = B(i \leq i')(b)$.)

Proof of well-formedness: To show that this definition is a lens, we should begin by checking that it is well typed—i.e., that the t' we build in $putr$ really lies in the complement (the argument for $putl$ will be symmetric). So suppose that $j \leq j'$ and $b \in B(j)$. There are two cases to consider:

1. $b = b_0|j$ for some (unique) $b_0 \in B(i \wedge j)$. Then $b|j' = b_0|j'$ so we are in the "then" branch in both $t'(j')(b|j')$ and $t'(j)(b)$, and the results are equal by the fact that $t \in C$.
2. b is not of the form $b_0|j$ for some (unique) $b_0 \in B(i \wedge j)$. We claim that then $b|j'$ is not of the form $b_1|j'$ for any $b_1 \in B(i \wedge j')$, so that we are in the "else" branch in both applications of t'. Since $t \in C$, this will conclude the proof of this case. To see the claim, assume for a contradiction that $b|j' = b_1|j'$ for some $b_1 \in B(i \wedge j')$. Applying the pullback property to the situation $i \wedge j \leq j \leq j'$ and $i \wedge j \leq i \wedge j' \leq j'$ yields a unique $b_0 \in B(i \wedge j)$ such that $b = b_0|j$ and $b_1 = b_0|(i \wedge j')$, contradicting the assumption.

It now remains to verify the lens laws. We will check PUTRL; the PUTLR law can be checked similarly. Suppose that

$$F_{I,B}(\ell).putr((i,f),t) = ((i,f_r),t_r)$$
$$F_{I,B}(\ell).putl((i,f_r),t_r) = ((i,f_{rl}),t_{rl})$$

We must check that $f_{rl} = f$ and $t_{rl} = t_r$.

Let us check that $f_{rl} = f$. Choose arbitrary $b \in B(i)$. Then $f_{rl}(b) = \mathsf{fst}(\ell.putl(f_r(b),t_r(i)(b)))$. Inspecting the definition of t_r, we find that $t_r(i)(b) = \mathsf{snd}(\ell.putr(f(b),t(i)(b)))$, and from the definition of f_r, we find that $f_r(b) = \mathsf{fst}(\ell.putr(f(b),t(i)(b)))$. Together, these two facts imply that

$$f_{rl}(b) = \mathsf{fst}(\ell.putl(\ell.putr(f(b),t(i)(b))))$$

Applying PUTRL to ℓ, this reduces to $f_{rl}(b) = f(b)$, as desired.

Finally, we must show that $t_{rl} = t_r$. Choose arbitrary $j \in I$ and $b \in B(j)$. There are two cases: either we have $b_0|j = b$ or not.

- Suppose $b_0|j = b$. Then we find that

$$t_{rl}(j)(b) = \mathsf{snd}(\ell.putl(f_r(b_0|i),t_r(j)(b)))$$

Now, inspecting the definitions of f_r and t_r, we find that this amounts to saying

$$t_{rl}(j)(b) = \mathsf{snd}(\ell.putl(\ell.putr(f(b_0|i),t(j)(b))))$$

Furthermore, we have $t_r(j)(b) = \mathsf{snd}(\ell.putr(f(b_0|i),t(j)(b)))$, so the PUTRL law applied to ℓ tells us that $t_{rl}(j)(b) = t_r(j)(b)$, as desired.

- Otherwise, there is no b_0 with that property. Then we find that $t_{rl}(j)(b) = t_r(j)(b)$ immediately from the definition of t_{rl}. $\square$

Proof of preservation of equivalence: If R witnesses $k \equiv \ell$, then we relate functions that yield related outputs for each possible input:

$$R_{I,B} = \{(t_k,t_\ell) \mid \forall i,b.\ t_k(i)(b)\ R\ t_\ell(i)(b)\}$$

For any i and b, we can show

$$
\begin{aligned}
F_{I,B}(k).missing(i)(b) &= k.missing \\
k.missing\ \ R\ \ \ell.missing \\
\ell.missing &= F_{I,B}(\ell).missing(i)(b)
\end{aligned}
$$

so the *missing* elements are related by $R_{I,B}$. Now suppose the following relationships hold:

$$
\begin{aligned}
t_k\ R_{I,B}\ t_\ell \\
F_{I,B}(k).putr((i,f),t_k) &= ((i,f_k),t_k') \\
F_{I,B}(\ell).putr((i,f),t_\ell) &= ((i,f_\ell),t_\ell')
\end{aligned}
$$

