Contents

Chapter 1

Introduction

Recent years have seen increased interest in the area of bidirectional programming. Broadly speaking, the problem domain involves maintaining a connection between two different representations of otherwise very similar information. The strong connections between an in-memory representation of a data structure and its serialized form; a piece of source code and its parsed abstract syntax tree [36]; tool-specific configuration formats and a common configuration format [28]; a database and some particular summary of interest; or two distant but partially replicated computers [35] are all examples of areas where two pieces of data are very similar. We will call each of the two objects in these pairings *repositories*. In each case, one would like the two repositories to stay "in synch": modifications to one repository should be propagated and reflected in the other.

At the moment, one common way of tackling this problem is to design by hand two programs that work together. Calling the two repositories X and Y, the first program translates updates to X into updates to Y, and the second translates in the other direction, turning updates to Y into updates to X. (Taking the example of connecting a piece of source code and its abstract syntax tree from above, these two programs might be a parser and a pretty-printer.) Programming in this style, however, quickly grows unmanageable. Recent developments in bidirectional transformations have suggested that a language-based approach—that is, the creation of a language where each term represents two transformations—may be more practical in many ways. Existing languages have a uniform interface across terms: different programs are run in the same manner. This means that such bidirectional programs are easy to extend to accommodate the evolution of the data structures being connected. Moreover, the language itself can provide evidence that the transformations are correct, for example, by guaranteeing that any transformation that can be constructed within the language will restore synchrony on each run, will not discard too much information, will not disrupt synchrony unnecessarily, or similar behavioral guarantees.[1] Designing a language is also a more modular approach to solving the

[1] We will use "synchronize" and related words informally to mean simply "maintain a correspon-

	Alignment	Symmetry	Performance	Syntax
group-based lenses	✓			
symmetric delta lenses	✓	✓		
symmetric lenses		✓		✓
asymmetric delta lenses	✓			✓
matching lenses	✓			✓
hole-based delta lenses	✓			✓
constraint maintainers	✓	✓		✓
edit lenses	✓	✓	✓	✓

Table 1.1: Feature coverage for various approaches to bidirectional programming

bidirectional transformation problem, as the design of bidirectional building blocks can be separated from the process of gluing the blocks together into particular useful transformations.

The term "lens" is a broad term encompassing a large family of related language-based approaches to the bidirectional transformation problem. In §1.1, we will introduce one of the earliest language-based approaches, asymmetric lenses, as a way to ground our ongoing discussion of the features that make bidirectional programming attractive and practical. We will identify four key challenges in lens design: alignment (the ability to represent data evolution precisely, §1.2), symmetry (no more restrictions on one piece of connected data than on the other, §1.3), performance (handling data in an incremental way, §1.4), and syntax (the existence of example transformations, §1.5). Table 1.1 shows how other approaches stack up on these four key features. We will discuss this figure in depth in Chapter 4; for now, it suffices to observe that the edit lens framework as described in this document is the first framework to support all four features.

1.1 Asymmetric Lenses

One well-studied approach to bidirectional programming is the framework of *asymmetric*, *state-based* lenses. A thorough review of this work is available elsewhere [17], so we will give only a brief introduction to the core concepts. Suppose we have two repositories; one repository stores a piece of data represented by an element of the set S, and the other stores an element of V. Then a lens connecting the two repositories

dence between two repositories by propagating edits in both directions". A full-blown synchronization tool would also include, at a minimum, some mechanism for dealing with conflicts between disconnected edits to the two structures, which is outside the scope of this document. Note, though, that we will go beyond most existing synchronization tools in allowing the repositories to be structured differently and to share only a part of their information.

has three components:

$$\begin{aligned} \mathit{get} &\in S \to V \\ \mathit{put} &\in V \times S \to S \\ \mathit{create} &\in V \to S \end{aligned}$$

In this model, the V repository is a *view* of or *query* on the S repository (called a *source*): that is, it can be completely reconstructed from the other without additional outside information. The type of the *get* component of the lens reflects this assumption. In most cases, a query will keep only some of the information available in the source; as a result, the opposite reconstruction property—that the source can be completely reconstructed from the view—usually does not hold. Asymmetric, state-based lenses handle this situation by allowing their other major function component to have access to both a modified value from the view repository *and* an original value from the source repository to merge the new data into, as reflected in the type of *put*. As a technical detail, it is sometimes convenient to demand (and rarely difficult to supply) a way to generate a value in the source repository with some sane defaults. This is the *create* component of the lens.

Lenses have one more piece, which was alluded to above. The structure we have described so far already addresses the need to give two transformations (namely *get* and *put*), but does not yet address our desire to prove that these two transformations work well together. Let us first try to build an intuition for what "works well together" might mean before we formalize this. Suppose we have a lens ℓ; to simplify things, we will take $\ell.\mathit{get}$ to be unassailable[2] and phrase all our desires in terms of constraints on $\ell.\mathit{put}$. It is natural to expect two things from our lens: first, that $\ell.\mathit{put}$ changes enough—that whatever change we make to the view is faithfully reflected in the source so that future calls to $\ell.\mathit{get}$ give exactly the value we changed the view to—and second, that $\ell.\mathit{put}$ does not change too much—that only the parts of the source that are used to compute the view are modified. Three behavioral laws address this intuition:

$$\begin{aligned} \mathit{put}(\mathit{get}(s), s) &= s && \textsc{GetPut} \\ \mathit{get}(\mathit{put}(v, s)) &= v && \textsc{PutGet} \\ \mathit{get}(\mathit{create}(v)) &= v && \textsc{CreateGet} \end{aligned}$$

The PutGet law formalizes the expectation that $\ell.\mathit{put}$ changes enough; the GetPut law takes a step toward formalizing the expectation that $\ell.\mathit{put}$ does not change too much.[3] There is a third law, CreateGet, which serves a similar purpose to the

[2] We will use record notation for lens components, so that $\ell.\mathit{get}$ is the *get* component of ℓ.

[3] To be precise, it actually only guarantees that unmodified views result in unmodified sources. It is very hard to come up with a better generic guarantee than this within the asymmetric, state-based lens framework, but Chapter 3 takes a step towards better guarantees.

PUTGET law. Collectively, these behavioral laws are often also called *roundtrip laws*: another way to read them is that in a given "round trip" through the lens the repository returns to exactly the same state it started at. In the remainder, we will write $\ell \in S \overset{a}{\leftrightarrow} V$ to assert that ℓ is an asymmetric, state-based lens—that is, that it is a triple of functions whose types are as above that satisfy the three behavioral laws discussed.

In the remaining sections, we explore some scenarios which are challenges for the asymmetric, state-based lenses described here.

1.2 Alignment

One very common operation when doing functional (unidirectional) programming is the map operation, which runs a computation on each element of a list. To give an idea of how common, as of April 3, 2012, there were 3878 packages on Hackage [40], the central code repository for Haskell projects, which made a total of 90,040 calls to map—an average of more than twenty calls per project.[4] Most serious attempts at designing a bidirectional language therefore provide some variant of a mapping operation. Since it is such a popular operation, it is important to do a really good job of designing the bidirectional map, and that job turns out to be surprisingly difficult! To see why, let us implement map in the most obvious way; then we can discuss the deficiencies of this approach.

Like the unidirectional map, which is parameterized by a unidirectional function to apply to list elements, our bidirectional map will be parameterized by a bidirectional operation. That is, writing $S^\star$ for the set of lists with elements drawn from S, when $\ell \in S \overset{a}{\leftrightarrow} V$, we will have $\mathsf{map}(\ell) \in S^\star \overset{a}{\leftrightarrow} V^\star$. Figures 1.1 and 1.2 define the map lens. In these figures, and in the remainder of the text, we distinguish the unidirectional mapping operation map_U from the bidirectional map. The *get* and *create* operations are fairly straightforward—direct analogues of the unidirectional version—but the *put* operation is more delicate. Since *put* takes one value from each repository, our $\mathsf{map}(\ell).put$ operation takes two lists, of types $S^\star$ and $V^\star$. When these lists are the same length we can just zip them together with $\ell.put$. When they are different lengths, there have been insertions or deletions. Deletions can be reflected directly by deleting the last few elements of the $S^\star$ list until the lengths match. For insertions, we recover elements of S by treating the last few elements of the $V^\star$ list as the insertions and using *create* to fabricate S elements to insert.

Figure 1.3 defines a lens *lower* that converts a character to lower case so that we

[4] In fact, the program used to calculate these numbers itself makes two calls to map:

```
ack -cl '\bmap\b' | cut -d: -f2 |
ghc -e 'interact $ unlines .  map show .  scanl (+) 0 .  map read .  lines'
```

$$\mathsf{map}_U(f,t) = \begin{cases} \langle\rangle & t = \langle\rangle \\ f(x)\text{:}\mathsf{map}_U(f,t') & t = x\text{:}t' \end{cases}$$

$$\mathsf{zip}(f,g,h,t,u) = \begin{cases} f(x,y)\text{:}\mathsf{zip}(f,g,h,t',u') & t = x:t' \wedge u = y:u' \\ g(t) & t = x:t' \wedge u = \langle\rangle \\ h(u) & t = \langle\rangle \wedge u = y:u' \\ \langle\rangle & t = u = \langle\rangle \end{cases}$$

$$\mathsf{const}(x) = \lambda y.\ x$$

Figure 1.1: Auxiliary unidirectional functions used in the definition of map

$$\begin{aligned} \mathsf{map}(\ell).get(t) &= \mathsf{map}_U(\ell.get, t) \\ \mathsf{map}(\ell).create(u) &= \mathsf{map}_U(\ell.create, u) \\ \mathsf{map}(\ell).put(t,u) &= \mathsf{zip}(\ell.put, \mathsf{map}_U(\ell.create), \mathsf{const}(\langle\rangle), t, u) \end{aligned}$$

Figure 1.2: A naive implementation of the bidirectional map operation

$$\begin{aligned} lower.get(c) &= \text{the lower case version of } c \\ lower.put(c',c) &= \begin{cases} \text{the upper case version of } c' & \texttt{A} < c < \texttt{Z} \\ c' & \text{otherwise} \end{cases} \\ lower.create(c) &= c \end{aligned}$$

Figure 1.3: The *lower* lens to convert a possibly-upper-case letter into a definitely-lower-case one

can demonstrate the behavior of map.

$$\begin{aligned}
\mathsf{map}(\mathit{lower}).\mathit{get}(\texttt{UpperCasedQord}) &= \texttt{uppercasedqord} \\
\mathsf{map}(\mathit{lower}).\mathit{put}(\texttt{uppercasedword}, \texttt{UpperCasedQord}) &= \texttt{UpperCasedWord} \\
\mathsf{map}(\mathit{lower}).\mathit{put}(\texttt{uppercased}, \texttt{UpperCasedWord}) &= \texttt{UpperCased} \\
\mathsf{map}(\mathit{lower}).\mathit{put}(\texttt{uppercasedsentence}, \texttt{UpperCasedWord}) &= \texttt{UpperCasedSentence}
\end{aligned}$$

All of these examples behave essentially optimally. However, not all is well; a simple example of the so-called *alignment problem* is something like this, where we have an insertion in the middle of the word to correct the spelling of "upper":

$$\mathsf{map}(\mathit{lower}).\mathit{put}(\texttt{uppercasedword}, \texttt{UperCasedWord}) = \texttt{uppeRcaseDword}$$

Because $\mathsf{map}(\mathit{lower}).\mathit{put}$ only looks at a lower-cased element's position when deciding which mixed-case character to match it up with, we have incorrectly *aligned* the new view with the old source this way:

```
U p e r C a s e d W o r d
| | | | | | | | | | | | |
u p p e r c a s e d w o r d
```

A better alignment would look like this:

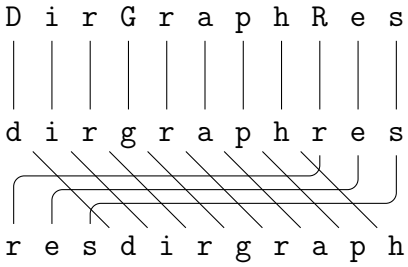
```
U p e r C a s e d W o r d
| |  \ \ \ \ \ \ \ \ \ \ \
u p p e r c a s e d w o r d
```

One natural reaction to this infelicity is to think of the `diff` algorithm [24] or something similar. This idea has been developed quite far [6]; let us see how. At first blush, it seems difficult to use the `diff` algorithm directly. Elements of the source and view have different types, so it is not clear how to compare them.[5] However, the alignment diagram above may be broken into two stages:

```
U p e r C a s e d W o r d          old source (of type S*)
| | | | | | | | | | | | |
u p e r c a s e d w o r d          old view (of type V*)
| |  \ \ \ \ \ \ \ \ \ \ \
u p p e r c a s e d w o r d          new view (of type V*)
```

[5]One might be tempted to use `diff` anyway, or to use a case-insensitive `diff`. In the general case, the elements of the source and view lists are very different kinds of objects, so that kind of trick does not scale well.

In this restructured alignment diagram, the upper alignment (which connects elements of different types) will always be completely flat, and hence requires no sophisticated tools to generate. In contrast, the lower alignment contains all the interesting information, and is the one we hope to compute with `diff`. Moreover, the connections in the lower alignment are now between elements of the same type, making the use of `diff` much more plausible. One wrinkle is that the data in this example is unrealistically simple, and more complicated data often needs more complicated tools for specifying the cost function that `diff` uses. Another wrinkle is that in real-world situations, one often wants to discover alignments with crossings like



which require more sophisticated algorithms than the traditional `diff`.

Indeed, it is not even clear that it is always possible to correctly guess the alignment given just an old and a new copy of the data. Figure 1.4 gives an example of a particularly tricky situation involving a school's employee database. Part a shows the full database, which includes a listing of all the teachers and their salaries. It transpires that the school secretary finds it useful to have access to this database; however, the secretary should not be privy to the confidential salary information. Consequently there is a secretarial view, shown in Part b, with salaries redacted, and we would like to keep the database and view synchronized using a `map` lens. Now, suppose one of two scenarios happens:

- Mary Jones gets married and changes her name to Mary Smith.
- Mary Jones retires, and the school hires a replacement who, by coincidence, shares her first name: Mary Smith.

In both cases, when the secretary updates her document, it will look as it does in part c. As shown in part d, there are really two feasible alignments, corresponding to whether the dotted edge should be present or not. In the first scenario above, the edge should be present: we should align Mary with her former self, and reflect the change as an update to her name (but keep her old salary). In contrast, in the second scenario, the edge should not be present: we should not align the new Mary with any of the teachers that used to teach at the school. Since only the old and new copies of the secretary's document are available to a lens, the lens cannot choose correctly. The context under which the change was made is invisible to the lens, and it has no way to distinguish between these two scenarios merely by observing what has changed.

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

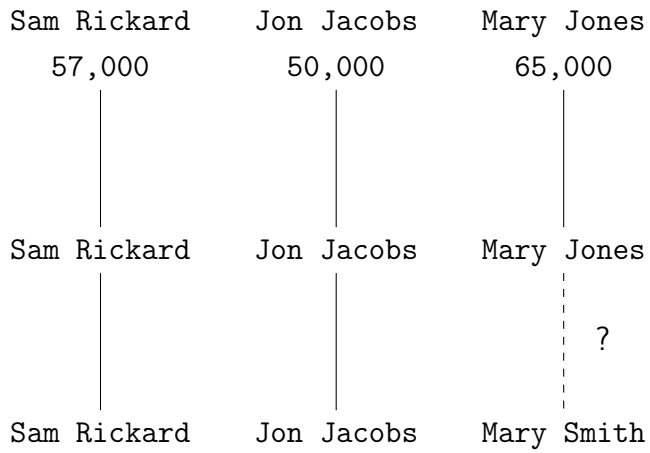
(a) HR's view

Teacher name
Sam Rickard
Jon Jacobs
Mary Jones

(b) A secretary's view

Teacher name
Sam Rickard
Jon Jacobs
Mary Smith

(c) After an update



(d) Whether the marked edge should be included or not depends on invisible context

Figure 1.4: A school's staff list, as seen by HR and by the principal's secretary

Clearly, discovering alignment information is a tricky business. Additionally, most current lens frameworks treat such alignment information as a second-class citizen: it is not passed, stored, or returned by the lens. Because of this, it is not possible for an outside tool to provide hints about the alignment; the implementation of alignment discovery is intermingled with the implementation of alignment usage and propagation inside each lens' definition; and alignment information cannot be internally communicated between lens components. The conclusion we must draw is that doing a really good job of implementing map involves rethinking some or all of the theoretical foundations of lenses to address the representation, propagation, and use of alignment information.

1.3 Symmetry

Let us turn our attention to a second fundamental challenge in lens design: symmetry. The asymmetric lenses discussed above assume that one repository is a view of the other. In the following, we will discuss two bidirectional scenarios, one that highlights the need to relax this assumption, and one that identifies a useful feature of asymmetric lenses that has long been thought incompatible with symmetry.

Continuing the example from 1.2, suppose our school secretary decided to begin tracking which room each teacher taught in. The two lower tables in Figure 1.5 shows how the two repositories might look after this schema change. As before, the salary information should be hidden from the secretary for privacy reasons; on the other hand, in our new scenario the human resources department is not interested in room assignments. Unfortunately, this slight modification puts our scenario firmly outside the realm of problems that asymmetric, state-based lens tools can help with: neither repository can be completely reconstructed just from the information available in the other.

Since the problem is that neither repository contains all the information, one thing that can be done is to design a third repository that *does* contain all the information. One would then design two lenses with that third repository as a common source, as shown in the remainder of Figure 1.5. The new repository sits at the top, and contains teacher names, salaries, and room assignments all in one location. The two repositories we are really interested in sit below, and are derived via two lenses. (We introduce the notation $\pi_{i_1,\ldots,i_n}$ for the lens which projects out distinct fields $i_1, \ldots, i_n$ of a tuple. To be really precise, each omitted field would need an additional annotation giving a value to return from the *create* operation, but we elide these annotations to avoid clutter.)

Suppose the secretary updates the room assignments document. The process to find a corresponding update for the salary document involves two lens operations: first, we use $\mathsf{map}(\pi_{1,3}).put$ to update the common source, then $\mathsf{map}(\pi_{1,2}).get$ to regenerate the salary document from the common source.

This approach is workable, and is fairly comprehensive. However, it is a little

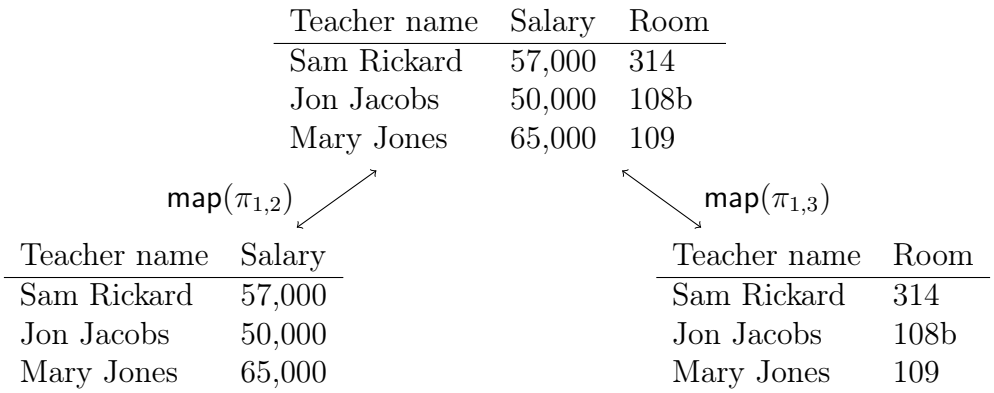
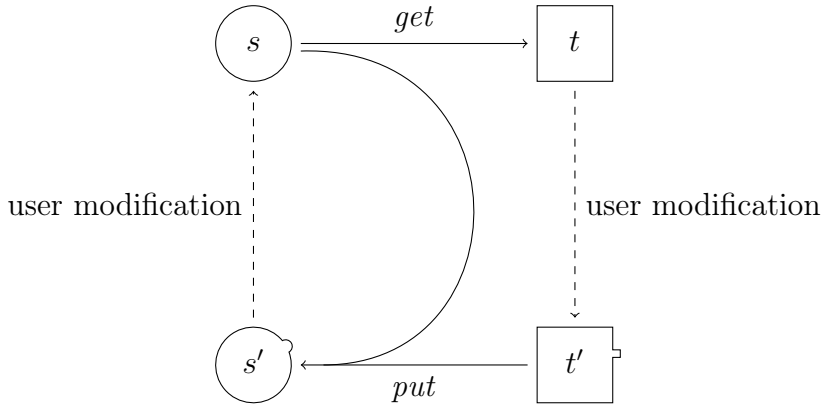


Figure 1.5: A slightly more complicated synchronization scenario

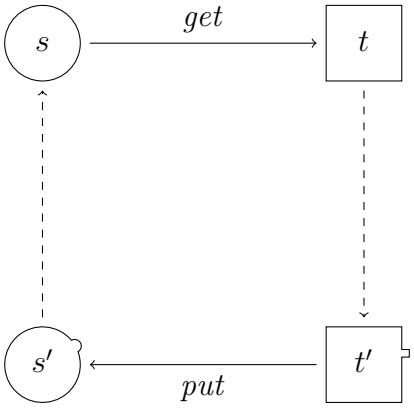
bit awkward in a few ways, the most notable of which is that we are now constructing two lenses. Even in this simple example, we can see that the structure of the lenses are very similar. All of the arguments which led people to prefer bidirectional languages over pairs of unidirectional programs in the first place—uniformity, guaranteed correctness, maintainability, modularity, etc.—arise here against writing pairs of bidirectional programs, too. It would be better to develop some theory which models the two operations together, so that we can write a single program and derive the two synchronization operations of interest. One could continue by designing a "bi-bidirectional" language—where each term could be interpreted as two lenses which are intended to be run back-to-back as in this example—but we choose instead to reconsider the foundations of lens theory and design a framework of symmetric bidirectional transformations that natively handles symmetric scenarios.

Let us address ourselves to what makes asymmetric lenses asymmetric in the first place. Figure 1.6a shows the typical life-cycle of an asymmetric lens $\ell \in S \overset{a}{\leftrightarrow} T$, ignoring *create* for the moment. Drawing the types of the *get* and *put* operations this way highlights their asymmetry, and quickly suggests two ways of symmetrizing the theory. Parts b and c illustrate these two ways, namely, removing the extra arc in the type of *put*, or adding an extra arc to the type of *get*. Together with some appropriate roundtrip laws, the former are known as isomorphisms, and several languages whose terms represent invertible functions in this way have been designed [9, 36]. They are especially useful as a formalism when the extra information available in the repositories is unimportant. For example, when parsing text, the exact whitespace used may not be available in the abstract syntax tree, but often a few simple rules will produce very similar replacement whitespace; and moreover the whitespace has aesthetic but not semantic significance. In the example given above, however, the extra information *is* important, and cannot be replaced with default data: resetting room assignments and salaries on each roundtrip would be very undesirable behavior.

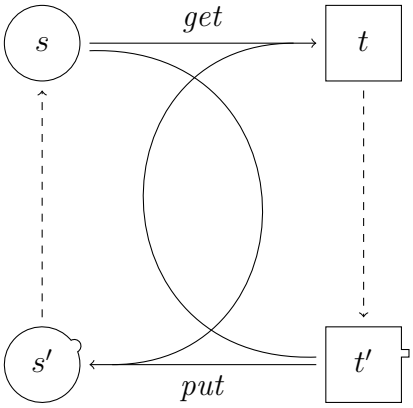
The latter (again with some appropriate roundtrip laws) are known as constraint



(a) Asymmetric, state-based lenses



(b) (Partial) isomorphisms



(c) Constraint maintainers

Figure 1.6: Asymmetric lens life cycle, and some proposed symmetric variants

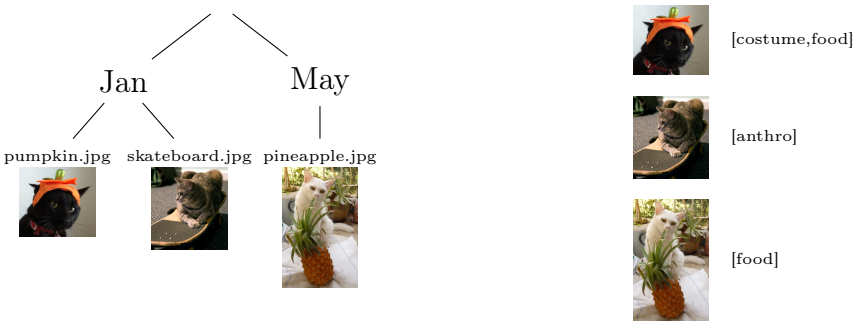


Figure 1.7: A whimsical symmetric synchronization scenario

maintainers [30], and do handle extra information quite explicitly. Constraint maintainers would be a good formalism to use when designing a bidirectional transformation for the school scenario above. They can express the connection between salaries and room numbers—that is, no connection at all—well, and support a map-like combinator to turn this single-record maintainer into one which handles lists of records like the ones stored in the repositories. However, constraint maintainers do not support *sequential composition*, the ability to run one maintainer after the other, and experience with asymmetric lenses shows that this is a very common tool when designing bidirectional programs. To see why, we will introduce a bidirectional transformation which is most naturally modeled using composition.

The whimsical situation shown in Figure 1.7 involves a web server, which must keep a file system storing pictures of cats synchronized with a user-modifiable web page (modeled here as a list of cat pictures with descriptive tags).[6] One natural approach to implementing this transformation is pictured in Figure 1.8. First, we separately implement two constraint maintainers: a *flatten* maintainer that flattens trees to lists by extracting the leaves, and a *relabel* maintainer that describes the connection between a single leaf in our original tree and a single list entry in our final list. We would then like to run these maintainers back-to-back; that is, we would like a sequential composition operator $-;-$ with a typing rule like:

$$\frac{k \in A \overset{c}{\leftrightarrow} B \qquad \ell \in B \overset{c}{\leftrightarrow} C}{k;\ell \in A \overset{c}{\leftrightarrow} C}$$

Unfortunately, implementing this combinator is not possible: we must design the $(k;\ell).get \in A \times C \to C$ component using the components $k.get$, $k.put$, $\ell.get$, and $\ell.put$, all of which require a B as input. It is certainly possible to build a constraint maintainer which has the desired behavior wholesale, but this involves writing both constraint maintenance functions and proving that they are consistent with each other—the exact task we set out to avoid by designing a language. Alternately,

[6] Pictures used with permission [4, 10, 42].

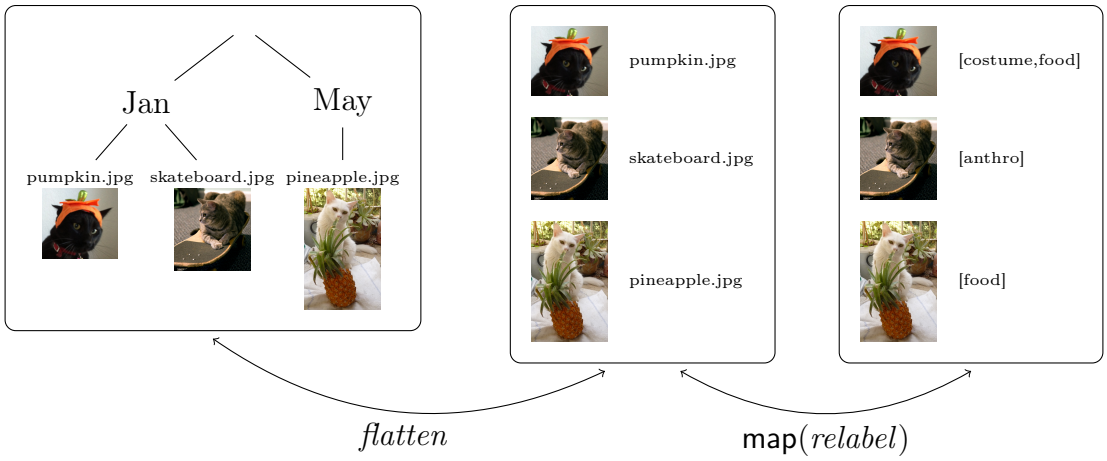


Figure 1.8: Adding an intermediate structure can improve modularity

one can step a little bit outside the constraint maintainer framework by keeping a copy of the "intermediate" repository around somewhere and running the constraint maintainers in sequence on each update. Making this choice, however, leads one to immediately ask how to model such maintainer chains and what behavioral guarantees one can expect!

So an ideal a model would capture the behavior of "sequencing", retain a symmetric presentation, and allow each repository to retain information not available in the other.

1.4 Performance

Real-world synchronization tools inevitably address a third concern: performance. Typical repositories are large objects; consequently, there can be significant time or memory costs associated with processing the data in a repository or transmitting a repository across a network. For existing file system synchronization tools like rsync, DropBox, and Unison and revision control systems like CVS, SVN, darcs, mercurial, and git [7, 11, 14, 18, 34, 35, 37, 39], network speed is a significant bottleneck. For these tools, where little computation on the repositories themselves is required, the relatively simple *delta compression* technique, which involves noting what has changed since a previous run of the tool, provides a serious network transmission performance boost. For bidirectional transformations, where repositories must be not just copied but transformed, computation time or memory usage may also be concerns. Extending the use of delta compression to address processing speed and memory requirements involves, in part, showing that the computations of interest can be performed solely by inspecting the deltas—that is, without decompressing and traversing the original repositories. Since it seems likely that a practical tool

will need to avoid incurring high resource usage, a theory that faithfully models a successful tool should therefore model not just repository states but also repository edits, edit transformations, and the connection between edits and repository states.

1.5 Syntax

One of the outstanding features of the body of asymmetric, state-based lens work and its closest variants is the devotion to retaining a rich syntax—that is, a large collection of particular lenses and lens combinators that have the appropriate shape and behavior for the given lens framework. This is a feature well worth emulating, for several reasons. The simplest stems from the variety of examples given in previous sections. Even in seemingly simple scenarios, there is often endless variation. Instead of designing a synchronization tool that addresses one of these scenarios, we set our sights higher: we wish to design a synchronization-tool-making tool that makes it easy to address any of the scenarios. Thus we want to find a collection of basic building blocks and ways of combining those blocks that can be used together to customize the bidirectional transformation for many different use cases.

Additionally, designing a syntax in parallel with the language semantics is a valuable cross-validation technique. On several occasions during the development of the framework described in this document, we found a desirable transformation which could not be implemented within the type or behavioral guarantees of our framework. Each time this happens, one then has a valuable opportunity to reevaluate both the lens framework and the transformation. A particularly good example of this, which we will discuss in greater depth in Chapter 2, is the transformation which duplicates information.[7] Many lens frameworks rule this transformation illegal (including ours), because supporting it involves relaxing the restriction that a single run of the synchronization tool produces a synchronized state. Whether one prioritizes behavioral guarantees like single-pass synchronization or richer syntax like duplication lenses may be a matter of taste; but the choice would not be readily apparent without attempting to design the syntax in parallel with the semantics.

Finally, syntax is a proving ground for the practicality of other features. The ultimate goal of a lens theory is to be an integral part of a widely-used tool, and designing a collection of instantiations is a critical first step on that path. A good lens framework can have the potential to solve alignment, symmetry, or performance problems, and attempting to design a syntax can quickly realize or dispel that potential.

Experience from the asymmetric, state-based lens work shows that supporting the majority of synchronization scenarios requires only a handful of lenses. The most basic lenses simply copy, insert, or delete data. Modular developments make

[7] For an example where such a transformation would be useful, imagine the process of turning wiki markup into an HTML page that includes both a table of contents and a content body. One might structure this as duplicating the markup, then extracting the section titles from one duplicate and rendering the other.

heavy use of sequential composition for running two lenses one after the other (as discussed in §1.3). One also often wants operations on sums and products such as parallel composition, projection, injection, and conditionals. Some support for lists (especially a mapping operation) and other structured data (typically inductive types with fold and unfold operations) rounds out a fairly complete set of operations for a practical framework to support.

1.6 Contributions

This document espouses a foundational effort to rebuild the lens formalism with the above challenges in mind from the beginning. Many previous efforts to address these challenges have begun with asymmetric, state-based lenses as a base and built additional capabilities on top. Adding to the existing theory in this way is quite useful, but a system which attempts to combine these additional features quickly grows baroque. By starting from first principles, we have a unique opportunity to address these concerns in the base theory in a new way. We focus on semantics first (modeling the core behaviors that lenses must allow primitively) and let the syntax (that is, what transformations are possible with the core building blocks) fall subordinate. As we hoped, our commitment to this approach has resulted in an elegant core lens theory which nevertheless has the ability to address each of the four challenges discussed above.

Chapter 2 develops the machinery needed for a symmetric lens theory in isolation from the issues of alignment and performance. The key observation is that we can think of the two transformations in a lens as sharing some state that is independent of the two repositories. We will show that all the usual lens combinators can be construed as pairs of stateful transformations. However, there is a price to pay for symmetry: though the usual transformations are available, they do not have all the same nice properties one expects from the asymmetric world; for example, lens composition is not directly associative. As a result, the machinery developed includes a notion of *lens equivalence*; most properties (including associativity of composition) then hold, but only up to lens equivalence.

In Chapter 3, these observations about how to achieve symmetry will be used as the basis for a system that tackles the alignment and performance problems (while retaining symmetry). Because the exact nature of alignment information is so different between data structures—and even between different transformations on the same structure—the framework proposed in this chapter will treat such information as almost completely abstract. It then becomes the responsibility of each lens definition to specify what information it expects to receive. We then go on to again implement many of the usual lens combinators, and show that many of them are capable of disambiguating between edits that are traditionally the source of serious alignment headaches. Additionally, we observe that the natural way of implementing these lenses results in a lens which operates on relatively small descriptions of what has

changed rather than on large repositories, which addresses some of the performance issues raised above.

We will explore related approaches in Chapter 4, with a special focus on work which addresses alignment issues. We will observe that edit lenses occupy a unique niche in the design space: most other approaches are either asymmetric or do not address the machinery needed to provide key symmetric combinators, and even among the asymmetric approaches it is uncommon to have an elegant theory that is nevertheless capable of addressing performance concerns. Finally, we will give some short concluding remarks and discuss several avenues for future research in Chapter 5.

1.7 Notation and Conventions

This section is intended to be a reference for the most common notation used in this dissertation. All non-standard notation will also be introduced and explained inline before its first use, so this section can safely be skimmed or even completely skipped; however it might nevertheless be useful for the reader who has forgotten what some particular piece of notation means and would not like to pore through the entire document to find its first use.

Naming When naming a set, we will make the choices that follow (perhaps appending a subscript or prime) unless there is a compelling local reason to choose another name:

- S and V for the source and view of asymmetric lenses,
- V, W, X, Y, Z for the kinds of values synchronized by symmetric lenses,
- and C for complement sets.

If we need a set of edits for a named set, its default name is formed by prepending ∂; for example, ∂X is the set of edits to values of type X. Set members will be named with lower case letters that match the set name; for example, $x \in X$ or $s \in S$. The lower case version of ∂ is d; for example, $\mathrm{d}x \in \partial X$. Lenses are named k, ℓ, m, and n.

Lists We use $X^\star$ to denote the set of lists with elements drawn from X. A length n list with x_i in the ith position is written $\langle x_1, \ldots, x_n \rangle$. A notable special case of this is $\langle\rangle$, the empty list. We will also use $x{:}t$ to denote the list whose first element is x and whose remaining elements are in t. When there is only one list involved in the nearby discussion, we will use n to denote the length of that list; otherwise, the notation $|x|$ gives the length of list x. To avoid clutter, we will write singleton lists $\langle x \rangle$ simply as x when it is clear from context both that x is a list element and that we expect a list, not an element. If there is a list $\langle x_1, \ldots, x_n \rangle$ (with exactly the subscripts 1 through

Notation	Meaning
$A \to B$	normal functions from A to B
$A \rightharpoonup B$	partial functions from A to B
$S \overset{a}{\leftrightarrow} V$	asymmetric, state-based lenses connecting S and V
$X \overset{c}{\leftrightarrow} Y$	constraint maintainers connecting X and Y
$X \leftrightarrow Y$	symmetric, state-based lenses connecting X and Y
$X \overset{\Delta}{\leftrightarrow} Y$	(symmetric) edit lenses connecting X and Y

Table 1.2: Function and lens types

n), we will also denote this list simply by x with no subscript. We will write $x[i \mapsto v]$ for the list x with index i replaced by element v, that is,

$$x[i \mapsto v] = \langle x_1, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_n \rangle .$$

We will also need to deal with the set of infinite lists, which we denote X^ω when the elements are drawn from X. The infinite list with x_i in the ith position is written $\langle x_1, \ldots \rangle$, and the infinite list where there is a single element x in every position is written x^ω. As with finite lists, $x{:}t$ denotes the infinite list whose first element is x and whose remaining elements are in t.

Miscellaneous notation We will use CamelCasedSmallCaps for the names of behavioral laws; a `monospaced font` for data; and a sans serif font for code and globally-scoped defined values. We name the canonical single-element set and its single element by the definition $\mathit{Unit} = \{()\}$. When defining and using lenses and similar structures, we will use record notation; for example, $\ell.\mathit{get}$ is the *get* component of lens ℓ. We deal with many variations on functions in this document; Table 1.2 summarizes them.

Chapter 2

Symmetric Lenses

In this chapter, we address the problem of symmetry without regard for alignment or performance issues. We will begin from asymmetric, state-based lenses and build a theory of symmetric, state-based lenses from them, and show how to recover the rich asymmetric syntax in symmetric form. In particular, we will show how to implement lens composition—the process of running two bidirectional transformations, one after the other—long thought to be an operation fundamentally in conflict with symmetric bidirectional presentations. In order to support this operation with the usual algebraic properties like associativity, we will need to develop a theory of behavioral equivalence. Unlike asymmetric theories, where ordinary equality suffices, our symmetric lenses have hidden state whose importance should be discounted when checking whether two lenses compute the same transformation. We will also discuss a collection of bidirectional operations which correspond to common transformations of container-based data types as well as inductive data types built up from products, sums, and type-level recursion. Finally, we will give an account of the connection between asymmetric and symmetric lenses: asymmetric lenses can be lifted to symmetric lenses, and symmetric lenses can be represented as a span of asymmetric lenses.

