# 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 [35]; tool-specific configuration formats and a common configuration format [27]; a database and some particular summary of interest; or two distant but partially replicated computers [34] 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. Designing a language is also a more modular approach to solving the bidirectional transformation problem, as the design of bidirectional

| | 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

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 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 $\ell$; to simplify things, we will take $\ell.get$ to be unassailable[1] and phrase all our desires in terms of constraints on $\ell.put$. It is natural to expect two things from our lens: first, that $\ell.put$ changes enough—that whatever change we make to the view is faithfully reflected in the source so that future calls to $\ell.get$ give exactly the value we changed the view to—and second, that $\ell.put$ 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:

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

The PutGet law formalizes the expectation that $\ell.put$ changes enough; the GetPut law takes a step toward formalizing the expectation that $\ell.put$ does not change too much.[2] There is a third law, CreateGet, which serves a similar purpose to the 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 $\ell$ 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.

[1] We will use record notation for lens components, so that $\ell.get$ is the *get* component of $\ell$.

[2] 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.

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 [39], 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.[3] 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 $\mathsf{map}_U$ 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 $\mathsf{map}(\ell).\mathit{put}$ 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 $\ell.\mathit{put}$. 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 can demonstrate the behavior of map.

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

[3]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'
```

$$
\begin{aligned}
\mathsf{map}_U(f,t) &= \begin{cases} \langle\rangle & t = \langle\rangle \\ f(x){:}\mathsf{map}_U(f,t') & t = x{:}t' \end{cases} \\
\mathsf{zip}(f,g,h,t,u) &= \begin{cases} f(x,y){:}\mathsf{zip}(f,g,h,t',u') & t = x : t' \wedge u = y : u' \\ g(t) & t = x : t' \wedge u = \langle\rangle \\ h(u) & t = \langle\rangle \wedge u = y : u' \\ \langle\rangle & t = u = \langle\rangle \end{cases} \\
