Generalizing Lenses

Daniel Wagner

A Dissertation Proposal

in

Computer Science

2013

ABSTRACT

Generalizing Lenses

Daniel Wagner

Benjamin C. Pierce, Advisor

Bidirectional situations are all around us: we synchronize bookmarks, keep clones of our data on our mobile devices, edit collaboratively, use GUIs to visualize and modify chunks of our data, and more. In these situations, we must write programs that translate back and forth between two data sets which potentially use different storage formats and sometimes even record differing characteristics of the data. The framework of lenses [9] aids in the creation and maintenance of these translations by allowing the programmer to write a single program which can be interpreted as both a forward and backward translation.

The basic lens framework, however, makes some assumptions about how the transformations they describe will be used:

1. One data set is less informative than the other, and can be completely recreated just by examining the more informative data set.

2. It is reasonable to transmit and traverse entire data sets.

3. There are just two data sets being synchronized (or at least, if there are many data sets, they can reasonably be linked pairwise).

For some decentralized applications, there are two data sets which share some data, but which also each have their own unshared data; because there is no master data set, assumption (1) is violated. For large data sets, it is reasonable to hope that one could store and transmit small descriptions of recent changes (e.g. transmitting a diff rather than an entire updated file), which violates assumption (2). Finally, a variant of spreadsheets (where formulas for computing cell values are made bidirectional) can be understood as many data sets—one for each cell—with intricate webs of connections between them, violating assumption (3).

We will describe the generalized lens frameworks of symmetric lenses, which relax assumption (1), and edit lenses, which relax assumption (2). We propose a novel lens framework based on local constraint propagation which relaxes assumption (3).

Contents

1 Introduction **1**

2 Finished work **3**
2.1 Background 4
2.2 Limitations 4
2.3 Symmetric lenses 5
2.4 Edit lenses 9
2.4.1 Building edit languages 10
2.4.2 Complements 10
2.4.3 Formal definition 12

3 Spreadsheets **16**
3.1 Simple example 17
3.2 Goal statement 18
3.3 Simplistic solutions 18
3.3.1 Spreadsheets 19
3.3.2 Tree topology 19
3.3.3 Linear constraints 20
3.3.4 Biased graph combination 21
3.4 Design axes 23
3.4.1 Sources of ambiguity 24
3.4.2 Sources of insolubility 25
3.4.3 Other difficulties 26
3.5 Timeline 27

4 Related work **28**
4.1 Symmetric lenses 29
4.2 Edit lenses 30
4.3 Spreadsheets 30

Chapter 1

Introduction

As the electronic world grows increasingly interconnected, it grows increasingly common to need good tools for synchronizing replicated data. In many cases, the data stores being replicated are not identical—each store is tailored to the device or application that is using the replica. Traditional tools for maintaining synchrony between differing data formats provide separate transformation tools for each pair of formats. However, as the formats and transformations grow complex, maintaining separate tools can grow more difficult and error-prone.

The lens framework of [9] attempts to address this problem by giving a language of transformations that can be interpreted *bidirectionally*: a lens between data formats A and B describes *both* a transformation from A to B *and* a transformation from B to A. Recent work has developed some nice tooling for lenses, but there remain many use cases that call for generalizations of lenses. We will discuss three such situations in this document.

One core assumption of the lens framework is that one of the two replicas being synchronized is "canonical": it stores enough information to reconstruct the other replica in its entirety. In many cases, this is untrue: the two replicas have some shared information, but each also has some information that is not shared with the other. Section 2.3 briefly discusses a symmetric variant of lenses that relaxes this assumption; the complete dissertation will also discuss in-depth the additional machinery needed for lens equivalence, the algebraic structure of symmetric lenses, and a syntax for writing symmetric lenses that includes rich support for lists and generic containers.

A second core assumption is that lenses can operate on entire replicas as a single object. For large data sets, this can be a problem: one may not wish to transmit entire replicas during synchronization, but instead short descriptions of what has changed since the last synchronization point. Section 2.4 briefly discusses the formalism needed to generalize lenses so that they operate on edits; the complete dissertation will also discuss (analogously to the discussion for symmetric lenses) edit lens equivalence, algebraic structure, and a syntax for edit lenses that includes some support for list and generic container operations.