2.1 Fundamental Definitions

Complements The key step toward symmetric lenses is the notion of *complements*. The idea dates back to a famous paper in the database literature on the view update problem [5] and was adapted to lenses in [6] (and, for a slightly different definition, [29]), and it is quite simple. If we think of the *get* component of a lens as a sort of projection function, then we can find 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. Now the *put* function doesn't need a whole $x \in X$ to recombine with some updated $y \in Y$; 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 $\mathit{missing} \in C$ and define $\mathit{create}(y)$ as $\mathit{put}(y, \mathit{missing})$.

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

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

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

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

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

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 symbols, $\mathit{Lens}(X, Y) = \exists C.\ \{\mathit{missing} : C,\ \mathit{get} : X \to Y \times C,\ \mathit{put} : Y \times C \to X\}$.

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*.

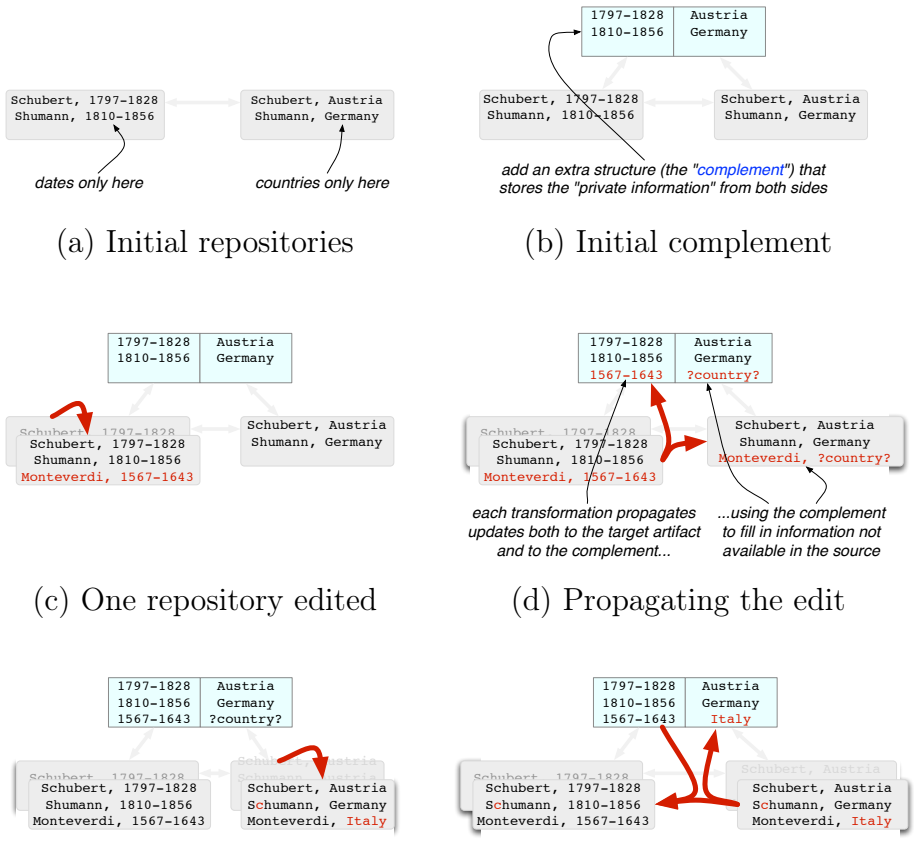
$$\begin{array}{rcl} \mathit{get} & \in & X \times C_Y \to Y \times C_X \\ \mathit{put} & \in & Y \times C_X \to X \times C_Y \end{array}$$

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

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

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

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



(a) Initial repositories (b) Initial complement

(c) One repository edited (d) Propagating the edit

(e) Second repository is edited (f) This change is propagated

Figure 2.1: Behavior of a symmetric lens

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.

2.1.1 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 $\mathit{missing} \in C$, and two functions

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

satisfying the following round-tripping laws:

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

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

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

The force of the PUTRL and PUTLR laws is to establish some "consistent" or "steady-state" triples (x, y, c), for which *put*s of x from the left or y from the right will have no effect—that is, will not change the complement. The conclusion of each rule has the same variable c' on both sides of the equation to reflect this. We will use the equation $putr(x, c) = (y, c)$ to characterize the steady states. In general, a *put* of a new x' from the left entails finding a y' and a c' that restore consistency. Additionally, we often wish this process to involve the complement c from the previous steady state; as a result, it can be delicate to choose a good value of *missing*. This value can often be chosen compositionally; each of our primitive lenses and lens combinators specify one good choice for *missing*.

Examples Figure 2.1 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 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 this lens e. Its *putr* component is given (x, y) as input and has (x', z) in its complement; it constructs a new complement by replacing x' by x to form (x, z), and it constructs its output by pairing the y from its input and the z from its complement to form (y, z). The *putl* component does the opposite, replacing the z part of the complement and retrieving the x part. Then the top-level lens in Figure 2.1—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 $e^\star$, with 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 maintain the invariant that the complement list in the output

is the same length as the input list. When the lists in the input have different lengths, we can restore the invariant by either truncating the complement list or padding it with $e.missing$. For example, taking $X = \{a, b, c, \ldots\}$, $Y = \{1, 2, 3, \ldots\}$, $Z = \{A, B, C, \ldots\}$, and $e.missing = (m, M)$, and writing $\langle a, b, c\rangle$ for the sequence with the three elements a, b, and c, we could have:

$$\begin{array}{ll} & putr(\langle (a,1)\rangle\,,\ \langle (p,P),(q,Q)\rangle) \\ = & putr(\langle (a,1)\rangle\,,\ \langle (p,P)\rangle) \qquad\qquad \text{(truncating)} \\ = & (\langle (1,P)\rangle\,,\ \langle (a,P)\rangle) \end{array}$$

$$\begin{array}{ll} & putr(\langle (a,1),(b,2)\rangle\,,\ \langle (a,P)\rangle) \\ = & putr(\langle (a,1),(b,2)\rangle\,,\ \langle (a,P),(m,M)\rangle) \qquad \text{(padding)} \\ = & (\langle (1,P),(2,M)\rangle\,,\ \langle (a,P),(b,M)\rangle) \end{array}$$

Notice that, after the first *putr*, the information in the second element of the complement list (q, Q) is lost. The second *putr* creates a brand new second element for the list, so the value Q is gone forever; what's left is the default value M.

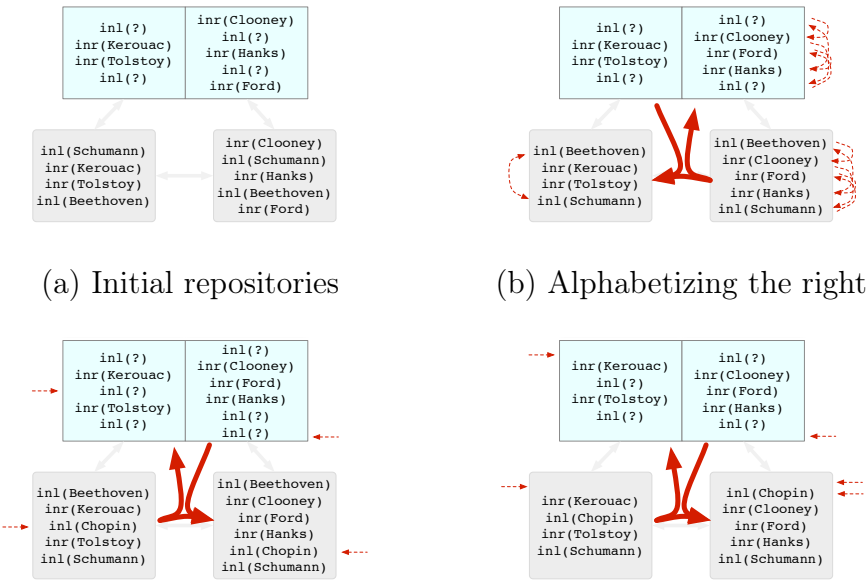
Another possibility—arguably better behaved—is to keep an *infinite* list of complements. Whenever we do a *put*, we use (and update) a prefix of the complement list of the same length as the current value being *put*, but we keep the infinite tail so that, later, we have values to use when the list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

$$\begin{array}{ll} & putr(\langle (a,1)\rangle\,,\ \langle (p,P),(q,Q),(m,M),(m,M),\ldots\rangle) \\ = & (\langle (1,P)\rangle\,,\ \langle (a,P),(q,Q),(m,M),(m,M),\ldots\rangle) \end{array}$$

$$\begin{array}{ll} & putr(\langle (a,1),(b,2)\rangle\,,\ \langle (a,P),(q,Q),(m,M),(m,M),\ldots\rangle) \\ = & (\langle (1,P),(2,Q)\rangle\,,\ \langle (a,P),(b,Q),(m,M),\ldots\rangle) \end{array}$$

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

Figure 2.2 illustrates another use of symmetric lenses. The structures in this example are lists of categorized data; each name on the left is either a composer (tagged `inl`) or an author (tagged `inr`), and each name on the right is either a composer or an actor. The lens under consideration will synchronize just the composers between the two lists, leaving the authors untouched on the left and the actors untouched on the right. The synchronized state (a) shows a complement with two lists, each with holes for the composers. If we re-order the right-hand structure (b), the change in order will be reflected on the left by swapping the two composers. Adding another composer on the left (c) involves adding a new hole to each complement; on the left, the location of the hole is determined by the new list, and on the right it simply shows up at the end. Similarly, if we remove a composer (d), the final hole on the other side disappears.



(a) Initial repositories (b) Alphabetizing the right

(c) Inserting Chopin on the left (d) Deleting Beethoven from the left

Figure 2.2: Synchronizing lists of sums

Abstractly, to achieve this behavior we need to define a lens *comp* between $(X+Y)^\star$ and $(X+Z)^\star$. To do this, it is convenient to first define a lens that connects $(X+Y)^\star$ and $X^\star \times Y^\star$; call this lens *partition*. The complement of the *partition* is a list of booleans telling whether the corresponding element of the left list is an X or a Y. The *putr* function is fairly simple: we separate the $(X+Y)$ list into X and Y lists by checking the tag of each element, and set the complement to exactly match the tags. For example:

$$\begin{aligned} \mathit{putr}(\langle \mathsf{inl}\ a, \mathsf{inl}\ b, \mathsf{inr}\ 1\rangle, c) &= ((\langle a, b\rangle, \langle 1\rangle), \langle \mathsf{false}, \mathsf{false}, \mathsf{true}\rangle) \\ \mathit{putr}(\langle \mathsf{inl}\ a, \mathsf{inr}\ 1, \mathsf{inl}\ b\rangle, c) &= ((\langle a, b\rangle, \langle 1\rangle), \langle \mathsf{false}, \mathsf{true}, \mathsf{false}\rangle) \end{aligned}$$

These examples demonstrate that *putr* ignores the complement entirely, fabricating a completely new one from its input. The *putl* function, on the other hand, relies entirely on the complement for its ordering information. When there are extra entries (not accounted for by the complement), it adds them at the end. Consider taking the output of the second *putr* above and adding c to the X list and 2 to the Y list:

$$\begin{aligned} &\mathit{putl}((\langle a, b, c\rangle, \langle 1, 2\rangle), \langle \mathsf{false}, \mathsf{true}, \mathsf{false}\rangle) = \\ &\qquad (\langle \mathsf{inl}\ a, \mathsf{inr}\ 1, \mathsf{inl}\ b, \mathsf{inl}\ c, \mathsf{inr}\ 2\rangle, \\ &\qquad \langle \mathsf{false}, \mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{true}\rangle) \end{aligned}$$

The *putl* fills in as much of the beginning of the list as it can, using the complement to indicate whether to draw elements from $X^\star$ or from $Y^\star$. (How the remaining X and Y elements are interleaved is a free choice, not specified by the lens laws, since this case only arises when we are *not* in a round-tripping situation. The strategy shown

here, where all new X entries precede all new Y entries, is just one possibility.)

Given *partition*, we can obtain *comp* by composing three lenses in sequence: from $(X + Y)^\star$ we get to $X^\star \times Y^\star$ using *partition*, then to $X^\star \times Z^\star$ using a variant of the lens e discussed above, and finally to $(X + Z)^\star$ using a "backwards" *partition*.

Put-Put Laws

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

$$\frac{putr(x, c) = (y, c')}{putr(x, c') = (y, c')} \qquad \text{(PutR2)}$$

$$\frac{putl(y, c) = (x, c')}{putl(y, c') = (x, c')} \qquad \text{(PutL2)}$$

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:

$$\frac{putr(x, c) = (\_\,, c')}{putr(x', c') = putr(x', c)} \qquad \text{(Strong-PutPutR*)}$$

$$\frac{putl(y, c) = (\_\,, c')}{putl(y', c') = putl(y', c)} \qquad \text{(Strong-PutPutL*)}$$

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 repository, 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 §2.5 satisfy these laws.

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

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

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

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 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 chapter is dealing with the issue of alignment. 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 will incorporate a richer notion of alignment in Chapter 3.

2.2 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 now 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.

2.2.1 Definition [R-similarity]: Given sets X, Y, C_f, C_g and a relation $R \subset C_f \times C_g$, 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:

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

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

2.2.3 Lemma: Lens equivalence is an equivalence relation.

Proof: Reflexivity and symmetry are obvious. We briefly sketch transitivity.

Suppose $k \equiv \ell$ (as witnessed by $R_{k\ell}$) and $\ell \equiv m$ (as witnessed by $R_{\ell m}$). We show that the relation

$$R_{km} = R_{k\ell} \circ R_{\ell m} = \{(c_k, c_m) \mid \exists c_\ell.\ c_k\ R_{k\ell}\ c_\ell \wedge c_\ell\ R_{\ell m}\ c_m\}$$

witnesses the equivalence $k \equiv m$. It is clear that

$$(k.missing, m.missing) \in R_{km},$$

since we can choose $c_\ell = \ell.missing$. Next, we show that $k.putr \sim_{R_{km}} m.putr$. We may assume three things:

$$\begin{aligned}(c_k, c_m) &\in R_{km} \\ k.putr(x, c_k) &= (y_k, c'_k) \\ m.putr(x, c_m) &= (y_m, c'_m)\end{aligned}$$

Since $(c_k, c_m) \in R_{km}$, we can choose c_ℓ such that $(c_k, c_\ell) \in R_{k\ell}$ and $(c_\ell, c_m) \in R_{\ell m}$. Choosing $(y_\ell, c'_\ell) = \ell.putr(x, c_\ell)$, we then conclude that $y_k = y_\ell$ and $(c'_k, c'_\ell) \in R_{k\ell}$, since $k.putr \sim_{R_{k\ell}} \ell.putr$. Similarly, we can conclude that $y_\ell = y_m$ and $(c'_\ell, c'_m) \in R_{\ell m}$ because $\ell.putr \sim_{R_{\ell m}} m.putr$. Thus $y_k = y_m$ and because of the existence of c'_ℓ, we know $(c'_k, c'_m) \in R_{km}$. But these are exactly the two facts we need to conclude that $k.putr \sim_{R_{km}} m.putr$. A similar argument shows that $k.putl \sim_{R_{km}} m.putl$, and hence that $k \equiv m$. □

2.2.4 Definition [Put object]: 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):

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

2.2.5 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

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

2.2.6 Theorem [Equivalence of equivalence]: $k \approx \ell$ 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

$$R = \{(c, d) \mid apply(k, c, P) = apply(\ell, d, P) \text{ for all } P\}.$$

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. □

2.3 Basic Constructions

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

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

$$id_X \in X \leftrightarrow X$$

C	$=$	$Unit$
$missing$	$=$	$()$
$putr(x, ())$	$=$	$(x, ())$
$putl(x, ())$	$=$	$(x, ())$

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.

2.3.2 Definition [Lens composition]:

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

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

Proof of well-formedness: We show that the lens satisfies PutRL; the proof that it satisfies PutLR is entirely symmetric. Assume that k and ℓ each satisfy PutRL, and that $(k;\ell).\mathit{putr}(x,(c_k,c_\ell)) = (z,(c'_k,c'_\ell))$. From the definition of $(k;\ell).\mathit{putr}$, we can conclude that there is a y such that $k.\mathit{putr}(x,c_k) = (y,c'_k)$ and $\ell.\mathit{putr}(y,c_\ell) = (z,c'_\ell)$.

$$\begin{array}{rcll}
(k;\ell).\mathit{putl}(z,(c'_k,c'_\ell)) & = & \mathsf{let}\ (y',c''_\ell) = \ell.\mathit{putl}(z,c'_\ell)\ \mathsf{in} & (2.3.1) \\
 & & \mathsf{let}\ (x',c''_k) = k.\mathit{putl}(y',c'_k)\ \mathsf{in} & \\
 & & (x',(c''_k,c''_\ell)) & \\
 & = & \mathsf{let}\ (y',c''_\ell) = (y,c'_\ell)\ \mathsf{in} & (2.3.2) \\
 & & \mathsf{let}\ (x',c''_k) = k.\mathit{putl}(y',c'_k)\ \mathsf{in} & \\
 & & (x',(c''_k,c''_\ell)) & \\
 & = & \mathsf{let}\ (x',c''_k) = k.\mathit{putl}(y,c'_k)\ \mathsf{in} & (2.3.3) \\
 & & (x',(c''_k,c'_\ell)) & \\
 & = & \mathsf{let}\ (x',c''_k) = (x,c'_k)\ \mathsf{in} & (2.3.4) \\
 & & (x',(c''_k,c'_\ell)) & \\
 & = & (x,(c'_k,c'_\ell)) & (2.3.5)
\end{array}$$

Equation 2.3.1 comes from expanding the definition of $(k;\ell).\mathit{putl}$; equation 2.3.2 from applying PutRL to ℓ; equation 2.3.3 from substituting let-bound variables; equation 2.3.4 from applying PutRL to k; and equation 2.3.5 from again substituting let-bound variables. Moreover, this last equation is exactly what is demanded from applying PutRL to $k;\ell$, so we are done. □

This definition specifies what it means to compose two lenses. To show that this definition lifts to equivalence classes of lenses, we need to check the following congruence property.

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

2.3.4 Definition: The following function on relations is convenient here:

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

Proof of 2.3.3: If the simulation R_k witnesses $k \equiv k'$ and R_ℓ witnesses $\ell \equiv \ell'$ then it is straightforward to verify that $R = R_k \times R_\ell$ witnesses $k;\ell \equiv k';\ell'$. There are three things to show.

1. We wish to show the first line:

$$\begin{aligned} & (k;\ell).\mathit{missing}\ R\ (k';\ell').\mathit{missing} \\ \iff & (k.\mathit{missing}, \ell.\mathit{missing})\ R\ (k'.\mathit{missing}, \ell'.\mathit{missing}) \\ \iff & k.\mathit{missing}\ R_k\ k'.\mathit{missing} \wedge \ell.\mathit{missing}\ R_\ell\ \ell'.\mathit{missing} \end{aligned}$$

   But the final line is certainly true, since R_k and R_ℓ are simulation relations.

2. We must show that $(k;\ell).\mathit{putr} \sim_R (k';\ell').\mathit{putr}$. So take $c_k, c_\ell, c_{k'}, c_{\ell'}$ such that $(c_k, c_\ell)\ R\ (c_{k'}, c_{\ell'})$ and choose an input x. Define the following:

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

   We can then compute:

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

   We need to show that $z = z'$ and that $(c_k', c_\ell')\ R\ (c_{k'}', c_{\ell'}')$. Since $c_k\ R_k\ c_{k'}$, we can conclude that $y = y'$ and $c_k'\ R_k\ c_{k'}'$; similarly, since $c_\ell\ R_\ell\ c_{\ell'}$ and $y = y'$, we know that $z = z'$ (discharging one of our two proof burdens) and $c_\ell'\ R_\ell\ c_{\ell'}'$. Combining the above facts, we find that $(c_k', c_\ell')\ R\ (c_{k'}', c_{\ell'}')$ by definition of R (discharging the other proof burden).

3. The proof that $(k;\ell).\mathit{putl} \sim_R (k';\ell').\mathit{putl}$ is similar to the *putr* case. □

2.3.5 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

$$R = \{((c_1, (c_2, c_3)), ((c_1, c_2), c_3)) \mid c_1 \in j.C, c_2 \in k.C, c_3 \in \ell.C\}.$$

The verification is then straightforward. □

2.3.6 Lemma [Identity arrows]: The identity lens 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 $((), c)\ R\ c$ whenever $c \in \ell.C$. The verification is direct.

The proof of the right-identity law $\ell; id \equiv \ell$ is analogous. □

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

2.3.7 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 implementing a bidirectional language, we might not want to expose *bij* in its syntax, 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.)

Proof of well-formedness: We verify that the PUTRL law holds for bijection lenses; the proof that PUTLR holds is symmetric. Observe that $bij_f.putr(x, ()) = (f(x), ())$. We can therefore compute that $bij_f.putl(f(x), ()) = (f^{-1}(f(x)), ()) = (x, ())$. Thus, after a round-trip, we return to the same x we started from—and the same complement, (), validating the law. □

In fact, any stateless lens is an instance of a bijection lens:

2.3.8 Lemma: If $\ell \in X \leftrightarrow Y$ and $h \in \ell.C \to \mathit{Unit}$ is a bijection, then there exists a bijection $f \in X \to Y$ such that $\ell \equiv \mathit{bij}_f$.

Proof: We define:

$$f(x) = \mathsf{fst}(\ell.\mathit{putr}(x, h^{-1}(())))$$

We must show that f is bijective and that $\mathit{bij}_f \equiv \ell$. For the former, we exhibit its inverse in g:

$$g(y) = \mathsf{fst}(\ell.\mathit{putl}(y, h^{-1}(())))$$

The round-trip law PUTRL guarantees that $g(f(x)) = x$, and the round-trip law PUTLR guarantees that $f(g(y)) = y$.

To show the latter, we argue that h witnesses the equivalence. Clearly

$$h(\mathit{bij}_f.\mathit{missing}) = \ell.\mathit{missing}$$

because all elements of $\ell.C$ are equal (and hence $(\mathit{bij}_f.\mathit{missing}, \ell.\mathit{missing}) \in h$). The definition of f makes it clear that $\mathit{bij}_f.\mathit{putr} \sim_h \ell.\mathit{putr}$; similarly, the definition of f's inverse g makes it clear that $\mathit{bij}_f.\mathit{putl} \sim_h \ell.\mathit{putl}$. □

2.3.9 Corollary: If $\ell.C$ is a singleton set $\{c\}$ and $\mathsf{fst}(\ell.\mathit{putr}(x, c)) = x$ for all x, then $\ell \equiv \mathit{id}$.

This transformation (like several others we will see) respects much of the structure available in our category. Formally, *bij* is a functor. Recall that 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 $F(\mathit{id}_X) = \mathit{id}_{F(X)}$ 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.

2.3.10 Lemma [Embedding bijections]: The *bij* operator forms a functor from the category ISO, whose objects are sets and whose arrows are isomorphic functions, to LENS—that is, $\mathit{bij}_{\mathit{id}_X} = \mathit{id}_X$ and $\mathit{bij}_f; \mathit{bij}_g = \mathit{bij}_{f;g}$.

Proof: Showing that $bij_{id_X} = id_X$ is a straightforward application of Corollary 2.3.9. Now consider $bij_f; bij_g$. Since its complement is a singleton set, Lemma 2.3.8 tells us that $bij_f; bij_g \equiv bij_h$, where

$$h(x) = \mathsf{fst}((bij_f; bij_g).putr(x, ((), ()))),$$

which can be reduced to:

$$h(x) = g(f(x))$$

Thus $bij_f; bij_g \equiv bij_{f;g}$ as desired. □

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. This is because a bijective lens displays no dependency on the complement at all, whereas an isomorphism in the category of lenses still allows for some limited interaction with the complement as in 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:

$$strange \in Trit \leftrightarrow Trit$$

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

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

2.3.11 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 $R \subset k.C \times \ell.C$ 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 $c = (c_k, c_\ell)$ where $c_k \; R_1 \; c_\ell$ 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, $c' \in R_2$. □

2.3.12 Corollary [Isomorphisms are indexed bijections]: 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: For any $x \in X$, we know $f_{\ell,c_\ell}(f_{k,c_k}(x)) = x$ by 2, and for any $y \in Y$, we know $f_{k,c_k}(f_{\ell,c_\ell}(y)) = y$ by 4. Thus, not only is f_{k,c_k} a bijection, we actually have its inverse: $f^{-1}_{k,c_k} = f_{\ell,c_\ell}$! Similarly, $g^{-1}_{k,c_k} = g_{\ell,c_\ell}$. □

2.3.13 Definition [Dual of a lens]:

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

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

Proof of well-formedness: We observe that saying ℓ^{op} satisfies PutRL is an identical condition to saying ℓ satisfies PutLR, and likewise having ℓ^{op} satisfy PutLR is identical to having ℓ satisfy PutRL. □

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

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

Proof: We must show three things:

1. The mapping $[\ell] \mapsto [\ell^{op}]$ is well-defined, that is, that if $k \equiv \ell$, then $k^{op} \equiv \ell^{op}$.
2. The mapping respects identities, that is, that $id^{op} \equiv id$.
3. The mapping respects composition, that is, $(k;\ell)^{op} \equiv \ell^{op}; k^{op}$.

We sketch the proofs in that order.

1. If $k \equiv \ell$ is witnessed by R then $k^{op} \equiv \ell^{op}$ is also witnessed by R;
2. In fact, $id^{op} = id$; and

3. The relation $(c_k, c_\ell)\ R\ (c_\ell, c_k)$ whenever $c_k \in k.C$ and $c_\ell \in \ell.C$ witnesses the equivalence. □

The existence of $(-)^{op}$ is one of the two canonical constructions that motivate the name "symmetric lenses" (the other being *disconnect*, which we discuss below). Before we formalize this intuition, we review two standard constructions from category theory.

2.3.15 Definition: The *opposite* of a category $\mathcal{C}$, denoted $\mathcal{C}^{op}$, has backwards composition compared to $\mathcal{C}$. That is, whenever $f; g = h$ in $\mathcal{C}$, we have $g; f = h$ in $\mathcal{C}^{op}$. This induces the remaining components of $\mathcal{C}^{op}$:

Objects The objects of $\mathcal{C}^{op}$ are exactly the objects of $\mathcal{C}$.

Arrows The arrows $f : X \to Y$ of $\mathcal{C}^{op}$ are the arrows $f : Y \to X$ of $\mathcal{C}$.

Identities The identities of $\mathcal{C}^{op}$ are exactly the identities of $\mathcal{C}$.

That is, forming the opposite of a category means formally reversing the "direction" of each arrow. In general, a category and its opposite can have very different structure. What we want to show is that the directionality of arrows in LENS is not important; we can formalize this by saying that LENS and LENS^{op} have the same structure, provided we can formalize what it means for two categories have the same structure. There are many ways to define equivalence between categories; we give a particularly strong one here.

2.3.16 Definition: Categories $\mathcal{C}$ and $\mathcal{D}$ are isomorphic if there are functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ for which $F; G$ is the identity on $\mathcal{C}$ and $G; F$ is the identity on $\mathcal{D}$.

2.3.17 Corollary: The category LENS is self dual, i.e., isomorphic to LENS^{op}. (Note that this does not mean that each arrow is its own inverse!)

Proof: The arrow part of $(-)^{op}$ is bijective. □

The lenses we have discussed so far maintain all the information in the domain and codomain. It is sometimes useful to discard some information in one direction of the lens. The terminal lens does this, recording the discarded information in the complement so that the other direction of the lens can restore it.

2.3.18 Definition [Terminal lens]:

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

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

Proof of well-formedness: The PUTLR law is trivially true, since

$$putr(putl((), c)) = putr(c, c) = ((), c)$$

and in particular since c does not change at all in this round trip. We also observe:

$$putl(putr(x, c)) = putl((), x) = (x, x)$$

Since the complement x does not change during the *putl* and we arrive back at the value x that we started with, this verifies that PUTRL holds as well. □

2.3.19 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 $k \equiv term_x$.

Of course, there may be many (pairwise non-equivalent) terminal lenses of a particular type; for any two $x, y \in X$ with $x \neq y$, it's clear that $term_x \not\equiv term_y$. Proposition 2.3.19 tells us that there are exactly as many arrows $\ell : X \Longleftrightarrow Unit$ as there are elements of 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 as follows:

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

It's clear that $k.missing\ R\ x$, since we have chosen x specifically so that

$$\mathsf{fst}(k.putl((), k.missing)) = x.$$

Let us show next that $k.putl \sim_R term_x.putl$. Choose arbitrary $v \in Unit$ and choose c and y such that $\mathsf{fst}(k.putl((), c)) = y$. Clearly, $v = ()$, so we can compute:

$$k.putl(v, c) = k.putl((), c) = (y, c')$$
$$term_x.putl(v, y) = term_x.putl((), y) = (y, y)$$

Clearly, $y = y$, and law PUTL2 tells us that $k.putl((), c') = (y, c')$, and hence that $c'\ R\ y$.

Finally, we must show that $k.putr \sim_R term_x.putr$. Again, choose c and y such that $\mathsf{fst}(k.putl((), c)) = y$ and arbitrary $z \in X$.

$$k.putr(z, c) = ((), c')$$
$$term_x.putr(z, y) = ((), z)$$

It's clear that $() = ()$, and law PUTRL tells us that $k.putl((), c') = (z, c')$, and hence $c'\ R\ z$. □

2.3.20 Definition [Disconnect lens]:

$$\frac{x \in X \qquad y \in Y}{\mathit{disconnect}_{xy} \in X \leftrightarrow Y}$$

$$\mathit{disconnect}_{xy} \;=\; \mathit{term}_x ; \mathit{term}_y^{op}$$

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

(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.)

2.4 Products

A few additional notions from elementary category theory will be useful 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.

2.4.1 Theorem [No products]: LENS does not have products.

Proof idea: Suppose we have lenses $k \in Z \Longleftrightarrow X$ and $\ell \in Z \Longleftrightarrow Y$. Informally, the lens k includes a way to take any Z and choose a corresponding X and a way to take any X and find a corresponding Z. Many common categories with products include the former, but the latter is somewhat unique to lens categories, so we focus on the return trip here.

The lenses k and ℓ together mean we have a way to take any X and choose a corresponding Z, and we have a (separate) way to take any Y and choose a corresponding Z. Assume temporarily that the object part of the product of two objects is simply the Cartesian product. To complete the product, we must construct $\langle k, \ell \rangle \in Z \Longleftrightarrow X \times Y$, that is, we must find a way to take an X and a Y and choose a Z that corresponds to both simultaneously. But there may not be any such Z—the Z that k gives us from X may not be the same as the Z that ℓ gives us from Y.

To complete the proof, we simply choose X and Y carefully to rule out the possibility of a corresponding Z, regardless of whether we choose $X \times Y$ to be the Cartesian product or to be some other construction.

Proof: Uniqueness of pairing shows that there is exactly one lens from *Unit* to *Unit* $\times$ *Unit* (whatever this may be). Combined with Prop. 2.3.19 this shows that *Unit* $\times$ *Unit* is a one-element set. Again by Prop. 2.3.19 this then means that lenses between *Unit* $\times$ *Unit* and any other set X are constant which leads to cardinality clashes once $|X| > 1$.

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 2.3.19 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. □

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

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

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

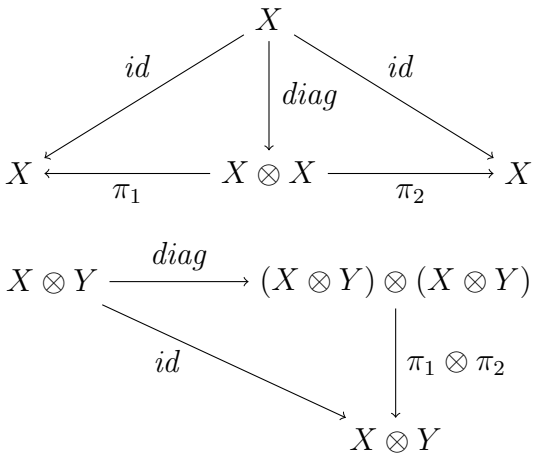
such that (suppressing subscripts to reduce clutter)

$$(f \otimes g); \pi_1 = \pi_1; f \tag{2.4.1}$$

$$(f \otimes g); \pi_2 = \pi_2; g \tag{2.4.2}$$

$$diag; (f \otimes f) = f; diag \tag{2.4.3}$$

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 2.4.1 and 2.4.2 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, arrows satisfying law 2.4.3), none of them make the additional diagrams commute.

2.4.2 Definition [Tensor product lens]:

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

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

Proof of well-formedness: We will show that PUTRL holds; a similar argument shows that PUTLR holds. Suppose

$$k.putr(x, c_k) = (z, c'_k)$$
$$\ell.putr(y, c_\ell) = (w, c'_\ell)$$

so that:

$$(k \otimes \ell).putr((x, y), (c_k, c_\ell)) = ((z, w), (c'_k, c'_\ell))$$

Applying PUTRL to the lenses k and ℓ, we learn that

$$k.putl(z, c'_k) = (x, c'_k)$$
$$\ell.putl(w, c'_\ell) = (y, c'_\ell)$$

so that:

$$(k \otimes \ell).putl((z, w), (c'_k, c'_\ell)) = ((x, y), (c'_k, c'_\ell))$$

But this is exactly what we need to show for rule PUTRL. □

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 $R = R_k \times R_\ell$ witnesses $k \otimes \ell \equiv k' \otimes \ell'$.

Since $k.missing\ R_k\ k'.missing$ and $\ell.missing\ R_\ell\ \ell'.missing$, we know that

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

that is:

$$(k \otimes \ell).missing\ R\ (k' \otimes \ell').missing$$

Choose arbitrary $(x, y) \in X \times Y$ and related complements $(c_k, c_\ell)\ R\ (c_{k'}, c_{\ell'})$. Define:

$$(z, c'_k) = k.putr(x, c_k)$$
$$(z', c'_{k'}) = k'.putr(x, c_{k'})$$
$$(w, c'_\ell) = \ell.putr(y, c_\ell)$$
$$(w', c'_{\ell'}) = \ell'.putr(y, c_{\ell'})$$

Since $c_k\ R_k\ c_{k'}$ and $k.putr \sim_{R_k} k'.putr$, we can conclude that $z = z'$ and $c'_k\ R_k\ c'_{k'}$. Similarly, $w = w'$ and $c'_\ell\ R_\ell\ c'_{\ell'}$. But we can compute

$$(k \otimes \ell).putr((x, y), (c_k, c_\ell)) = ((w, z), (c'_k, c'_\ell))$$
$$(k' \otimes \ell').putr((x, y), (c_{k'}, c_{\ell'})) = ((w', z'), (c'_{k'}, c'_{\ell'}))$$

where $(w, z) = (w', z')$ and $(c'_k, c'_\ell)\ R\ (c'_{k'}, c'_{\ell'})$. Thus, $(k \otimes \ell).putr \sim_R (k' \otimes \ell').putr$.

Showing that $(k \otimes \ell).putl \sim_R (k' \otimes \ell').putl$ is similar. □

2.4.3 Lemma [Functoriality of $\otimes$]: The tensor product operation on lenses induces a bifunctor on the category LENS, that is,

$id_X \otimes id_Y \equiv id_{X \times Y}$, and

$(k_1; \ell_1) \otimes (k_2; \ell_2) \equiv (k_1 \otimes k_2); (\ell_1 \otimes \ell_2)$.

Proof of functoriality: Corollary 2.3.9 implies the former equivalence. The latter has an intricate (but uninteresting) witness:

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

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. □

2.4.4 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.

$$\begin{array}{rcl} k.putr((x,y),((),())) & = & \mathsf{let}\ (x', c_1) = bij_f.putr(x,()) \ \mathsf{in} \\ & & \mathsf{let}\ (y', c_2) = bij_g.putr(y,()) \ \mathsf{in} \\ & & ((x', y'), (c_1, c_2)) \\ & = & ((f(x), g(y)), ((), ())) \\ \ell.putr((x,y),()) & = & ((f(x), g(y)), ()) \end{array}$$

The *putl* direction is similar. □

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

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

These are known as the *associator*, *left-unitor*, *right-unitor*, and *symmetry*, respectively. In addition to the equations implied by these being natural isomorphisms, they must also satisfy the coherence equations:

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

2.4.5 Proposition [LENS,$\otimes$ is symmetric monoidal]: 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 §5.1).

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.[2] The five coherence conditions follow from coherence in SET, functoriality of *bij*, and Lemma 2.4.4. □

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

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

$$\pi_{1y} = (id_X \otimes term_y);\rho^\otimes_X$$

[2]For example, showing that $\gamma^\otimes$ is natural requires showing that for any two lenses $k : X \leftrightarrow Z$ and $\ell : Y \leftrightarrow W$,

$$(k \otimes \ell);\gamma^\otimes_{Z,W} \equiv \gamma^\otimes_{X,Y};(\ell \otimes k).$$

The complements for these two lenses are $(k.C \times \ell.C) \times Unit$ and $Unit \times (\ell.C \times k.C)$; the isomorphism that simply rearranges the parts of the complement is a witness to the lenses' equivalence. The story is similar for the other naturality properties.

The other projection is defined similarly.

Returning to the example in the introduction, recall that we wish to create a lens $e : X \times Y \leftrightarrow Y \times Z$ with missing elements $m \in X$ and $M \in Z$. We now have the machinery necessary to construct this lens:

$$e = \pi_{2m} \,;\, \pi^{op}_{1M}$$

The extra parameter to the projection (e.g. m or M above) needs to be chosen with some care. Some sets may have clear neutral elements; for example, a projection from $A \times B^\star \to A$ will likely use the empty list $\langle\rangle$ as its neutral element. Other projections may need additional domain knowledge to choose a good neutral element—for example, a projection $A \times \text{Country} \to A$ might use the country with the most customers as its default.

In some cases, the algebraic laws that one wants the projection to satisfy may guide the choice as well. The extra parameter 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.

2.4.7 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:

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

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. □

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 [16])

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

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

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

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

—$eq(x,x')$ yields $\mathsf{inl}\ ()$ if $x = x'$ and yields x' if not. This assumes that X possesses a decidable equality, a reasonable assumption for the applications of lenses that we know about. However, neither of these proposals satisfy all the required laws.