\mathsf{const}(x) &= \lambda y.\ x
\end{aligned}
$$

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

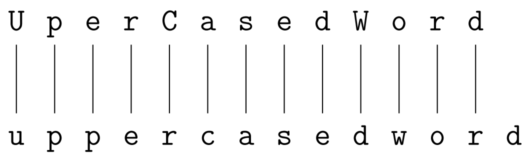
$$
\begin{aligned}
lower.get(c) &= \text{the lower case version of } c \\
lower.put(c',c) &= \begin{cases} \text{the upper case version of } c' & \mathtt{A} < c < \mathtt{Z} \\ c' & \text{otherwise} \end{cases} \\
lower.create(c) &= c
\end{aligned}
$$

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

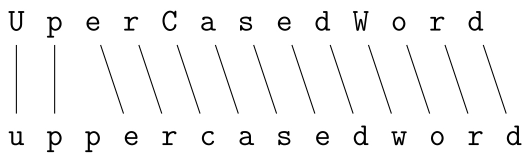
All of these examples behave essentially optimally. However, not all is well; a simple example of the so-called *alignment problem* is something like this, where we have an insertion in the middle of the word to correct the spelling of “upper”:

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

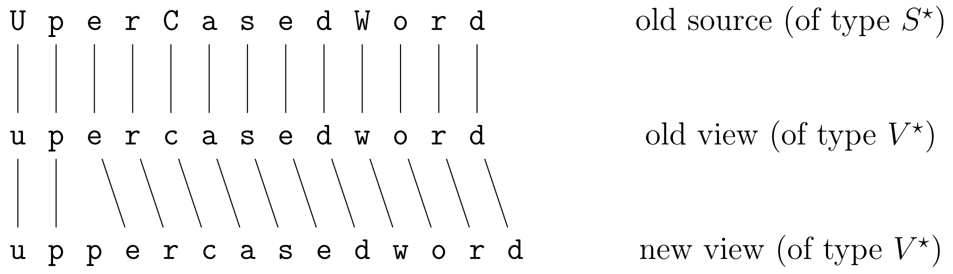
Because $\mathsf{map}(\mathit{lower}).\mathit{put}$ only looks at a lower-cased element’s position when deciding which mixed-case character to match it up with, we have incorrectly *aligned* the new view with the old source this way:



A better alignment would look like this:



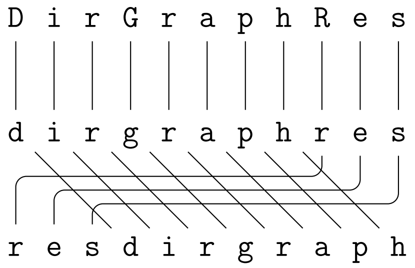
One natural reaction to this infelicity is to think of the `diff` algorithm [24] or something similar. This idea has been developed quite far [6]; let us see how. At first blush, it seems difficult to use the `diff` algorithm directly. Elements of the source and view have different types, so it is not clear how to compare them.[4] 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 $S^\star$)
u p e r c a s e d w o r d
old view (of type $V^\star$)
u p p e r c a s e d w o r d
new view (of type $V^\star$)

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

[4]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.

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 |

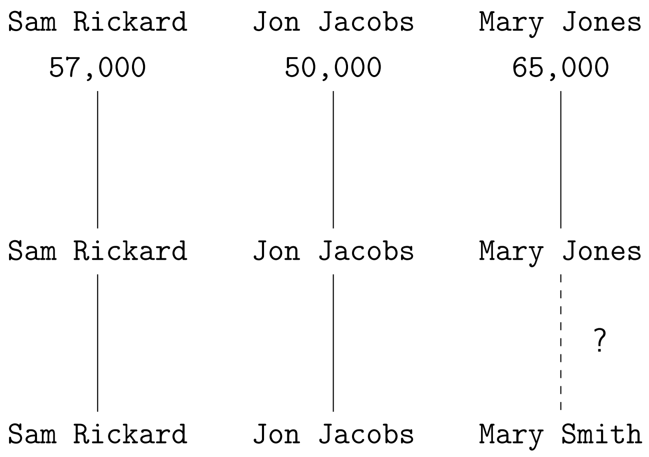
(a) HR's view

| 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

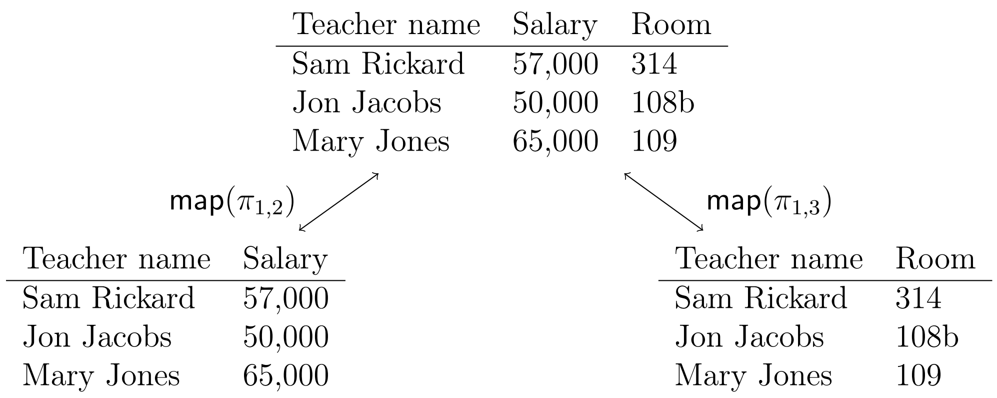
Let us turn our attention to a second fundamental challenge in lens design: symmetry. The asymmetric lenses discussed above assume that one repository is a view of the other. In the following, we will discuss two bidirectional scenarios, one that highlights the need to relax this assumption, and one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



| Teacher name | Salary | Room |
|---|---|---|
| Sam Rickard | 57,000 | 314 |
| Jon Jacobs | 50,000 | 108b |
| Mary Jones | 65,000 | 109 |

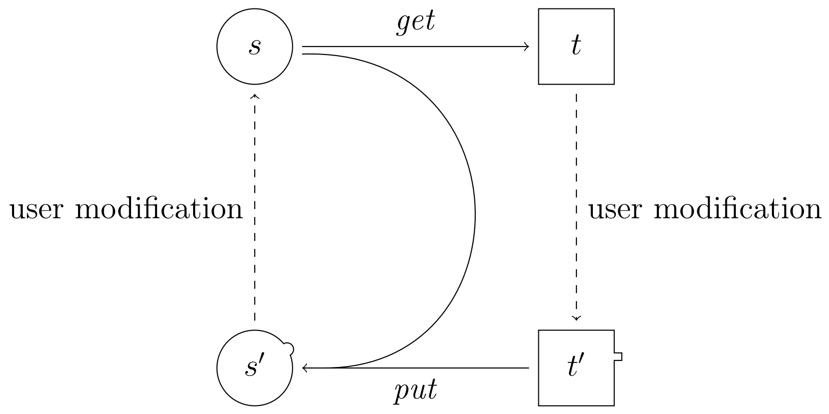
| Teacher name | Salary |
|---|---|
| Sam Rickard | 57,000 |
| Jon Jacobs | 50,000 |