Finally, the lens framework focuses itself on the problem of synchronizing two (potentially large) replicas at a time. The main new work proposed in this document is to produce a generalization of lenses that can synchronize very many (potentially quite small and loosely-related) replicas. For concreteness, imagine a multi-directional spreadsheet: each cell is a replica. Some cells are computed from others; these computations are the transformations that we would like to bidirectionalize. Section 3 discusses the major goals of this work as well as some simple solutions that we have already explored and the major challenges we foresee.

In the remainder of the document, we will formally introduce the lens framework (2.1), briefly discuss the formalisms of symmetric and edit lenses (2.3 and 2.4), introduce the hyperlens multi-directional spreadsheet project's goals (3.2), challenges (3.4), and timeline (3.5), and give an overview of related work (4).

Chapter 2

Finished work

2.1 Background

To set the stage, let's review one standard definition of asymmetric lenses. Suppose X is some set of source structures (say, the possible states of a database) and Y a set of target structures (views of the database). An asymmetric state-based lens from X to Y has two components:

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

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

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

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

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

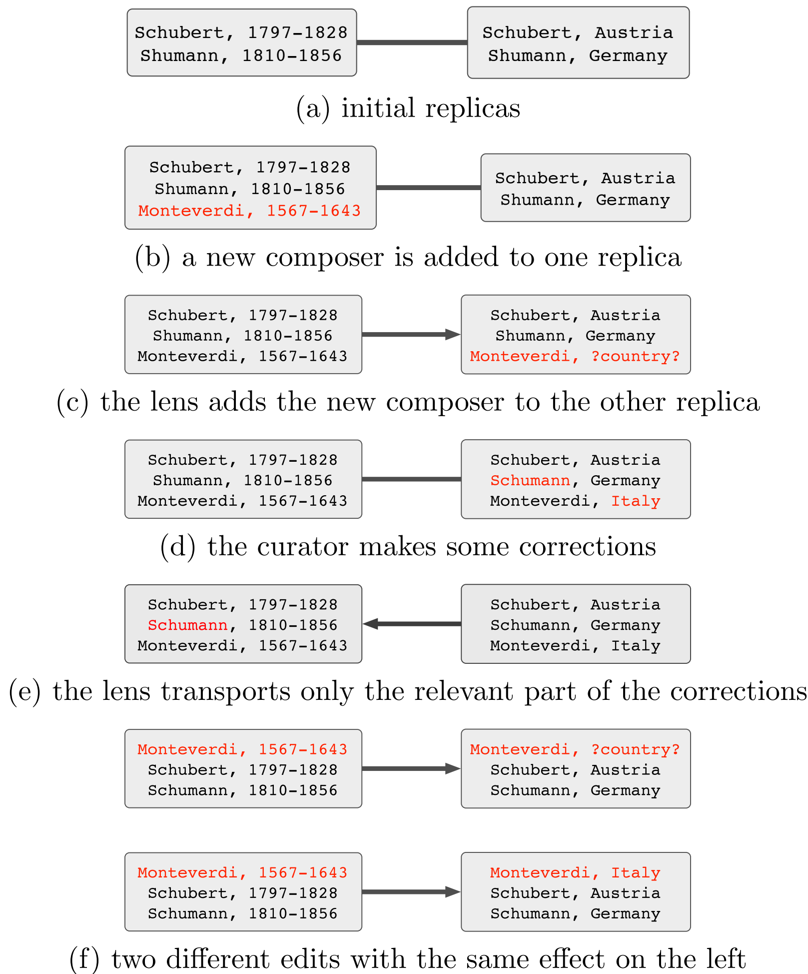
$$get\ (create\ y) = y \qquad \text{(CreateGet)}$$

2.2 Limitations

Figure 2.1 gives a simple example of a pair of repositories and operations on those repositories that the asymmetric, state-based lenses above do not model well, along with the behavior that we desire from our symmetric and edit lens generalizations. In part (a), we see the initial replicas, which are in a synchronized state. On the left, the replica 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. Of particular relevance here is that the right-hand repository contains countries, which do not appear in the left-hand repository—this means we cannot write a *get* function from left to right—while the left-hand repository contains dates, which do not appear in the right-hand repository—meaning we also cannot write a *get* function from right to left.