Proof of well-formedness:
PutLR:

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

PutRL:

$$\begin{array}{rcll} putl(putr(x, \mathsf{inl}\ ())) & = & putl((x,x), \mathsf{inl}\ ()) & \\ & = & (x, \mathsf{inl}\ ()) & \\ putl(putr(x, \mathsf{inr}\ x')) & = & putl((x,x'), eq(x,x')) & \\ & = & (x, eq(x,x')) & \square \end{array}$$

2.5 Sums and Lists

Historically, 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 2.3.17 and Theorem 2.4.1 imply that LENS does not have sums. But we do have a tensor whose object part is a set-theoretic sum—in fact, there are at least two interestingly different ones—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.

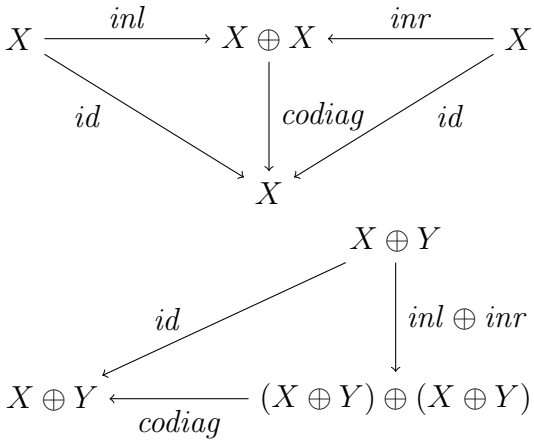
As with products, a tensor can be extended to a sum by providing three natural transformations—this time written *inl*, *inr*, and *codiag*; that is, for each pair of objects X and Y, there must be arrows

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

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:



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 §2.1, differ in how they handle the complement when the new value being *put* is from a different branch of the sum than the old value 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*.

2.5.1 Definition [Retentive tensor sum lens]:

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

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

Proof of well-formedness: We show that PutRL holds; the proof that PutLR holds is similar. Choose arbitrary $c_k \in k.C$ and $c_\ell \in \ell.C$. There are two cases to consider for the starting value: it will be either $\mathsf{inl}\ x$ for some $x \in X$ or $\mathsf{inr}\ y$ for some $y \in Y$. In the former case, define $(z, c'_k) = k.\mathit{putr}(x, c_k)$ so that applying PutRL to k tells us that $k.\mathit{putl}(z, c'_k) = (x, c'_k)$. But now we can compute:

$$\mathit{putl}(\mathit{putr}(\mathsf{inl}\ x, (c_k, c_\ell))) = \mathit{putl}(\mathsf{inl}\ z, (c'_k, c_\ell)) = (\mathsf{inl}\ x, (c'_k, c_\ell)).$$

Thus, the value has round-tripped exactly as $\mathsf{inl}\ x$, and the complement changed only after the *putr* (and not after the *putl*) – exactly what we needed to show.

The other case is similar: define $(w, c'_\ell) = \ell.\mathit{putr}(y, c_\ell)$ so that applying PutRL to ℓ tells us that $\ell.\mathit{putl}(w, c'_\ell) = (y, c'_\ell)$. Computation then shows that:

$$\mathit{putl}(\mathit{putr}(\mathsf{inr}\ y, (c_k, c_\ell))) = \mathit{putl}(\mathsf{inr}\ w, (c_k, c'_\ell)) = (\mathsf{inr}\ y, (c_k, c'_\ell)).\ \square$$

Proof of preservation of equivalence: Suppose $k \equiv k'$ and $\ell \equiv \ell'$, as witnessed by relations R_k and R_ℓ, respectively. Then $R = R_k \times R_\ell$ witnesses the equivalence $k \oplus \ell \equiv k' \oplus \ell'$. Since $k.\mathit{missing}\ R_k\ k'.\mathit{missing}$ and $\ell.\mathit{missing}\ R_\ell\ \ell'.\mathit{missing}$, we have $(k \oplus \ell).\mathit{missing}\ R\ (k' \oplus \ell').\mathit{missing}$.

We now show that $(k \oplus \ell).\mathit{putr} \sim_R (k' \oplus \ell').\mathit{putr}$. Choose arbitrary $v \in X + Y, c_k \in k.C, c_{k'} \in k'.C, c_\ell \in \ell.C, c_{\ell'} \in \ell'.C$ such that $(c_k, c_\ell)\ R\ (c_{k'}, c_{\ell'})$. By the definition of

R, we can conclude that $c_k \; R_k \; c_{k'}$ and that $c_\ell \; R_\ell \; c_{\ell'}$. There are two cases to consider: either $v = \mathsf{inl}\ x$ for some $x \in X$ or $v = \mathsf{inr}\ y$ for some $y \in Y$. In the first case, define

$$(z, c'_k) = k.putr(x, c_k)$$
$$(z', c'_{k'}) = k'.putr(x, c_{k'})$$

Since $c_k \; R_k \; c_{k'}$, we can conclude $z = z'$ and $c'_k \; R_k \; c'_{k'}$. Therefore,

$$(k \oplus \ell).putr(v, (c_k, c_\ell)) = (\mathsf{inl}\ z, (c'_k, c_\ell))$$
$$(k' \oplus \ell').putr(v, (c_{k'}, c_{\ell'})) = (\mathsf{inl}\ z, (c'_{k'}, c_{\ell'}))$$

where $(c'_k, c_\ell) \; R \; (c'_{k'}, c_{\ell'})$ as desired. The second case, where $v = \mathsf{inr}\ y$, is similar.

Showing that $(k \oplus \ell).putl \sim_R (k' \oplus \ell').putl$ is symmetric to the argument for $putr$. □

2.5.2 Lemma [Functoriality of $\oplus$]: The tensor sum operation on lenses induces a bifunctor on LENS.

Proof of functoriality: Corollary 2.3.9 gives us $id_X \oplus id_Y \equiv id_{X+Y}$ with fairly minor computation. We must also show that composition is preserved. Suppose we have four lenses:

$$k \in X \leftrightarrow Y \qquad k' \in X' \leftrightarrow Y'$$
$$\ell \in Y \leftrightarrow Z \qquad \ell' \in Y' \leftrightarrow Z'$$

The obvious isomorphism between complements witnesses the equivalence $(k; \ell) \oplus (k'; \ell') \equiv (k \oplus k'); (\ell \oplus \ell')$, namely:

$$((c_k, c_\ell), (c'_k, c'_\ell)) \; R \; ((c_k, c'_k), (c_\ell, c'_\ell))$$

Define abbreviations $a = (k; \ell) \oplus (k'; \ell')$ and $b = (k \oplus k'); (\ell \oplus \ell')$. Expanding definitions,

$$a.missing = ((k.missing, \ell.missing), (k'.missing, \ell'.missing))$$
$$b.missing = ((k.missing, k'.missing), (\ell.missing, \ell'.missing))$$

so $a.missing \; R \; b.missing$. We must also show $a.putr \sim_R b.putr$ and $a.putl \sim_R b.putl$. We will show only the former; the proof of the latter is similar.

Choose arbitrary $v \in X + X', c_a \in a.C, c_b \in b.C$ such that $c_a \; R \; c_b$. This means there are $c_k \in k.C, c_{k'} \in k'.C, c_\ell \in \ell.C, c_{\ell'} \in \ell'.C$ such that $c_a = ((c_k, c_\ell), (c_{k'}, c_{\ell'}))$ and $c_b = ((c_k, c_{k'}), (c_\ell, c_{\ell'}))$. There are two cases to consider: either $v = \mathsf{inl}\ x$ or $v = \mathsf{inr}\ x'$. In the first case, we can define

$$(y, c'_k) = k.putr(x, c_k)$$
$$(z, c'_\ell) = \ell.putr(y, c_\ell),$$

and compute:

$$a.putr(\mathsf{inl}\ x, ((c_k, c_\ell), (c_{k'}, c_{\ell'}))) = (\mathsf{inl}\ z, ((c'_k, c'_\ell), (c_{k'}, c_{\ell'})))$$
$$b.putr(\mathsf{inl}\ x, ((c_k, c_{k'}), (c_\ell, c_{\ell'}))) = (\mathsf{inl}\ z, ((c'_k, c_{k'}), (c'_\ell, c_{\ell'})))$$

Since $\mathsf{inl}\ z = \mathsf{inl}\ z$ and $((c'_k, c'_\ell), (c_{k'}, c_{\ell'}))\ R\ ((c'_k, c_{k'}), (c'_\ell, c_{\ell'}))$, we have finished the first case. The second case, where $v = \mathsf{inr}\ x'$, is nearly identical, and we conclude that $a.putr \sim_R b.putr$. □

2.5.3 Definition [Forgetful tensor sum]:

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

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

putl is similar

Proof of well-formedness: As for the retentive sum, the round-trip laws for k and ℓ guarantee that $k \oplus^f \ell$ round-trips. The only difference is that there are additional cases to consider when the tag on the value and the tag on the complement do not match at the beginning of the trip; however, this poses no real difficulty, as the tags *will* match after the first put. □

Proof of preservation of equivalence: Let $a = k \oplus^f \ell$ and $b = k' \oplus^f \ell'$. If R_k witnesses $k \equiv k'$ and R_ℓ witnesses $\ell \equiv \ell'$ then $a \equiv b$ may be witnessed by

$$R = \{(\mathsf{inl}\ c, \mathsf{inl}\ c') \mid c\ R_k\ c'\} \cup \{(\mathsf{inr}\ c, \mathsf{inr}\ c') \mid c\ R_\ell\ c'\}$$

Since $k.missing\ R_k\ k'.missing$, we know $a.missing\ R\ b.missing$.

We must still show that $a.putr \sim_R b.putr$ and that $a.putl \sim_R b.putl$; for each of these proofs, there are cases to consider where the input is tagged inl and cases

where the input is tagged inr. Below, we will consider only the inl case for *putr*; the remaining cases are similar.

Therefore, consider arbitrary $x \in X, c_a \in a.C, c_b \in b.C$ such that $c_a \; R \; c_b$. Project these complements into $k.C$ and $k'.C$, respectively, as follows:

$$c'_a = \begin{cases} c_k & c_a = \mathsf{inl}\ c_k \\ k.missing & c_a = \mathsf{inr}\ c_\ell \end{cases}$$
$$c'_b = \begin{cases} c_{k'} & c_b = \mathsf{inl}\ c_{k'} \\ k'.missing & c_b = \mathsf{inr}\ c_{\ell'} \end{cases}$$

Since $c_a \; R \; c_b$, we know they have the same tag, and hence that c'_a and c'_b follow the same "branch" in their definition; in either branch, we find that $c'_a \; R_k \; c'_b$, because $c_a \; R \; c_b$ and $k.missing \; R_k \; k'.missing$. But now we can compute:

$$a.putr(x, c_a) = \mathsf{let}\ (z, c'_k) = k.putr(x, c'_a)\ \mathsf{in}\ (\mathsf{inl}\ z, \mathsf{inl}\ c'_k)$$
$$b.putr(x, c_b) = \mathsf{let}\ (z, c'_{k'}) = k'.putr(x, c'_b)\ \mathsf{in}\ (\mathsf{inl}\ z, \mathsf{inl}\ c'_{k'})$$

The desired properties now arise because $k.putr \sim_{R_k} k'.putr$ and $c'_a \; R_k \; c'_b$. □

Proof of functoriality: There are two things to show:

$$id_X \oplus^f id_Y \equiv id_{X+Y}$$

$$(k \oplus^f k'); (\ell \oplus^f \ell') \equiv (k; \ell) \oplus^f (k'; \ell')$$

For identity preservation, we use the total relation:

$$c \; R \; ()$$

Clearly the initial condition $(id \oplus^f id).missing \; R \; id.missing$ holds; we will also show that $(id \oplus^f id).putr \sim_R id.putr$, eliding the similar proof relating the *putl* functions. So, choose arbitrary $v \in X + Y$ and $c \in Unit + Unit$.

$$(id \oplus^f id).putr(v, c) = \begin{cases} \mathsf{let}\ (x', c') = id.putr(x, ()) & \\ \mathsf{in}\ (\mathsf{inl}\ x', \mathsf{inl}\ c') & v = \mathsf{inl}\ x \\ \mathsf{let}\ (y', c') = id.putr(y, ()) & \\ \mathsf{in}\ (\mathsf{inr}\ y', \mathsf{inr}\ c') & v = \mathsf{inr}\ y \end{cases}$$
$$= \begin{cases} (\mathsf{inl}\ x, \mathsf{inl}\ ()) & v = \mathsf{inl}\ x \\ (\mathsf{inr}\ y, \mathsf{inr}\ ()) & v = \mathsf{inr}\ y \end{cases}$$
$$= \left(v, \begin{cases} \mathsf{inl}\ () & v = \mathsf{inl}\ x \\ \mathsf{inr}\ () & v = \mathsf{inr}\ y \end{cases}\right)$$
$$id.putr(v, c) = (v, ())$$

Since $v = v$ and the complements are always related, this shows that

$$(id \oplus^f id).putr \sim_R id.putr.$$

For preservation of composition, we use the relation R defined by:

$$\{((\mathsf{inl}\ c_k, \mathsf{inl}\ c_\ell), \mathsf{inl}\ (c_k, c_\ell)) \mid c_k \in k.C, c_\ell \in \ell.C\} \cup$$
$$\{((\mathsf{inr}\ c_k, \mathsf{inr}\ c_\ell), \mathsf{inr}\ (c_k, c_\ell)) \mid c_k \in k'.C, c_\ell \in \ell'.C\}$$

Abbreviating $a = (k \oplus^f k'); (\ell \oplus^f \ell')$ and $b = (k; \ell) \oplus^f (k'; \ell')$, we can quickly see that $a.missing = (\mathsf{inl}\ k.missing, \mathsf{inl}\ \ell.missing)\ R\ \mathsf{inl}\ (k.missing, \ell.missing) = b.missing$. We will also show that $a.putr \sim_R b.putr$, eliding the similar proof that $a.putl \sim_R b.putl$.

Choose arbitrary $v \in X_0 + X_1, c_a \in a.C, c_b \in b.C$ such that $c_a\ R\ c_b$. There are many cases to consider, but two of them are representative of the remainder. In the first representative case, we have

$$\begin{aligned} v &= \mathsf{inl}\ x_0 \\ c_a &= (\mathsf{inl}\ c_k, \mathsf{inl}\ c_\ell) \\ c_b &= \mathsf{inl}\ (c_k, c_\ell) \end{aligned}$$

Then:

$$\begin{aligned} a.putr(v, c_a) &= \mathsf{let}\ (y_0, c'_k) = k.putr(x_0, c_k)\ \mathsf{in} \\ &\quad \mathsf{let}\ (z_0, c'_\ell) = \ell.putr(y_0, c_\ell)\ \mathsf{in} \\ &\quad (\mathsf{inl}\ z_0, (\mathsf{inl}\ c'_k, \mathsf{inl}\ c'_\ell)) \\ b.putr(v, c_b) &= \mathsf{let}\ (z_0, (c'_k, c'_\ell)) = (k; \ell).putr(x_0, (c_k, c_\ell))\ \mathsf{in} \\ &\quad (\mathsf{inl}\ z_0, \mathsf{inl}\ (c'_k, c'_\ell)) \\ &= \mathsf{let}\ (y_0, c'_k) = k.putr(x_0, c_k)\ \mathsf{in} \\ &\quad \mathsf{let}\ (z_0, c'_\ell) = \ell.putr(y_0, c_\ell)\ \mathsf{in} \\ &\quad (\mathsf{inl}\ z_0, \mathsf{inl}\ (c'_k, c'_\ell)) \end{aligned}$$

Since z_0, c'_k, c'_ℓ are computed identically in the two equations, the relation is preserved in this case.

In the second representative case, we have

$$\begin{aligned} v &= \mathsf{inl}\ x_0 \\ c_a &= (\mathsf{inr}\ c_{k'}, \mathsf{inr}\ c_{\ell'}) \\ c_b &= \mathsf{inr}\ (c_{k'}, c_{\ell'}) \end{aligned}$$

Then:

$$\begin{aligned}
a.putr(v, c_a) &= \mathsf{let}\ (y_0, c_k) = k.putr(x_0, k.missing)\ \mathsf{in} \\
&\quad\ \mathsf{let}\ (z_0, c_\ell) = \ell.putr(x_0, \ell.missing)\ \mathsf{in} \\
&\quad\ (\mathsf{inl}\ z_0, (\mathsf{inl}\ c_k, \mathsf{inl}\ c_\ell)) \\
b.putr(v, c_b) &= \mathsf{let}\ (z_0, c') = (k;\ell).putr(x_0, (k;\ell).missing)\ \mathsf{in} \\
&\quad\ (\mathsf{inl}\ z_0, \mathsf{inl}\ c') \\
&= \mathsf{let}\ (y_0, c_k) = k.putr(x_0, k.missing)\ \mathsf{in} \\
&\quad\ \mathsf{let}\ (z_0, c_\ell) = \ell.putr(y_0, \ell.missing)\ \mathsf{in} \\
&\quad\ (\mathsf{inl}\ z_0, \mathsf{inl}\ (c_k, c_\ell))
\end{aligned}$$

Again, since z_0, c_k, c_ℓ are computed identically in both equations, the relation is preserved. □

2.5.4 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 $\ell = bij_{f+g}$. The total relation $R \subset (Unit \times Unit) \times Unit$ is a witness that $k \equiv \ell$ and the total relation $R^f \subset (Unit + Unit) \times Unit$ 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.

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

The *putl* direction is similar. □

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

The types of these natural isomorphisms are:

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

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:

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

The five coherence conditions follow from coherence in SET, functoriality of *bij*, and Lemma 2.5.4. □

Unlike the product unit, there are no interesting lenses whose domain is the sum's unit, so this cannot be used to define the injection lenses; we have to do it by hand.

2.5.6 Definition [Injection lenses]:

$$\frac{x \in X}{\mathit{inl}_x \in X \leftrightarrow X + Y}$$

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

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

Proof of well-formedness: For PUTRL, we consider two cases: either the complement has the form $(x_c, \mathsf{inl}\ ())$ or the form $(x_c, \mathsf{inr}\ y)$.

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

Thus, in each case, the output value is equal to the input value and the complement is unaffected by the *putl*, as required by PUTRL.

To show PUTLR holds, we again consider two cases: either we start with $\mathsf{inl}\ x$ or $\mathsf{inr}\ y$.

$$\begin{aligned} putr(putl(\mathsf{inl}\ x, (x_c, y_c))) &= putr(x, (x, \mathsf{inl}\ ())) \\ &= (\mathsf{inl}\ x, (x, \mathsf{inl}\ ())) \\ putr(putl(\mathsf{inr}\ y, (x_c, y_c))) &= putr(x_c, (x_c, \mathsf{inr}\ y)) \\ &= (\mathsf{inr}\ y, (x_c, \mathsf{inr}\ y)) \end{aligned}$$

In both cases, the value output matches the value input and the complement remains unaffected by *putr*. □

As with the projection lenses for tensor products, we may ask whether the injection lenses for tensor sums are natural. If they were, we would expect a diagram like the following one to commute for all f:

$$\begin{array}{ccc} X & \xrightarrow{\quad f \quad} & Y \\ {\scriptstyle inl_x}\downarrow & & \downarrow{\scriptstyle inl_y} \\ X + Z & \xrightarrow{\quad f \oplus id \quad} & Y + Z \end{array}$$

Now, even if we carefully choose x and y to be related by f as we did for the projection lenses, this diagram may not commute. When running the *putr* function, the path along the top always invokes *f.putr*, whereas the path along the bottom may sometimes invoke *id.putr* instead; at that moment, the complements of f (on the top path) and $f \oplus id$ (on the bottom path) get out of synch. As we show in the following proposition this can be used to produce a subsequent observable difference, i.e., not only at the level of complements.

The situation with the forgetful sum is similar, but offers an additional way to desynchronize the two complements: when resetting f's complement along the bottom path to *missing*.

2.5.7 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:

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

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

We delay the proof that this lens is well-formed temporarily. 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 objects

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

The first two put objects in the list 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 objects 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. □

Proof of well-formedness: For completeness, we must also show that $count_x$ satisfies the lens laws.

PUTLR: There are two cases to consider. Both are simple calculations.

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

PUTRL: There are three cases to consider. For the first case, choose distinct $x \neq x'$.

$$\begin{aligned} putl(putr(x, (x', b, n))) &= putl(\mathsf{inr}\ (n+1), (x, \mathsf{true}, n+1)) \\ &= (x, (x, \mathsf{true}, n+1)) \end{aligned}$$

In the remaining cases, both the value and the complement round-trip exactly, which is even more than the PUTRL law requires.

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

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.

2.5.8 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:

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

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.

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

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

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

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

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

$$\begin{array}{rcl} union'_{XY} & = & bij_{(f+g);\alpha^+;(id+(\alpha^+)^{-1})}; \\ & & (id_X \oplus (union_{X\cap Y, X\cap Y} \oplus id_Y)); \\ & & bij_h \end{array}$$

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:

2.5.10 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 $X = Y$:

2.5.11 Lemma:

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

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

2.5.12 Definition [Retentive case lens]:

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

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

2.5.13 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 only consider mapping here, because in the next section we show 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.

2.5.14 Definition [Retentive list mapping lens]:

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

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

The map lens gives us the machinery we need to complete the first example in the introduction: simply define $e^\star = \mathsf{map}(e)$. Additionally, as we saw in §2.1, 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, state-based lenses [8, 16].

2.5.15 Definition [Forgetful list mapping lens]:

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

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

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:

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

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

2.6 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* [31].

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

2.6.1 Lists

Let $\mathit{fold} \in \mathit{Unit} + (X \times X^\star) \to X^\star$ be the bijection between "unfolded" lists and lists; fold takes $\mathsf{inl}\ ()$ to $\langle\rangle$ and $\mathsf{inr}\ (x, xs)$ to $x{:}xs$. Note that $\mathit{bij}_{\mathit{fold}} \in \mathit{Unit} + (X \times X^\star) \Longleftrightarrow X^\star$

is then a bijective arrow in the category LENS.

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

The function w here plays the role of a termination measure. We will be iterating $\ell.\mathit{putl}$, producing a stream of values of type Z, which we would like to guarantee eventually ends.

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

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

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 $F(Z) \to Z$. An arrow between F-algebras (Z, f) and (Z', f') is an arrow $u \in Z \to Z'$ such that $f; u = F(u); f'$. The F-algebras thus form a category themselves. An initial F-algebra is an initial object in that category (an initial object has exactly one arrow to each other object, and is unique up to isomorphism). F-algebras can be used to model a wide variety of inductive datatypes, including lists and various kinds of trees [41]. Using this terminology, Theorem 2.6.1.2 says that $\mathit{bij}_{\mathit{fold}}$ is an initial object in the subcategory consisting of those $T^\star_X$-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:

$$\mathit{Unit} + (X \times Y^\star) \xrightarrow{\mathit{id}_{\mathit{Unit}} \oplus (k \otimes \mathit{id}_{Y^\star})} \mathit{Unit} + (Y \times Y^\star) \xrightarrow{\mathit{bij}_{\mathit{fold}}} Y^\star$$

A suitable weight function is given by $w(ys) = \mathit{length}(ys)$. The induced lens $\mathit{It}(\ell) \in X^\star \Longleftrightarrow Y^\star$ 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 $\mathit{It}(\ell)$ is the sneakier variant of the lens $e^\star$. (Again, we are ignoring the important question of alignment here. A hand-written map lens could perform a more sophisticated alignment analysis to associate "similar" items in a sequence of puts and recover more appropriate data from the complement; the process described above results in a simple positional alignment scheme.)

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

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

by distributing the sum and unfolding the list type for f and by factoring the polynomial and folding the list type for g.[3]

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

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

Then we can create

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

$$\begin{array}{rcl} \ell & = & (\mathit{id}_{\mathit{Unit}} \oplus \mathit{bij}_f); \\ & & (\mathit{id}_{\mathit{Unit}} \oplus (\mathit{id}_{X_1^+ + X_2^+} \oplus \mathit{switch}_{X_1^+ \times X_2^+})); \\ & & \mathit{bij}_g \end{array}$$

A suitable weight function for ℓ is given by

$$w((xs_1, xs_2)) = \mathit{length}(xs_1) + \mathit{length}(xs_2).$$

The lens $\mathit{It}(\ell) \in (X_1 + X_2)^\star \Longleftrightarrow X_1^\star \times X_2^\star$ that we obtain from iteration partitions the input list in one direction and uses a stream of booleans from the state to put them

[3]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.

back in the right order in the other direction. Indeed, $It(\ell)$ is exactly the *partition* lens described in the introductory examples. Composing it with a projection yields a filter lens. (Alternatively, the filter lens could be obtained directly by iterating a slightly trickier ℓ.) Consequently, we now have the machinery we need to define *comp* from the introduction:

$$\begin{aligned} \mathit{filter} &= \mathit{partition}; \pi_{1\langle\rangle} \\ \mathit{comp} &= \mathit{filter}; \mathit{filter}^{op} \end{aligned}$$

Proof of 2.6.1.2: We define the lens $It(\ell)$ explicitly.

$$\frac{\ell \in T^{\star}_{X}(Z) \leftrightarrow Z \qquad \exists \text{ suitable } w}{It(\ell) \in X^{\star} \leftrightarrow Z}$$

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

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 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 of the *putr* of a one-element 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 $It(\ell).\mathit{putr}$ is clearly terminating because the first argument to the recursive call is always a shorter list; the recursive definition of $It(\ell).\mathit{putl}$ 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 $k \in X^\star \iff Z$ be another lens for which the diagram commutes—i.e., such that:

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

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:

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

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

$$\frac{(d, (d', c)) \in R \qquad k.putr(x{:}xs, d) = (z, d_1) \qquad k.putr(xs, d') = (z', d'_1) \qquad \ell.putr(\mathsf{inr}\ (x, z'), c) = (z'', c_1)}{(d_1, (d'_1, c_1)) \in R \ \wedge\ z = z''}$$

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

$$\frac{(d, (d', c)) \in R \qquad k.putl(z, d) = (x{:}xs, d_1)}{\begin{array}{rl} & \ell.putl(z, c) = (\mathsf{inr}\ (x, z'), c_1) \\ \wedge & k.putl(z', d') = (xs, d'_1) \\ \wedge & (d_1, (d'_1, c_1)) \in R \end{array}}$$

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:

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

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

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

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

Let

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

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

$$\begin{array}{rcl} It(\ell).putl(z, c{:}cs) & = & (x{:}xs, c'{:}cs') \\ (\mathsf{inr}\ (x, z'), c') & = & \ell.putl(z, c) \\ (xs, cs') & = & It(\ell).putl(z', cs). \end{array}$$

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. $\square$

2.6.1.3 Corollary [Hylomorphism]: 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:

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

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

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 [31]. 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.

2.6.2 Other Datatypes

Analogs of Theorem 2.6.1.2 and Corollary 2.6.1.3 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 §2.7), 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.

Iterators with multiple arguments The list iterator allows us to define a lens between $X^\star$ and some other set Z, but Theorem 2.6.1.2 cannot be directly used to define a lens between $X^\star \times Y$ and Z (think of Y as modeling parameters). In standard functional programming, a map from $X^\star \times Y$ to Z is tantamount to a map from $X^\star$ to $Y \to Z$, so iteration with parameters is subsumed by the parameterless

case. Unfortunately, LENS does not seem to have the function spaces required to play this trick.

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

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

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

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

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

We can use a version of Corollary 2.6.1.3 for this data structure to implement tail recursive constructions. Consider, for instance, the $T^{\star}_{\mathit{Unit},X^{\star}}$-algebra $k : X^{\star} + X^{\star} \times X^{\star} \leftrightarrow X^{\star} \times X^{\star}$ where

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

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

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

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

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

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

2.7 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 [19] or an *analytic functor* in the terminology of Joyal [27]. (These structures are also related to the *shapely types* of Jay and Cockett [25].)

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

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

The arrow part of $F_{I,B}$ maps a function $u \in X \to Y$ to a function $F_{I,B}(u) \in F_{I,B}(X) \to F_{I,B}(Y)$ given by $(i, f) \mapsto (i, f; u)$.

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

2.7.2 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 $B(i \leq j) \in B(i) \to B(j)$ 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 arrow 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.

2.7.3 Definition [Container mapping lens]:

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

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

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

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

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}. □

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

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

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

$$t_k\ R_{I,B}\ t_\ell$$

$$\begin{array}{rcl} F_{I,B}(k).putr((i,f), t_k) & = & ((i, f_k), t'_k) \\ F_{I,B}(\ell).putr((i,f), t_\ell) & = & ((i, f_\ell), t'_\ell) \end{array}$$

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 $t'_\ell(j)(b) = t_\ell(j)(b)$. □

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 $F_{I,B}(k; \ell) \equiv F_{I,B}(k); F_{I,B}(\ell)$. □

For the case of lists, this mapping lens coincides with the retentive map that we obtained from the iterator in §2.6. In general, two pieces of data synchronized by one of these mapping lenses will have exactly the same shape; any shape change to one of the sides will be precisely mirrored in the other side. For example, the tree version

of this lens will transport the deletion of a node by deleting the node in the same position on the other side. We believe it should also be possible to define a forgetful version where the complement is just $F_{I,B}(\ell.C)$.

The notion of *combinatorial species* provides an alternative to the container framework. One of their attractions is that there are species corresponding to containers whose $B(i) \to X$ family is quotiented by some equivalence relation; we can obtain multisets in this way, for example. However, we have not explored this generalization in the case of lenses, because it is then not clear how to match up positions.

2.8 Asymmetric Lenses as Symmetric Lenses

The final step in our investigation is to formalize the connection between symmetric lenses and the more familiar asymmetric ones, and to show how known constructions on asymmetric lenses 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 §2.1, with *get*, *put*, and *create* components).

2.8.1 Definition [Symmetrization]: Every asymmetric lens can be embedded in a symmetric one.

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

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

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

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:

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

PutLR:

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

□

2.8.2 Definition [Asymmetric lenses]: Here are several useful asymmetric lenses (based on string lenses from [8]).

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

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

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

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

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

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

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

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

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

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

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

$$\begin{array}{rcl}
get(\langle x_1, \ldots, x_n \rangle) & = & \langle \ell.get(x_1), \ldots, \ell.get(x_n) \rangle \\
put(\langle y_1, \ldots, y_m \rangle, \langle x_1, \ldots, x_n \rangle) & & \\
& = & \langle x'_1, \ldots, x'_m \rangle \\
\text{where } x'_i & = & \begin{cases} \ell.put(y_i, x_i) & i \leq \min(m, n) \\ \ell.create(y_i) & n+1 \leq i \end{cases} \\
create(\langle y_1, \ldots, y_n \rangle) & = & \langle \ell.create(y_1), \ldots, \ell.create(y_n) \rangle
\end{array}$$

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

$$copy_X^{sym} \equiv id_X \tag{2.8.1}$$
$$(k; \ell)^{sym} \equiv k^{sym}; \ell^{sym} \tag{2.8.2}$$
$$aconst_x^{sym} \equiv term_x \tag{2.8.3}$$
$$(k \cdot \ell)^{sym} \equiv k^{sym} \otimes \ell^{sym} \tag{2.8.4}$$
$$(k|\ell)^{sym} \equiv (k^{sym} \oplus^f \ell^{sym}); union \tag{2.8.5}$$
$$(\ell^\star)^{sym} \equiv \mathsf{map}^f(\ell^{sym}) \tag{2.8.6}$$

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. Defining f to be the identity function $f(x) = x$, the singleton relation $f\ R\ ()$ witnesses the equivalence. Since $a.missing(x) = x$, we have $a.missing\ R\ b.missing$. Furthermore:

$$\begin{array}{rcl}
a.putr(x, f) & = & (x, x' \mapsto copy_X.put(x', x)) \\
& = & (x, x' \mapsto x') \\
& = & (x, f) \\
b.putr(x, ()) & = & (x, ()) \\
a.putl(x, f) & = & (f(x), x' \mapsto copy_X.put(x', x)) \\
& = & (x, f) \\
b.putl(x, ()) & = & (x, ())
\end{array}$$

   This establishes that $a.putr \sim_R b.putr$ and that $a.putl \sim_R b.putl$.

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 $a.putr(x, f_{k\ell})$.

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

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

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

It's now clear that

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

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 $f_{k\ell}\ R\ (f_k, f_\ell)$.

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

Similarly,

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

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:

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

Observing that

$$\begin{array}{rll} k.get(f_k(f_\ell(z))) = f_\ell(z) & & \text{because } f_k \in k^{sym}.C \\ f_k(f_\ell(z)) = f_{k\ell}(z) & & \text{because } f_{k\ell}\ R\ (f_k, f_\ell), \end{array}$$

that last line becomes

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

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

3. The relation

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

witnesses the equivalence. Since $a.missing = () \mapsto x$ and $b.missing = x$, we see $a.missing\ R\ b.missing$.

To show that $a.putr \sim_R b.putr$, choose arbitrary $x, c \in X$ and define $f_c(()) = c$:

$$\begin{array}{c} a.putr(x, f_c) = ((), () \mapsto x) \\ b.putr(x, c) = ((), x) \end{array}$$

These clearly satisfy $() = ()$ and $(() \mapsto x)\ R\ x$, so we can conclude that $a.putr \sim_R b.putr$.

To show that $a.putl \sim_R b.putl$, choose arbitrary $c \in X$ and define $f_c(()) = c$ as

before. Then:

$$\begin{aligned} a.putl((), f_c) &= (f_c(()), u \mapsto aconst_x.put(u, f_c(()))) \\ &= (c, u \mapsto c) \\ &= (c, () \mapsto c) \\ b.putl((), c) &= (c, c) \end{aligned}$$

These again clearly satisfy $c = c$ and $(() \mapsto c)\ R\ c$, so $b.putl \sim_R b.putl$.

4. The relation

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

witnesses the equivalence. We can compute

$$\begin{aligned} a.missing &= (y, w) \mapsto (k.create(y), \ell.create(w)) \\ b.missing &= (y \mapsto k.create(y), w \mapsto \ell.create(w)), \end{aligned}$$

so clearly $a.missing\ R\ b.missing$.

Let us show that $a.putr \sim_R b.putr$. Choose $(x, z) \in X \times Z$ and arbitrary $f_{k\ell}, f_k, f_\ell$ (we will not need the assumption that $f_{k\ell}\ R\ (f_k, f_\ell)$). Then:

$$\begin{aligned} a.putr((x, z), f_{k\ell}) &= ((k.get(x), \ell.get(z)), \\ &\qquad (y, w) \mapsto (k.put(y, x), \ell.put(w, z))) \\ b.putr((x, z), (f_k, f_\ell)) &= ((k.get(x), \ell.get(z)), \\ &\qquad (y \mapsto k.put(y, x), w \mapsto \ell.put(w, z)) \end{aligned}$$

It's clear that the first elements of these tuples are equal, and the second elements are just as clearly related by R, so it is indeed true that $a.putr \sim_R b.putr$.

Similarly, choose $(y, w) \in Y \times W$ and suppose $f_{k\ell}\ R\ (f_k, f_\ell)$ – which in particular means that $f_{k\ell}(y, w) = (f_k(y), f_\ell(w))$. Then we can define a few things:

$$\begin{aligned} (v_a, f_a) &= a.putl((y, w), f_{k\ell}) \\ &= \mathsf{let}\ (x, z) = f_{k\ell}(y, w)\ \mathsf{in} \\ &\quad ((x, z), (y', w') \mapsto (k.put(y', x), \ell.put(w', z)) \\ &= \mathsf{let}\ (x, z) = (f_k(y), f_\ell(w))\ \mathsf{in} \\ &\quad ((x, z), (y', w') \mapsto (k.put(y', x), \ell.put(w', z)) \\ &= ((f_k(y), f_\ell(w)), \\ &\quad (y', w') \mapsto (k.put(y', f_k(y)), \ell.put(w', f_\ell(w)))) \end{aligned}$$

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

So $v_a = v_b$ and $f_a\ R\ f_b$ – that is, $a.putl \sim_R b.putl$.

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

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

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

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

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

$$\begin{aligned} \mathsf{tag}(\mathsf{inl}\ f_k) &= \mathsf{false} \\ \mathsf{tag}(\mathsf{inr}\ f_\ell) &= \mathsf{true} \end{aligned}$$

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 whenever $f([y_1, \ldots, y_n]) = [x_1, \ldots, x_m]$ we can conclude $m = n$ and $\ell.get(x_i) = y_i$.

   The complement $\mathsf{map}^f(\ell^{sym}).C$ 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 $\mathsf{map}^f(\ell^{sym})$ 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 $\mathsf{map}^f(\ell^{sym})$ complement is $[g_1, \ldots, g_m]$ where $g_i(y) = \ell.put(x_i, y)$. These are clearly related again. □

We suspect that there might be an asymmetric *fold* construction similar to our iteration lens above satisfying an equivalence like

$$fold(\ell)^{sym} \equiv It(\ell^{sym}),$$

but have not explored this carefully.

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, nor do either of the example lenses presented in the introduction. However, we *can* characterize symmetric lenses in terms of asymmetric ones in a slightly more elaborate way.

2.8.4 Theorem [Lenses are spans]: Given any arrow ℓ of LENS, there are asymmetric lenses k_1, k_2 such that

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

This suggests that the category LENS could be constructed from spans in ALENS. A full account of the machinery necessary to realize this approach is given by Johnson and Rosebrugh [26]. It is quite involved for two reasons: first, composition of spans is typically given via a pullback construction, but pullbacks in the appropriate category do not always exist, and second, one must develop a span-based analog for our lens equivalence to retain associativity of composition.

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

2.8.5 Definition [Asymmetrization]: We can view a symmetric lens as a pair of asymmetric lenses joined "tail to tail" whose common domain is consistent triples. For any lens $\ell \in X \leftrightarrow Y$, define

$$S_\ell = \{(x, y, c) \in X \times Y \times \ell.C \mid \ell.putr(x, c) = (y, c)\}.$$

Now define:

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

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

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

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

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

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

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

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

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. □

Proof of 2.8.4: 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 $\ell_r.putr(x, f_c)$:

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

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. □

2.9 Conclusion

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. The category of symmetric lenses 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 data types both inductively and as "containers" and found the category of symmetric lenses to support powerful mapping and folding constructs. In the next chapter, we will extend this approach to address performance—significantly reducing the amount of information a lens must process—and alignment—giving precise details about the correspondence between old and new copies of a complex repository.