We must show that $f_k = f_\ell$ and that $t_k'\ R_{I,B}\ t_\ell'$. The former follows directly; for any b, we have $f_k(b) = f_\ell(b)$ because $t_k(i)(b)\ R\ t_\ell(i)(b)$. For the latter, consider an arbitrary j and b. There are two cases. If $b_0|j = b$ for some $b_0 \in B(i \wedge j)$, then $t_k'(j)(b)\ R\ t_\ell'(j)(b)$ because k and ℓ preserve R-states; otherwise, $t_k'(j)(b)\ R\ t_\ell'(j)(b)$ because $t_k'(j)(b) = t_k(j)(b)$ and $t_\ell'(j)(b) = t_\ell(j)(b)$. $\square$

Proof of functoriality: The complete relation (which has only one element) witnesses the equivalence $F_{I,B}(id_X) \equiv id_{F_{I,B}(X)}$. The relation

$$\{(t,(t_l,t_r)) \mid \forall i,b.\ t(i)(b) = (t_l(i)(b),t_r(i)(b))\}$$

witnesses the equivalence $F_{I,B}(k;\ell) \equiv F_{I,B}(k);F_{I,B}(\ell)$. $\square$

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 $F_{I,B}(\ell.C)$.

In the literature on containers, the notion of *combinatorial species* further generalizes the container framework by allowing the family $B(i) \to X$ to be quotiented by some equivalence relation; we can obtain multisets in this way, for example. However, we do not see a way to apply this generalization in the case of lenses, because it is then not clear how to match up positions.

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 asymmetric 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.

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

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

(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.

Proof of well-formedness: The CREATEGET law guarantees that $\ell.create \in C$ and the PUTGET law guarantees that $f_x \in C$ for all $x \in X$, so we need merely check the round-trip laws.
PUTRL:

$$
\begin{aligned}
putl(putr(x,c)) &= putl(\ell.get(x),f_x) \\
&= \mathsf{let}\ x' = f_x(\ell.get(x))\ \mathsf{in}\ (x',f_{x'}) \\
&= \mathsf{let}\ x' = \ell.put(\ell.get(x),x)\ \mathsf{in}\ (x',f_{x'}) \\
&= (x,f_x)
\end{aligned}
$$

PUTLR:

$$
\begin{aligned}
putr(putl(y,f)) &= putr(\mathsf{let}\ x = f(y)\ \mathsf{in}\ (x,f_x)) \\
&= putr(f(y),f_{f(y)}) \\
&= (\ell.get(f(y)),f_{f(y)}) \\
&= (y,f_{f(y)})
\end{aligned}
$$

$\square$

9.2 Definition [Asymmetric lenses]: Here are several useful asymmetric lenses (based on string lenses from [7]).

$$copy_X \in X \overset{a}{\leftrightarrow} X$$

$$
\begin{aligned}
get(x) &= x \\
put(x,x') &= x \\
create(x) &= x
\end{aligned}
$$

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

$$
\begin{aligned}
get(x) &= \ell.get(k.get(x)) \\
put(z,x) &= k.put(\ell.put(z, k.get(x)), x) \\
create(z) &= k.create(\ell.create(z))
\end{aligned}
$$

$$\frac{v \in X}{aconst_v \in X \overset{a}{\leftrightarrow} Unit}$$

$$
\begin{aligned}
get(x) &= () \\
put((),x) &= x \\
create(()) &= v
\end{aligned}
$$

$$\frac{k \in X \overset{a}{\leftrightarrow} Y \qquad \ell \in Z \overset{a}{\leftrightarrow} W}{k \cdot \ell \in X \times Z \overset{a}{\leftrightarrow} Y \times W}$$