In part (b), the user interacting with the left-hand replica decides to add a new composer, Monteverdi, at the end of the list. The lens connecting the two replicas now converts this into a corresponding change that adds Monteverdi to the right-hand replica, shown in part (c). Note that the translation includes the name component but leaves the country component with its default value, "?country?." This is the best it can do, since the left-hand replica 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 from right to left, but propagates the spelling correction.

In some extraordinary cases, there may be many reasonable ways to keep the two repositories synchronized. Consider part (f) of Figure 2.1, where the left-hand replica ends up with a row for Monteverdi at the beginning of the list, instead of at the end. There are at least two reasonable user intentions that could lead to this effect: either the user could mean to delete the old Monteverdi



(a) initial replicas

(b) a new composer is added to one replica

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

(d) the curator makes some corrections

(e) the lens transports only the relevant part of the corrections

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

Figure 2.1: The desired behavior of a simple lens.

record and insert a brand new one (which happens to have similar data to the old record), or the user could mean to rearrange the order of the records. The upper row shows how the other repository should change in the former situation (it leaves Monteverdi with a default country), while the lower row shows what should happen in the latter situation (it reorders the other repository's records, preserving all the information associated with Monteverdi).

2.3 Symmetric lenses

Lenses can be generalized from the asymmetric presentation above—where one of the structures is always a "view" of the other—to a fully *symmetric* version where each of the two structures may contain information that is not present in the other. Although symmetric variants of lenses have been studied [15, 24, 4], they all lack a notion of sequential composition of lenses, a significant technical and practical limitation. The extra structure needed to support composition is nontrivial; in particular, constructions involving symmetric lenses need to be proved correct modulo a notion of *behavioral equivalence*. However, once that structure is in place, we find that there is a rich *algebra*

on the space of lenses, which can be used as the theoretical basis for a language of symmetric lenses.

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 [1] and was adapted to lenses in [2] (and, for a slightly different definition, [14]), 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 $\mathit{put}(y, \mathit{missing})$.

So far, this perspective has retained the assumption that elements of X have richer structure than elements of Y; hence, the complement need only store the extra rich parts of X. For symmetry, we will lose this assumption, and allow the complement set C to contain information about both X elements and Y elements. As a result, *both* the *get* and *put* functions may need to inspect and update the complement. It will be the responsibility of those functions to decompose the complement into the "private information from X" and the "private information from Y"; then we expect that *get* will read the part about Y and write the part about X and *put* will read the part about X and write the part about Y. Thus, our new types for *get* and *put* are

$$\mathit{get} \in X \times C \to Y \times C$$
$$\mathit{put} \in Y \times C \to X \times C$$

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

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.