Chapter 3

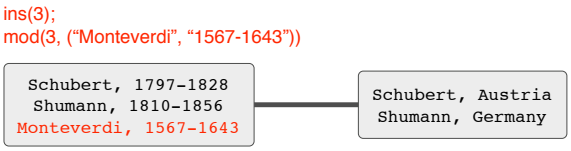
Edit Lenses

3.1 Overview

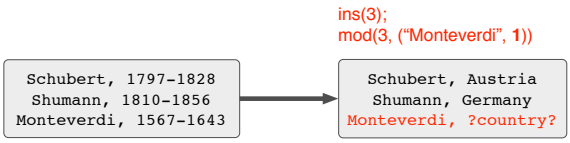
Before diving into the technicalities of edit lenses, let's take a brief tour of the main ideas via some examples. Figure 3.1 demonstrates a simple use of edit lenses to synchronize two repositories. In part (a), we see the initial repositories, which are in a synchronized state. On the left, the repository is a list of records describing composers' birth and death years; on the right, a list of records describing the same composers' countries of origin. In part (b), the user interacting with the left-hand repository decides to add a new composer, Monteverdi, at the end of the list. This change is described by the edit script ins(3); mod(3, ("Monteverdi", "1567-1643")). The script says to first *insert* a dummy record at index three, then *modify* this record by replacing the left field with "Monteverdi" and replacing the right field with "1567-1643". (One could of course imagine other edit languages where the insertion would be done in one step. We represent it this way because this is closer to how our generic "container mapping" combinator in §3.4 will do things.) The lens connecting the two repositories now converts this edit script into a corresponding edit script that adds Monteverdi to the right-hand repository, shown in part (c): ins(3); mod(3, ("Monteverdi", **1**)). Note that the translated mod command overwrites the name component but leaves the country component with its default value, "?country?". This is the best it can do, since the edit was in the left-hand repository, which doesn't mention countries. Later, an eagle-eyed editor notices the missing country information and fills it in, at the same time correcting a spelling error in Schumann's name, as shown in (d). In part (e), we see that the lens discards the country information when translating the edit from right to left, but propagates the spelling correction.

Of course, a particular new repository state can potentially be achieved by many different edits, and these edits may be translated differently. Consider part (f) of Figure 3.1, where the left-hand repository ends up with a row for Monteverdi at the beginning of the list, instead of at the end. Two edit scripts that achieve this effect are shown. The upper script deletes the old Monteverdi record and inserts a brand new

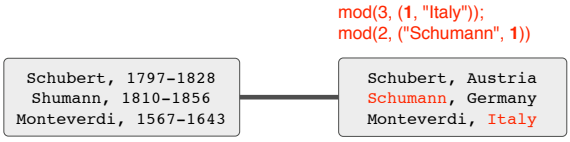
(a) initial repositories



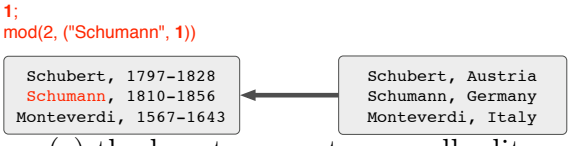
(b) a new composer is added to one repository



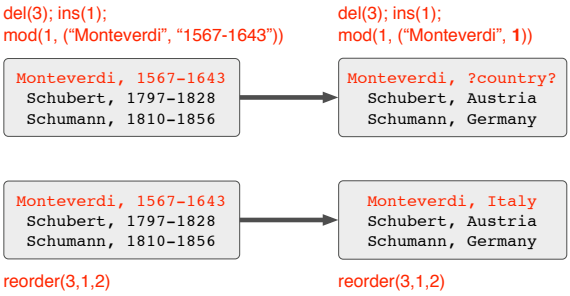
(c) the lens adds the new composer to the other repository



(d) the curator makes some corrections



(e) the lens transports a small edit



(f) two different edits with the same effect on the left

Figure 3.1: A simple (complement-less) edit lens in action.

one (which happens to have the same data) at the top; the lower script rearranges the order of the list. The translation of the upper edit leaves Monteverdi with a default country, while the lower edit is translated to a rearrangement, preserving all the information associated with Monteverdi.

We do not address the question of where these edits come from or who decides, in cases like part (f), which of several possible edits is intended. As argued in [6], answers to these questions will tend to be intertwined with the specifics of particular editing and/or diffing tools and will tend to be messy, heuristic, and domain-specific—unpromising material for a foundational theory. Rather, our aim is to construct a theory that shows how edits, however generated, can be translated between repositories of different shapes.

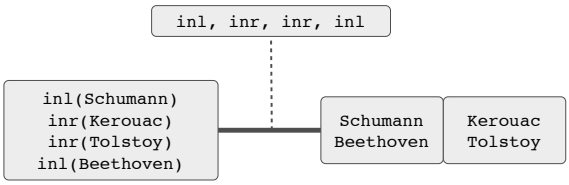
Abstractly, the lens we are discussing maps between structures of the form $(X \times Y)^\star$ and ones of the form $(X \times Z)^\star$, where X is the set of composer names, Y the set of date strings, and Z the set of countries. We want to build it compositionally—that is, the whole lens should have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

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

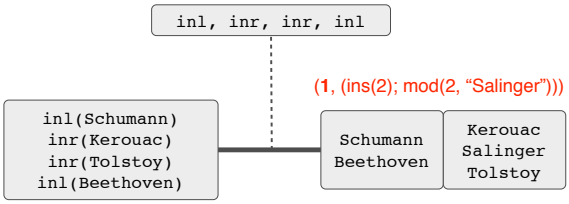
Our lens $\ell^\star$ will consist of two components—one for transporting edits from the left side to the right, written $(\ell^\star).\Rrightarrow \in \partial(X \times Y)^\star \to \partial(X \times Z)^\star$,[1] and another for transporting edits from right to left, written $(\ell^\star).\Lleftarrow \in \partial(X \times Z)^\star \to \partial(X \times Y)^\star$.

We sometimes need lenses to have a little more structure than this simple example suggests. To see why, consider defining a *partitioning* lens p between the sets $\partial((X + Y)^\star)$ and $\partial(X^\star \times Y^\star)$. Figure 3.2 demonstrates the behavior of this lens. In part (a), we show the original repositories: on the left, a single list that intermingles authors and composers (with inl/inr tags showing which is which), and on the right a pair of homogeneous (untagged) lists, one for authors and one for composers. Now consider

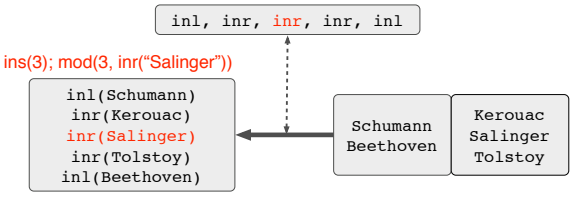
[1] The symbol $\Rrightarrow$ is pronounced "put an edit through the lens from left to right", or just "put right". It is the edit-analog of the *putr* function of the state-based symmetric lenses in Chapter 2 and the *put* function of the state-based asymmetric lenses in [8, 15].



(a) the initial repositories: a tagged list of composers and authors on the left; a pair of lists on the right; a complement storing just the tags



(b) an element is added to one of the partitions



(c) the complement tells how to translate the index

Figure 3.2: A lens with complement.

an edit, as in (b), that inserts a new element somewhere in the author list on the right. It is clear that we should transport this into an insertion on the left repository, but where, exactly, should we insert it? If the $\Leftarrow$ function is given just an insertion edit for the homogeneous author list and nothing else, there is no way it can translate this edit into a sensible position in the combined list on the left, since it doesn't know how the lists of authors and composers are interleaved on the left.

The solution is to store a small list, called a *complement*, off to the side, recording the *tags* (inl or inr) from the original, intermingled list, and pass this list as an extra argument to translation. We then enrich the types of the edit translation functions to accept a complement and return a new complement, so that

$$p.\Rightarrow \in \partial((X + Y)^\star) \times C \to \partial(X^\star \times Y^\star) \times C$$

and

$$p.\Leftarrow \in \partial(X^\star \times Y^\star) \times C \to \partial((X + Y)^\star) \times C.$$

Part (c) demonstrates the use (and update) of the complement when translating the insertion.

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

3.2 Edit Lenses

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

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

$$x \cdot_M (y \cdot_M z) = (x \cdot_M y) \cdot_M z$$
$$\mathbf{1}_M \cdot_M x = x = x \cdot_M \mathbf{1}_M.$$

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

The unit element represents a "change nothing" edit. Multiplication of edits corresponds to packaging up multiple edits into a single one representing their combined effects (this might be useful, for example, for offline editing).

Modeling edits as monoid elements gives us great flexibility in concrete representations. The simplest edit language is a free monoid whose elements are just words over some set of primitive edits and whose multiplication is concatenation. However, it may be useful to put more structure on edits, either (a) to allow more compact representations or (b) to capture the intuition that edits to different parts of a structure do not interfere with each other and can thus be applied in any order. We will see an example of (b) in §3.5. For a simple example of (a), recall from §3.1 that, for every set X, we can form an *overwrite* monoid where the edits are just the elements of X together with a fresh unit element—i.e., edits can be represented as elements of the disjoint union $\mathit{Unit} + X$. Combining two edits in this monoid simply drops the second (unless the first is the unit):

$$\mathsf{inl}(()) \cdot e = e \qquad \mathsf{inr}(x) \cdot e = \mathsf{inr}(x)$$

These equations allow this edit language to represent an arbitrarily long sequence of updates using a single element of X (and, *en passant*, to recover state-based lenses as a special case of edit lenses). The monoid framework can also accommodate more abstract notions of edit. For example, the set of all total (respectively, partial) functions from a set X to itself forms a monoid, where the multiplication operation is function composition (and the unit is the identity function). This is essentially the form of edits considered by Stevens [38]. We mostly focus on the simple case where edit languages are free monoids. §3.5 considers how additional laws can be added to the product and sum lens constructions (laws for lists and general containers are left for future work).

3.2.2 Definition: Given monoids M and N, a *monoid homomorphism* is a function $h \in M \to N$ that satisfies two laws:

$$h(\mathbf{1}_M) = \mathbf{1}_N$$
$$h(m \cdot_M m') = h(m) \cdot_N h(m')$$

Monoid homomorphisms are structure-preserving maps, and we will see many specializations of this definition below. An example of a homomorphism in the case where the two monoids M and N are both free monoids is any operation that acts pointwise on the elements of the lists. Having defined monoids, which model descriptions of edits, we will now model the operation that performs an edit on a particular object.

3.2.3 Definition: Given a monoid M and a set X, a *monoid action* of M on X is a monoid homomorphism from M to the monoid of partial functions $X \rightharpoonup X$. Unrolling this definition, this means an action is a partial function $\odot \in M \to (X \rightharpoonup X)$, or equivalently, $\odot \in M \times X \rightharpoonup X$, satisfying two laws:

$$\mathbf{1} \odot x = x$$
$$(m \cdot n) \odot x = m \odot (n \odot x)$$

We use $X \rightharpoonup Y$ for the space of partial functions with domain X and codomain Y. If $f \in X \rightharpoonup Y$, we will write $f(x)\downarrow$ to mean that f is defined at x. As with monoid multiplication, we often elide the monoid action symbol, writing mx for $m \odot x$. In standard mathematical terminology, a monoid action in our sense might instead be called a "partial monoid action", but since we always work with partial actions we find it convenient to drop the qualifier.

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

Why deal with failure explicitly, rather than keeping edit application total and simply defining our monoid actions so that applying an edit in a state where it is not appropriate yields the same state again (or perhaps some other state)? One reason is that it seems natural to directly address the fact that some edits are not applicable in some states, and to have a canonical outcome in all such cases. A more technical reason is that, when we work with monoids with nontrivial equations, making inapplicable edits behave like the identity is actually wrong.[2]

However, although the framework allows for the possibility of edits failing, we still want to know that the edits produced by our lenses will never actually fail when applied to repository states arising in practice. This requirement, corresponding to the *totality* property of previous presentations of lenses [15], is formalized in Theorem 3.2.9. In general, we adopt the design principle that partiality should be kept to a minimum; this simplifies the definitions.

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

3.2.4 Definition: A *module* is a tuple $\langle X, \mathit{init}_X, \partial X, \odot_X \rangle$ comprising a set X, an element $\mathit{init}_X \in X$, a monoid ∂X, and a monoid action $\odot_X$ of ∂X on X.

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

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

We will use modules to represent the structures connected by lenses. Before coming to the definition of lenses, however, we need one last ingredient: the notion of a *stateful homomorphism* between monoids. As we saw in §3.1, there are situations where the information in an edit may be insufficient to determine how it should be translated—we may need to know something more about how the two structures correspond. The exact nature of the extra information needed varies according to the lens. To give lenses a place to store such auxiliary information, we follow Chapter 2 and allow the edit-transforming components of a lens (the $\Rrightarrow$ and $\Lleftarrow$ functions) to take a *complement* as an extra input and return an updated complement as an extra output.

3.2.5 Definition: Given a monoid M and a *complement set* C, one can define the *stateful monoid* $State_C(M)$ whose elements are functions $C \to M \times C$. The unit and multiplication are given by

$$\begin{aligned} \mathbf{1}(c) &= (\mathbf{1}, c) \\ (m \cdot n)(c) &= \mathsf{let}\ (m', c') = m(c) \\ &\qquad\quad (n', c'') = n(c') \\ &\quad\ \mathsf{in}\ (m' \cdot n', c'') \end{aligned}$$

Functional programmers will recognize this monoid's multiplication as a lifting of the underlying monoid's multiplication into the state monad (and likewise the unit is a lifting of the underlying unit).

3.2.6 Definition: Given monoids M and N and a complement set C, a *stateful monoid homomorphism* from M to N over C is a monoid homomorphism $h \in M \to State_C(N)$. In the following, we will typically treat h as if it were a two-argument function; so, unrolling the definition of homomorphism, h satisfies two laws:

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

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

The intended usage of an edit lens is as follows. There are two users, one holding an element of X the other one an element of Y. Initially, they hold $init_X$ and $init_Y$, respectively, and the lens is initialized with complement $\ell.missing$. The users then perform actions and propagate them across the lens. An action consists of producing an edit $\mathrm{d}x$ (or $\mathrm{d}y$), applying it to one's current repository x (resp. y), putting the edit through the lens to obtain an edit $\mathrm{d}y$ (resp. $\mathrm{d}x$), and asking the user on the other side to apply $\mathrm{d}y$ ($\mathrm{d}x$) to their repository. In the process, the internal state c of the lens is updated to reflect the new correspondence between the two repositories.

We further assume there is some *consistency* relation K between X, Y, and C, which describes the "synchronized states" of the repositories and complement. This gives us a natural way to state the totality requirement discussed above: if we start in a consistent state, make a successful edit (one that does not fail at the initiating side), and put it through the lens, the resulting edit is guaranteed (a) to be applicable on the receiving side and (b) to lead again to a consistent state. We make no guarantees about edits that fail at the initiating side: these should not be put through the lens.

3.2.7 Definition: A *symmetric edit lens* between modules X and Y consists of a complement set C, a distinguished element $\mathit{missing} \in C$, two stateful monoid homomorphisms

$$\begin{array}{rcl} \Rrightarrow & \in & \partial X \times C \to \partial Y \times C \\ \Lleftarrow & \in & \partial Y \times C \to \partial X \times C \end{array}$$

and a ternary *consistency relation* $K \subseteq |X| \times C \times |Y|$ such that

- $(\mathit{init}_X, \mathit{missing}, \mathit{init}_Y) \in K$;
- if $(x, c, y) \in K$ and $\mathrm{d}x\ x$ is defined and $\Rrightarrow(\mathrm{d}x, c) = (\mathrm{d}y, c')$, then $\mathrm{d}y\ y$ is also defined and $(\mathrm{d}x\ x, c', \mathrm{d}y\ y) \in K$;
- if $(x, c, y) \in K$ and $\mathrm{d}y\ y$ is defined and $\Lleftarrow(\mathrm{d}y, c) = (\mathrm{d}x, c')$, then $\mathrm{d}x\ x$ is also defined and $(\mathrm{d}x\ x, c', \mathrm{d}y\ y) \in K$.[3]

Since symmetric edit lenses are the main topic of this chapter, we will generally write "edit lens" or just "lens" for these, deploying additional adjectives to talk about other variants such as the state-based symmetric lenses of Chapter 2. Similarly, we will co-opt the notation of the previous chapter, reusing many component names, $\leftrightarrow$ for the type of edit lenses, $\equiv$ for lens equivalence, and so on. When it is important to differentiate, we will use a subscript s for state-based concepts, as in $\underset{s}{\leftrightarrow}$ or $\equiv_s$.

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

3.2.8 Definition: Let $\ell \in X \leftrightarrow Y$ be a lens. A *dialogue* is a sequence of edits—a word in $(\partial X + \partial Y)^\star$. The partial function

$$\ell.\mathit{run} \in (\partial X + \partial Y)^\star \rightharpoonup X \times \ell.C \times Y$$

is defined by:

$$\overline{\ell.\mathit{run}(\langle\rangle) = (\mathit{init}_X, \ell.\mathit{missing}, \mathit{init}_Y)}$$

[3]One might consider a more general format with "creation" operations $\mathit{creater} \in X \to Y \times C$ and symmetrically $\mathit{createl}$. This format actually arises as a special case of the one above by choosing the edit monoids to include operations of the form $\mathrm{set}(x)$ for $x \in X$, with action $\mathrm{set}(x) \odot x' = x$. One can then define $\mathit{creater}(x, c) = \Rrightarrow(\mathrm{set}(x), c)$.

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

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

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

Proof: By induction on w we can easily show that $(x, c, y) \in \ell.K$. The claim then follows from the axioms for lenses. □

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

One technical issue arising from the definition of edit lenses is that the hidden complements cause many important laws—like associativity of composition—to hold only up to *behavioral equivalence*. This phenomenon was also observed in §2.2 for the case of symmetric state-based lenses, and the appropriate behavioral equivalence for edit lenses is a natural refinement of the one used there (taking the consistency relations into account).

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

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

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

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

3.2.11 Theorem: Lenses $k, \ell : X \leftrightarrow Y$ are equivalent iff there exists a relation $S \subseteq X \times k.C \times \ell.C \times Y$ such that

- $(init_X, k.missing, \ell.missing, init_Y) \in S$;
- if $(x, c, d, y) \in S$ and $\mathrm{d}x\ x$ is defined, then if $(\mathrm{d}y_1, c') = k.\Rrightarrow(\mathrm{d}x, c)$ and $(\mathrm{d}y_2, d') = \ell.\Rrightarrow(\mathrm{d}x, d)$, then $\mathrm{d}y_1 = \mathrm{d}y_2$ and $(\mathrm{d}x\ x, c', d', \mathrm{d}y_1\ y) \in S$; and
- analogously for $\Lleftarrow$.

Proof: For the "if" direction we prove by induction on dialogues that if $k.run(w)$ is defined then so is $\ell.run(w)$ and vice versa and if $k.run(w) = (x, c, y)$ and $\ell.run(w) = (x', d, y')$ then $x = x'$ and $y = y'$ and $(x, c, d, y) \in S$. For the converse we define $(x, c, d, y) \in S$ iff there exists a dialogue w such that $k.run(w) = (x, c, y)$ and $\ell.run(w) = (x, d, y)$ □

3.2.12 Theorem: Lens equivalence is an equivalence relation.

Proof: Reflexivity: the set $\{(x, c, c, y) \mid (x, c, y) \in \ell.K\}$ witnesses the equivalence $\ell \equiv \ell$ for any ℓ.

Symmetry: if the set S witnesses the equivalence $k \equiv \ell$, then the set $\{(x, d, c, y) \mid (x, c, d, y) \in S\}$ witnesses the equivalence $\ell \equiv k$.

Transitivity: if S witnesses $j \equiv k$ and T witnesses $k \equiv \ell$, then

$$\{(x, c, e, y) \mid \exists d.(x, c, d, y) \in S \wedge (x, d, e, y) \in T\}$$

witnesses $j \equiv \ell$. The verification is straightforward. □

3.3 Edit Lens Combinators

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

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

3.3.1 Definition [Identity]:

$$id_X \in X \leftrightarrow X$$

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

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

3.3.2 Lemma: $id.\Rrightarrow$ and $id.\Lleftarrow$ are stateful homomorphisms, and the relation $id.K$ is preserved.

Proof: Showing that $\Rrightarrow$ is a homomorphism involves showing that $id.\Rrightarrow(\mathbf{1}, ()) = (\mathbf{1}, ())$, which is direct, and that if $id.\Rrightarrow(\mathrm{d}x, c) = (\mathrm{d}y, c')$ and $id.\Rrightarrow(\mathrm{d}x', c') = (\mathrm{d}y', c'')$, then $id.\Rrightarrow(\mathrm{d}x'\mathrm{d}x, c) = (\mathrm{d}y'\mathrm{d}y, c'')$. Since $c = c' = c'' = ()$, it follows directly that $\mathrm{d}y = \mathrm{d}x$ and $\mathrm{d}y' = \mathrm{d}x'$, so the final claim is true. A similar argument shows that $\Lleftarrow$ is a homomorphism.

To show that K is preserved, choose a consistent triple $(x, (), x)$ and observe that $\Rrightarrow(\mathrm{d}x, ()) = (\mathrm{d}x, ())$ results in another consistent triple $(\mathrm{d}x\ x, (), \mathrm{d}x\ x)$. A similar argument for $\Lleftarrow$ applies. □

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

3.3.3 Definition [Composition]:

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

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

3.3.4 Lemma: Given that k and ℓ are lenses, this construction defines a lens:

- $\Rrightarrow$ and $\Lleftarrow$ are stateful monoid homomorphisms,
- relation K is preserved, and

- it respects lens equivalence: if $k \equiv k'$ and $\ell \equiv \ell'$, then $k;\ell \equiv k';\ell'$.

Proof:

$\Rrightarrow$ is a stateful monoid homomorphism. Since $k.\Rrightarrow$ and $\ell.\Rrightarrow$ are homomorphisms, we know that

$$k.\Rrightarrow(\mathbf{1}, c_k) = (\mathbf{1}, c_k)$$
$$\ell.\Rrightarrow(\mathbf{1}, c_\ell) = (\mathbf{1}, c_\ell)$$

and hence that

$$\Rrightarrow(\mathbf{1}, (c_k, c_\ell)) = (\mathbf{1}, (c_k, c_\ell)).$$

Choosing arbitrary $\mathrm{d}x, \mathrm{d}x', c_k, c_\ell$, we can define

$$(\mathrm{d}y, c_k') = k.\Rrightarrow(\mathrm{d}x, c_k)$$
$$(\mathrm{d}y', c_k'') = k.\Rrightarrow(\mathrm{d}x', c_k')$$
$$(\mathrm{d}z, c_\ell') = \ell.\Rrightarrow(\mathrm{d}y, c_\ell)$$
$$(\mathrm{d}z', c_\ell'') = \ell.\Rrightarrow(\mathrm{d}y', c_\ell')$$

and observe that since $k.\Rrightarrow$ and $\ell.\Rrightarrow$ are homomorphisms, we then know:

$$k.\Rrightarrow(\mathrm{d}x'\mathrm{d}x, c_k) = (\mathrm{d}y'\mathrm{d}y, c_k'')$$
$$\ell.\Rrightarrow(\mathrm{d}y'\mathrm{d}y, c_\ell) = (\mathrm{d}z'\mathrm{d}z, c_\ell'')$$

We can now calculate

$$(k;\ell).\Rrightarrow(\mathrm{d}x, (c_k, c_\ell)) = (\mathrm{d}z, (c_k', c_\ell'))$$
$$(k;\ell).\Rrightarrow(\mathrm{d}x', (c_k', c_\ell')) = (\mathrm{d}z', (c_k'', c_\ell''))$$
$$(k;\ell).\Rrightarrow(\mathrm{d}x'\mathrm{d}x, (c_k, c_\ell)) = (\mathrm{d}z'\mathrm{d}z, (c_k'', c_\ell''))$$

as necessary.

$\Lleftarrow$ is a stateful monoid homomorphism. The argument is very similar to the above.

The relation K is respected. The triple $(\mathit{init}_X, (k.\mathit{missing}, \ell.\mathit{missing}), \mathit{init}_Z)$ is in K because we can choose $y = \mathit{init}_Y$ and observe that $(\mathit{init}_X, k.\mathit{missing}, \mathit{init}_Y) \in k.K$ and $(\mathit{init}_Y, \ell.\mathit{missing}, \mathit{init}_Z) \in \ell.K$.

Next, consider consistent triple $(x, (c_k, c_\ell), z)$ and some particular y for which $(x, c_k, y) \in k.K$ and $(y, c_\ell, z) \in \ell.K$. (Such a y is guaranteed to exist by the definition of K.) Take $\mathrm{d}x$ for which $\mathrm{d}x\ x$ is defined and define:

$$(\mathrm{d}y, c_k') = k.\Rrightarrow(\mathrm{d}x, c_k)$$
$$(\mathrm{d}z, c_\ell') = \ell.\Rrightarrow(\mathrm{d}y, c_\ell)$$

By consistency of k, we know $\mathrm{d}y\ y$ is defined, and hence by consistency of ℓ we also

know $\mathrm{d}z\ z$ is defined. Furthermore, $(\mathrm{d}x\ x, c_k, \mathrm{d}y\ y) \in k.K$ and $(\mathrm{d}y\ y, c_\ell, \mathrm{d}z\ z) \in \ell.K$, and hence $\mathrm{d}y\ y$ is a witness to the fact that $(\mathrm{d}x\ x, (c_k, c_\ell), \mathrm{d}z\ z) \in (k;\ell).K$, as needed. A similar argument shows that $\Leftarrow$ respects the consistency relation.

The combinator respects lens equivalence. Suppose for simplicity that k and k' are identical (the general case then follows by symmetry and transitivity of $\equiv$). Using Theorem 3.6.2 assume furthermore that $\ell \equiv \ell' : X \leftrightarrow Y$ by virtue of relation $S \subseteq X \times C \times C' \times Y$ assuming that C and C' are the complements of ℓ, ℓ'. We note D the complement of $k \in Y \leftrightarrow Z$.

Define simulation relation $T \subseteq X \times (C \times D) \times (C' \times D) \times Z$ by

$$T = \{(x, (c, d), (c', d), z) \mid \exists y.(x, c, c', y) \in S \wedge (y, d, z) \in k.K\}$$

Suppose that $(x, c, c', y) \in S$ and $(y, d, z) \in k.K$ thus $(x, (c, d), (c', d), z) \in T$ and $\mathrm{d}x \in \partial X$ such that $\mathrm{d}x\ x$ is defined. Let $(\mathrm{d}y, c_1) = \ell.\Rrightarrow(\mathrm{d}x, c)$ and $(\mathrm{d}y', c_1') = \ell'.\Rrightarrow(\mathrm{d}x, c')$ and further $(\mathrm{d}z, d_1) = k.\Rrightarrow(\mathrm{d}y, d)$ and $(\mathrm{d}z', d_1') = k.\Rrightarrow(\mathrm{d}y', d)$.

We should prove $\mathrm{d}z = \mathrm{d}z'$ and $d_1 = d_1'$ and $(\mathrm{d}x\ x, (c_1, d_1), (c_1', d_1), \mathrm{d}z\ z) \in T$. From $(x, c, c', y) \in S$ we get $\mathrm{d}y = \mathrm{d}y'$ and $(\mathrm{d}x\ x, c_1, c_1', \mathrm{d}y\ y) \in S$ and $\mathrm{d}z = \mathrm{d}z'$ and $d_1 = d_1'$. From $(y, d, z) \in k.K$ we then get $(\mathrm{d}y\ y, d_1, \mathrm{d}z\ z) \in k.K$ and thus all that is required. $\square$

The following theorem establishes the properties necessary to show that there is a category with modules as objects and equivalence classes of lenses as arrows. In what follows, we will sometimes note how the properties of our lens constructions can be restated in terms of standard categorical jargon, but these observations are intended just as sanity checks; nothing depends on them, and they can safely be ignored.

3.3.5 Theorem:

- $id_X; \ell \equiv \ell; id_Y \equiv \ell$
- $(k;\ell); m \equiv k; (\ell; m)$

Proof: The two relations given below witness $id_X; \ell \equiv \ell$ and $\ell; id_Y \equiv \ell$ respectively.

$$\{(x, (c, ()), c, y) \mid (x, c, y) \in \ell.K\}$$

$$\{(x, c, (c, ()), y) \mid (x, c, y) \in \ell.K\}$$

The relation that re-associates the complements is a witness that $(k;\ell); m \equiv k; (\ell; m)$:

$$R = \{(w, ((c_k, c_\ell), c_m), (c_k, (c_\ell, c_m)), z) \mid c_k \in k.C, c_\ell \in \ell.C, c_m \in m.C\}$$

Suppose we have an element of this relation and an edit $\mathrm{d}w$ for which $\mathrm{d}w\ w$ is defined; then define:

$$(\mathrm{d}x, c'_k) = k.\Rrightarrow(\mathrm{d}w, c_k)$$
$$(\mathrm{d}y, c'_\ell) = \ell.\Rrightarrow(\mathrm{d}x, c_\ell)$$
$$(\mathrm{d}z, c'_m) = m.\Rrightarrow(\mathrm{d}y, c_m)$$

We can compute that:

$$((k;\ell);m).\Rrightarrow(\mathrm{d}w, ((c_k, c_\ell), c_m)) = (\mathrm{d}z, ((c'_k, c'_\ell), c'_m))$$
$$(k;(\ell;m)).\Rrightarrow(\mathrm{d}w, (c_k, (c_\ell, c_m))) = (\mathrm{d}z, (c'_k, (c'_\ell, c'_m)))$$

Thus, the two lenses output the same edit $\mathrm{d}z$ and transition to related complements, as required. □

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

3.3.6 Definition [Dual]:

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

$$\begin{array}{lcl} C & = & \ell.C \\ \mathit{missing} & = & \ell.\mathit{missing} \\ K & = & \{(y, c, x) \mid (x, c, y) \in \ell.K\} \\ \Rrightarrow(\mathrm{d}y, c) & = & \ell.\Lleftarrow(\mathrm{d}y, c) \\ \Lleftarrow(\mathrm{d}x, c) & = & \ell.\Rrightarrow(\mathrm{d}x, c) \end{array}$$

3.3.7 Lemma: Given that ℓ is a lens, ℓ^{op} is a lens: $\Rrightarrow$ and $\Lleftarrow$ are stateful monoid homomorphisms, the consistency relation is preserved, and if $k \equiv \ell$ then $k^{op} \equiv \ell^{op}$.

Proof: $\Rrightarrow$ and $\Lleftarrow$ are homomorphisms because $\ell.\Lleftarrow$ and $\ell.\Rrightarrow$ are, respectively. The preservation of K is a direct consequence of ℓ preserving $\ell.K$. If S is a bisimulation relation witnessing $k \equiv \ell$, then $S^{op} = \{(y, c, d, x) \mid (x, c, d, y) \in S\}$ is a bisimulation relation witnessing $k^{op} \equiv \ell^{op}$. □

The name ^{op} is justified by the following theorem, which establishes that $(-)^{op}$ is an involutive contravariant functor and hence that the category of lenses is self-dual.

3.3.8 Theorem:

- $(\ell^{op})^{op} \equiv \ell$
- $id_X \equiv id_X^{op}$
- $k^{op}; \ell^{op} \equiv (\ell; k)^{op}$

Proof: In fact, $(\ell^{op})^{op} = \ell$ and $id_X = id_X^{op}$.

To show that $k^{op}; \ell^{op} \equiv (\ell; k)^{op}$, consider the relation:

$$S = \{(z, (c_k, c_\ell), (c_\ell, c_k), x) \mid (z, (c_k, c_\ell), x) \in (k^{op}; \ell^{op}).K\}$$

It is clear that the initial complements and initial x, y values are in this relation by simply unraveling the definitions of composition and dual. So suppose we have consistent z, c_k, c_ℓ, x and choose an edit $\mathrm{d}z$ for which $\mathrm{d}z\ z$ is defined. We can see that $(z, (c_\ell, c_k), x) \in (\ell; k)^{op}.K$, again by simply unrolling definitions to compare the consistency relations for the compositions. Define

$$(\mathrm{d}y, c'_\ell) = \ell.\Leftarrow(\mathrm{d}z, c_\ell)$$
$$(\mathrm{d}x, c'_k) = k.\Leftarrow(\mathrm{d}y, c_k)$$

Then we can calculate that:

$$(\mathrm{d}x, (c'_k, c'_\ell)) = (k^{op}; \ell^{op}).\Rightarrow(\mathrm{d}z, (c_k, c_\ell))$$
$$(\mathrm{d}x, (c'_\ell, c'_k)) = (\ell; k)^{op}.\Rightarrow(\mathrm{d}z, (c_\ell, c_k))$$

The output edits are equal, as required. Since both compositions preserve their respective consistency relations, we also know that $\mathrm{d}x\ x$ is defined and

$$(\mathrm{d}z\ z, (c'_k, c'_\ell), \mathrm{d}x\ x) \in (k^{op}; \ell^{op}).K.$$

So we have reached another consistent quadruple. □

3.3.9 Definition [Disconnect]:

$$disconnect_{XY} \in X \leftrightarrow Y$$

$$\begin{array}{lcl} C & = & Unit \\ K & = & X \times Unit \times Y \\ \Rightarrow(\mathrm{d}x, ()) & = & (\mathbf{1}, ()) \\ \Leftarrow(\mathrm{d}y, ()) & = & (\mathbf{1}, ()) \end{array}$$

3.3.10 Lemma: This is a good lens: $\Rightarrow$ and $\Leftarrow$ are homomorphisms, and K is preserved.

Proof: First we show that $\Rrightarrow$ is a stateful monoid homomorphism. There are two things to show; first, that:

$$\Rrightarrow(\mathbf{1}, c) = (\mathbf{1}, c)$$

Since $c = ()$, this follows immediately. Secondly, that if

$$\Rrightarrow(\mathrm{d}x, c) = (\mathrm{d}y, c') \quad \wedge \quad \Rrightarrow(\mathrm{d}x', c') = (\mathrm{d}y', c'')$$

then

$$\Rrightarrow(\mathrm{d}x'\mathrm{d}x, c) = (\mathrm{d}y'\mathrm{d}y, c'').$$

Since $c = c' = c'' = ()$ and $\mathrm{d}y = \mathrm{d}y' = \mathrm{d}y'\mathrm{d}y = \mathbf{1}$, this is trivially true. The argument showing that $\Lleftarrow$ is a homomorphism is similar.

Since K is the complete relation, there are no proof obligations to show that it is preserved except that $\mathbf{1}\ x$ is defined for all x—which follows from the definition of a module. □

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

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

3.3.11 Definition [Terminal]:

$$term_X \in X \leftrightarrow Unit$$

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

3.3.12 Lemma: This is a good lens: $\Rrightarrow$ and $\Lleftarrow$ are homomorphisms, and K is preserved.

Proof: Immediate, by observing $term_X = disconnect_{X\,Unit}$. □

3.3.13 Lemma: The *disconnect* and *term* lenses are closely related by the two equations $term_X \equiv disconnect_{X\,Unit}$ and $disconnect_{XY} \equiv term_X; term_Y^{op}$.

Proof: The former equivalence is actually an equality: $term_X = disconnect_{X\,Unit}$ can be verified by inspecting the two definitions. The complete relation $\{((), ())\}$ is a witness to the equivalence $disconnect_{XY} \equiv term_X; term_Y^{op}$. □

The *disconnect* lens that we saw in §3.1 can be built from *term*. The *term* lens is also unique (up to equivalence): the implementation of $\Rrightarrow$ is forced by the size of its range monoid *Unit*, and the implementation of $\Lleftarrow$ is forced by the homomorphism laws.

There is a trivial lens between any two isomorphic modules.

3.3.14 Definition: A *module homomorphism* (f, h) between modules X and Y is a function $f \in X \to Y$ and a monoid homomorphism $h \in \partial X \to \partial Y$ such that:

$$f(init_X) = init_Y \qquad f(\mathrm{d}x\, x) = h(\mathrm{d}x)\, f(x)$$

There is an identity $(\lambda x.\, x,\ \lambda \mathrm{d}x.\, \mathrm{d}x)$ for every module, and the point-wise composition of module homomorphisms is also a homomorphism, so modules form a category. If module homomorphisms $(e, g) \in X \to Y$ and $(f, h) \in Y \to X$ satisfy $(e, g); (f, h) = id_X$ and $(f, h); (e, g) = id_Y$, then (e, g) is an *isomorphism* and (f, h) is *inverse* to (e, g).

3.3.15 Definition [Isomorphism]:

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

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

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

3.3.16 Lemma: This is a good lens: $\Rrightarrow$ and $\Lleftarrow$ are stateful monoid homomorphisms, and K is preserved.

Proof: $\Rrightarrow$ and $\Lleftarrow$ are stateful monoid homomorphisms because h and h^{-1} are homomorphisms (and the state is trivial).