$$
\begin{aligned}
get(x,z) &= (k.get(x), \ell.get(z)) \\
put((y,w),(x,z)) &= (k.put(y,x), \ell.put(w,z)) \\
create((y,w)) &= (k.create(y), \ell.create(w))
\end{aligned}
$$

$$\frac{k \in X \overset{a}{\leftrightarrow} Y \qquad \ell \in Z \overset{a}{\leftrightarrow} W}{k | \ell \in X + Z \overset{a}{\leftrightarrow} Y \cup W}$$

$$
\begin{aligned}
get(\mathsf{inl}\ x) &= k.get(x) \\
get(\mathsf{inr}\ z) &= \ell.get(z) \\
put(yw, \mathsf{inl}\ x) &= \begin{cases} \mathsf{inl}\ k.put(yw, x) & yw \in Y \\ \mathsf{inr}\ \ell.create(yw) & yw \in W \setminus Y \end{cases} \\
put(yw, \mathsf{inr}\ z) &= \begin{cases} \mathsf{inr}\ \ell.put(yw, z) & yw \in W \\ \mathsf{inl}\ k.create(yw) & yw \in Y \setminus W \end{cases} \\
create(yw) &= \begin{cases} \mathsf{inl}\ k.create(yw) & yw \in Y \\ \mathsf{inr}\ \ell.create(yw) & yw \in W \setminus Y \end{cases}
\end{aligned}
$$

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

$$
\begin{aligned}
get(\langle x_1, \ldots, x_n \rangle) &= \langle \ell.get(x_1), \ldots, \ell.get(x_n) \rangle \\
put(\langle y_1, \ldots, y_m \rangle, \langle x_1, \ldots, x_n \rangle) & \\
&= \langle x_1', \ldots, x_m' \rangle \\
\text{where } x_i' &= \begin{cases} \ell.put(y_i, x_i) & i \le \min(m,n) \\ \ell.create(y_i) & n+1 \le i \end{cases} \\
create(\langle y_1, \ldots, y_n \rangle) &= \langle \ell.create(y_1), \ldots, \ell.create(y_n) \rangle
\end{aligned}
$$

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

$$
\begin{aligned}
copy_X^{sym} &\equiv id_X & (1) \\
(k; \ell)^{sym} &\equiv k^{sym}; \ell^{sym} & (2) \\
aconst_x^{sym} &\equiv term_x & (3) \\
(k \cdot \ell)^{sym} &\equiv k^{sym} \otimes \ell^{sym} & (4) \\
(k | \ell)^{sym} &\equiv case_{k^{sym}, \ell^{sym}}^{f} & (5) \\
(\ell^\star)^{sym} &\equiv map^f(\ell^{sym}) & (6)
\end{aligned}
$$

The first two show that $(-)^{sym}$ is a functor.

Proof: Throughout the proofs, we will use a to refer to the left-hand side of the equivalence, and b to refer to the right-hand side.

1. The complete relation $R \in \{x \mapsto x\} \times Unit$ witnesses the equivalence.
2. The relation

$$R = \{(f_{k\ell}, (f_k, f_\ell)) \mid f_{k\ell} = f_\ell; f_k\}$$

witnesses the equivalence. The fact that $a.missing\ R\ b.missing$ is immediate from the definitions.

Now, to show that $a.putr \sim_R b.putr$, suppose $f_{k\ell}\ R\ (f_k, f_\ell)$. We first compute $a.putr(x, f_{k\ell})$.

$$
\begin{aligned}
a.putr(x, f_{k\ell}) &= ((k;\ell).get(x), z \mapsto (k;\ell).put(z,x)) \\
&= (\ell.get(k.get(x)), \\
&\qquad z \mapsto k.put(\ell.put(z, k.get(x)), x)) \\
&= (x_a, f'_{k\ell})
\end{aligned}
$$

And now $b.putr(x, (f_k, f_\ell))$:

$$
\begin{aligned}
k^{sym}.putr(x, f_k) &= (k.get(x), y \mapsto k.put(y,x)) \\
\ell^{sym}.putr(k.get(x), f_\ell) &= (\ell.get(k.get(x)), \\
&\qquad z \mapsto \ell.put(z, k.get(x))) \\
b.putr(x, (f_k, f_\ell)) &= (x_b, (f'_k, f'_\ell))
\end{aligned}
$$

It's now clear that

$$
\begin{aligned}
f'_k(f'_\ell(z)) &= f'_k(\ell.put(z, k.get(x))) \\
&= k.put(\ell.put(z, k.get(x)), x) \\
&= f'_{k\ell}(z)
\end{aligned}
$$

and that $x_a = x_b$, so $a.putr \sim_R b.putr$.