| Mary Jones | 65,000 |

| Teacher name | Room |
|---|---|
| Sam Rickard | 314 |
| Jon Jacobs | 108b |
| Mary Jones | 109 |

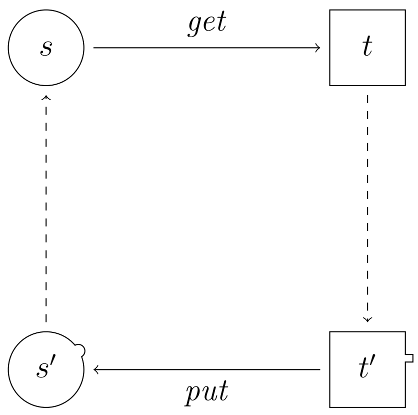
Figure 1.5: A slightly more complicated synchronization scenario

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.

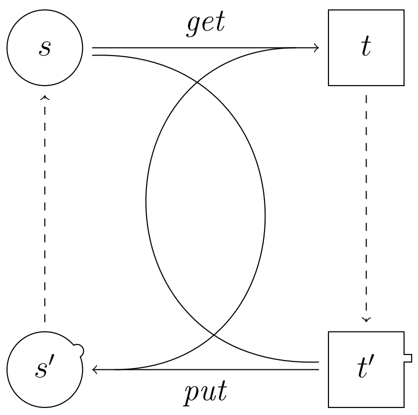
Let us address ourselves to what makes asymmetric lenses asymmetric in the first place. Figure 1.6a shows the typical life-cycle of an asymmetric lens $\ell \in S \overset{a}{\leftrightarrow} T$, ignoring *create* for the moment. Drawing the types of the *get* and *put* operations this way highlights 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, 35]. 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



(a) Asymmetric, state-based lenses



(b) (Partial) isomorphisms



(c) Constraint maintainers

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

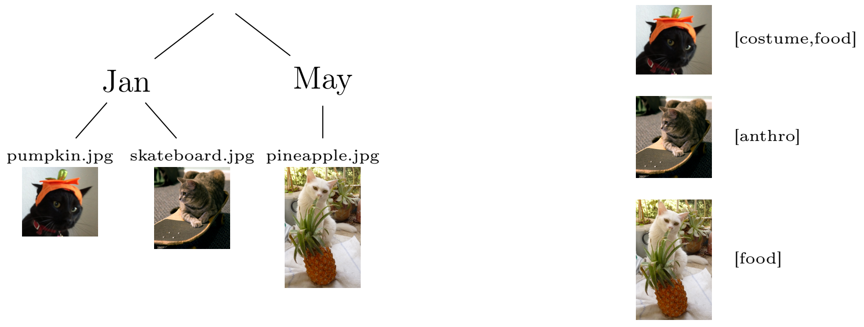


Figure 1.7: A whimsical symmetric synchronization scenario

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 maintainers [29], 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).[5] 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

[5]Pictures used with permission [4, 10, 41].

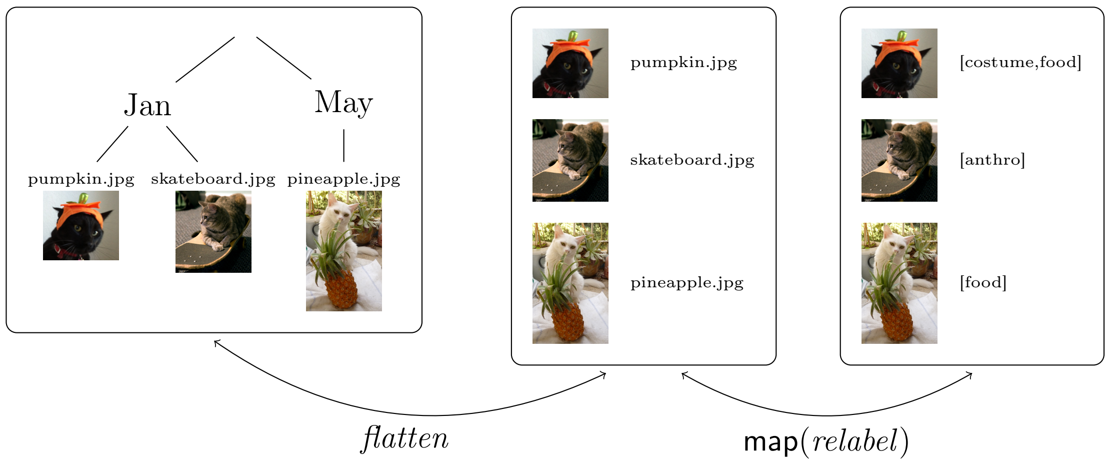


Figure 1.8: Adding an intermediate structure can improve modularity

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, 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, 33, 34, 36, 38], 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.[6] 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

[6]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.

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 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 three 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 $\partial$; for example, $\partial X$ 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 $\partial$ is d; for example, $\mathrm{d}x \in \partial X$. Lenses are named $k$, $\ell$, $m$, and $n$.

| 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

**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 $i$th 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 $n$), 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 $X^\omega$ when the elements are drawn from $X$. The infinite list with $x_i$ in the $i$th position is written $\langle x_1, \ldots \rangle$, and the infinite list where there is a single element $x$ in every position is written $x^\omega$. 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 $\ell$. We deal with many variations on functions in this document; Table 1.2 summarizes them.