The definition of module homomorphisms give exactly the facts needed to show that K is preserved. In particular, we must show that $(init_X, (), init_Y) \in K$, but the definition of a module homomorphism tells us that $init_Y = f(init_X)$ as necessary. Moreover, whenever $\mathrm{d}x\ x$ is defined, the equation $f(\mathrm{d}x\ x) = h(\mathrm{d}x) f(x)$ from the definition of module homomorphism tells us what we need to know about $\Rrightarrow$. Similarly, the equation $f^{-1}(\mathrm{d}y\ y) = h^{-1}(\mathrm{d}y) f^{-1}(y)$ tells us what we need to know about $\Lleftarrow$ whenever $\mathrm{d}y\ y$ is defined. □

3.3.17 Theorem:

- $bij_{(id,id)} \equiv id$
- Given isomorphisms $(e, g) \in X \to Y$ and $(f, h) \in Y \to Z$,

$$bij_{(e,g)}; bij_{(f,h)} \equiv bij_{(e,g);(f,h)}.$$

- If (f, h) is inverse to (f^{-1}, h^{-1}), then

$$bij^{op}_{(f,h)} \equiv bij_{(f^{-1},h^{-1})}.$$

- If (f, h) is inverse to (f^{-1}, h^{-1}), then

$$bij_{(f,h)}; bij_{(f^{-1},h^{-1})} \equiv id.$$

Proof:

- We know $bij_{(id,id)} \equiv id$ because $bij_{(id,id)} = id$.
- It is easy to verify that the following relation satisfies the conditions of Theorem 3.2.11:

$$\{(x, ((), ()), (), f(e(x))) \mid x \in X\}$$

- In fact, the equivalence is an equality, because $(h^{-1})^{-1} = h$.
- By the first and second equivalences in the theorem,

$$bij_{(f,h)}; bij_{(f^{-1},h^{-1})} \equiv bij_{(f,h);(f^{-1},h^{-1})} = bij_{(id,id)} \equiv id. \qquad \square$$

Generators for free monoids For writing practical lenses, we want not only generic combinators like the ones presented above, but also more specific lenses for structured data such as products, sums, and lists. We show in the rest of this section how to define simple versions of these constructors whose associated edit monoids are freely generated. §3.4 shows how to generalize the list mapping lens to other forms of containers, and §3.5 discusses edit languages with nontrivial laws.

Given a set G of generators, one commonly-used monoid is the *free monoid*: the set of lists $G^\star$ together with sequence concatenation as the binary operation and $\langle\rangle$ as the identity. Defining homomorphisms from this monoid to another is often most conveniently done by specifying the homomorphism's behavior on each generator. Given a function $f_{\mathsf{g}} \in G \to M$ on generators[4], the monoid homomorphism $f \in$

[4]We use a different typeface in the subscript of f_{g} so that it is clear that it is not intended to be an index; thus the notation f_g is for the gth element of a family of functions, while f_{g} is for a particular function which we are thinking of as specifying a homomorphism.

$G^\star \to M$ is defined by $f(\langle\rangle) = \mathbf{1}$ and $f(g{:}gs) = f_{\mathsf{g}}(g)f(gs)$. Since this is generic over the codomain monoid, we can specialize this to give specifications of monoid actions and stateful monoid homomorphisms. As before, we will often treat the specification function as if it were a function of two arguments rather than a function whose codomain space contains functions.

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

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

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

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

The full module is then given by

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

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

3.3.18 Definition [Tensor Product]:

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

$$\begin{aligned}
C &= k.C \times \ell.C\\
\mathit{missing} &= (k.\mathit{missing}, \ell.\mathit{missing})\\
K &= \{\ ((x, z), (c_k, c_\ell), (y, w)) \mid\\
&\qquad (x, c_k, y) \in k.K\\
&\qquad \wedge\, (z, c_\ell, w) \in \ell.K\ \}\\
\Rrightarrow_{\mathsf{g}}(\mathsf{left}(\mathrm{d}x), (c_k, c_\ell)) &= \mathsf{let}\ (\mathrm{d}z, c'_k) = k.\Rrightarrow(\mathrm{d}x, c_k)\ \mathsf{in}\\
&\quad (\mathsf{left}(\mathrm{d}z), (c'_k, c_\ell))\\
\Rrightarrow_{\mathsf{g}}(\mathsf{right}(\mathrm{d}y), (c_k, c_\ell)) &= \mathsf{let}\ (\mathrm{d}w, c'_\ell) = \ell.\Rrightarrow(\mathrm{d}y, c_\ell)\ \mathsf{in}\\
&\quad (\mathsf{right}(\mathrm{d}w), (c_k, c'_\ell))\\
\Lleftarrow_{\mathsf{g}} \text{ similarly}&
\end{aligned}$$

3.3.19 Theorem:

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

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

The remaining statements are direct consequences of the definitions, together with Theorem 3.2.11; for example, the third equivalence can be witnessed by the simulation relation

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

This theorem asserts that $\otimes$ is a symmetric, associative bifunctor. Thus, the category of edit lenses with tensor product is almost a symmetric monoidal closed category; the only missing ingredient being an isomorphism between X and $X \otimes \mathit{Unit}$. With the present definition of tensor product such an isomorphism is available if ∂X is a free monoid, in which case we can map a free generator $\mathrm{d}x$ to $\mathsf{left}(\mathrm{d}x)$ and extend homomorphically. In order for $\mathrm{d}x \mapsto \langle \mathsf{left}(\mathrm{d}x) \rangle$ to be a homomorphism of modules in general, we would need equations $\langle \mathsf{left}(\mathrm{d}x) \rangle \cdot \langle \mathsf{left}(\mathrm{d}x') \rangle = \langle \mathsf{left}(\mathrm{d}x\, \mathrm{d}x') \rangle$ and $\langle \mathsf{left}(\mathbf{1}) \rangle = \langle \rangle$. See §3.5 for more detail on this alteration.

As in Chapter 2, the tensor construction is not quite a full categorical product, because duplicating information does not give rise to a well-behaved lens—there is no lens with type $X \leftrightarrow X \otimes X$ that satisfies all the equivalences a lens programmer would want.

Sum We now present one way (not the only one—see footnote 5) of constructing a sum module and a sum lens. Given sets of edits ∂X and ∂Y, we can describe the

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

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

Figure 3.3: The sum lens

generators for the free monoid of edits to a sum by:

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

The idea is that edits to a sum can either change just the content or change the tag (and therefore necessarily also the content, which is superseded by the given new content). That is, we want the "atoms" of the edit language to express the operations of editing content and switching sides. This gives us the switch_{LR}, switch_{RL}, and stay edits. For present purposes, we could leave it at this and define the monoid of edits to be the free monoid over just these generators. However, in §3.5 we will introduce a more compact representation that allows multiple edits to be combined into one, and this representation will give rise to the other two switch operations; for example, switch_{LL} represents a switch_{LR} followed by a switch_{RL}. To avoid having two similar but subtly different definitions, we include these edits here in the basic generators as well. Finally, we introduce an always-failing edit to represent sequences of edits that are internally inconsistent—e.g., a switch to the left side followed by an attempt to apply an edit which stays on the right side. These intuitions are formalized in the application function:

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

We then define the sum of modules X and Y as

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

There is a free choice of initial element for this module; one could also quite naturally choose $\mathsf{inr}(\mathit{init}_Y)$. We use $\overleftarrow{\oplus}$ to emphasize that this is the left-biased sum, and define a similar module, denoted $\overrightarrow{\oplus}$, whose only difference is that $\mathit{init}_{X \overrightarrow{\oplus} Y} = \mathsf{inr}(\mathit{init}_Y)$. We will use the left-biased sum almost exclusively in the remainder, writing simply $\oplus$ instead of $\overleftarrow{\oplus}$. This consideration extends to the lens definition below, where the tag of the *missing* state must match the tag of the module's *init*, and we will use the same notational convention to differentiate between the two lenses when necessary.

We now wish to give a lens combinator $k \oplus \ell$ that runs lens k on the parts of edits

that apply to inl values and ℓ on the parts of edits that apply to inr values.[5]

3.3.20 Definition [Sum]: Figure 3.3 defines the sum of two lenses.

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

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

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

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

Case $\mathrm{d}v = \mathsf{switch}_{LL}(\mathrm{d}x)$: We define $(\mathrm{d}y, c') = k.\mathit{putr}(\mathrm{d}x, k.\mathit{missing})$ and $(x', y') = (\mathrm{d}x\,\mathit{init}_X, \mathrm{d}y\,\mathit{init}_Y)$. Since k is a lens, we know $(\mathit{init}_X, k.\mathit{missing}, \mathit{init}_Y) \in k.K$ and therefore that $(x', c', y') \in k.K$. This means $(\mathsf{inl}(x'), \mathsf{inl}(c'), \mathsf{inl}(y')) \in K$. Since we now know the three equations

$$\begin{aligned} (k \oplus \ell).\Rrightarrow(\mathrm{d}v, \mathsf{inl}(c)) &= (\mathsf{switch}_{LL}(\mathrm{d}y), \mathsf{inl}(c')) \\ \mathrm{d}v\,\mathsf{inl}(x) &= \mathsf{inl}(x') \\ \mathsf{switch}_{LL}(\mathrm{d}y)\,\mathsf{inl}(y) &= \mathsf{inl}(y'), \end{aligned}$$

this shows that K is preserved in this case.

Case $\mathrm{d}v = \mathsf{switch}_{LR}(\mathrm{d}z)$: Nearly identical to the previous one, but using the fact that $\ell.K$ is preserved instead of $k.K$.

Case $\mathrm{d}v = \mathsf{stay}_L(\mathrm{d}x)$: We define $(\mathrm{d}y, c') = k.\Rrightarrow(\mathrm{d}x, c)$ and use similar reasoning to the above cases to observe that then $(\mathsf{inl}(\mathrm{d}x\,x), \mathsf{inl}(c'), \mathsf{inl}(\mathrm{d}y\,y)) \in K$ is both what we want to show and true because $k.K$ is preserved by $k.\Rrightarrow$. □

[5] In Chapter 2, there is some discussion regarding "forgetful" and "retentive" sum lenses, with the distinction revolving around what to do with the complement when an edit switches between sides of the sum. For state-based lenses, lenses on recursive structures like lists were given in terms of lenses on the non-recursive structure, and the retentive sum lens gave rise to a retentive list mapping lens whereas the forgetful sum lens gave rise to a forgetful list mapping lens. The poor alignment strategies given in that chapter were mediated somewhat by the retentive map's ability to use complements from previous versions of a list, making retentive sums somewhat more attractive than forgetful ones. In this presentation, however, the mapping lens has much better alignment information, so we eschew the more complicated retentive lenses in favor of simpler forgetful versions.

3.3.22 Lemma: Suppose that whenever $(x, c, y) \in k.K$ and $(x, d, y) \in \ell.K$ and $\mathrm{d}x\ x{\downarrow}$ then $\pi_1(k.{\Rrightarrow}(\mathrm{d}x, c)) = \pi_1(\ell.{\Rrightarrow}(\mathrm{d}x, d))$ (and similarly for $\Lleftarrow$). Then $k \equiv \ell$.

Proof: We build the witnessing relation:

$$S = \{(x, c, d, y) \mid (x, c, y) \in k.K \wedge (x, d, y) \in \ell.K\}$$

Since k is a lens, we know $(\mathit{init}_X, k.\mathit{missing}, \mathit{init}_Y) \in k.K$ (and similarly for ℓ), so we conclude $(\mathit{init}_X, k.\mathit{missing}, \ell.\mathit{missing}, \mathit{init}_Y) \in S$.

To show that $\Rrightarrow$ preserves S, we suppose $(x, c, d, y) \in S$ and $\mathrm{d}x\ x{\downarrow}$. Defining $(\mathrm{d}y_k, c') = k.{\Rrightarrow}(\mathrm{d}x, c)$ and $(\mathrm{d}y_\ell, d') = \ell.{\Rrightarrow}(\mathrm{d}x, d)$, we observe that our assumption gives us $\mathrm{d}y_k = \mathrm{d}y_\ell$. Moreover, since k and ℓ are lenses, the behavioral law for consistency relations tells us that $\mathrm{d}y_k\ y{\downarrow}$, that $(\mathrm{d}x\ x, c', \mathrm{d}y_k\ y) \in k.K$, and that $(\mathrm{d}x\ x, d', \mathrm{d}y_\ell\ y) \in \ell.K$. These last two tell us $(\mathrm{d}x\ x, c', d', \mathrm{d}y_k\ y) \in S$ (since $\mathrm{d}y_k = \mathrm{d}y_\ell$), as needed.

The proof that $\Lleftarrow$ preserves S is similar. □

3.3.23 Theorem:

- If $k \equiv k'$ and $\ell \equiv \ell'$, then $k \oplus \ell \equiv k' \oplus \ell'$.
- $\mathit{id} \oplus \mathit{id} \equiv \mathit{id}$.
- $(k \oplus \ell); (k' \oplus \ell') \equiv (k; k') \oplus (\ell; \ell')$.
- $\left(k \overleftarrow{\oplus} \ell\right); \mathit{bij}_{\mathit{swap}} \equiv \ell \overrightarrow{\oplus} k$, where *swap* is the obvious module isomorphism between $X \overleftarrow{\oplus} Y$ and $Y \overrightarrow{\oplus} X$.

Proof:

- If $k \equiv k'$ and $\ell \equiv \ell'$, then $k \oplus \ell \equiv k' \oplus \ell'$.

  Suppose sets S_k and S_ℓ witness the two given equivalences. Then we can construct a witness S for the desired equivalence as follows:

$$\begin{aligned}
S'_k &= \{(x, c_k, c_{k'}, y) \mid (x, c_k, c_{k'}, y) \in S_k \\
&\qquad \wedge (x, c_k, y) \in k.K \wedge (x, c_{k'}, y) \in k'.K\} \\
S'_\ell &= \{(z, c_\ell, c_{\ell'}, w) \mid (z, c_\ell, c_{\ell'}, w) \in S_\ell \\
&\qquad \wedge (z, c_\ell, w) \in \ell.K \wedge (z, c_{\ell'}, w) \in \ell'.K\} \\
S &= \{(\mathsf{inl}(x), \mathsf{inl}(c_k), \mathsf{inl}(c_{k'}), \mathsf{inl}(y)) \mid (x, c_k, c_{k'}, y) \in S'_k\} \\
&\quad \cup \{(\mathsf{inr}(z), \mathsf{inr}(c_\ell), \mathsf{inr}(c_{\ell'}), \mathsf{inr}(w)) \mid (z, c_\ell, c_{\ell'}, w) \in S'_\ell\}
\end{aligned}$$

  It is clear that

$$\begin{aligned}
&(\mathit{init}_{X \oplus Y}, (k \oplus \ell).\mathit{missing}, (k' \oplus \ell').\mathit{missing}, \mathit{init}_{Z \oplus W}) \\
&= (\mathsf{inl}(\mathit{init}_X), \mathsf{inl}(k.\mathit{init}), \mathsf{inl}(k'.\mathit{init}), \mathsf{inl}(\mathit{init}_Z)) \\
&\in S
\end{aligned}$$

because $(init_X, k.init, k'.init, init_Z) \in S_k$ and $(init_X, k.init, init_Z) \in k.K$ and $(init_X, k'.init, init_Z) \in k'.K$.

To show that S is preserved by $\Rrightarrow$ and $\Lleftarrow$, it is sufficient to consider only the generator edits (since $\Rrightarrow$ and $\Lleftarrow$ are homomorphisms). We show here that $\Rrightarrow_{\mathsf{g}}$ preserves S when starting from $(\mathsf{inl}(x), \mathsf{inl}(c_k), \mathsf{inl}(c_{k'}), \mathsf{inl}(y))$; the arguments for $\Lleftarrow_{\mathsf{g}}$ and for starting quadruples with inrs are nearly identical. Choose arbitrary $\mathrm{d}v$ for which $\mathrm{d}v\ \mathsf{inl}(x)$ is defined. There are three cases to consider.

1. If $\mathrm{d}v = \mathsf{switch}_{LL}(\mathrm{d}x)$ and $\mathrm{d}x\ init_X$ is defined, define:

$$\begin{aligned}(\mathrm{d}y_k, c'_k) &= k.\Rrightarrow(\mathrm{d}x, k.missing)\\ (\mathrm{d}y_{k'}, c'_{k'}) &= k'.\Rrightarrow(\mathrm{d}x, k'.missing)\\ x' &= \mathrm{d}x\ init_X\\ y' &= \mathrm{d}y_k\ init_Y\end{aligned}$$

   Since $(init_X, k.missing, k'.missing, init_Y) \in S_k$, we can conclude that that $\mathrm{d}y_k = \mathrm{d}y_{k'}$, that $\mathrm{d}y_k\ init_Y$ is defined, and that $(x', c'_k, c'_{k'}, y') \in S_k$. But now we can calculate that:

$$\begin{aligned}(k \oplus \ell).\Rrightarrow(\mathrm{d}v, \mathsf{inl}(c_k)) &= (\mathsf{switch}_{LL}(\mathrm{d}y_k), \mathsf{inl}(c'_k))\\ (k' \oplus \ell').\Rrightarrow(\mathrm{d}v, \mathsf{inl}(c_{k'}) &= (\mathsf{switch}_{LL}(\mathrm{d}y_{k'}), \mathsf{inl}(c'_{k'}))\end{aligned}$$

   Then the facts we must show (that $\mathsf{switch}_{LL}(\mathrm{d}y_k) = \mathsf{switch}_{LL}(\mathrm{d}y_{k'})$ and that $(\mathsf{inl}(x'), \mathsf{inl}(c'_k), \mathsf{inl}(c'_{k'}), \mathsf{inl}(y')) \in S$) follow immediately.

2. If $\mathrm{d}v = \mathsf{switch}_{LR}(\mathrm{d}z)$ and $\mathrm{d}z\ init_Z$ is defined, the argument is similar to above, but using ℓ and S_ℓ and inr everywhere instead of k and S_k and inl.

3. If $\mathrm{d}v = \mathsf{stay}_L(\mathrm{d}x)$ and $\mathrm{d}x\ x$ is defined, define:

$$\begin{aligned}(\mathrm{d}y_k, c'_k) &= k.\Rrightarrow(\mathrm{d}x, c_k)\\ (\mathrm{d}y_{k'}, c'_{k'}) &= k'.\Rrightarrow(\mathrm{d}x, c_{k'})\\ x' &= \mathrm{d}x\ x\\ y'_k &= \mathrm{d}y_k\ y\\ y'_{k'} &= \mathrm{d}y_{k'}\ y\end{aligned}$$

   Since k preserves $k.K$, we can conclude that y'_k is defined and $(x', c'_k, y'_k) \in k.K$; since k' preserves $k'.K$, we can conclude that $y'_{k'}$ is defined and $(x', c'_{k'}, y'_{k'}) \in k'.K$; since k and k' preserve S_k, we can conclude that $\mathrm{d}y_k = \mathrm{d}y_{k'}$ (hence $y'_k = y'_{k'}$) and $(x', c'_k, c'_{k'}, y'_k) \in S_k$. We may now compute

$$\begin{aligned}(k \oplus \ell).\Rrightarrow(\mathrm{d}x, \mathsf{inl}(c_k)) &= (\mathsf{stay}_L(\mathrm{d}y_k), \mathsf{inl}(c'_k))\\ (k' \oplus \ell').\Rrightarrow(\mathrm{d}x, \mathsf{inl}(c_{k'})) &= (\mathsf{stay}_L(\mathrm{d}y_{k'}), \mathsf{inl}(c'_{k'}))\end{aligned}$$

and observe that the above facts are exactly what we need to show that $y' = \mathsf{stay}_L(\mathrm{d}y_k)\ \mathsf{inl}(y)$ is defined and the two necessary conclusions:

$$\mathsf{stay}_L(\mathrm{d}y_k) = \mathsf{stay}_L(\mathrm{d}y_{k'})$$
$$(x', c'_k, c'_{k'}, y') \in S$$

- $id \oplus id \equiv id$

  We appeal to Lemma 3.3.22. We will show the argument for $\Rrightarrow$ (applied to generators, because the monoid is free); the argument for $\Lleftarrow$ is symmetric. Hence we may assume we have consistent triples $(x, c, y) \in (id \oplus id).K$ and $(x, d, y) \in id.K$ and an edit generator $\mathrm{d}x$ for which $\mathrm{d}x\ x{\downarrow}$. Since $id.\Rrightarrow(\mathrm{d}x, d) = (\mathrm{d}x, ())$, we must show $(id \oplus id).\Rrightarrow(\mathrm{d}x, c) = (\mathrm{d}x, c')$ for some c'. This property is clearly true by inspecting the various cases; one need only observe that $\Rrightarrow_{\mathbf{g}}$ outputs fail only in cases where $\mathrm{d}x\ x$ is visibly not defined.

- $(k \oplus \ell); (k' \oplus \ell') \equiv (k; k') \oplus (\ell; \ell')$

  We will refer to $(k \oplus \ell); (k' \oplus \ell')$ and $(k; k') \oplus (\ell; \ell')$ as a and b, respectively. The key insight is that the two sum lenses being composed always agree about which side of the sum they are on. This insight is embodied in the split function:

  $$\mathsf{split}(\mathsf{inl}((c, c'))) = (\mathsf{inl}(c), \mathsf{inl}(c'))$$
  $$\mathsf{split}(\mathsf{inr}((c, c'))) = (\mathsf{inr}(c), \mathsf{inr}(c'))$$

  Our witness relation uses this function.

  $$S = \{(x, \mathsf{split}(c), c, z) \mid (x, c, z) \in b.K\}$$

  Supposing we have the types

  $$k \in X_k \leftrightarrow Y_k \qquad k' \in Y_k \leftrightarrow Z_k$$
  $$\ell \in X_\ell \leftrightarrow Y_\ell \qquad \ell' \in Y_\ell \leftrightarrow Z_\ell$$

  we will first show that $(init_{X_k \oplus X_\ell}, a.missing, b.missing, init_{Z_k \oplus Z_\ell}) \in S$. It suffices to show that $a.missing = \mathsf{split}(b.missing)$, since $b.K$ is a correct consistency relation. But $b.missing = \mathsf{inl}((k.missing, k'.missing))$ and $a.missing = (\mathsf{inl}(k.missing), \mathsf{inl}(\ell.missing))$, so the necessary equation holds.

  We must also show that $\Rrightarrow$ and $\Lleftarrow$ preserve the relation S. We will show the $\Rrightarrow$ cases for stay_L and switch_{LR}; the remaining cases are very similar or mere induction. Since we may assume the supplied edit applies cleanly, we know that we have $(\mathsf{inl}(x), \mathsf{inl}((c_k, c_{k'})), \mathsf{inl}(z)) \in b.K$. A simple calculation assures us that consequently $(\mathsf{inl}(x), (\mathsf{inl}(c_k), \mathsf{inl}(c_{k'})), \mathsf{inl}(z)) \in a.K$. We will use the

abbreviations

$$c_a = (\mathsf{inl}(c_k), \mathsf{inl}(c_{k'})) \qquad\qquad c_b = \mathsf{inl}((c_k, c_{k'}))$$

in the following, noting that $c_a = \mathsf{split}(c_b)$.

Case $\mathsf{stay}_L(\mathrm{d}x)$: Defining

$$\begin{aligned} (\mathrm{d}y, c'_k) &= k.\Rrightarrow(\mathrm{d}x, c_k) & c'_a &= (\mathsf{inl}(c'_k), \mathsf{inl}(c'_{k'})) \\ (\mathrm{d}z, c'_{k'}) &= k'.\Rrightarrow(\mathrm{d}y, c_{k'}) & c'_b &= \mathsf{inl}((c'_k, c'_{k'})) \end{aligned}$$

we may now compute:

$$\begin{aligned} a.\Rrightarrow(\mathsf{stay}_L(\mathrm{d}x), c_a) &= (\mathsf{stay}_L(\mathrm{d}z), c'_a) \\ b.\Rrightarrow(\mathsf{stay}_L(\mathrm{d}x), c_b) &= (\mathsf{stay}_L(\mathrm{d}z), c'_b) \end{aligned}$$

Since $k; k'$ is a lens, and we know (from the assumption that $\mathsf{stay}_L(\mathrm{d}x)\ \mathsf{inl}(x)\!\downarrow$) that $\mathrm{d}x\ x\!\downarrow$, we can conclude that $\mathrm{d}z\ z\!\downarrow$ (one of two facts we must show to decide that S is preserved in this case). Setting

$$x' = \mathsf{inl}(\mathrm{d}x\ x) \qquad\qquad z' = \mathsf{inl}(\mathrm{d}z\ z)$$

we can conclude from $(\mathrm{d}x\ x, (c'_k, c'_{k'}), \mathrm{d}z\ z) \in (k; k').K$ that $(x', c'_b, z') \in b.K$, hence that $(x', c'_a, c'_b, z') = (x', \mathsf{split}(c'_b), c'_b, z') \in S$, the second necessary fact.

Case $\mathsf{switch}_{LR}(\mathrm{d}x)$: Defining some abbreviations,

$$\begin{aligned} (\mathrm{d}y, c'_\ell) &= \ell.\Rrightarrow(\mathrm{d}x, \ell.\mathit{missing}) & c'_a &= (\mathsf{inr}(c'_\ell), \mathsf{inr}(c'_{\ell'})) \\ (\mathrm{d}z, c'_{\ell'}) &= \ell'.\Rrightarrow(\mathrm{d}y, \ell'.\mathit{missing}) & c'_b &= \mathsf{inr}((c'_\ell, c'_{\ell'})) \end{aligned}$$

we can then compute:

$$\begin{aligned} a.\Rrightarrow(\mathsf{switch}_{LR}(\mathrm{d}x), c_a) &= (\mathsf{switch}_{LR}(\mathrm{d}z), c'_a) \\ b.\Rrightarrow(\mathsf{switch}_{LR}(\mathrm{d}x), c_b) &= (\mathsf{switch}_{LR}(\mathrm{d}z), c'_b) \end{aligned}$$

We would like to show that $\mathsf{switch}_{LR}(\mathrm{d}z)\ \mathsf{inl}(z)\!\downarrow$, that is, that $\mathrm{d}z\ \mathit{init}_{Z_\ell}\!\downarrow$. On the other hand, we know that $\mathsf{switch}_{LR}(\mathrm{d}x)\ \mathsf{inl}(x)\!\downarrow$, that is, that $\mathrm{d}x\ \mathit{init}_{X_\ell}\!\downarrow$. Since $\ell; \ell'$ is a lens, it translates an applicable edit to a consistent state to an applicable edit that restores consistency, and $(\mathit{init}_{X_\ell}, (\ell; \ell').\mathit{missing}, \mathit{init}_{Z_\ell}) \in (\ell; \ell').K$ is a consistent state. Hence we can conclude $\mathrm{d}z\ \mathit{init}_{Z_\ell}$ is defined as necessary, and furthermore that $(\mathrm{d}x\ \mathit{init}_{X_\ell}, (c'_\ell, c'_{\ell'}), \mathrm{d}z\ \mathit{init}_{Z_\ell}) \in (\ell; \ell').K$. From this we can conclude the second fact that we need, namely that

$$(\mathsf{inr}(\mathrm{d}x\ \mathit{init}_{X_\ell}), c'_a, c'_b, \mathsf{inr}(\mathrm{d}z\ \mathit{init}_{Z_\ell})) \in S.$$

- $\left(k \overleftarrow{\oplus} \ell\right); bij_{swap} \equiv \ell \overrightarrow{\oplus} k$

  We give a witnessing relation that rearranges the bits of the complements:

  $$S = \{(x, (swap(c), ()), c, y) \mid (x, c, y) \in (\ell \overrightarrow{\oplus} k).K\}$$

  As in previous proofs, we will name the two lenses in question $a = \left(k \overleftarrow{\oplus} \ell\right); bij_{swap}$ and $b = \ell \overrightarrow{\oplus} k$. Supposing that the types are $k \in X_k \leftrightarrow Y_k$ and $\ell \in X_\ell \leftrightarrow Y_\ell$, we first observe that

  $$\begin{aligned}
  &\left(init_{X_\ell \overrightarrow{\oplus} X_k}, a.missing, b.missing, init_{Y_\ell \overrightarrow{\oplus} Y_k}\right) \\
  &\quad = (\mathsf{inr}(init_{X_k}), (\mathsf{inl}(k.missing), ()), \mathsf{inr}(k.missing), \mathsf{inr}(init_{Y_k})) \\
  &\quad = (\mathsf{inr}(init_{X_k}), (swap(\mathsf{inr}(k.missing)), ()), \mathsf{inr}(k.missing), \mathsf{inr}(init_{Y_k})) \\
  &\quad \in S
  \end{aligned}$$

  because $(init_{X_k}, k.missing, init_{Y_k}) \in k.K$. It only remains to show that S is preserved by $\Rrightarrow$ and $\Lleftarrow$ applied to applicable edits. The reasoning needed in showing that $\Rrightarrow$ preserves S when applied to switch_{LR} edits is representative of the reasoning needed in the other cases, so we satisfy ourselves with the proof of that case.

  So we assume that we have some $(x, c_a, c_b, y) \in S$ for which $\mathsf{switch}_{LR}(\mathrm{d}x)\ x{\downarrow}$. We can immediately deduce a few facts:

  $$c_a = (swap(c_b), ()) \tag{3.3.1}$$
  $$x = \mathsf{inl}(x_k) \qquad \text{for some } x_k \in X_k \tag{3.3.2}$$
  $$(x, c_a, y) \in a.K \tag{3.3.3}$$
  $$(x, c_b, y) \in b.K \tag{3.3.4}$$
  $$\mathrm{d}x; init_{X_\ell}{\downarrow} \tag{3.3.5}$$

  Since we may assume that $\mathsf{switch}_{LR}(\mathrm{d}x)\ x{\downarrow}$, we know that $x = \mathsf{inl}(x_k)$ for some x_k and $\mathrm{d}x\ init_{X_\ell}{\downarrow}$.

  □

This theorem does not attempt to show that $\oplus$ is associative, that is, to connect $(k \oplus \ell) \oplus m$ and $k \oplus (\ell \oplus m)$ in any way. This is because the edits of the modules $(X \oplus Y) \oplus Z$ and $X \oplus (Y \oplus Z)$ as we have defined them are fundamentally different. For example, the former has an operation $\mathsf{switch}_{LR}(\mathbf{1})$ which takes any X or Y value and turns it into $init_Z$; this operation is not matched by any edit operation in the latter. Investigation into an associative sum module (and associative sum lens) is left for future work.

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

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

Figure 3.4: The list mapping lens

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

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

For compatibility with the generalization to arbitrary containers in §3.4, we slightly change the behavior of these operations from what we saw in §3.1. Insertions and deletions are now always performed at the end of the list; to insert in the middle of the list, you first insert at the end, then reorder the list. The argument i to $\mathsf{ins}(i)$ and $\mathsf{del}(i)$ now specifies how *many* elements to insert or delete.

$$\begin{array}{rcl}
\mathsf{mod}(p, \mathrm{d}x) \odot_{\mathsf{g}} x & = & x[p \mapsto \mathrm{d}x\, x_p] \\
\mathsf{ins}(i) \odot_{\mathsf{g}} x & = & x \cdot \underbrace{\langle \mathit{init}_X, \ldots, \mathit{init}_X \rangle}_{i \text{ times}} \\
\mathsf{del}(i) \odot_{\mathsf{g}} x & = & \langle x_1, \ldots, x_{n-i} \rangle \\
\mathsf{reorder}(f) \odot_{\mathsf{g}} x & = & \langle x_{f(n)(1)}, \ldots, x_{f(n)(n)} \rangle \\
\mathsf{fail} \odot_{\mathsf{g}} x & & \text{undefined}
\end{array}$$

We take $\mathsf{mod}(p, \mathrm{d}x) \odot_{\mathsf{g}} x$ to be undefined when $p > |x|$, and similarly take $\mathsf{del}(i) \odot_{\mathsf{g}} x$ to be undefined when $i > |x|$. The list module is then $X^\star = \left\langle |X|^\star, \langle\rangle, (G_X^{\text{list}})^\star, \odot \right\rangle$.

Mapping lens The list mapping lens $\ell^\star$ uses ℓ to translate mod edits from X to Y and vice versa. Other kinds of edits (ins, del, and $\mathsf{reorder}$) are carried across unchanged. When translating non-modification edits, we update the complement in a way almost identical to the way the two repositories are updated; to reflect this similarity, we use edit application from the $Unit^\star_{\ell.missing \in \ell.C}$ module to define the new complement.

3.3.24 Definition [Map]: Figure 3.4 defines the list mapping lens.

3.3.25 Lemma: The mapping lens is well-behaved:

- If ℓ is a lens, then $\ell^\star$ is a lens.
- If $k \equiv \ell$ then $k^\star \equiv \ell^\star$.
- $id^\star \equiv id$
- $k^\star; \ell^\star \equiv (k; \ell)^\star$

Proof:

- Because the lens is defined by specification over a free monoid, nothing needs to be verified for the monoid homomorphism laws. However, we must still verify that the initial repositories are consistent and that consistent triples are mapped to consistent triples. Since $init_X = \mathsf{map}(\ell).missing = init_Y = \langle\rangle$, it is clear that the consistency condition is satisfied: these lists all have the same length, and the pointwise-consistent constraint is degenerate. To prove that consistent triples are mapped to consistent triples, we argue that because the generating function $\Rrightarrow_{\mathsf{g}}$ preserves consistency, the resulting function $\Rrightarrow$ also preserves consistency.

  To show that $\Rrightarrow_{\mathsf{g}}$ maintains consistency, choose an arbitrary consistent triple (x, c, y) and applicable basic edit $\mathrm{d}x$; these two conditions mean that $|x| = |c| = |y|$, that $(x_p, c_p, y_p) \in \ell.K$ for all p, and that $\mathrm{d}x \odot_{\mathsf{g}} x$ is defined. We now consider each of the cases for $\mathrm{d}x$.

  If $\mathrm{d}x = \mathsf{mod}(p, \mathrm{d}v_x)$, then we know that $1 \leq p \leq |x|$ and $x'_p = \mathrm{d}v \odot x_p$ is defined (because $\mathrm{d}x \odot_{\mathsf{g}} x$ is). Defining $(\mathrm{d}v_y, c'_p) = \ell.\Rrightarrow(\mathrm{d}v_x, c_p)$ and $y'_p = \mathrm{d}v_y \odot y_p$, we observe that since ℓ is a lens, we must have $(x'_p, c'_p, y'_p) \in \ell.K$. Hence we know that $(x[p \mapsto x'_p], c[p \mapsto c'_p], y[p \mapsto y'_p]) \in \mathsf{map}(\ell).K$, and, by definition of $\odot_{\mathsf{g}}$, that:

$$(\mathsf{mod}(p, \mathrm{d}v_x) \odot_{\mathsf{g}} x, c[p \mapsto c'_p], \mathsf{mod}(p, \mathrm{d}v_y) \odot_{\mathsf{g}} y) \in \mathsf{map}(\ell).K$$

This is what we needed to show for this case.

If $\mathrm{d}x = \mathsf{ins}(i)$, then we need merely show that the additional elements in each list are synchronized. Since ℓ is a lens, we know that $(\mathit{init}_X, \ell.\mathit{missing}, \mathit{init}_Y) \in \ell.K$, so this is trivially true.

Suppose $\mathrm{d}x = \mathsf{del}(i)$ (and hence $i \leq |x|$), and let $n = |x| - i$. We observe that $|\mathrm{d}x \odot_{\mathsf{g}} x| = |\mathrm{d}x \odot_{\mathsf{g}} c| = |\mathrm{d}x \odot_{\mathsf{g}} y| = n$. Moreover, the pointwise-consistent part of the condition is clearly satisfied: we must show that $\forall 1 \leq p \leq n.\ (x_p, c_p, y_p) \in \ell.K$, but we know the stronger condition that $\forall 1 \leq p \leq |x|.\ (x_p, c_p, y_p) \in \ell.K$.

Suppose $\mathrm{d}x = \mathsf{reorder}(f)$, and let $n = |x|$. Since the lengths of the three lists x, c, and y are all n, the effect of $\mathrm{d}x$ on each is to apply the permutation $f(n)$. Permutations do not affect length or pointwise properties, so the resulting permuted lists are also in $\mathsf{map}(\ell).K$, as desired.

Finally, we need not consider the case where $\mathrm{d}x = \mathsf{fail}$ because this contradicts the assumption that $\mathrm{d}x \odot_{\mathsf{g}} x$ is defined.

The argument that $\Leftarrow$ maintains consistency is similar.

- Suppose S is a witness that $k \equiv \ell$. Define S' by the rule:

$$\frac{\forall i.(x_i, c_i, d_i, y_i) \in S}{(\langle x_1, \ldots, x_n \rangle, \langle c_1, \ldots, c_n \rangle, \langle d_1, \ldots, d_n \rangle, \langle y_1, \ldots, y_n \rangle) \in S'}$$

  Then S' is a witness that $k^\star \equiv \ell^\star$. The initial quadruple

$$(\mathit{init}_{X^\star}, k^\star.\mathit{init}, \ell^\star.\mathit{init}, \mathit{init}_{Y^\star}) = (\langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle)$$

  is in S' because the head of the inference rule is degenerate. The verification that defined edits preserve the S' relation is long, but straightforward. In the mod case, we rely on the analogous preservation of S for individual points, and in all other cases the pointwise property of the inference rule is preserved because the same pointwise operation is applied to each of the four lists in question (and because $(\mathit{init}_X, k.\mathit{init}, \ell.\mathit{init}, \mathit{init}_Y) \in S$).

- Let f be the function that takes a list and returns a list of equal length, all of whose elements are $()$. Then the relation

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

  witnesses the equivalence $\mathit{id}^\star_X \equiv \mathit{id}_X$.

- Define the function unzip as follows:

$$\begin{aligned}&\mathsf{unzip}(\langle (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \rangle) \\ &\quad = (\langle x_1, x_2, \ldots, x_n \rangle, \langle y_1, y_2, \ldots, y_n \rangle)\end{aligned}$$

Then the requisite simulation relation is:

$$R = \{(x, \mathsf{unzip}(d), d, y) \mid (x, d, y) \in (k; \ell)^\star.K\}$$

The interesting property to verify is that if $(x, c, d, y) \in R$ and $\mathrm{d}x\, x$ is defined, then $(k^\star; \ell^\star).\Rrightarrow$ and $(k; \ell)^\star.\Rrightarrow$ produce the same edit $\mathrm{d}y$ in related states. As in the other proofs here, we can show this property for the restricted set of edits which contain only one atomic edit by case analysis; the stateful homomorphism property of $\Rrightarrow$ then guarantees the same property for the set of all edits.

Suppose $\mathrm{d}x = \mathsf{mod}(p, \mathrm{d}x')$ and $d_p = (s_k, s_\ell)$. Then by definition of unzip, we will have $c = (c_k, c_\ell)$ and $(c_k)_p = s_k$ and $(c_\ell)_p = s_\ell$. Hence we will be running $k.\Rrightarrow$ followed by $\ell.\Rrightarrow$ with complements s_k and s_ℓ respectively in both cases, and will receive related outputs as required.