Finally, to show that $a.putl \sim_R b.putl$, suppose again that $f_{k\ell}\ R\ (f_k, f_\ell)$.

$$
\begin{aligned}
a.putl(z, f_{k\ell}) &= \mathsf{let}\ x = f_{k\ell}(z)\ \mathsf{in} \\
&\qquad (x, z' \mapsto (k;\ell).put(z', x)) \\
&= \mathsf{let}\ x = f_{k\ell}(z)\ \mathsf{in} \\
&\qquad (x, z' \mapsto k.put(\ell.put(z', k.get(x)), x))
\end{aligned}
$$

Similarly,

$$
\begin{aligned}
\ell^{sym}.putl(z, f_\ell) &= \mathsf{let}\ y = f_\ell(z)\ \mathsf{in} \\
&\qquad (y, z' \mapsto \ell.put(z', y)) \\
k^{sym}.putl(f_\ell(z), f_k) &= \mathsf{let}\ x = f_k(f_\ell(z))\ \mathsf{in} \\
&\qquad (x, y' \mapsto k.put(y', x)) \\
b.putl(z, (f_k, f_\ell)) &= (f_k(f_\ell(z)), \\
&\qquad (y' \mapsto k.put(y', f_k(f_\ell(z))), \\
&\qquad z' \mapsto \ell.put(z', f_\ell(z))))
\end{aligned}
$$

Now, we want to show that the first parts of the outputs are equal, that is, that $f_{kl}(z) = f_k(f_\ell(z))$, which is immediate from $f_{kl}\ R\ (f_k, f_\ell)$, and that the second parts of the outputs are related:

$$
\begin{aligned}
f'_k(f'_\ell(z')) &= f'_k(\ell.put(z, f_\ell(z))) \\
&= k.put(\ell.put(z, f_\ell(z)), f_k(f_\ell(z)))
\end{aligned}
$$

Observing that $k.get(f_k(f_\ell(z))) = f_\ell(z)$ because $f_k \in k^{sym}.C$ and that $f_k(f_\ell(z)) = f_{k\ell}(z)$ because $f_{k\ell}\ R\ (f_k, f_\ell)$, that last line becomes

$$
\begin{aligned}
f'_k(f'_\ell(z')) &= k.put(\ell.put(z, k.get(f_{k\ell}(z))), f_{k\ell}(z)) \\
&= f'_{k\ell}(z')
\end{aligned}
$$

so the second parts of the outputs are related after all, and $a.putl \sim_R b.putl$.

3. The relation

$$R = \{(() \mapsto x, x) \mid x \in X\}$$

witnesses the equivalence.

4. The relation

$$R = \{(f_{k\ell}, (f_k, f_\ell)) \mid f_{k\ell}(y, w) = (f_k(y), f_\ell(w))\}$$

witnesses the equivalence.

5. Suppose $k \in X \overset{a}{\leftrightarrow} Y$ and $\ell \in Z \overset{a}{\leftrightarrow} W$. Define the following functions:

$$g \in ((Y \to X) + (W \to Z)) \times (Y \cup W) \to X + Z$$

$$g(\mathsf{inl}\, f_k, yw) = \begin{cases} \mathsf{inl}\, f_k(yw) & yw \in Y \\ \mathsf{inr}\, \ell.create(yw) & yw \in W \setminus Y \end{cases}$$

$$g(\mathsf{inr}\, f_\ell, yw) = \begin{cases} \mathsf{inr}\, f_\ell(yw) & yw \in W \\ \mathsf{inl}\, k.create(yw) & yw \in Y \setminus W \end{cases}$$

$$\mathsf{tag} \in (Y \to X) + (W \to Z) \to Bool$$

$$\mathsf{tag}(\mathsf{inl}\, f_k) = \mathsf{false}$$
$$\mathsf{tag}(\mathsf{inr}\, f_\ell) = \mathsf{true}$$

Then we can define the relation

$$R = \{(g(f), (f, \mathsf{tag}(f))) \mid f \in (k^{sym} \oplus^f \ell^{sym}).C\}.$$

It is tedious but straightforward to verify that this witnesses the equivalence.

6. $(\ell^*)^{sym}.C$ comprises functions $f : Y^* \to X^*$ such that $f([y_1, \ldots, y_n]) = [x_1, \ldots, x_m]$ implies $m = n$ and $\ell.get(x_i) = y_i$.
The complement $map^f(\ell^{sym}).C$ on the other hand comprises lists of functions $[f_1, \ldots, f_n]$ where $f_i : Y \to X$ and $\ell.get(f_i(y)) = y$. Relate two such complements f and $[f_1, \ldots, f_n]$ if $f([y_1, \ldots, y_m]) = [x_1, \ldots, x_m]$ implies $x_i = f_i(y_i)$ when $i \leq n$ and $x_i = \ell.create(y_i)$ otherwise.
Clearly, the two "missings" are thus related and it is also easy to see that $putr$ is respected. As for the $putl$ direction consider that f and $[f_1, \ldots, f_n]$ are related and that $ys = [y_1, \ldots, y_m]$ is do be $putl$-ed. Let $[x_1, \ldots, x_k]$ be the result in the $(f^*)^{sym}$ direction. It follows $k = m$ and $[x_1, \ldots, x_m] = f([y_1, \ldots, y_m])$. If $[x'_1, \ldots, x'_m]$ is the result in the $map^f(\ell^{sym})$ direction then $x'_i = f_i(y_i)$ if $i \leq n$ and $x'_i = \ell.create(y_i)$ otherwise. Now $x_i = x'_i$ follows by relatedness.
The new $(\ell^*)^{sym}$ complement then is $\lambda ys.(\ell^*).put(ys, xs)$. The new $map^f(\ell^{sym})$ complement is $[g_1, \ldots, g_m]$ where $g_i(y) = \ell.put(x_i, y)$. These are clearly related again. $\quad\square$

The $(-)^{sym}$ 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 ℓ of LENS, there are asymmetric lenses k_1, k_2 such that

$$(k_1^{sym})^{op} ; k_2^{sym} = \ell.$$

This suggests that the category LENS could be constructed from spans in ALENS. We choose not to try this because it makes many things more awkward. While it is possible to define composition of spans with a pullback construction, it would—as in our account—not be associative unless some equivalence would be imposed. However, in the span presentation there does not seem to be a natural and easy-to-use candidate for such an equivalence. Of course, going back-and-forth via symmetric lenses does induce such an equivalence.

To see this, we need to know how to "asymmetrize" a symmetric lens.

9.5 Definition: 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:

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

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

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

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

Proof of well-formedness: We show only that ℓ_r^{asym} is well-formed; the proof for ℓ_l^{asym} is similar.
GETPUT:

$$
\begin{aligned}
put(get((x, y, c)), (x, y, c)) &= put(x, (x, y, c)) \\
&= \mathsf{let}\,(y', c') = \ell.putr(x, c) \\
&\quad\ \mathsf{in}\,(x, y', c') \\
&= (x, y, c)
\end{aligned}
$$

The final equality is justified because (x, y, c) is a consistent triple.
PUTGET:

$$
\begin{aligned}
get(put(x', (x, y, c))) &= \mathsf{let}\,(y', c') = \ell.putr(x', c) \\
&\quad\ \mathsf{in}\,get((x', y', c')) \\
&= x'
\end{aligned}
$$

CREATEGET:

$$
\begin{aligned}
get(create(x)) &= \mathsf{let}\,(y, c) = \ell.putr(x, \ell.missing) \\
&\quad\ \mathsf{in}\,get((x, y, c)) \\
&= x
\end{aligned}
$$

In addition to the three round-trip laws, we must show that put and $create$ yield consistent triples. But this is clear: the PUTR2 law is exactly what we need. $\quad\square$