Otherwise, $\mathrm{d}x$ is an insertion, deletion, or rearrangement, and both $k^\star; \ell^\star$ and $(k; \ell)^\star$ will merely apply the appropriate insertion, deletion, or rearrangement to the tuple of lists and list of tuples, respectively.

□

Partition lens Figures 3.5, 3.6, and 3.7 give the definition of a list partitioning lens that (as we saw in §3.1) separates a list of tagged elements into those tagged inl and those tagged inr. Additionally, as with the mapping lens, we consider the complement to belong to a module; this time, to the module $\mathit{Unit}^\star_{L \in \{L,R\}}$.

These figures may be a bit intimidating at first, but there is nothing very deep going on—just some everyday functional programming over lists. To illustrate how it all works, let's consider a few example invocations of the *partition* lens. Each of them begins with the consistent triple illustrated in Figure 3.8. Note that only the middle part—the complement—is actually available to the partition lens as it runs: its other input is just an edit.

As a warm-up, consider a simple edit: changing Dvorak's name to Dvořák (with correct diacritics) in the left repository. The edit describing this has the form

$$\mathsf{mod}(5, \mathsf{stay}_L(\mathrm{d}n)),$$

where $\mathrm{d}n$ describes the string edit to the name. To translate this edit, we first need to translate the index 5 to an index into the list of composers in the right-hand repository (line 12 in Figure 3.6). We can do this by simply counting how many composers appear up to and including Dvorak, that is, how many L values appear in the complement list up to index 5—in this case, 3. We then wrap this index up, along with the $\mathrm{d}n$ edit, in a new edit of the form $\mathsf{left}(\mathsf{mod}(3, \mathrm{d}n))$; the complement need not change because we have not changed the structure of the lists. This pattern—count to translate the index, then re-tag the edit appropriately—can be generalized to all

$$\mathit{partition} \in (X \oplus Y)^\star \leftrightarrow X^\star \otimes Y^\star$$

$$
\begin{array}{lcl}
C & = & \{L, R\}^\star \\
\mathit{missing} & = & \langle\rangle \\
K & = & \{(z, \mathsf{map}_{\mathsf{tagof}}(z), (\mathsf{lefts}(z), \mathsf{rights}(z))) \mid z \in (|X| + |Y|)^\star\} \\
\Leftarrow_{\mathsf{g}}(\mathsf{left}(\mathsf{mod}(p, \mathrm{d}x)), c) & = & (\mathsf{mod}(p', \mathsf{stay}_L(\mathrm{d}x)), c), \text{ where} \quad (1) \\
\multicolumn{3}{r}{p' = \mathsf{iso}(c)^{-1}(\mathsf{inl}(p))} \\
\Leftarrow_{\mathsf{g}}(\mathsf{left}(\mathsf{reorder}(f)), c) & = & (\mathsf{reorder}(f'), c), \text{ where} \quad (2) \\
\multicolumn{3}{r}{g(\mathsf{inr}(p)) = \mathsf{inr}(p) \qquad f'(n \neq |c|) = \lambda p.\, p} \\
\multicolumn{3}{r}{g(\mathsf{inl}(p)) = \mathsf{inl}(f(n_L)(p)) \qquad f'(|c|) = h; g; h^{-1}} \\
\multicolumn{3}{r}{(n_L + 1, n_R + 1) = \mathsf{count}(|c| + 1, c) \qquad h = \mathsf{iso}(c)} \\
\Leftarrow_{\mathsf{g}}(\mathsf{left}(\mathsf{ins}(i)), c) & = & (\mathsf{ins}(i), \mathsf{ins}(i)\ c) \quad (3) \\
\Leftarrow_{\mathsf{g}}(\mathsf{left}(\mathsf{del}(0)), c) & = & (\langle\rangle, c) \quad (4) \\
\Leftarrow_{\mathsf{g}}(\mathsf{left}(\mathsf{del}(i)), c) & = & (d''\ \mathsf{del}'(p), c''), \text{ where} \quad (5) \\
\multicolumn{3}{r}{h = \mathsf{iso}(c) \qquad (n_L + 1, n_R + 1) = \mathsf{count}(|c| + 1, c)} \\
\multicolumn{3}{r}{p = h^{-1}(\mathsf{inl}(n_L)) \qquad (d'', c'') = \Leftarrow_{\mathsf{g}}(d', c')} \\
\multicolumn{3}{r}{c' = \mathsf{del}'(p)\ c \qquad d' = \mathsf{left}(\mathsf{del}(i{-}1))} \\
\multicolumn{3}{r}{\text{when } 1 \leq i \leq n_L} \\
\Leftarrow_{\mathsf{g}}(\mathsf{left}(\mathsf{del}(i)), c) & = & (\mathsf{fail}, c) \text{ otherwise} \quad (6) \\
\Leftarrow_{\mathsf{g}}(\mathsf{left}(\mathsf{fail}), c) & = & (\mathsf{fail}, c) \quad (7) \\
\Leftarrow_{\mathsf{g}}(\mathsf{right}(\mathrm{d}y), c) & & \text{similar}
\end{array}
$$

Figure 3.5: Part of the *partition* lens (see also Figure 3.6)

modifications that stay on the same side of the sum; the count and tag functions defined in Figure 3.7 implement these two steps.

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

Perhaps the most interesting of these is an in-place modification to the left repository that switches sides of a sum (line 11). For example, suppose we want to replace

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

$$\Rrightarrow_{\mathsf{g}}(\mathsf{mod}(p, \mathrm{d}v), c) = (\mathsf{left}(\mathsf{fail}), c) \text{ when } p > |c| \quad (8)$$

$$\Rrightarrow_{\mathsf{g}}(\mathsf{mod}(p, \langle\rangle), c) = (\langle\rangle, c) \text{ when } 1 \le p \le |c| \quad (9)$$

$$\Rrightarrow_{\mathsf{g}}(\mathsf{mod}(p, \mathrm{d}v{:}\mathrm{d}vs), c) = (d'\, d, c''), \text{ where} \quad (10)$$
$$1 < n \qquad (d, c') = \Rrightarrow_{\mathsf{g}}(\mathsf{mod}(p, \mathrm{d}vs), c)$$
$$1 \le p \le |c| \quad (d', c'') = \Rrightarrow_{\mathsf{g}}(\mathsf{mod}(p, \mathrm{d}v), c')$$

$$\Rrightarrow_{\mathsf{g}}(\mathsf{mod}(p, \mathsf{switch}_{jk}(\mathrm{d}v)), c) = (d_2 d_1 d_0, c[p \mapsto k]), \text{ where} \quad (11)$$
$$(p_L, p_R) = \mathsf{count}(p, c) \qquad d_0 = \mathsf{map}_{\lambda d.\, \mathsf{tag}(j,d)}(\mathsf{del}'(p_j))$$
$$d_2 = \mathsf{tag}(k, \mathsf{mod}(p_k, \mathrm{d}v)) \quad d_1 = \mathsf{map}_{\lambda d.\, \mathsf{tag}(k,d)}(\mathsf{ins}'(p_k))$$

$$\Rrightarrow_{\mathsf{g}}(\mathsf{mod}(p, \mathsf{stay}_j(\mathrm{d}v)), c) = (\mathsf{tag}(j, \mathsf{mod}(p_j, \mathrm{d}v)), c), \text{ where} \quad (12)$$
$$(p_L, p_R) = \mathsf{count}(p, c)$$

$$\Rrightarrow_{\mathsf{g}}(\mathsf{mod}(p, \mathsf{fail}), c) = (\mathsf{left}(\mathsf{fail}), c) \quad (13)$$

$$\Rrightarrow_{\mathsf{g}}(\mathsf{ins}(i), c) = (\mathsf{left}(\mathsf{ins}(i)), \mathsf{ins}(i)\, c) \quad (14)$$

$$\Rrightarrow_{\mathsf{g}}(\mathsf{del}(i), c) = (d_1 d_0, \mathsf{del}(i)\, c), \text{ where} \quad (15)$$
$$c' = \mathsf{reverse}(c) \qquad d_0 = \mathsf{left}(\mathsf{del}(n_L{-}1))$$
$$(n_L, n_R) = \mathsf{count}(i{+}1, c') \quad d_1 = \mathsf{right}(\mathsf{del}(n_R{-}1))$$

$$\Rrightarrow_{\mathsf{g}}(\mathsf{reorder}(f), c) = (d_L d_R, c'), \text{ where} \quad (16)$$
$$h = \mathsf{iso}(c) \qquad c' = \mathsf{reorder}(f)\, c$$
$$h' = \mathsf{iso}(c') \qquad (n_L + 1, n_R + 1) = \mathsf{count}(|c| + 1, c)$$
$$h'' = h'^{-1}; f(|c|); h \qquad f_k(n \neq n_k) = \lambda p.\, p$$
$$d_L = \mathsf{left}(\mathsf{reorder}(f_L)) \qquad f_L(n_L) = \mathsf{inl}; h''; \mathsf{out}$$
$$d_R = \mathsf{right}(\mathsf{reorder}(f_R)) \qquad f_R(n_R) = \mathsf{inr}; h''; \mathsf{out}$$

$$\Rrightarrow_{\mathsf{g}}(\mathsf{fail}, c) = (\mathsf{left}(\mathsf{fail}), c) \quad (17)$$

Figure 3.6: Part of the *partition* lens (see also Figure 3.5)

Beethoven with Plato. The edit to do this has the form $\mathsf{mod}(2, \mathsf{switch}_{LR}(\mathrm{d}n))$—that is, at position 2, switch from an inl to an inr. Here, the translated edit must do *four* things: delete Beethoven from the left list, insert a new element into the right list, re-tag $\mathrm{d}n$ so that it changes the new element to Plato, and finally fix up the complement to match the new interleaving. As before, we can use count to translate the position 2 in the interleaved list into a position in the left list in the right repository. But then we hit a minor snag: deletions only occur at the end of a list. The solution is to first reorder the list, so that Beethoven appears at the end, then delete one element. Figure 3.7 defines the cycle function, which constructs permutations to do

$$\mathsf{tagof}(\mathsf{inl}(x)) = L \qquad \mathsf{map}_f(\langle\rangle) = \langle\rangle$$
$$\mathsf{tagof}(\mathsf{inr}(y)) = R \qquad \mathsf{map}_f(c{:}w) = f(c){:}\mathsf{map}_f(w)$$

$$\mathsf{lefts}(\langle\rangle) = \langle\rangle \qquad \mathsf{rights}(\langle\rangle) = \langle\rangle$$
$$\mathsf{lefts}(\mathsf{inl}(x){:}w) = x{:}\mathsf{lefts}(w) \qquad \mathsf{rights}(\mathsf{inl}(x){:}w) = \mathsf{rights}(w)$$
$$\mathsf{lefts}(\mathsf{inr}(y){:}w) = \mathsf{lefts}(w) \qquad \mathsf{rights}(\mathsf{inr}(y){:}w) = y{:}\mathsf{rights}(w)$$

$$\mathsf{tag}(L, \mathrm{d}x) = \mathsf{left}(\mathrm{d}x) \qquad \mathsf{out}(\mathsf{inl}(x)) = x$$
$$\mathsf{tag}(R, \mathrm{d}y) = \mathsf{right}(\mathrm{d}y) \qquad \mathsf{out}(\mathsf{inr}(y)) = y$$

$$\mathsf{count}(p, \langle\rangle) = (1, 1) \qquad \mathsf{count}(1, w) = (1, 1)$$

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

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

$$\mathsf{reverse}(\langle c_1, \ldots, c_n\rangle) = \langle c_n, c_{n-1}, \ldots, c_1\rangle$$
$$\mathsf{del}'(p) = \big\langle \mathsf{del}(1), \mathsf{reorder}(\mathsf{cycle}_p)\big\rangle$$
$$\mathsf{ins}'(p) = \big\langle \mathsf{reorder}(\lambda n.\ \mathsf{cycle}_p(n)^{-1}), \mathsf{ins}(1)\big\rangle$$
$$\mathsf{iso}(c) = \lambda p.\ \mathsf{let}\ (n_L, n_R) = \mathsf{count}(p, c)\ \mathsf{in}$$
$$\begin{cases} \mathsf{inl}(n_L) & c_p = L \\ \mathsf{inr}(n_R) & c_p = R \end{cases}$$

Figure 3.7: Supplementary functions for *partition*

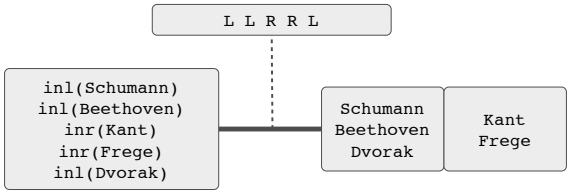


Figure 3.8: A consistent triple for the partition lens.

this reordering. The function $\mathsf{cycle}_p(n)$ permutes lists of size n by moving position p to the end of the list, and shifting all the other elements after p down one to fill in the resulting hole. For example, $\mathsf{cycle}_2(5)$ looks like this:

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

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

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

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

The edit itself is then $\mathsf{left}(\mathsf{reorder}(f))$. Our job is now to compute some f' for which $\mathsf{reorder}(f')$ swaps inl(Schumann) and inl(Dvorak) in the left repository (line 2). There is one wrinkle: f and f' are parameterized by the length of the lists they permute. Translating f naively would therefore seem to require a way for f' to *guess* the number of composers in lists whose lengths do not match that of the complement. Fortunately, f' need only behave correctly for exactly those lists that are consistent with the current complement, for which our "guess" about how many composers there are is guaranteed to be accurate. So we need only construct a single permutation (and use, say, the identity permutation for all inconsistent list lengths). We use the count function to construct this permutation. It is convenient to derive an isomorphism between positions in the left repository and positions tagged by which list they are indexing into in the right repository; the iso function shows how to use count to do this. In our example, the resulting isomorphism looks like this:

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

We can use $f(3)$ as a permutation on the inl elements, defining:

$$g(p) = \begin{cases} \mathsf{inr}(p') & p = \mathsf{inr}(p') \\ \mathsf{inl}(f(3)(p')) & p = \mathsf{inl}(p') \end{cases}$$

Then, to find out where position p in the left repository should come from, we can simply translate p into an index into the right repository using iso, apply g to find out where that index came from, and translate back into the left repository using iso^{-1}. Expanding the table above with these translations yields:

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

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

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

Translating a reordering of the left repository follows a similar path (line 16): restrict the reordering to lists consistent with the current complement, then compose the permutation with isomorphisms between the indices in the two repositories. There is one subtlety here: a reordering of the list in the left repository may shuffle which positions are inl's and which are inr's. As a result, we must take care to construct *two* separate position isomorphisms: one for "before" the reordering, and one for "after".

3.3.26 Lemma:

$$\mathsf{mod}(p, \mathrm{d}v{:}\mathrm{d}vs)z = \mathsf{mod}(p, \mathrm{d}v)\mathsf{mod}(p, \mathrm{d}vs)z$$

Proof: Let $n = |z|$. Either $p > n$ or not. If it is, then both sides are undefined; otherwise:

$$\begin{aligned} \mathsf{mod}(p, \mathrm{d}v{:}\mathrm{d}vs)z &= z[p \mapsto (\mathrm{d}v{:}\mathrm{d}vs)z_p] \\ &= z[p \mapsto \mathrm{d}v(\mathrm{d}vs\ z_p)] \\ &= \mathsf{mod}(p, \mathrm{d}v)(z[p \mapsto \mathrm{d}vs\ z_p]) \\ &= \mathsf{mod}(p, \mathrm{d}v)\mathsf{mod}(p, \mathrm{d}vs)z \end{aligned}$$

□

3.3.27 Lemma: If $1 \leq p \leq n$, then:

$$\mathsf{del}'(p) \odot v = \langle v_1, \ldots, v_{p-1}, v_{p+1}, \ldots, v_n \rangle$$

Proof: The only tricky part of this proof is evaluating $\mathsf{cycle}_p(n)$:

$$\begin{aligned}
\mathsf{del}'(p) \odot v &= \left\langle \mathsf{del}(1), \mathsf{reorder}(\mathsf{cycle}_p) \right\rangle \odot v \\
&= \langle \mathsf{del}(1) \rangle \odot \left\langle v_{\mathsf{cycle}_p(n)(1)}, \ldots, v_{\mathsf{cycle}_p(n)(n)} \right\rangle \\
&= \langle \mathsf{del}(1) \rangle \odot \langle v_1, \ldots, v_{p-1}, v_{p+1}, \ldots, v_{n-1}, v_p \rangle \\
&= \langle v_1, \ldots, v_{p-1}, v_{p+1}, \ldots, v_{n-1} \rangle
\end{aligned}$$

If $p = n$, then neither of the first two conditions in the definition of cycle will ever be true, so $\mathsf{cycle}_p(n)(m) = m$, making the evaluation given in these equations a special case where the interval from $p+1$ to $n-1$ is empty and $v_p = v_n$. On the other hand, when $p < n$, the value of $\mathsf{cycle}_p(n)$ is exactly in the form given here. □

3.3.28 Lemma: When $1 \leq p \leq n+1$:

$$\mathsf{ins}'(p) \odot \langle v_1, \ldots, v_n \rangle = \langle v_1, \ldots, v_{p-1}, \mathit{init}, v_p, \ldots, v_n \rangle$$

Proof: Set $v_{n+1} = \mathit{init}$ so that:

$$\begin{aligned}
\mathsf{ins}'(p) \odot v_1 \cdots v_n &= \mathsf{reorder}(\lambda n.\ \mathsf{cycle}_p(n)^{-1})\mathsf{ins}(1) \odot \langle v_1, \ldots, v_n \rangle \\
&= \mathsf{reorder}(\lambda n.\ \mathsf{cycle}_p(n)^{-1}) \odot \langle v_1, \ldots, v_n, \mathit{init} \rangle \\
&= \mathsf{reorder}(\lambda n.\ \mathsf{cycle}_p(n)^{-1}) \odot \langle v_1, \ldots, v_{n+1} \rangle \\
&= \left\langle v_{\mathsf{cycle}_p(n+1)^{-1}(1)}, \ldots, v_{\mathsf{cycle}_p(n+1)^{-1}(n+1)} \right\rangle \\
&= \langle v_1, \ldots, v_{p-1}, v_{n+1}, v_p, \ldots, v_n \rangle \\
&= \langle v_1, \ldots, v_{p-1}, \mathit{init}, v_p, \ldots, v_n \rangle
\end{aligned}$$

As with Lemma 3.3.27, the only tricky part is arguing that this evaluation of cycle_p is correct, and the argument is similar to the one given there, but in reverse. □

3.3.29 Lemma: The lefts and rights functions are list homomorphisms, that is,

$$\mathsf{lefts}(vw) = \mathsf{lefts}(v)\mathsf{lefts}(w),$$

and similarly for rights.

Proof: We will show the proof for lefts. We argue by induction on v. In the base case, $v = \langle\rangle$, and:

$$\begin{aligned}
\mathsf{lefts}(vw) &= \mathsf{lefts}(w) \\
&= \langle\rangle\, \mathsf{lefts}(w) \\
&= \mathsf{lefts}(\langle\rangle)\mathsf{lefts}(w) \\
&= \mathsf{lefts}(v)\mathsf{lefts}(w)
\end{aligned}$$

Otherwise, $v = v_1{:}v'$, we know from the induction hypothesis that $\mathsf{lefts}(v'w) = \mathsf{lefts}(v')\mathsf{lefts}(w)$, and by case analysis either $v_1 = \mathsf{inl}(x)$ or $v_1 = \mathsf{inr}(y)$. In the former case:

$$\begin{aligned}\mathsf{lefts}(vw) &= \mathsf{lefts}(\mathsf{inl}(x){:}v'w)\\ &= x{:}\mathsf{lefts}(v'w)\\ &= x{:}\mathsf{lefts}(v')\mathsf{lefts}(w)\\ &= \mathsf{lefts}(\mathsf{inl}(x){:}v')\mathsf{lefts}(w)\\ &= \mathsf{lefts}(v)\mathsf{lefts}(w)\end{aligned}$$

In the latter:

$$\begin{aligned}\mathsf{lefts}(vw) &= \mathsf{lefts}(\mathsf{inr}(y){:}v'w)\\ &= \mathsf{lefts}(v'w)\\ &= \mathsf{lefts}(v')\mathsf{lefts}(w)\\ &= \mathsf{lefts}(\mathsf{inr}(y){:}v')\mathsf{lefts}(w)\\ &= \mathsf{lefts}(v)\mathsf{lefts}(w)\end{aligned}$$

□

3.3.30 Lemma: The isomorphism produced by iso is coherent in the following sense. Choose arbitrary $v \in (X+Y)^\star$ and let $c = \mathsf{map}_{\mathsf{tagof}}(v)$ be the list of tags of v. If $\mathsf{iso}(c)(p) = \mathsf{inl}(p')$ then $\mathsf{inl}(\mathsf{lefts}(v)_{p'}) = v_p$ and likewise if $\mathsf{iso}(c)(p) = \mathsf{inr}(p')$ then $\mathsf{inr}(\mathsf{rights}(v)_{p'}) = v_p$.

Proof: Suppose there are n_L copies of L and n_R copies of R in $\langle c_1, \ldots, c_{p-1}\rangle$ and $p \leq n{+}1$. Then it is easy to show (by induction on c) that $\mathsf{count}(p, \langle c_1, \ldots, c_n\rangle) = (1{+}n_L, 1{+}n_R)$. Inspecting the definition of iso, it is therefore clear that $\mathsf{iso}(c)(p) = \mathsf{inl}(p')$ exactly when $c_p = L$ and there are p' copies of L in $\langle c_1, \ldots, c_p\rangle$. This implies there are exactly p' elements tagged inl in $\langle v_1, \ldots, v_p\rangle$ (and that v_p itself is tagged inl), hence that $\mathsf{inl}(\mathsf{lefts}(v)_{p'}) = v_p$.

The argument that iso is coherent with rights is similar. □

3.3.31 Corollary: If $c = \mathsf{map}_{\mathsf{tagof}}(v)$ and $1 \leq p \leq |v|$, then

$$\mathsf{tagof}(v_p) = \mathsf{tagof}(\mathsf{iso}(c)(p)).$$

3.3.32 Lemma: Suppose $\mathsf{iso}(c)(m) = \mathsf{inl}(n)$ (respectively, $\mathsf{inr}(n)$) and $\mathsf{iso}(c)(m') = \mathsf{inl}(n')$ (resp. $\mathsf{inr}(n')$). Then $m < m'$ if and only if $n < n'$.

Proof: As shown in the proof of Lemma 3.3.30, we have $\mathsf{iso}(c)(m) = \mathsf{inl}(n)$ exactly when $c_m = L$ and there are n copies of L in $\langle c_1, \ldots, c_m\rangle$. A similar statement relates m' and n'. Since $\langle c_1, \ldots, c_m\rangle$ and $\langle c_1, \ldots, c_{m'}\rangle$ share a common prefix, if one has more

copies of L than the other then it must be longer—that is, $n' > n$ implies $m' > m$. On the other hand, since $c_m = c_{m'} = L$, if one is longer than the other than it definitely has more copies of L—that is, $m' > m$ implies $n' > n$. □

3.3.33 Theorem: The *partition* operation defined in Figures 3.5, 3.6, and 3.7 is indeed a symmetric edit lens.

Proof: According to Definition 3.2.7, we must show three things. First, *partition*.$\Rrightarrow$ and *partition*.$\Lleftarrow$ must be stateful monoid homomorphisms; since the edit monoid for the list module is freely generated and the two functions in question are defined by specification, this is immediate. Second, the initial state

$$(init_{(X \oplus Y)^\star}, \langle\rangle, init_{X^\star \otimes Y^\star})$$

must be an element of K; this is easily verified from the definitions of the initial elements of the list and product modules. And third, the $\Rrightarrow$ and $\Lleftarrow$ operations must preserve consistent states; this is where some work is required. We show that $\Rrightarrow_{\mathsf{g}}$ and $\Lleftarrow_{\mathsf{g}}$ respect K; since $\Rrightarrow$ and $\Lleftarrow$ are defined by specification from these, the fact that they respect K follows by induction on the number of atomic edits they are handed.

For the two parts of the proof that follow, choose some $(z, c, (x, y)) \in K$. We will define $n = |z|$, $n_L = |x|$, and $n_R = |y|$ in the following. By the definition of K, we know

$$\begin{aligned} c &= \mathsf{map}_{\mathsf{tagof}}(z) \\ x &= \mathsf{lefts}(z) \\ y &= \mathsf{rights}(z). \end{aligned}$$

In many of the cases below, the definition of $\Rrightarrow_{\mathsf{g}}$ or $\Lleftarrow_{\mathsf{g}}$ has its own bindings for n_L and n_R using the idiom

$$(n_L + 1, n_R + 1) = \mathsf{count}(|c| + 1, c).$$

At first blush, these definitions conflict with the convention we are establishing here. However, Lemma 3.3.30 tells us that these are in fact coincident definitions; so we will not remark on them further in the cases where they occur.

First we show that $\Lleftarrow_{\mathsf{g}}$ respects K. We will give the proofs for atomic edits of the form $\mathsf{left}(\mathrm{d}x)$; the proofs for edits of the form $\mathsf{right}(\mathrm{d}y)$ are similar. Choose $\mathrm{d}x \in G_X^{\mathrm{list}}$ such that $\mathrm{d}x\ x$ is defined. We define $(\mathrm{d}z, c') = \Lleftarrow_{\mathsf{g}}(\mathsf{left}(\mathrm{d}x), c)$ and must show that $\mathrm{d}z\ z$ is defined and that $(\mathrm{d}z\ z, c', (\mathrm{d}x\ x, y)) \in K$. We proceed by induction on the size of $\mathrm{d}x$.

Case 1: In this case, we have the following equalities:

$$\begin{aligned} \mathrm{d}x &= \mathsf{mod}(p, \mathrm{d}v) \\ \mathrm{d}z &= \mathsf{mod}(p', \mathsf{stay}_L(\mathrm{d}v)) \\ c' &= c \\ p' &= \mathsf{iso}(c)^{-1}(\mathsf{inl}(p)) \end{aligned}$$

By Lemma 3.3.30, $z_{p'} = \mathsf{inl}(\mathsf{lefts}(z)_p) = \mathsf{inl}(x_p)$. This gives us enough to know that $\mathrm{d}z\ z$ is defined and in fact that

$$\begin{aligned} \mathrm{d}z\ z &= z[p' \mapsto \mathsf{stay}_L(\mathrm{d}v)\ z_{p'}] \\ &= z[p' \mapsto \mathsf{stay}_L(\mathrm{d}v)\ \mathsf{inl}(x_p)] \\ &= z[p' \mapsto \mathsf{inl}(\mathrm{d}v\ x_p)] \end{aligned}$$

Since none of the tags of z changes during this operation, this makes the computation of lefts, rights, and $\mathsf{map}_{\mathsf{tagof}}$ easy:

$$\begin{aligned} \mathsf{map}_{\mathsf{tagof}}(\mathrm{d}z\ z) &= \mathsf{map}_{\mathsf{tagof}}(z) \\ &= c \\ &= c' \\ \mathsf{rights}(\mathrm{d}z\ z) &= \mathsf{rights}(z) \\ &= y \\ \mathsf{lefts}(\mathrm{d}z\ z) &= \mathsf{lefts}(z)[p \mapsto \mathrm{d}v\ x_p] \\ &= x[p \mapsto \mathrm{d}v\ x_p] \\ &= \mathrm{d}x\ x \end{aligned}$$

These three computations establish that $(\mathrm{d}z\ z, c', (\mathrm{d}x\ x, y)) \in K$, as desired.

Case 2: We have a slew of equalities in hand to begin with. We have some chosen f and three main equalities:

$$\begin{aligned} \mathrm{d}x &= \mathsf{reorder}(f) \\ \mathrm{d}z &= \mathsf{reorder}(f') \\ c' &= c \end{aligned}$$

These depend on the additional definitions:

$$\begin{aligned} g(\mathsf{inr}(p)) &= \mathsf{inr}(p) & \qquad f'(n \neq |c|) &= \lambda p.\, p \\ g(\mathsf{inl}(p)) &= \mathsf{inl}(f(n_L)(p)) & \qquad f'(|c|) &= h; g; h^{-1} \\ h &= \mathsf{iso}(c) \end{aligned}$$

We first observe that $\mathsf{reorder}(f')$ does not affect tags at all. To be precise, for $1 \leq p \leq n$, we have:

$$\begin{aligned}
\mathsf{tagof}((\mathsf{reorder}(f')\ z)_p) &= \mathsf{tagof}(z_{(h;g;h^{-1})(p)}) && (3.3.6)\\
&= \mathsf{tagof}((h;g;h^{-1};h)(p)) && (3.3.7)\\
&= \mathsf{tagof}((h;g)(p)) && (3.3.8)\\
&= \mathsf{tagof}(h(p)) && (3.3.9)\\
&= \mathsf{tagof}(z_p) && (3.3.10)
\end{aligned}$$

Equation 3.3.6 follows from the definition of f' and edit application. Equation 3.3.7 is an application of Corollary 3.3.31; we can then simplify significantly in equations 3.3.8 and 3.3.9 because h is an isomorphism and g does not modify tags, as is evident from its definition. A second application of Corollary 3.3.31, this time "in reverse", gives us the final equation 3.3.10. We conclude that

$$\mathsf{map}_{\mathsf{tagof}}(\mathrm{d}z\ z) = \mathsf{map}_{\mathsf{tagof}}(z) = c = c',$$

part of what we need to show that $(\mathrm{d}z\ z, c', (\mathrm{d}x\ x, y)) \in K$. (It also means that h is the appropriate isomorphism to use when applying Lemma 3.3.30 to $\mathrm{d}z\ z$.)

Let us now turn our attention to showing that $\mathrm{d}z\ z$ and $\mathrm{d}x\ x$ have the appropriate relationship. We reason as follows:

$$\begin{aligned}
\mathsf{inl}(\mathsf{lefts}(\mathrm{d}z\ z)_p) &= (\mathrm{d}z\ z)_{h^{-1}(\mathsf{inl}(p))} && (3.3.11)\\
&= z_{(h^{-1};h;g;h^{-1})(\mathsf{inl}(p))} && (3.3.12)\\
&= z_{(g;h^{-1})(\mathsf{inl}(p))} && (3.3.13)\\
&= z_{h^{-1}(f(n_L)(p))} && (3.3.14)\\
&= \mathsf{inl}(\mathsf{lefts}(z)_{f(n_L)(p)}) && (3.3.15)
\end{aligned}$$

Equation 3.3.11 is an application of Lemma 3.3.30. The next three equations, 3.3.12 through 3.3.14, are mere computations that invoke the definitions of $\mathrm{d}z$, edit application, and g. The final equation 3.3.15 follows from the previous by Lemma 3.3.30. A similar argument to the above, differing only in line 3.3.14 where the definition of g is used, shows that

$$\mathsf{inr}(\mathsf{rights}(\mathrm{d}z\ z)_p) = \mathsf{inr}(\mathsf{rights}(z)_p).$$

We can therefore conclude that $\mathsf{lefts}(\mathrm{d}z\ z) = \mathrm{d}x\ \mathsf{lefts}(z) = \mathrm{d}x\ x$ and that $\mathsf{rights}(\mathrm{d}z\ z) = \mathsf{rights}(z) = y$, that is, that $(\mathrm{d}z\ z, c', (\mathrm{d}x\ x, y)) \in K$ as desired.

Case 3: We know $\mathrm{d}x = \mathsf{ins}(i)$ and $\mathrm{d}z = \mathsf{ins}(i)$ and $c' = \mathsf{ins}(i)\ c$. We compute:

$$\begin{aligned}
\mathsf{map}_{\mathsf{tagof}}(\mathrm{d}z\ z) &= \mathsf{map}_{\mathsf{tagof}}(z\ \langle \underbrace{init_{X\oplus Y}, \ldots, init_{X\oplus Y}}_{i \text{ times}}\rangle) \\
&= \mathsf{map}_{\mathsf{tagof}}(z)\mathsf{map}_{\mathsf{tagof}}(\langle \underbrace{init_{X\oplus Y}, \ldots, init_{X\oplus Y}}_{i \text{ times}}\rangle) \\
&= c\ \langle \underbrace{L, \ldots, L}_{i \text{ times}}\rangle \\
&= \mathsf{ins}(i)\ c \\
&= c'
\end{aligned}$$

There's a slight left-bias here; in the right- version of this proof, we find that $\Leftarrow_{\mathsf{g}}$ would have to produce a c' with many replicated Rs instead of Ls, and so would not have quite as compact a syntax for this output.

$$\begin{aligned}
\mathsf{lefts}(\mathrm{d}z\ z) &= \mathsf{lefts}(z\ \langle \underbrace{init_{X\oplus Y}, \ldots, init_{X\oplus Y}}_{i \text{ times}}\rangle) \\
&= \mathsf{lefts}(z)\mathsf{lefts}(\langle \underbrace{init_{X\oplus Y}, \ldots, init_{X\oplus Y}}_{i \text{ times}}\rangle) \\
&= x\ \langle \underbrace{init_{X}, \ldots, init_{X}}_{i \text{ times}}\rangle \\
&= \mathsf{ins}(i)\ x \\
&= \mathrm{d}x\ x
\end{aligned}$$

Again, the left-bias means the right- version of this proof relies on $\Leftarrow_{\mathsf{g}}$ being slightly more complicated for the right- analog. In particular, $\Leftarrow_{\mathsf{g}}$ would need to output an edit which did the insertion above followed by a series of modifications that turned the i final copies of $\mathsf{inl}(init_X)$ into i copies of $\mathsf{inr}(init_Y)$.

A similar computation to the previous one shows that $\mathsf{rights}(\mathrm{d}z\ z) = \mathsf{rights}(z) = y$. This concludes the proof of this case, since our three computations have shown that $(\mathrm{d}z\ z, c', (\mathrm{d}x\ x, y)) \in K$.

Case 4: We have $\mathrm{d}x = \mathsf{del}(0)$ and $\mathrm{d}z = \langle\rangle$ and $c' = c$. Since $\mathrm{d}z\ z = z$, $\mathrm{d}x\ x = x$, and $c' = c$, we are in the happy situation of having assumed exactly what we need to prove, namely that $(\mathrm{d}z\ z, c', (\mathrm{d}x\ x, y)) = (z, c, (x, y)) \in K$.

Case 5: To fit in with the surrounding conventions in the proof, we will rename a

few of the bindings of this case. To be specific, we have

$$\begin{aligned} \mathrm{d}x &= \mathsf{del}(i) \\ \mathrm{d}z &= d'd \\ (d', c') &= \Leftarrow_{\mathsf{g}}(d'', c'') \\ d'' &= \mathsf{left}(\mathsf{del}(i-1)) \\ c'' &= d\ c \\ d &= \mathsf{del}'(\mathsf{iso}(c)^{-1}(\mathsf{inl}(n_L))) \end{aligned}$$

and we know that $1 \leq i \leq n_L$. Our strategy is to show that $(d\ z, c'', (\mathsf{del}(1)\ x, y)) \in K$; the induction hypothesis then tells us that $(d'\ (d\ z), c', d''\ (\mathsf{del}(1)\ x, y)) \in K$. This means that $((d'd)\ z, c', (\mathsf{del}(i)\ x, y)) \in K$, which concludes this case. In the remainder of this case, let $m = \mathsf{iso}(c)^{-1}(\mathsf{inl}(n_L))$ so that $d = \mathsf{del}'(m)$.

Let us begin by showing that $\mathsf{map}_{\mathsf{tagof}}(d\ z) = d\ c$. Then Lemma 3.3.27 tells us two things:

$$\begin{aligned} d\ z &= \langle z_1, \ldots, z_{m-1}, z_{m+1}, \ldots, z_n \rangle \\ d\ c &= \langle c_1, \ldots, c_{m-1}, c_{m+1}, \ldots, c_n \rangle \end{aligned}$$

The desired equality then follows from the fact that map is a list homomorphism and that $\mathsf{map}_{\mathsf{tagof}}(z) = c$.

We must also show that $\mathsf{lefts}(d\ z) = \mathsf{del}(1)\ x$. By Lemma 3.3.30, $z_m = \mathsf{inl}(x_{n_L})$, and by Lemma 3.3.32, $z_{m'}$ is an inr for all $m' > m$. Since lefts is a list homomorphism, we can conclude that

$$\begin{aligned} \mathsf{lefts}(d\ z) &= \mathsf{lefts}(\langle z_1, \ldots, z_{m-1}, z_{m+1}, \ldots, z_n \rangle) \\ &= \mathsf{lefts}(\langle z_1, \ldots, z_{m-1} \rangle)\ \mathsf{lefts}(\langle z_{m+1}, \ldots, z_n \rangle) \\ &= \mathsf{lefts}(\langle z_1, \ldots, z_{m-1} \rangle) \\ &= \mathsf{del}(1)\ (\mathsf{lefts}(\langle z_1, \ldots, z_{m-1} \rangle)\ \mathsf{lefts}(\langle \mathsf{inl}(x_{n_L}) \rangle)) \\ &= \mathsf{del}(1)\ (\mathsf{lefts}(\langle z_1, \ldots, z_{m-1} \rangle)\ \mathsf{lefts}(\langle \mathsf{inl}(x_{n_L}) \rangle)\ \mathsf{lefts}(\langle z_{m+1}, \ldots, z_n \rangle)) \\ &= \mathsf{del}(1)\ (\mathsf{lefts}(z)) \\ &= \mathsf{del}(1)\ x \end{aligned}$$

as desired.

Next, we show that $\mathsf{rights}(d\ z) = y$. By Lemma 3.3.30, we know $z_m = \mathsf{inl}(x_{n_L})$. Since rights is a list homomorphism and $\mathsf{rights}(\mathsf{inl}(v)) = \langle\rangle$ for any v, we then can

compute that:

$$\begin{aligned}
\mathsf{rights}(d\ z) &= \mathsf{rights}(\langle z_1, \ldots, z_{m-1}, z_{m+1}, \ldots, z_n \rangle) \\
&= \mathsf{rights}(\langle z_1, \ldots, z_{m-1} \rangle)\ \mathsf{rights}(\langle z_{m+1}, \ldots, z_n \rangle) \\
&= \mathsf{rights}(\langle z_1, \ldots, z_{m-1} \rangle)\ \mathsf{rights}(\langle \mathsf{inl}(x_{n_L}) \rangle)\ \mathsf{rights}(\langle z_{m+1}, \ldots, z_n \rangle) \\
&= \mathsf{rights}(\langle z_1, \ldots, z_{m-1}, z_m, z_{m+1}, \ldots, z_n \rangle) \\
&= \mathsf{rights}(z) \\
&= y
\end{aligned}$$

The previous three paragraphs establish that $(d\ z, c'', (\mathsf{del}(1)\ x, y)) \in K$. Since $d'' = \mathsf{left}(\mathsf{del}(i-1))$ is a smaller edit than $\mathsf{left}(\mathrm{d}x) = \mathsf{left}(\mathsf{del}(i))$, we can apply the induction hypothesis to conclude that $(d'\ (d\ z), c', d''\ (\mathsf{del}(1)\ x, y)) \in K$. Since edit application is a monoid action, we know $d'\ (d\ z) = (d'd)\ z$. By definition of the edit application, we know $d''\ (\mathsf{del}(1)\ x, y) = (\mathsf{del}(i-1)\ (\mathsf{del}(1)\ x), y) = (\mathsf{del}(i)\ x, y)$. These last two equalities directly mean that $(\mathrm{d}z\ z, c', (\mathrm{d}x\ x, y)) \in K$, which completes this case.

Case 6: We know $\mathrm{d}x = \mathsf{del}(i)$ and, since the previous cases did not apply, $i > n_L + 1$. Hence we know $\mathrm{d}x\ x$ is not defined, a contradiction to our assumption that it is.

Case 7: Since $\mathrm{d}x = \mathsf{fail}$, the assumption that $\mathrm{d}x$ successfully applies is immediately contradicted, so there is nothing to prove in this case.

We now show that $\Rrightarrow_{\mathsf{g}}$ respects K. We are given some $\mathrm{d}z \in G^{\mathrm{list}}_{X \oplus Y}$ such that $\mathrm{d}z\ z$ is defined. We can define $(\mathrm{d}z', c') = \Rrightarrow_{\mathsf{g}}(\mathrm{d}z, c)$; then we must show that $\mathrm{d}z'(x, y)$ is defined and that $(\mathrm{d}z\ z, c', \mathrm{d}z'(x, y)) \in K$. We proceed by induction on the size of $\mathrm{d}z$.

Case 8: $\mathrm{d}v \in X \oplus Y$ and $\mathrm{d}z = \mathsf{mod}(p, \mathrm{d}v)$ and $p > |c|$.
Since $|z| = |c|$, we conclude that $\mathrm{d}z\ z$ is undefined, a contradiction.

Case 9: $\mathrm{d}z = \mathsf{mod}(p, \langle\rangle)$ and $1 \leq p \leq |c|$.
We calculate:

$$\begin{aligned}
\mathrm{d}z' &= \langle\rangle \\
c' &= c \\
\mathrm{d}z\ z &= z \\
\mathrm{d}z'(x, y) &= (x, y)
\end{aligned}$$

So $(\mathrm{d}z\ z, c', \mathrm{d}z'(x, y)) \in K$ by assumption: $(z, c, (x, y)) \in K$.

Case 10: We have all of the following:

$$\mathrm{d}v \in G^{\oplus}_{X,Y} \tag{3.3.16}$$
$$\mathrm{d}vs \in \partial(X \oplus Y) \tag{3.3.17}$$
$$\mathrm{d}z = \mathsf{mod}(p, \mathrm{d}v{:}\mathrm{d}vs) \tag{3.3.18}$$
$$1 \leq p \leq |c| \tag{3.3.19}$$
$$1 < n \tag{3.3.20}$$
$$(d, c'') = \Rrightarrow_{\mathsf{g}}(\mathsf{mod}(p, \mathrm{d}vs), c) \tag{3.3.21}$$
$$(d', c') = \Rrightarrow_{\mathsf{g}}(\mathsf{mod}(p, \mathrm{d}v), c'') \tag{3.3.22}$$
$$\mathrm{d}z' = d'\ d \tag{3.3.23}$$

By Lemma 3.3.26 and the assumption that $\mathsf{mod}(p, \mathrm{d}v{:}\mathrm{d}vs)z$ is defined, we know $\mathsf{mod}(p, \mathrm{d}v)(\mathsf{mod}(p, \mathrm{d}vs)z)$ is defined, and hence $\mathsf{mod}(p, \mathrm{d}vs)z$ is defined. The induction hypothesis for equation 3.3.21 therefore tells us that $d\ (x, y)$ is defined and that

$$(\mathsf{mod}(p, \mathrm{d}vs)z, c'', d\ (x, y)) \in K.$$

Again appealing to the induction hypothesis, this time for equation 3.3.22, we also know that $d'\ (d\ (x, y))$ is defined and

$$(\mathsf{mod}(p, \mathrm{d}v)(\mathsf{mod}(p, \mathrm{d}vs)z), c', d'\ (d\ (x, y))) \in K.$$

By one final appeal to Lemma 3.3.26, we therefore conclude that

$$(\mathrm{d}z\ z, c', \mathrm{d}z'(x, y)) \in K$$

as desired.

Case 11: We have:

$$\begin{aligned}
\mathrm{d}z &= \mathsf{mod}(p, \mathsf{switch}_{jk}(\mathrm{d}v)) \\
1 &\leq p \leq |c| \\
\mathrm{d}v &\in \partial X \text{ when } k = L \\
\mathrm{d}v &\in \partial Y \text{ when } k = R \\
\mathrm{d}z' &= d_2 d_1 d_0 \\
c' &= c[p \mapsto k] \\
d_0 &= \mathsf{map}_{\lambda d.\, \mathsf{tag}(j,d)}(\mathsf{del}'(p_j)) \\
d_1 &= \mathsf{map}_{\lambda d.\, \mathsf{tag}(k,d)}(\mathsf{ins}'(p_k)) \\
d_2 &= \mathsf{tag}(k, \mathsf{mod}(p_k, \mathrm{d}v)) \\
(p_L, p_R) &= \mathsf{count}(p, c)
\end{aligned}$$

Let us consider the case when $j = k = L$, whose argument is representative of the other cases.

Since $\mathrm{d}z\ z$ is defined, we know that $z_p = \mathsf{inl}(v)$ for some $v \in X$. Taking $v' = \mathrm{d}v\ \mathit{init}_X$, we can now compute:

$$\begin{aligned}
\mathsf{map}_{\mathsf{tagof}}(\mathrm{d}z\ z) &= \mathsf{map}_{\mathsf{tagof}}(z[p \mapsto \mathsf{inl}(v')]) \\
&= \mathsf{map}_{\mathsf{tagof}}(\langle z_1, \ldots, z_{p-1}\rangle)\ \langle k\rangle\ \mathsf{map}_{\mathsf{tagof}}(\langle z_{p+1}, \ldots, z_n\rangle) \\
&= \langle c_1, \ldots, c_{p-1}, k, c_{p+1}, \ldots, c_n\rangle \\
&= c[p \mapsto k] \\
&= c'
\end{aligned}$$

The second line follows from the first because map is a list homomorphism. Hence $\Rrightarrow_{\mathsf{g}}$ maintains consistency of c in this case; it remains to show that $\Rrightarrow_{\mathsf{g}}$ maintains consistency of the output. We calculate the effects of $\mathrm{d}z$ and $\mathrm{d}z'$, starting with $\mathrm{d}z'$:

$$\begin{aligned}
\mathrm{d}z'\ (x, y) &= d_2 d_1 d_0 (x, y) \\
&= d_2 d_1 (\mathsf{map}_{\lambda d.\mathsf{tag}(j,d)}(\mathsf{del}'(p_j)))(x, y) \\
&= d_2 d_1 (\mathsf{map}_{\mathsf{left}}(\mathsf{del}'(p_L)))(x, y) \\
&= d_2 d_1 (\mathsf{del}'(p_L)x, y) \\
&= d_2 (\mathsf{map}_{\lambda d.\mathsf{tag}(k,d)}(\mathsf{ins}'(p_k)))(\mathsf{del}'(p_L)x, y) \\
&= d_2 (\mathsf{map}_{\mathsf{left}}(\mathsf{ins}'(p_L)))(\mathsf{del}'(p_L)x, y) \\
&= d_2 (\mathsf{ins}'(p_L)\mathsf{del}'(p_L)x, y) \\
&= \mathsf{tag}(k, \mathsf{mod}((, p)_k, \mathrm{d}v))(\mathsf{ins}'(p_L)\mathsf{del}'(p_L)x, y) \\
&= (\mathsf{mod}(p_L, \mathrm{d}v)\mathsf{ins}'(p_L)\mathsf{del}'(p_L)x, y)
\end{aligned}$$

We can use Lemmas 3.3.27 and 3.3.28 to simplify:

$$\begin{aligned}
\mathrm{d}z'\ (x, y) &= (\mathsf{mod}(p_L, \mathrm{d}v)\mathsf{ins}'(p_L)\mathsf{del}'(p_L)\ \langle x_1, \ldots, x_{n_L}\rangle, y) \\
&= (\mathsf{mod}(p_L, \mathrm{d}v)\mathsf{ins}'(p_L)\ \langle x_1, \ldots, x_{p_L-1}, x_{p_L+1}, \ldots, x_{n_L}\rangle, y) \\
&= (\mathsf{mod}(p_L, \mathrm{d}v)\ \langle x_1, \ldots, x_{p_L-1}, \mathit{init}_X, x_{p_L+1}, \ldots, x_{n_L}\rangle, y) \\
&= (\langle x_1, \ldots, x_{p_L-1}, v', x_{p_L+1}, \ldots, x_{n_L}\rangle, y) \\
&= (x[p_L \mapsto v'], y)
\end{aligned}$$

We now make some observations about the effects of $\mathrm{d}z$, making crucial use of

Lemma 3.3.29:

$$\begin{aligned}
\mathsf{rights}(\mathrm{d}z\ z) &= \mathsf{rights}(\langle z_1, \ldots, z_{p-1}, \mathsf{inl}(v'), z_{p+1}, \ldots, z_n\rangle) \\
&= \mathsf{rights}(\langle z_1, \ldots, z_{p-1}\rangle)\mathsf{rights}(\mathsf{inl}(v'))\mathsf{rights}(\langle z_{p+1}, \ldots, z_n\rangle) \\
&= \mathsf{rights}(\langle z_1, \ldots, z_{p-1}\rangle)\mathsf{rights}(\mathsf{inl}(v))\mathsf{rights}(\langle z_{p+1}, \ldots, z_n\rangle) \\
&= \mathsf{rights}(\langle z_1, \ldots, z_{p-1}\rangle)\mathsf{rights}(z_p)\mathsf{rights}(\langle z_{p+1}, \ldots, z_n\rangle) \\
&= \mathsf{rights}(\langle z_1, \ldots, z_{p-1}, z_p, z_{p+1}, \ldots, z_n\rangle) \\
&= \mathsf{rights}(z) \\
&= y
\end{aligned}$$

We now observe that Lemma 3.3.30 implies that $\mathsf{lefts}(\langle z_1, \ldots, z_{p-1}\rangle) = \langle x_1, \ldots, x_{p_L-1}\rangle$ and likewise that $\mathsf{lefts}(\langle z_{p+1}, \ldots, z_n\rangle) = \langle x_{p_L+1}, \ldots, x_{n_L}\rangle$.

$$\begin{aligned}
\mathsf{lefts}(\mathrm{d}z\ z) &= \mathsf{lefts}(z[p \mapsto \mathsf{inl}(v')]) \\
&= \mathsf{lefts}(\langle z_1, \ldots, z_{p-1}\rangle)\mathsf{lefts}(\mathsf{inl}(v'))\mathsf{lefts}(\langle z_{p+1}, \ldots, z_n\rangle) \\
&= \langle x_1, \ldots, x_{p_L-1}, v', x_{p_L+1}, \ldots, x_{n_L}\rangle \\
&= x[p_L \mapsto v']
\end{aligned}$$

Taken together, these last three computations show that

$$\mathrm{d}z'\ (x, y) = (\mathsf{lefts}(\mathrm{d}z\ z), \mathsf{rights}(\mathrm{d}z\ z))$$

which is just what we needed.

Case 12: Let us consider specifically the case where $j = L$; the argument for $j = R$ is very similar. Then we have

$$\begin{aligned}
\mathrm{d}z &= \mathsf{mod}(p, \mathsf{stay}_L(\mathrm{d}v)) \\
\mathrm{d}z' &= \mathsf{left}(\mathsf{mod}(p_L, \mathrm{d}v)) \\
(p_L, p_R) &= \mathsf{count}(p, c)
\end{aligned}$$

Moreover, since $\mathrm{d}z\ z$ is defined, we know that there is some $v \in X$ for which $\mathrm{d}v\ v$ is defined such that $z_p = \mathsf{inl}(v)$ and, by appeal to Lemma 3.3.30, we know in particular that $v = \mathsf{lefts}(z)_{p_L} = x_{p_L}$. Hence $\mathrm{d}z'\ (x, y)$ is defined.

We now observe that $\mathsf{mod}(p, \mathsf{stay}_L(\mathrm{d}v))$ does not change any tags or any non-inl values, so $\mathsf{map}_{\mathsf{tagof}}(\mathrm{d}z\ z) = \mathsf{map}_{\mathsf{tagof}}(z) = c$ and $\mathsf{rights}(\mathrm{d}z\ z) = \mathsf{rights}(z) = y$. Furthermore:

$$\begin{aligned}
\mathsf{lefts}(\mathrm{d}z\ z) &= \mathsf{lefts}(\mathsf{mod}(p, \mathsf{stay}_L(\mathrm{d}v))\ z[p \mapsto \mathsf{inl}(x_{p_L})]) \\
&= \mathsf{lefts}(z[p \mapsto \mathsf{inl}(\mathrm{d}v\ x_{p_L})]) \\
&= x[p_L \mapsto \mathrm{d}v\ x_{p_L}] \\
&= \mathsf{mod}(p_L, \mathrm{d}v)\ x
\end{aligned}$$

That is, $\mathrm{d}z\ z$ and $\mathrm{d}z'\ (x, y) = (\mathsf{mod}(p_L, \mathrm{d}v)\ x, y)$ are synchronized as desired.

Case 13: When $\mathrm{d}z = \mathsf{mod}(p, \mathsf{fail})$ there is nothing to prove, because the assumption that the edit application is defined is immediately contradicted.

Case 14:

$$\begin{aligned} \mathrm{d}z &= \mathsf{ins}(i) \\ \mathrm{d}z' &= \mathsf{left}(\mathsf{ins}(i)) \\ c' &= \mathsf{ins}(i)\, c \end{aligned}$$

We calculate:

$$\begin{aligned} \mathrm{d}z\ z &= z\,\langle \underbrace{init_{X \oplus Y}, \ldots, init_{X \oplus Y}}_{i \text{ times}} \rangle \\ &= z\,\langle \underbrace{\mathsf{inl}(init_X), \ldots, \mathsf{inl}(init_X)}_{i \text{ times}} \rangle \\ \mathrm{d}z'(x, y) &= (x\,\langle \underbrace{init_X, \ldots, init_X}_{i \text{ times}} \rangle,\ y) \\ c' &= c\,\langle \underbrace{L, \ldots, L}_{i \text{ times}} \rangle \end{aligned}$$

Now, since map is a list homomorphism, we have:

$$\begin{aligned} \mathsf{map}_{\mathsf{tagof}}(\mathrm{d}z\ z) &= \mathsf{map}_{\mathsf{tagof}}(z)\mathsf{map}_{\mathsf{tagof}} \left(\langle \underbrace{\mathsf{inl}(init_X), \ldots, \mathsf{inl}(init_X)}_{i \text{ times}} \rangle \right) \\ &= c\,\langle \underbrace{L, \ldots, L}_{i \text{ times}} \rangle \\ &= c' \end{aligned}$$

Likewise, by Lemma 3.3.29:

$$\begin{aligned} \mathsf{lefts}(\mathrm{d}z\ z) &= \mathsf{lefts}(z)\mathsf{lefts} \left(\langle \underbrace{\mathsf{inl}(init_X), \ldots, \mathsf{inl}(init_X)}_{i \text{ times}} \rangle \right) \\ &= x\,\langle \underbrace{init_X, \ldots, init_X}_{i \text{ times}} \rangle \\ \mathsf{rights}(\mathrm{d}z\ z) &= \mathsf{rights}(z)\mathsf{rights} \left(\langle \underbrace{\mathsf{inl}(init_X), \ldots, \mathsf{inl}(init_X)}_{i \text{ times}} \rangle \right) \\ &= y \end{aligned}$$

These latter two computations amount to showing that

$$\mathrm{d}z'(x, y) = (\mathsf{lefts}(\mathrm{d}z\ z), \mathsf{right}(\mathrm{d}z\ z))$$

which, together with the observation above that $\mathsf{map}_{\mathsf{tagof}}(\mathrm{d}z\ z) = c'$, shows that $\Rrightarrow_{\mathsf{g}}$ preserves consistency in this case.

Case 15:

$$\begin{aligned} \mathrm{d}z &= \mathsf{del}(i) \\ \mathrm{d}z' &= \mathsf{right}(\mathsf{del}(n_L - 1))\mathsf{left}(\mathsf{del}(n_R - 1)) \\ (n_L, n_R) &= \mathsf{count}(i + 1, \mathsf{reverse}(c)) \end{aligned}$$

(Take careful notice of the definition of n_L and n_R here: it differs from the convention established at the beginning of the proof! We will use these local definitions for the remainder of this case.)

The interesting thing to prove is that $\mathsf{lefts}(\mathsf{del}(i)\ z) = \mathsf{del}(n_L - 1)\mathsf{lefts}(z)$ (and similarly for rights). We proceed by an inner induction on i.

When $i = 0$, we have $\mathsf{lefts}(\mathsf{del}(0)\ z) = \mathsf{lefts}(z)$ and

$$(n_L, n_R) = \mathsf{count}(1, \mathsf{reverse}(c)) = (1, 1)$$

so that $\mathsf{del}(n_L - 1)\mathsf{lefts}(z) = \mathsf{del}(0)\mathsf{lefts}(z) = \mathsf{lefts}(z)$.

Suppose $i > 0$. Define the abbreviation $c^r = \mathsf{reverse}(c)$. Then the induction hypothesis says that

$$\mathsf{lefts}(\mathsf{del}(i - 1)\ z) = \mathsf{del}(\mathsf{fst}(\mathsf{count}(i, c^r)) - 1)\mathsf{lefts}(z).$$

Now, either $c^r_i = L$ or $c^r_i = R$. If the former, then $z_{n-i+1} = \mathsf{inl}(x)$ for some x and:

$$\begin{aligned} \mathsf{del}(n_L - 1)\mathsf{lefts}(z) &= \mathsf{del}(\mathsf{fst}(\mathsf{count}(i + 1, c^r)) - 1)\mathsf{lefts}(z) \\ &= \mathsf{del}(1 + \mathsf{fst}(\mathsf{count}(i, c^r)) - 1)\mathsf{lefts}(z) \\ &= \mathsf{del}(1)\mathsf{del}(\mathsf{fst}(\mathsf{count}(i, c^r)) - 1)\mathsf{lefts}(z) \\ &= \mathsf{del}(1)\mathsf{lefts}(\mathsf{del}(i - 1)\ z) \\ &= \mathsf{del}(1)\mathsf{lefts}(\langle z_1, \ldots, z_{n-i}, \mathsf{inl}(x)\rangle) \\ &= \mathsf{del}(1)(\mathsf{lefts}(\langle z_1, \ldots, z_{n-i}\rangle)x) \\ &= \mathsf{lefts}(\langle z_1, \ldots, z_{n-i}\rangle) \\ &= \mathsf{lefts}(\mathsf{del}(i)\ z) \end{aligned}$$

Otherwise, $z_{n-i+1} = \mathsf{inr}(y)$ for some y and:

$$\begin{aligned}
\mathsf{del}(n_L - 1)\mathsf{lefts}(z) &= \mathsf{del}(\mathsf{fst}(\mathsf{count}(i+1, c^r)) - 1)\mathsf{lefts}(z) \\
&= \mathsf{del}(\mathsf{fst}(\mathsf{count}(i, c^r)) - 1)\mathsf{lefts}(z) \\
&= \mathsf{lefts}(\mathsf{del}(i-1)\ z) \\
&= \mathsf{lefts}(\langle z_1, \ldots, z_{n-i}, \mathsf{inr}(y)\rangle) \\
&= \mathsf{lefts}(\langle z_1, \ldots, z_{n-i}\rangle) \\
&= \mathsf{lefts}(\mathsf{del}(i)\ z)
\end{aligned}$$

as desired.

A similar argument shows that:

$$\mathsf{rights}(\mathsf{del}(i)\ z) = \mathsf{del}(n_R - 1)\mathsf{rights}(z)$$

Case 16: The main idea of the proof for this case is to observe that c contains enough information to deduce the length of x, y, and z, and in particular which index the various reorder edits will be specialized to during edit application. We can focus on these indices. (We will see that the somewhat strange-looking clause defining $f_k(n \neq n_k) = \lambda p.\ p$ is never used – the lens could use any automorphism on $\{1, \ldots, n\}$ in place of the identity there.)

Because the application of $\mathrm{d}z$ and $\mathrm{d}z'$ are always defined, we need only show that the new complement and the output edits are consistent with the input edits. We begin by showing the new complement is consistent with $\mathrm{d}z\ z$.

$$\begin{aligned}
\mathsf{map}_{\mathsf{tagof}}(\mathrm{d}z\ z) &= \mathsf{map}_{\mathsf{tagof}}(\langle z_{f(n)(1)}, \ldots, z_{f(n)(n)}\rangle) && (3.3.24)\\
&= \langle \mathsf{map}_{\mathsf{tagof}}(z)_{f(n)(1)}, \ldots, \mathsf{map}_{\mathsf{tagof}}(z)_{f(n)(n)}\rangle && (3.3.25)\\
&= \langle c_{f(n)(1)}, \ldots, c_{f(n)(n)}\rangle && (3.3.26)\\
&= \mathsf{reorder}(f)\ c && (3.3.27)
\end{aligned}$$

Equation 3.3.24 follows by definition of edit application in the list module (and because $|z| = |c| = n$); equation 3.3.25 is a special property of map; equation 3.3.26 by definition of c; and equation 3.3.27 by the definition of reorder's edit application.

We will now show that $\mathsf{lefts}(\mathrm{d}z\ z) = \mathsf{reorder}(f_L)\ x$. A similar argument to the following also shows that $\mathsf{rights}(\mathrm{d}z\ z) = \mathsf{reorder}(f_R)\ y$, and these two facts together will conclude the proof (since $\mathrm{d}z'\ (x, y) = (\mathsf{reorder}(f_L)\ x, \mathsf{reorder}(f_R)\ y)$). By the

above fact about c' and Lemma 3.3.30:

$$\begin{aligned}
\mathsf{inl}(\mathsf{lefts}(\mathrm{d}z\ z)_i) &= (\mathrm{d}z\ z)_{\mathsf{iso}^{-1}(c')(\mathsf{inl}(i))} && (3.3.28)\\
&= (\mathrm{d}z\ z)_{h'^{-1}(\mathsf{inl}(i))} && (3.3.29)\\
&= z_{f(n)(h'^{-1}(\mathsf{inl}(i)))} && (3.3.30)\\
&= \mathsf{inl}(\mathsf{lefts}(z)_{\mathsf{out}(\mathsf{iso}(c)(f(n)(h'^{-1}(\mathsf{inl}(i)))))}) && (3.3.31)\\
&= \mathsf{inl}(\mathsf{lefts}(z)_{\mathsf{out}(h(f(n)(h'^{-1}(\mathsf{inl}(i)))))}) && (3.3.32)\\
&= \mathsf{inl}(\mathsf{lefts}(z)_{(\mathsf{inl};h'^{-1};f(n);h;\mathsf{out})(i)}) && (3.3.33)\\
&= \mathsf{inl}(\mathsf{lefts}(z)_{(\mathsf{inl};h'';\mathsf{out})(i)}) && (3.3.34)\\
&= \mathsf{inl}(\mathsf{lefts}(z)_{f_L(n_L)(i)}) && (3.3.35)\\
\mathsf{lefts}(\mathrm{d}z\ z)_i &= \mathsf{lefts}(z)_{f_L(n_L)(i)} && (3.3.36)
\end{aligned}$$

Equation 3.3.28 is a straightforward application of Lemma 3.3.30; equation 3.3.29 folds the definition of h'; and equation 3.3.30 applies edit $\mathrm{d}z$. Equation 3.3.31 applies Lemma 3.3.30 again, but with the knowledge that the tag of the previous line is inl (because that is the left-hand side of the equality we have already proved). Equations 3.3.32, 3.3.33, 3.3.34, and 3.3.35 just rewrite the equation by folding the definitions of h, h'', and f_L and rewriting explicit function application as the application of a function composition. The final equation 3.3.36 holds by injectivity of inl.

Now, since $x = \mathsf{lefts}(z)$ and $|x| = n_L$, we can conclude that $\mathsf{lefts}(\mathrm{d}z\ z) = \mathsf{reorder}(f_L)\ x$ as desired.

Case 17: As in Case 13, there is nothing to prove, as the assumption that the edit application is defined is immediately contradicted.

□

3.4 Containers

The list mapping lens from the previous section can be generalized to a much larger set of structures, called *containers*, that also includes trees, labeled graphs, etc. We will also provide a general construction for "reorganization lenses" between *different* container types (over the same type of entries). Together with composition and tensor product, this will provide a set of building blocks for constructing many useful lenses. The reorganization lenses also furnish further examples of lenses with nontrivial complements. (Only a small part of §3.5 depends on this material; it can safely be skimmed on a first reading.)

Containers were first proposed by Abbott, Altenkirch, and Ghani [2]. The idea is that a container type specifies a set I of shapes and, for each shape i, a set of positions $P(i)$. A container with entries in X and belonging to such a container type comprises a shape i and a function $f : P(i) \to X$. For example, lists are containers whose shapes

are the natural numbers and for which $P(i) = \{0, \ldots, i-1\}$, whereas binary trees are containers whose shapes are prefix-closed subsets of $\{0,1\}^\star$ (access paths) and where $P(i) = i$ itself. Even labeled graphs can be modeled using unlabeled graphs as shapes. One can further generalize the framework to allow the types of entries to depend on their position, but for the sake of simplicity we will not do so here.

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

In the case of the state-based symmetric lenses of Chapter 2, it has been observed that lens iterators akin to "fold left" for inductive data structures also permit the definition of powerful (state-based) lenses. In the edit-based framework iterators are less convenient because it is unclear how edits in an arbitrary module should be propagated to, say, list edits in such a way that the rich edit structure available for lists is meaningfully exploited. (Of course, it is possible to propagate everything to a "rebuild from scratch" edit, thus aping the state-based case.)

In the following we slightly deviate from the presentation of containers from §2.7 and [2] in that we do not allow the set of positions to vary with the shapes. We rather have a universal set of positions P and a predicate live that delineates a subset of P for each shape i. We can then obtain a container type in the original sense by putting $P(i) = \{p \mid p \in \mathsf{live}(i)\}$. Conversely, given a container type in the sense of [2], we can define $P = \{(i,p) \mid p \in P(i)\}$ and $\mathsf{live}(i) = \{(i,p) \mid p \in P\}$. Furthermore, as we already did in Chapter 2, we require a *partially-ordered* set of shapes I and ask that live be monotone. Formulating this in the original setting would require a coherent family of transition functions $P(i) \to P(i')$ when $i \leq i'$, which is more cumbersome. Another advantage of the present formulation of container types is that it lends itself more easily to an implementation in a programming language without dependent types.

3.4.1 Definition: A *container type* is a triple $\langle I, P, \mathsf{live}\rangle$ comprising

- a *module* I of *shapes* whose underlying set is partially ordered (but whose action need not be monotone);
- a set P of *positions*; and
- a *liveness predicate* in the form of a monotone function $\mathsf{live} \in I \to \mathcal{P}(P)$ which tells for each shape which positions belong to it.

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

Our aim is now to explain how the mapping $X \mapsto T(X)$ lifts to a functor on the category of lenses—i.e., for each module X, how to construct a module $T(X)$ whose

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

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

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

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

3.4.4 Definition [Container edits]: Given container T and module X we define edits for $T(|X|)$ as follows (we give some intuition after Definition 3.4.5):

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

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

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

$\mathsf{fail}\ (i, f)$ is always undefined

$\mathsf{mod}(p, \mathrm{d}x)\ (i, f) = (i, f[p \mapsto \mathrm{d}x\ f(p)])$ when $p \in \mathsf{live}(i)$

$\mathsf{ins}(\mathrm{d}i)\ (i, f) = (\mathrm{d}i\ i, f')$
where $f'(p) =$ if $p \in \mathsf{live}(i)$ then $f(p)$ else $init_X$

$\mathsf{del}(\mathrm{d}i)\ (i, f) = (\mathrm{d}i\ i, f{\restriction}\mathsf{live}(\mathrm{d}i\ i))$

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

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

The $\mathsf{rearr}(\mathrm{d}i, f)$ edit, finally, changes the shape of a container but neither adds nor removes entries. As already mentioned, a typical example is the left-rotation of a binary tree about the root. This rotation applies whenever the root has two grandchildren to the left and a child to the right. For this example, one may worry about the size of f, since it affects many positions; however, it can be serialized to a small, three line if-then-else. We do not, at this point, provide edits that copy the contents of some position into other positions; their investigation is left for future work.

We define the monoid $\partial T(X)$ as the free monoid generated by the basic edits defined above. In §3.5 we discuss the possibility of imposing equational laws, in particular with a view to compact normal forms of container edits.

Setting $init_{T(X)} = (init_I, \lambda p.init_X)$ when $T = \langle I, P, \mathsf{live}\rangle$ completes the definition of the module $T(X)$. Figure 3.9 gives the definition of a mapping lens for arbitrary container types. Given that this definition looks complex at first we state and prove explicitly that it is indeed a lens.

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

Proof: We begin by unraveling the definition. The complement of the $T(\ell)$ lens is itself a container of ℓ-complements; thus, even if ℓ has a trivial complement the complement in $T(\ell)$ can be nontrivial. The consistency relation requires that the shapes of the left and right states agree with the shape of the complement and that matching entries are consistent in the sense of ℓ.

A $\mathsf{mod}(p, \mathrm{d}x)$ edit is transported by sending $\mathrm{d}x$ through ℓ using the appropriate ℓ-complements contained in the complement (i, f) of the mapping lens. When no such ℓ-complement is available, the lens returns fail. If $((i, f), (j, g), (i', f')) \in K$ and $\mathsf{mod}(p, \mathrm{d}x)(i, f)\downarrow$, then $p \in \mathsf{live}(i)$, hence $p \in \mathsf{live}(j)$ and $p \in \mathsf{live}(i')$. So the result of propagating $\mathsf{mod}(p, \mathrm{d}x)$ will be $\mathsf{mod}(p, \mathrm{d}y)$ where $(\mathrm{d}y, c') = \ell.{\Rrightarrow}(\mathrm{d}x, g(p))$. Now since $(f(p), g(p), f'(p)) \in \ell.K$, we know that $\mathrm{d}y\ f'(p)\downarrow$ and $(\mathrm{d}x\ f(p), c', \mathrm{d}y\ f'(p)) \in \ell.K$. It follows that $\mathsf{mod}(p, \mathrm{d}y)\ (i', f')\downarrow$ and the new triple is again in K.

As success or failure of the other edit operations only depends on the shape, it is clear that success is preserved by the mapping lens when starting from a consistent triple. We must argue, though, that the resulting triples remain consistent. We show

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

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

Figure 3.9: Generic container-mapping lens

how this argument works using $\mathsf{rearr}(\mathrm{d}i, h)$ as an example. The resulting triple is $((\mathrm{d}i\ i, f \circ h(i)), (\mathrm{d}i\ i, g \circ h(i)), (\mathrm{d}i\ i, f' \circ h(i)))$. Now, since $h(i) \in \mathsf{live}(\mathrm{d}i\ i) \simeq \mathsf{live}(i)$ must be a bijection it follows immediately that this triple is in K.

Compatibility of $\Rrightarrow, \Lleftarrow$ with monoid multiplication is trivial here since the edit monoid for containers is freely generated.

Let $T(k); T(\ell)$ be the composition of two mapping lenses. The complement of this lens is $T(k.C) \times T(\ell.C)$. On the other hand, the complement of $T(k; \ell)$ is $T(k.C \times \ell.C)$. An appropriate simulation relation is defined by

$$\{(((i, g_k), (i, g_\ell)), (i, g_{k;\ell})) \mid \forall p. g_{k;\ell}(p) = (g_k(p), g_\ell(p))\}.$$

We omit the straightforward verification. We also have to show that $T(-)$ is well-defined on equivalence classes, so assume that $\ell \equiv k \in X \leftrightarrow Y$ and let $S \subseteq X \times \ell.C \times k.C \times Y$ be a witnessing simulation relation, cf. Thm. 3.2.11.

The following relation $T(S)$ then witnesses $T(\ell) \equiv T(k)$.

$$\begin{aligned} T(S) = \{&(i, f), (i, g), (i, g'), (i, f') \\ &\mid i \in I \wedge \forall p. (f(p), g(p), g'(p), f'(p)) \in S\} \end{aligned}$$

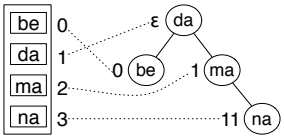
We omit the straightforward verification. □

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

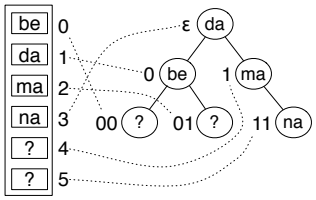
One typical instance of this kind of lens is list reversal; another is a lens between trees and lists which ensures that the list entries agree with the tree entries according to some fixed order, e.g. in-order or breadth first. Although the live positions of the containers to be synchronized are in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

To model a list, we take $I = \mathbb{N}$; $P = \mathbb{N}$; $\mathsf{live}(i) = \{p \mid p < i\}$; and for trees, I' comprises prefix closed subsets of $\{0, 1\}^\star$; $P' = \{0, 1\}^\star$; $\mathsf{live}'(i') = i'$. The monoid ∂I has increment and decrement operations; the monoid $\partial I'$ has operations for adding and removing nodes in leaf positions and also for rotating tree shapes. We illustrate the propagation of an $\mathsf{ins}(\mathrm{d}i)$ edit—which is one of the more complex cases.

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

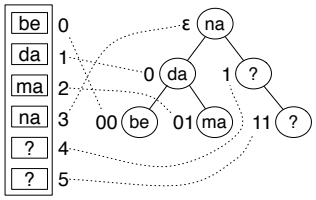


The family of bijections $f_{i,c,i'}$ models the in-order correspondence; thus, for example if $i = 4$ and $i' = \{\langle\rangle, \langle 0\rangle, \langle 1\rangle, \langle 1,1\rangle\}$ the bijection would be as shown above. (For illustration we also indicate possible X-contents of the positions.)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



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

Since the newly inserted elements in the list come at the end, we can restore consistency by moving the newly inserted tree elements to positions that come at the end of the in-order walk. This can be done essentially automatically just using the in-order walk position bijections: to decide where a position p in the old tree should reappear in the new tree, we can simply follow the position through its journey of being flattened and unflattened using $f_{i,c,i'}$ and $f^{-1}_{\mathrm{d}i\, i,c',\mathrm{d}i'\, i'}$, respectively. Thus to restore consistency we apply $\mathsf{rearr}(\mathbf{1}, f_{\mathsf{i}})$ where $f_{\mathsf{i}}(i') = \{(\langle 0,0\rangle, \langle 0\rangle), (\langle 0\rangle, \langle\rangle), (\langle 0,1\rangle, \langle 1\rangle), (\langle\rangle, \langle 1,1\rangle), (\langle 1\rangle, \langle 0,0\rangle), (\langle 1,1\rangle, \langle 0,1\rangle)\}$. We could also have chosen $f_{\mathsf{i}}(i') = \{\ldots, (\langle 1\rangle, \langle 0,1\rangle), (\langle 1,1\rangle, \langle 0,1\rangle)\}$; since in any case the new positions are uninitialized, this free choice has little impact. Of course $f_{\mathsf{i}}(i'')$ for $i'' \neq i'$ is also completely unconstrained. After applying $\mathsf{rearr}(\mathbf{1}, f_{\mathsf{i}})$ we end up with the desired consistent state.



With this intuition in hand, we are ready to see the details of the restructuring lens. As discussed above, we must have containers T and T', an edit lens ℓ between their shapes, and a family of bijections between live sets. We also require that ℓ maps insertions to insertions, deletions to deletions, and rearrangements to rearrangements. (This is well-defined on equivalence classes of lenses.) Given these data, we define the restructuring lens in Figure 3.10, with a few supplementary definitions below. The additional families of bijections $f_{\mathsf{i}}, f_{\mathsf{d}}, f_{\mathsf{r}}$ must be chosen in such a way that the container edits in which they appear are well-formed (this is possible since $\mathrm{d}i'$ is an insertion, deletion, or restructuring as appropriate) and such that the following three constraints are satisfied: in each case i, i', etc., refer to the current values from above

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

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

Figure 3.10: Container restructuring lens

and $p \in \mathsf{live}'(\mathrm{d}i'\ i')$ is an arbitrary position.

$$\begin{aligned} f_{\mathsf{i}}(\mathrm{d}i'\ i')(p) &= f^{-1}_{i,c,i'}(f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p)) \\ &\qquad \text{when } f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p) \in \mathsf{live}(i) \\ f_{\mathsf{d}}(i')(p) &= f^{-1}_{i,c,i'}(f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p)) \\ f_{\mathsf{r}}(i')(p) &= f^{-1}_{i,c,i'}(g(i)(f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p))) \end{aligned}$$

These conditions do not completely determine f_{i}, f_{d}, and f_{r}. In each case, these families are completely unconstrained on shapes other than $\mathrm{d}i'\ i'$. The propagated edits are supposed to be applied to a container of the current shape i', so the arbitrary decisions about other shapes do not really matter; nevertheless it would be nice if we could be a bit more uniform. This is indeed possible in the case where ℓ is an isomorphism lens, but we refrain from formulating details.