Proof of 9.4: Given arrow $[\ell]$, choose $k_1 = \ell_r^{asym}$ and $k_2 = \ell_l^{asym}$. Writing ℓ_r for $((\ell_r^{asym})^{sym})^{op}$ and ℓ_l for $(\ell_l^{asym})^{sym}$, we then need to show that $\ell_r; \ell_l \equiv \ell$. Define two functions:

$$
\begin{aligned}
f_c(x) &= \mathsf{let}\,(y, c') = \ell.putr(x, c)\,\mathsf{in}\,(x, y, c') \\
g_c(y) &= \mathsf{let}\,(x, c') = \ell.putl(y, c)\,\mathsf{in}\,(x, y, c')
\end{aligned}
$$

Then the relation $R = \{((f_c, g_c), c) \mid c \in C\}$ witnesses the equivalence. We can check the definitions to discover that

$$\ell_r.missing = \ell_r^{asym}.create = f_{\ell.missing}$$

$$\ell_l.missing = \ell_l^{asym}.create = g_{\ell.missing}$$

and hence that $(\ell_r; \ell_l).missing\ R\ \ell.missing$.

We also need to show that $(\ell_r; \ell_l).putr$ and $\ell.putr$ are well-behaved with respect to R. Suppose $\ell.putr(x, c) = (y, c')$; then we need to show that $(\ell_r; \ell_l).putr(x, (f_c, g_c)) = (y, (f_{c'}, g_{c'}))$. First we compute $\ell_r.putr(x, f_c)$:

$$
\begin{aligned}
\ell_r.putr(x, f_c) &= ((\ell_r^{asym})^{sym})^{op}.putr(x, f_c) \\
&= (\ell_r^{asym})^{sym}.putl(x, f_c) \\
&= \text{let } t = f_c(x) \text{ in } (t, x' \mapsto \ell_r^{asym}.put(x', t)) \\
&= \text{let } (y, c') = \ell.putr(x, c) \text{ in} \\
&\quad ((x, y, c'), x' \mapsto \ell_r^{asym}.put(x', (x, y, c'))) \\
&= ((x, y, c'), x' \mapsto \ell_r^{asym}.put(x', (x, y, c'))) \\
&= ((x, y, c'), f_{c'})
\end{aligned}
$$

We then compute $\ell_l.putr((x, y, c'), g_c)$:

$$
\begin{aligned}
\ell_l.putr((x, y, c'), g_c) &= (\ell_l^{asym})^{sym}.putr((x, y, c'), g_c) \\
&= (\ell_l^{asym}.get((x, y, c')), \\
&\quad y' \mapsto \ell_l^{asym}.put(y', (x, y, c'))) \\
&= (y, y' \mapsto \ell_l^{asym}.put(y', (x, y, c'))) \\
&= (y, g_{c'})
\end{aligned}
$$

We conclude from this that $(\ell_r; \ell_l).putr(x, (f_c, g_c)) = (y, (f_{c'}, g_{c'}))$ as desired.

The argument that $(\ell_r; \ell_l).putl$ and $\ell.putl$ are well-behaved with respect to R is almost identical. □

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, 15]. Connections to the literature on *view update* in databases are surveyed in [12].

The first symmetric approach to update propagation was proposed by Meertens [26] and followed up in the context of model-driven design by Stevens [29] and Diskin [11]. 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 choices of behavioral laws have been explored. Some of these are strictly 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.[25] 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. [12] and the more refined variants in [14].

A different foundation for defining lenses by recursion was explored by Foster et al. [12], 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:

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

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

$$\ell.putr \in A \to B$$
$$\ell.putl \in B \to A$$

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.

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 update. 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. Algebraic models for bidirectional model synchronization. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS*, pages 21–36. Springer, 2008.

[10] 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.

[11] 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.

[12] 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.

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

[14] J. Nathan Foster, Benjamin C. Pierce, and Steve Zdancewic. Updatable security views. In *IEEE Computer Security Foundations Symposium (CSF), Port Jefferson, NY*, July 2009.

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

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

[17] 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.

[18] S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Towards a compositional approach to model transformation for software development. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 468–475. ACM New York, NY, USA, 2009.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

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

[33] 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.