As discussed in the example above, the bijection f_{i} contains a little more choice, namely the behavior on the T' positions in $f^{-1}_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(\mathsf{live}(\mathrm{d}i\ i) \setminus \mathsf{live}(i))$. Fortunately, they all contain $init_X$ so that the choice does not affect the resulting state after application of the edit, and the alignment is decided not by f_{i} but by the family of bijections $f_{i,c',i'}$ that parameterize the lens.

3.4.7 Theorem: The restructuring lens is indeed a lens.

Proof: As the edit monoid is free, we only need to show that successful edits to consistent states get transported to successful edits resulting in consistent states. Thus suppose that $(i, c, i') \in \ell.K$ and $f(f_{i,c,i'}(p)) = f'(p)$ holds for all $p \in \mathsf{live}'(i')$ so that $((i, f), (i, c, i'), (i', f'))$ are consistent. We will show below that $\Rrightarrow$ is correct; the proof about $\Lleftarrow$ is very similar. In the cases below where there is an edit named $\mathrm{d}i$, we will write $(\mathrm{d}i', c') = \ell.\Rrightarrow(\mathrm{d}i, c)$ and abbreviate the bijections $f_{i,c,i'}$ and $f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}$ to f_{pre} and f_{post}, respectively.

Case fail is obvious.

Case $\mathsf{mod}(p, \mathrm{d}x)$: the complement does not change and the edit $\mathrm{d}x$ is applied to the same elements.

Case $\mathsf{ins}(\mathrm{d}i)$. The resulting new repository states are $(\mathrm{d}i\ i, f_1)$ and $(\mathrm{d}i'\ i', f_1')$:

$$\begin{aligned} f_1(p) &= \text{if } p \in \mathsf{live}(i) \text{ then } f(p) \text{ else } init_X \\ f_1'(p) &= \text{if } f_{\mathsf{i}}(\mathrm{d}i'\ i')(p) \in \mathsf{live}(i') \text{ then } f'(f_{\mathsf{i}}(\mathrm{d}i'\ i')(p)) \text{ else } init_X \end{aligned}$$

Also, the bijection $f_{\mathsf{i}}(\mathrm{d}i'\ i') \in \mathsf{live}'(\mathrm{d}i'\ i') \simeq \mathsf{live}'(\mathrm{d}i'\ i')$ satisfies

$$f_{post}(p) \in \mathsf{live}(i) \Rightarrow f_{\mathsf{i}}(\mathrm{d}i'\ i')(p) = f^{-1}_{pre}(f_{post}(p)).$$

This assumption lets us conclude that $f_{post}(p) \in \mathsf{live}(i)$ if and only if $f_{\mathsf{i}}(\mathrm{d}i'\ i')(p) \in$

$\mathsf{live}'(i')$. In the forward direction, we argue:

$$\begin{aligned}
f_{post}(p) &\in \mathsf{live}(i) && \text{assumption}\\
f_{pre}^{-1}(f_{post}(p)) &\in \mathsf{live}'(i') && f_{pre} \in \mathsf{live}(i) \simeq \mathsf{live}'(i')\\
f_{\mathsf{i}}(\mathrm{d}i'\ i')(p) &\in \mathsf{live}'(i') && \text{assumed condition of } f_{\mathsf{i}}
\end{aligned}$$

In the backward direction, define $q = f_{\mathsf{i}}(\mathrm{d}i'\ i')(p)$. Then:

$$\begin{aligned}
f_{\mathsf{i}}(\mathrm{d}i'\ i')(p) &\in \mathsf{live}'(i') && \text{assumption}\\
q &\in \mathsf{live}'(i') && \text{definition of } q\\
f_{pre}(q) &\in \mathsf{live}(i) && f_{pre} \in \mathsf{live}(i) \simeq \mathsf{live}'(i')\\
f_{post}(f_{post}^{-1}(f_{pre}(q))) &\in \mathsf{live}(i) && f_{post} \text{ is a bijection}\\
f_{pre}^{-1}(f_{post}(f_{post}^{-1}(f_{pre}(q)))) &= q && f_{post} \text{ and } f_{pre} \text{ are bijections}\\
f_{\mathsf{i}}(\mathrm{d}i'\ i')(f_{post}^{-1}(f_{pre}(q))) &= q && \text{assumed condition of } f_{\mathsf{i}}\\
f_{post}^{-1}(f_{pre}(q)) &= p && f_{\mathsf{i}}(\mathrm{d}i'\ i') \text{ is a bijection and}\\
& && \text{definition of } q\\
f_{post}(p) &\in \mathsf{live}(i) && f_{pre}(q) \in \mathsf{live}(i)
\end{aligned}$$

To conclude the case, we must show that arbitrary $p \in \mathsf{live}'(\mathrm{d}i'\ i')$ have $f_1'(p) = f_1(f_{post}(p))$. We consider two cases: either $f_{post}(p) \in \mathsf{live}(i)$ or not. If so then $f_1(f_{post}(p)) = f(f_{post}(p)) = f'(f_{pre}^{-1}(f_{post}(p))) = f_1'(p)$ where the first equation uses the above characterization of f_1; the second one uses consistency of f and f', and the third one uses the characterizations of f_1' and f_{i} (noting that $f_{post}(p) \in \mathsf{live}(i)$ implies $f_{\mathsf{i}}(\mathrm{d}i'\ i')(p) \in \mathsf{live}'(i')$). In the other case, $f_{post}(p) \notin \mathsf{live}(i)$, so $f_{\mathsf{i}}(\mathrm{d}i'\ i')(p) \notin \mathsf{live}'(i')$. Then $f_1'(p) = init_X$, but $f_1(f_{post}(p)) = init_X$, too, by the characterization of f_1.

Case $\mathsf{del}(\mathrm{d}i)$. Ignoring domain restrictions, the new repository states are $(\mathrm{d}i\ i, f)$ and $(\mathrm{d}i'\ i', f_{\mathsf{d}}(i'); f')$. For these to be consistent, we must show that $f(f_{post}(p)) = f'(f_{\mathsf{d}}(i')(p))$ whenever $p \in \mathsf{live}'(\mathrm{d}i'\ i')$. Since $\mathrm{d}i$ is a deletion, we know $\mathrm{d}i\ i \subset i$, so that $f_{post} \in \mathsf{live}(\mathrm{d}i\ i) \simeq \mathsf{live}'(\mathrm{d}i'\ i')$ implies $f_{post}(p) \in \mathsf{live}(i)$. Hence we can equate:

$$\begin{aligned}
f(f_{post}(p)) &= f'(f_{pre}^{-1}(f_{post}(p))) && \text{consistency of } f \text{ and } f'\\
&= f'(f_{\mathsf{d}}(i')(p)) && \text{assumed condition of } f_{\mathsf{d}}
\end{aligned}$$

Case $\mathsf{rearr}(\mathrm{d}i, g)$. The new repository states that we must show are consistent are $(\mathrm{d}i\ i, g(i); f)$ and $(\mathrm{d}i'\ i', f_{\mathsf{r}}(i'); f')$. Consider arbitrary $p \in \mathsf{live}'(\mathrm{d}i'\ i')$. Since $f_{post}; g(i) \in \mathsf{live}'(\mathrm{d}i'\ i') \simeq \mathsf{live}(i)$, we know $g(i)(f_{post}(p)) \in \mathsf{live}(i)$; this justifies the first equation below.

$$\begin{aligned}
f(g(i)(f_{post}(p))) &= f'(f_{pre}^{-1}(g(i)(f_{post}(p)))) && \text{consistency of } f \text{ and } f'\\
&= f'(f_{\mathsf{r}}(i')(p)) && \text{condition of } f_{\mathsf{r}}
\end{aligned}$$

□

Using the container lens combinators, the partition lens and lens mediating between "built-in" lists and "list containers" we can then plumb together a variety of useful lenses, e.g. one that partitions the entries of an $X + Y$ labeled tree into *inl*s and *inr*s and then presents the two resulting containers again as trees over X and Y. If one wants one can then use a mapping lens to change the representation of the Y's in some way.

3.5 Adding Monoid Laws

The edit languages accompanying the constructions in the previous two sections were all freely generated. This was a good place to begin as it is relatively easy to understand, but, as discussed in §3.2, there are good reasons for investigating richer languages. This section takes a first step in this direction by showing how to equip the product and sum combinators with more interesting edits.

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

$$\begin{aligned} X \otimes Y &= \langle |X| \times |Y|, (\mathit{init}_X, \mathit{init}_Y), \partial X \otimes \partial Y, \odot_{X \otimes Y} \rangle \\ \mathbf{1}_{M \otimes N} &= (\mathbf{1}_M, \mathbf{1}_N) \\ (m, n) \cdot_{M \otimes N} (m', n') &= (m\, m', n\, n') \\ (\mathrm{d}x, \mathrm{d}y) \odot_{X \otimes Y} (x, y) &= (\mathrm{d}x\, x, \mathrm{d}y\, y) \end{aligned}$$

3.5.1 Lemma: These definitions give rise to a module—that is, $\cdot_{M \otimes N}$ is associative with identity $\mathbf{1}_{M \otimes N}$ and $\odot_{X \otimes Y}$ satisfies the monoid action laws.

Proof: To show that $\cdot_{M \otimes N}$ is associative, using the fact that $\cdot_M$ and $\cdot_N$ are associative:

$$\begin{aligned} v_1 \cdot (v_2 \cdot v_3) &= (m_1, n_1) \cdot ((m_2, n_2) \cdot (m_3, n_3)) \\ &= (m_1, n_1) \cdot (m_2 \cdot m_3, n_2 \cdot n_3) \\ &= (m_1 \cdot (m_2 \cdot m_3), n_1 \cdot (n_2 \cdot n_3)) \\ &= ((m_1 \cdot m_2) \cdot m_3, (n_1 \cdot n_2) \cdot n_3) \\ &= (m_1 \cdot m_2, n_1 \cdot n_2) \cdot (m_3, n_3) \\ &= ((m_1, n_1) \cdot (m_2, n_2)) \cdot (m_3, n_3) \\ &= (v_1 \cdot v_2) \cdot v_3 \end{aligned}$$

To show that $\mathbf{1}_{M \otimes N}$ is an identity for $\cdot_{M \otimes N}$, assuming $\mathbf{1}_M$ and $\mathbf{1}_N$ are the respective

identities for $\cdot_M$ and $\cdot_N$:

$$\begin{aligned}(\mathbf{1},\mathbf{1}) \cdot (m,n) &= (\mathbf{1} \cdot m, \mathbf{1} \cdot n)\\ &= (m,n)\\ &= (m \cdot \mathbf{1}, n \cdot \mathbf{1})\\ &= (m,n) \cdot (\mathbf{1},\mathbf{1})\end{aligned}$$

To show the monoid action laws are satisfied by $\odot_{M\otimes N}$, assuming these laws are satisfied by $\odot_M$ and $\odot_N$:

$$\begin{aligned}(\mathbf{1},\mathbf{1}) \odot (x,y) &= (\mathbf{1} \odot x, \mathbf{1} \odot y)\\ &= (x,y)\\ ((m,n) \cdot (m',n')) \odot (x,y) &= (m \cdot m', n \cdot n') \odot (x,y)\\ &= ((m \cdot m') \odot x, (n \cdot n') \odot y)\\ &= (m \odot m' \odot x, n \odot n' \odot y)\\ &= (m,n) \odot (m' \odot x, n' \odot y)\\ &= (m,n) \odot (m',n') \odot (x,y)\end{aligned}$$

□

One might wonder whether the standard definition has any connection to the definition we give earlier. One way to bridge the gap is to add equational laws to the free monoid.[6] The equations below demand that left and right be monoid homomorphisms, and that they commute:

$$\begin{aligned}\langle \mathsf{left}(\mathbf{1}) \rangle &= \langle\rangle\\ \langle \mathsf{left}(\mathrm{d}x), \mathsf{left}(\mathrm{d}x') \rangle &= \langle \mathsf{left}(\mathrm{d}x\ \mathrm{d}x') \rangle\\ \langle \mathsf{right}(\mathbf{1}) \rangle &= \langle\rangle\\ \langle \mathsf{right}(\mathrm{d}y), \mathsf{right}(\mathrm{d}y') \rangle &= \langle \mathsf{right}(\mathrm{d}y\ \mathrm{d}y') \rangle\\ \langle \mathsf{left}(\mathrm{d}x), \mathsf{right}(\mathrm{d}y) \rangle &= \langle \mathsf{right}(\mathrm{d}y), \mathsf{left}(\mathrm{d}x) \rangle\end{aligned}$$

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

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

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

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

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

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

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

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

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

3.5.2 Lemma: Suppose k and ℓ are lenses. For each of the equations above, if that equation is in force in the modules on both sides of the $k \otimes \ell$ lens, then the $\Rrightarrow$ and $\Lleftarrow$ functions defined above for this lens respect that equation.

Proof: We will show that $\Rrightarrow$ treats left as a monoid homomorphism and lets left and right commute; the proofs that $\Rrightarrow$ treats right as a monoid homomorphism and that $\Lleftarrow_{\mathsf{g}}$ respects all these laws are similar.

To show that $\Rrightarrow$ respects the law $\langle \mathsf{left}(\mathbf{1}) \rangle = \langle\rangle$:

$$\begin{aligned}
\Rrightarrow(\langle \mathsf{left}(\mathbf{1}) \rangle, (c_k, c_\ell)) &= \mathsf{let}\ (\mathrm{d}z, c'_k) = k.\Rrightarrow(\mathbf{1}, c_k)\ \mathsf{in} \\
&\quad (\langle \mathsf{left}(\mathrm{d}z) \rangle, (c'_k, c_\ell)) \\
&= (\langle \mathsf{left}(\mathbf{1}) \rangle, (c_k, c_\ell)) && (3.5.1) \\
&= (\langle\rangle, (c_k, c_\ell)) && (3.5.2) \\
&= \Rrightarrow(\langle\rangle, (c_k, c_\ell)) && (3.5.3)
\end{aligned}$$

Equation 3.5.1 follows because k is a lens and hence $k.\Rrightarrow$ is a stateful monoid homomorphism. Equation 3.5.2 follows by assumption, and equation 3.5.3 follows by definition of $\Rrightarrow$.

Next we will show that $\Rrightarrow$ respects the law $\langle \mathsf{left}(\mathrm{d}x), \mathsf{left}(\mathrm{d}x') \rangle = \langle \mathsf{left}(\mathrm{d}x\mathrm{d}x') \rangle$. It will be convenient to name a few things. Pick a state c_k and define:

$$\begin{aligned}
(\mathrm{d}y', c'_k) &= k.\Rrightarrow(\mathrm{d}x', c_k) \\
(\mathrm{d}y, c''_k) &= k.\Rrightarrow(\mathrm{d}x, c'_k)
\end{aligned}$$

Since k is a lens and hence in particular $k.\Rrightarrow$ is a stateful monoid homomorphism, we can conclude that:

$$k.\Rrightarrow(\mathrm{d}x\mathrm{d}x', c_k) = (\mathrm{d}y\mathrm{d}y', c_k'')$$

We may now compute:

$$\begin{aligned}
&\Rrightarrow(\langle \mathsf{left}(\mathrm{d}x), \mathsf{left}(\mathrm{d}x')\rangle, (c_k, c_\ell)) \\
&\quad = \mathsf{let}\ (\mathrm{d}y', (c_k', c_\ell')) = \Rrightarrow_{\mathsf{g}}(\mathsf{left}(\mathrm{d}x'), (c_k, c_\ell))\ \mathsf{in} \\
&\qquad \mathsf{let}\ (\mathrm{d}y'', (c_k'', c_\ell'')) = \Rrightarrow_{\mathsf{g}}(\mathsf{left}(\mathrm{d}x), (c_k', c_\ell'))\ \mathsf{in} \\
&\qquad (\langle \mathrm{d}y'', \mathrm{d}y'\rangle, (c_k'', c_\ell'')) \\
&\quad = \mathsf{let}\ (\mathrm{d}y'', (c_k'', c_\ell'')) = \Rrightarrow_{\mathsf{g}}(\mathsf{left}(\mathrm{d}x), (c_k', c_\ell))\ \mathsf{in} \\
&\qquad (\langle \mathrm{d}y'', \mathsf{left}(\mathrm{d}y')\rangle, (c_k'', c_\ell'')) \\
&\quad = (\langle \mathsf{left}(\mathrm{d}y), \mathsf{left}(\mathrm{d}y')\rangle, (c_k'', c_\ell)) \\
&\quad = (\langle \mathsf{left}(\mathrm{d}y\mathrm{d}y')\rangle, (c_k'', c_\ell)) \\
&\quad = \Rrightarrow_{\mathsf{g}}(\mathsf{left}(\mathrm{d}x\mathrm{d}x'), (c_k, c_\ell)) \\
&\quad = \Rrightarrow(\langle \mathsf{left}(\mathrm{d}x\mathrm{d}x')\rangle, (c_k, c_\ell))
\end{aligned}$$

The final equation to preserve is $\langle \mathsf{left}(\mathrm{d}x), \mathsf{right}(\mathrm{d}y)\rangle = \langle \mathsf{right}(\mathrm{d}y), \mathsf{left}(\mathrm{d}x)\rangle$. As before, we choose a c_k and c_ℓ and name a few intermediate computations:

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

Now we may compute:

$$\begin{aligned}
&\Rrightarrow(\langle \mathsf{left}(\mathrm{d}x), \mathsf{right}(\mathrm{d}y)\rangle, (c_k, c_\ell)) \\
&\quad = \mathsf{let}\ (\mathrm{d}y', (c_k', c_\ell')) = \Rrightarrow_{\mathsf{g}}(\mathsf{right}(\mathrm{d}y), (c_k, c_\ell))\ \mathsf{in} \\
&\qquad \mathsf{let}\ (\mathrm{d}x', (c_k'', c_\ell'')) = \Rrightarrow_{\mathsf{g}}(\mathsf{left}(\mathrm{d}x), (c_k', c_\ell'))\ \mathsf{in} \\
&\qquad (\langle \mathrm{d}x', \mathrm{d}y'\rangle, (c_k'', c_\ell'')) \\
&\quad = \mathsf{let}\ (\mathrm{d}x', (c_k'', c_\ell'')) = \Rrightarrow_{\mathsf{g}}(\mathsf{left}(\mathrm{d}x), (c_k, c_\ell'))\ \mathsf{in} \\
&\qquad (\langle \mathrm{d}x', \mathsf{right}(\mathrm{d}y')\rangle, (c_k'', c_\ell'')) \\
&\quad = (\langle \mathsf{left}(\mathrm{d}x'), \mathsf{right}(\mathrm{d}y')\rangle, (c_k', c_\ell')) \\
&\quad = (\langle \mathsf{right}(\mathrm{d}y'), \mathsf{left}(\mathrm{d}x')\rangle, (c_k', c_\ell')) \\
&\quad = \Rrightarrow(\langle \mathsf{right}(\mathrm{d}y), \mathsf{left}(\mathrm{d}x)\rangle, (c_k, c_\ell))
\end{aligned}$$

The final line follows from the previous one by an argument almost identical (but reversed) to the argument showing that the second-to-last line follows from the first. □

Adding the first four equations lets us create a projection lens out of smaller parts by observing that there are some new isomorphisms available.

3.5.3 Definition [Projection lenses]: Let f and g be the obvious isomorphisms

connecting $X \otimes \mathit{Unit}$ to X and $\mathit{Unit} \otimes Y$ to Y.[7]

$$\pi_1 = (id_X \otimes term_Y); bij_f$$
$$\pi_2 = (term_X \otimes id_Y); bij_g$$

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

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

$$\begin{aligned}
\langle \mathsf{switch}_{jk}(m), \mathsf{switch}_{ij}(m') \rangle &= \langle \mathsf{switch}_{ik}(m) \rangle \\
\langle \mathsf{switch}_{ij}(m), \mathsf{stay}_i(m') \rangle &= \langle \mathsf{switch}_{ij}(m) \rangle \\
\langle \mathsf{stay}_j(m), \mathsf{switch}_{ij}(m') \rangle &= \langle \mathsf{switch}_{ij}(mm') \rangle \\
\langle \mathsf{stay}_i(m), \mathsf{stay}_i(m') \rangle &= \langle \mathsf{stay}_i(mm') \rangle \\
\langle d, d' \rangle &= \langle \mathsf{fail} \rangle \qquad \text{in all other cases}
\end{aligned}$$

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

As with products, we must check that the remaining definitions are well-formed.

3.5.4 Lemma: In the module defined above for sums, $\odot$ respects the above equations.

Proof: We will give proofs for the first four equations with i, j, and k instantiated to L (proofs for other instantiations are nearly identical). The final equation is respected because every pair of atomic edits not listed in the first four equations results in an edit that cannot be successfully applied to any value (just like the fail edit itself).

For each of the four equations (instantiated to L everywhere) $e = e'$, both $e \odot \mathsf{inr}(y)$

[7] Unlike the analogous state-based lenses from Chapter 2, these projections are *not* parameterized by an element of the set that is being projected away. Never fear: this element is still available, as the *init* value of the appropriate module.

and $e' \odot \mathsf{inr}(y)$ are undefined, so we focus on $e \odot \mathsf{inl}(x)$ and $e' \odot \mathsf{inl}(x)$.

$$\begin{aligned}
\langle \mathsf{switch}_{LL}(m), \mathsf{switch}_{LL}(m') \rangle \odot \mathsf{inl}(x) &= \langle \mathsf{switch}_{LL}(m) \rangle \odot \mathsf{inl}(m' \odot \mathit{init}) \\
&= \mathsf{inl}(m \odot \mathit{init}) \\
&= \langle \mathsf{switch}_{LL}(m) \rangle \odot \mathsf{inl}(x) \\
\langle \mathsf{switch}_{LL}(m), \mathsf{stay}_{L}(m') \rangle \odot \mathsf{inl}(x) &= \langle \mathsf{switch}_{LL}(m) \rangle \odot \mathsf{inl}(m' \odot x) \\
&= \mathsf{inl}(m \odot \mathit{init}) \\
&= \langle \mathsf{switch}_{LL}(m) \rangle \odot \mathsf{inl}(x) \\
\langle \mathsf{stay}_{L}(m), \mathsf{switch}_{LL}(m') \rangle \odot \mathsf{inl}(x) &= \langle \mathsf{stay}_{L}(m) \rangle \odot \mathsf{inl}(m' \odot \mathit{init}) \\
&= \mathsf{inl}(m \odot m' \odot \mathit{init}) \\
&= \mathsf{inl}(mm' \odot \mathit{init}) \\
&= \langle \mathsf{switch}_{LL}(mm') \rangle \odot \mathsf{inl}(x) \\
\langle \mathsf{stay}_{L}(m), \mathsf{stay}_{L}(m') \rangle \odot \mathsf{inl}(x) &= \langle \mathsf{stay}_{L}(m) \rangle \odot \mathsf{inl}(m' \odot x) \\
&= \mathsf{inl}(m \odot m' \odot x) \\
&= \mathsf{inl}(mm' \odot x) \\
&= \langle \mathsf{stay}_{L}(mm') \rangle \odot \mathsf{inl}(x) \qquad \square
\end{aligned}$$

3.5.5 Lemma: If k and ℓ are lenses, then $(k \oplus \ell).\Rrightarrow_{\mathsf{g}}$ and $(k \oplus \ell).\Lleftarrow_{\mathsf{g}}$ respect the above equations.

Proof: We will show only that $\Rrightarrow_{\mathsf{g}}$ respects the equations; the argument for $\Lleftarrow_{\mathsf{g}}$ is similar. Choose arbitrary sum edits e_1, e_2 and initial complement $c_0 \in C$ and define:

$$\begin{aligned}
e_{12} &= e_2 e_1 \\
(f_1, c_1) &= \Rrightarrow(e_1, c_0) \\
(f_2, c_2) &= \Rrightarrow(e_2, c_1) \\
(f_{12}, c_{12}) &= \Rrightarrow(e_2 e_1, c_0)
\end{aligned}$$

We must show that $f_{12} = f_2 f_1$ and $c_{12} = c_2$. We will go by case analysis on e_1 and e_2; however we can first rule out a few broad categories of such cases. When $\Rrightarrow(e_1, c) = (\mathsf{fail}, \mathsf{failed})$ fails, there is very little to prove; we know that $\Rrightarrow(e_2, \mathsf{failed}) = (f_2, \mathsf{failed})$ for some f_2, and hence that $f_2 f_1 = \mathsf{fail}$. Furthermore, it is not hard to see by inspecting the cases where $\Rrightarrow(e_1, c)$ fails that $\Rrightarrow(e_{12}, c)$ will also fail for any e_2. Hence $f_{12} = \mathsf{fail} = f_2 f_1$ and $c_{12} = \mathsf{failed} = c_2$. Similarly, when e_2 results in a failure, any combined edit e_{12} will also result in failure. As a final broad category, when the lens has already failed (that is, when $c_0 = \mathsf{failed}$), we observe that the lens preserves the "constructor" of the edit. Since the monoid multiplication inspects only the constructor, the required equation $f_2 f_1 = f_{12}$ will hold, and we will have $c_{12} = \mathsf{failed} = c_2$.

In the following, we therefore assume that no failure occurs. A few definitions will

be convenient:

$$\begin{array}{lll} s(L, \mathrm{d}x) = k.\Rrightarrow(\mathrm{d}x, k.\mathit{missing}) & t(L, x) = \mathsf{inl}(x) & u(\mathsf{inl}(x)) = L \\ s(R, \mathrm{d}x) = \ell.\Rrightarrow(\mathrm{d}x, \ell.\mathit{missing}) & t(R, y) = \mathsf{inr}(y) & u(\mathsf{inr}(y)) = R \\ & & u(\mathsf{failed}) = \mathsf{failed} \end{array}$$

The cases now proceed as follows:

Case $e_1 = \mathsf{switch}_{hi}(\mathrm{d}x'), e_2 = \mathsf{switch}_{ij}(\mathrm{d}x)$: We have $e_2 e_1 = \mathsf{switch}_{hj}(\mathrm{d}x)$. Since we consider only non-failing cases, we know $u(c_0) = h$—that is to say, the complement and the edit are consistent, and we are translating a "sensible" edit. We know two things: first, $f_1 = \mathsf{switch}_{hi}(\mathrm{d}y')$ for some $\mathrm{d}y'$, and second, $u(c_1) = i$. From there, we can define $(\mathrm{d}y, c) = s(j, \mathrm{d}x)$, so that $(f_2 f_1, c_2) = (\mathsf{switch}_{hj}(\mathrm{d}y), t(j, c))$. Moreover, simple calculation (again observing that $u(c_0) = h$ and hence that e_{12} is a sensible edit to apply, according to the complement) shows that also $(f_{12}, c_{12}) = (\mathsf{switch}_{hj}(\mathrm{d}y), t(j, c))$, which is equal to the previous tuple, as desired.

Case $e_1 = \mathsf{stay}_i(\mathrm{d}x), e_2 = \mathsf{switch}_{ij}(\mathrm{d}x')$: Since we know no failure occurs, we must have $u(c_0) = i$. Therefore, e_2 is a sensible edit to apply, and so

$$\Rrightarrow(e_2, c_0) = (\mathsf{switch}_{ij}(\mathrm{d}y'), t(j, c))$$

where $(\mathrm{d}y', c) = s(j, \mathrm{d}x')$. Furthermore, e_1 is a sensible edit to apply, so we know $u(c_1) = i$, and hence that $(f_2, c_2) = (\mathsf{switch}_{ij}(\mathrm{d}y'), t(j, c))$. Furthermore, since $f_1 = \mathsf{stay}_i(\mathrm{d}y)$ for some $\mathrm{d}y$, we know $f_2 f_1 = f_2$. But since $e_{12} = e_2$,

$$\begin{aligned} (f_{12}, c_{12}) &= \Rrightarrow(e_{12}, c_0) \\ &= \Rrightarrow(e_2, c_0) \\ &= \Rrightarrow(e_2, c_1) \\ &= (f_2, c_2) \\ &= (f_2 f_1, c_2) \end{aligned}$$

as desired.

Case $e_1 = \mathsf{switch}_{ij}(\mathrm{d}x'), e_2 = \mathsf{stay}_j(\mathrm{d}x)$: Again, we observe that we must have $u(c_0) = i$. Therefore we simply appeal to the homomorphism laws for $k.\Rrightarrow$ (when $j = L$) or $\ell.\Rrightarrow$ (when $j = R$). For example, when $j = L$ and hence $c_0 = \mathsf{inl}(c)$, we can define:

$$\begin{aligned} (\mathrm{d}y', c') &= k.\Rrightarrow(\mathrm{d}x', k.\mathit{missing}) \\ (\mathrm{d}y, c'') &= k.\Rrightarrow(\mathrm{d}x, c') \\ (\mathrm{d}y'', c''') &= k.\Rrightarrow(\mathrm{d}x\mathrm{d}x', k.\mathit{missing}) \end{aligned}$$

Then, by computation:

$$\begin{aligned}(f_1, c_1) &= (\mathsf{switch}_{il}(\mathrm{d}y'), \mathsf{inl}(c')) \\ (f_2, c_2) &= (\mathsf{stay}_L(\mathrm{d}y), \mathsf{inl}(c'')) \\ (f_{12}, c_{12}) &= (\mathsf{switch}_{il}(\mathrm{d}y''), \mathsf{inl}(c''')) \\ f_2 f_1 &= \mathsf{switch}_{il}(\mathrm{d}y\mathrm{d}y')\end{aligned}$$

Finally, appealing to $k.\Rrightarrow$'s homomorphism law, we conclude $\mathrm{d}y'' = \mathrm{d}y\mathrm{d}y'$ and $c''' = c''$, and hence that $(f_2 f_1, c_2) = (f_{12}, c_{12})$.

Case $e_1 = \mathsf{stay}_i(\mathrm{d}x'), e_2 = \mathsf{stay}_i(\mathrm{d}x)$: Much like the previous case, since $u(c_0) = i$, we appeal directly to the homomorphism law for the underlying $\Rrightarrow$ operations; the only difference from the previous case is that we begin with a complement that may not be $k.\mathit{missing}$ or $\ell.\mathit{missing}$. □

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

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

3.6 From State-Based to Edit Lenses and Back

Recall from Chapter 2 that a state-based symmetric lens ℓ between *sets* X and Y comprises a set of complements C, a distinguished element $\mathit{missing} \in C$, and two functions

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

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

We are now ready to give the definition of the lifting operation that turns any symmetric, state-based lens between inhabited types into a symmetric edit lens.[8]

$$\frac{\ell \in X \underset{s}{\leftrightarrow} Y \qquad x \in X \qquad \ell.putr(x, \ell.missing) = (y, c_0)}{\partial_x \ell \in X_x \leftrightarrow Y_y}$$

$$\begin{array}{lcl} C & = & \ell.C \\ missing & = & c_0 \\ K & = & \{(x, c, y) \mid \ell.putr(x, c) = (y, c)\} \\ \Rrightarrow_{\mathsf{g}} & = & \ell.putr \\ \Lleftarrow_{\mathsf{g}} & = & \ell.putl \end{array}$$

$\partial_x \ell$ is a symmetric edit lens and the passage from ℓ to $\partial_x \ell$ is compatible with the equivalences on symmetric lenses and symmetric edit lenses. The equations for $\Rrightarrow$ and $\Lleftarrow$ are well-defined because the round-trip law for symmetric lenses guarantees that putting the same value twice in a row results in the same output both times (hence if edits will be coalesced in X_x, they will be translated to edits that get coalesced in Y_y), and the consistency relation is likewise preserved because the roundtrip laws for symmetric lenses guarantee that any given *putr* or *putl* results in a "stable state".

3.6.1 Theorem: If k and ℓ are state-based lenses and $k \equiv \ell$, then $\partial_x k \equiv \partial_x \ell$.

Proof: Suppose S is a witness that $k \equiv \ell$. Then we define $\partial_x S$ as follows:

$$\partial_x S = \{(x, c_k, c_\ell, y) \mid (c_k, c_\ell) \in S \wedge x \in X \wedge y \in Y\}$$

Let us write $(y_k, c_{0k}) = k.putr(x, k.missing)$ and $(y_\ell, c_{0\ell}) = \ell.putr(x, \ell.missing)$. Since $(k.missing, \ell.missing) \in S$, we know $y_k = y_\ell$ and $(c_{0k}, c_{0\ell}) \in S$. The former equality tells us that at least the two lenses have the same type—that is, $Y_{y_k} = Y_{y_\ell}$ as modules—while the latter inclusion lets us observe that

$$(init_{X_x}, (\partial_x k).missing, (\partial_x \ell).missing, init_{Y_y}) = (x, k.missing, \ell.missing, y) \in \partial_x S$$

where we abbreviate y_k and y_ℓ by the name y.

It remains to show that $\partial_x S$ is preserved by $\Rrightarrow$ and $\Lleftarrow$ (definedness is not in question since the modules in question have no partial edits); these arguments are very similar, so we focus on the one for $\Rrightarrow$. We have $x_0 \in X, y_0 \in Y, (c_k, c_\ell) \in S, \mathrm{d}x \in X_x$.

[8]The unique state-based lens between uninhabited types can be lifted to the unique edit lens between degenerate modules.

We must show that computing $\Rrightarrow$ with these values produces values that form a tuple in $\partial_x S$. We proceed by induction on $\mathrm{d}x$.

In case $\mathrm{d}x = \langle\rangle$, we are done: after computing $\Rrightarrow$, we still have x_0, y_0, c_k, and c_ℓ. Otherwise, $\mathrm{d}x = x_1{:}\mathrm{d}x'$ and the induction hypothesis tells us that if the two equations

$$(\partial_x k).\Rrightarrow(\mathrm{d}x', c_k) = (\mathrm{d}y_k, c'_k)$$
$$(\partial_x \ell).\Rrightarrow(\mathrm{d}x', c_\ell) = (\mathrm{d}y_\ell, c'_\ell)$$

hold then $(c'_k, c'_\ell) \in S$ and $\mathrm{d}y_k = \mathrm{d}y_\ell$. We can then conclude that if $k.putr(x_1, c'_k) = (y_k, c''_k)$ and $\ell.putr(x_1, c'_\ell) = (y_\ell, c''_\ell)$ then $y_k = y_\ell$ and $(c''_k, c''_\ell) \in S$ because S is a witness that $k \equiv \ell$. We now compute with these definitions that $(\partial_x k).\Rrightarrow(\mathrm{d}x, c_k) = (y_k{:}\mathrm{d}y_k, c''_k)$ and $(\partial_x \ell).\Rrightarrow(\mathrm{d}x, c_\ell) = (y_\ell{:}\mathrm{d}y_\ell, c''_\ell)$. But we have already seen that $y_k{:}\mathrm{d}y_k = y_\ell{:}\mathrm{d}y_\ell$ and $(c''_k, c''_\ell) \in S$, so we are done. □

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

Given an edit lens ℓ between modules X and Y, both equipped with differs, we define a symmetric lens $|\ell|$. The passage $\ell \mapsto |\ell|$ is compatible with lens equivalence.

$$\frac{\ell \in X \leftrightarrow Y}{|\ell| \in |X| \underset{s}{\leftrightarrow} |Y|}$$

C	$=$	$\lvert X\rvert \times \ell.C \times \lvert Y\rvert$
$missing$	$=$	$(init_X, \ell.missing, init_Y)$
$putr(x, (x_0, c, y_0))$	$=$	$(\mathrm{d}y\ y_0, (x, c', \mathrm{d}y\ y_0))$
		where $(dy, c') = \ell.\Rrightarrow(dif(x_0, x), c)$
$putl(y, (x_0, c, y_0))$		analogous

3.6.2 Theorem: Let X, Y be sets with distinguished elements x and y and equip the associated modules X_x and Y_y with their canonical differs. The constructions $|-|$ and ∂_x then establish a one-to-one correspondence between equivalence classes of edit lenses between X_x and Y_y, on the one hand, and state-based lenses between X and Y for which x and y are already consistent, on the other.

Proof: Let ℓ be a state-based lens between sets X and Y and let $x \in X$, $y \in Y$ satisfy $\ell.putr(x, \ell.missing) = (y, \ell.missing)$. To show that $|\partial_x \ell| \equiv \ell$ we use the

simulation $R = \{((x, c, y), c) \mid \ell.putr(x, c) = (y, c)\}$. Conversely, if $\ell \in X_x \leftrightarrow Y_y$ then $\ell \equiv \partial_{init_X}|\ell|$. To see this, we use the simulation $S = \{(x, c, (x, c, y), y) \mid x \in X, y \in Y, c \in \ell.C\}$. We omit the verification of both simulations. □

The theorem's condition that $(x, \ell.missing, y)$ already be a consistent triple may look strong at first, but one can simply take an arbitrary lens and "step" it once by x (producing a new lens whose *missing* component is given by *putr*) to produce a lens with essentially the same behavior but a stable *missing* component. We conjecture that this "isomorphism" between state-based and certain edit lenses is also compatible with various lens constructors, in particular tensor product and sum.

3.7 Conclusion

Recall from Chapter 1 that there are four high-level challenges in the design of bidirectional programming languages: alignment, symmetry, performance, and syntax. The tools from Chapter 2—existentially quantified complement sets and an equivalence relation to quotient out uninteresting differences between them—enabled a symmetric system that nevertheless supports a range of useful transformations. This chapter's approach retains those tools while tackling the remaining two challenge areas of alignment and performance. We identified an abstract model for edits and edit transformations—monoids and monoid homomorphisms, respectively—and investigated instantiations of these models to the standard basic data types and operations. In our investigation, we showed that the natural instantiations (in particular our list and container edit monoids) enable rich alignment information to be provided to, processed by, and received from our transformations. Moreover, our lens transformations traverse edits and complements, but not repositories. Because our instantiations of edit monoids contain significantly less data than the repositories (and complements are typically trivial or at most represent the spine of the data), traversals of these data should require less processing power, memory, and transmission bandwidth. (Viewing edits as a compression scheme for updated repositories, we have shown that edit lenses can compute on the compressed data directly without decompressing; this is a significant proof burden for other compression techniques.) We have shown that these techniques for alignment representation and processing efficiency are applicable within the realm of symmetric lenses and compatible with most of the transformations needed for a comprehensive syntax of edit lenses.