Definition 1 (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*

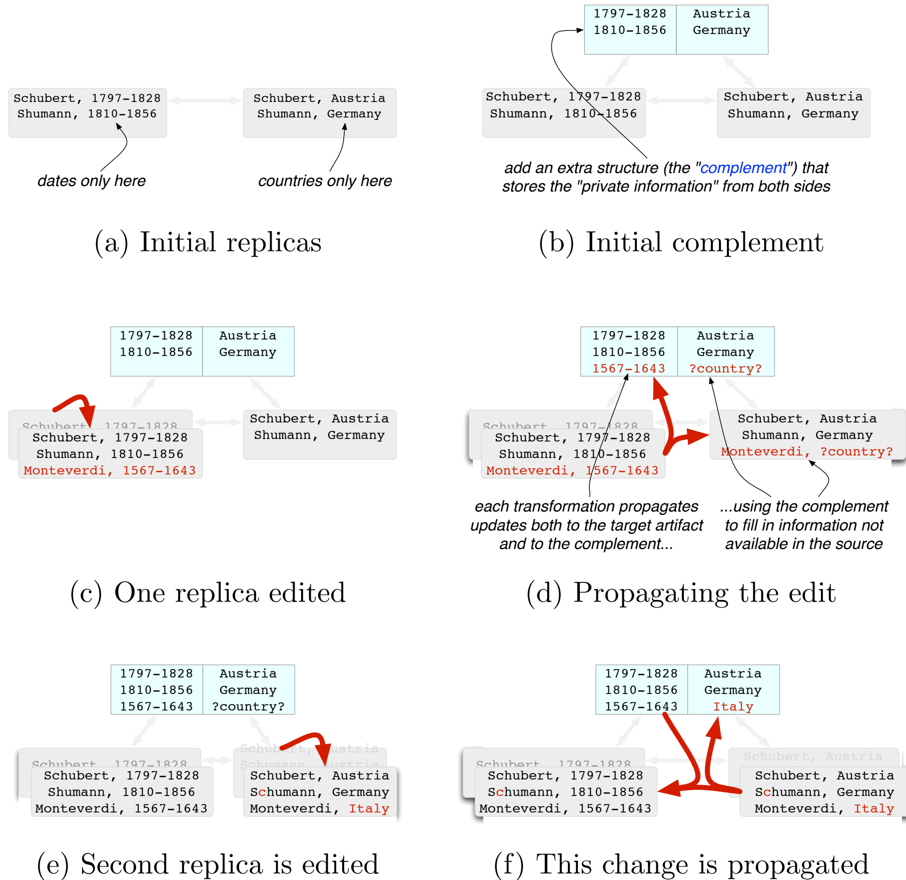
$$\begin{array}{rcl} \mathit{putr} & \in & X \times C \to Y \times C \\ \mathit{putl} & \in & Y \times C \to X \times C \end{array}$$

satisfying the following round-tripping laws:

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

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

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



(a) Initial replicas (b) Initial complement

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

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

Figure 2.2: Behavior of a symmetric lens

The force of the PutRL and PutLR laws is to establish some "consistent" or "steady-state" triples (x, y, c), for which *puts* 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 2.2 shows how this model might be instantiated to synchronize the repositories discussed earlier in Section 2.2. The complement (b) contains all the information that is discarded by both *puts*—all the dates from the left-hand structure and all the countries from the right-hand structure. (It can be viewed as a pair of lists of strings, or equivalently as a list of pairs of strings; the way we build list lenses later actually corresponds to the latter.) If we add a new record to the left hand structure (c) and use the *putr* operation to propagate it through the lens (d), we copy the shared information (the new name) directly from left to right, store the private information (the new dates) in the complement, and use a default string to fill in both the private information on the

right and the corresponding right-hand part of the complement. If we now update the right-hand structure to fill in the missing information and correct a typo in one of the other names (e), then a *putl* operation will propagate the edited country to the complement, propagate the edited name to the other structure, and use the complement to restore the dates for all three composers.

Viewed 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 2.2—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.

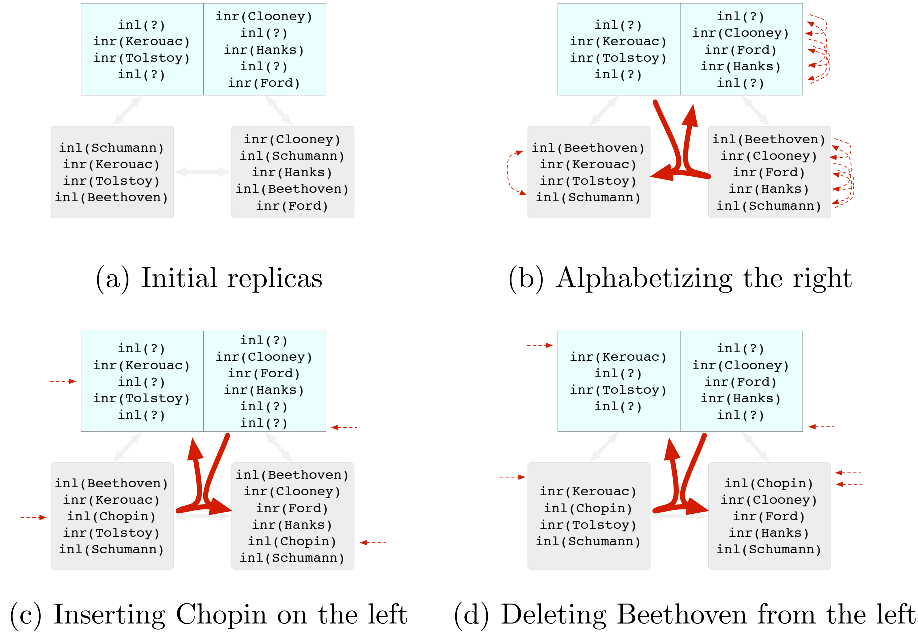
There are several plausible implementations of $e^\star$, with slightly different behaviors when list elements are added and removed—i.e., when the input and complement arguments to *putr* or *putl* are lists of different lengths. One possibility is to take $e^\star.C = (e.C)^\star$ and maintain the 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:

$$
\begin{array}{lll}
 & putr(\langle (a,1) \rangle\,,\ \langle (p,P),(q,Q) \rangle) & \\
= & putr(\langle (a,1) \rangle\,,\ \langle (p,P) \rangle) & \text{(truncating)} \\
= & (\langle (1,P) \rangle\,,\ \langle (a,P) \rangle) & \\
 & putr(\langle (a,1),(b,2) \rangle\,,\ \langle (a,P) \rangle) & \\
= & putr(\langle (a,1),(b,2) \rangle\,,\ \langle (a,P),(m,M) \rangle) & \text{(padding)} \\
= & (\langle (1,P),(2,M) \rangle\,,\ \langle (a,P),(b,M) \rangle) &
\end{array}
$$

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.

Figure 2.3 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.

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



(a) Initial replicas (b) Alphabetizing the right

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

Figure 2.3: Synchronizing lists of sums

to exactly match the tags. For example:

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

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:

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

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*.

2.4 Edit lenses

The two frameworks discussed, asymmetric state-based lenses and symmetric state-based lenses, are both somewhat "extensional". That is: the *put* functions have access only to extensional

information about the states of the repository before and after any user changes. As shown in (f) of Figure 2.1, this information is not always enough to determine a best modification to the unchanged repository to restore synchrony: one wants "intentional" information about how a change was made in addition to the effect that change had. Additionally, the description of what has changed since the last synchronization point can often be represented much more compactly than the new value of the repository. The goal of edit lenses is to address these two concerns.

We do not address the question of where these edits come from or who decides, in cases like part (f), which of several possible edits is intended. As argued in [2], answers to these questions will tend to be intertwined with the specifics of particular editing and/or diffing tools and will tend to be messy, heuristic, and domain-specific—unpromising material for a foundational theory. Rather, our aim is to construct a theory that shows how edits, however generated, can be translated between replicas of different shapes.

Below, we discuss how to build an edit lens synchronizing the data structures in the introductory example in Figure 2.1, paralleling the discussion of symmetric lenses. The primary difference is that each lens must also be associated with the set of edits that it knows how to translate. In the following, we will discuss how to build this association compositionally; the role that complements play in edit lenses; and the formal definition of edit lenses.

2.4.1 Building edit languages

As with symmetric lenses, we will build our edit lens between $(X \times Y)^*$ and $(X \times Z)^*$ compositionally—that is, the whole lens should have the form ℓ^*, where $*$ is a "list mapping" lens combinator and ℓ itself is a product $\ell_1 \times \ell_2$ of a lens $\ell_1 \in X \to X$ that translates composer edits verbatim, while $\ell_2 \in Y \to Z$ is a "disconnect" lens that maps every edit on either side to a trivial identity edit on the other side.

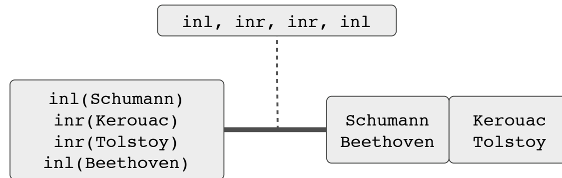
In analogous fashion, the edit languages for the top-level structures will be constructed compositionally. The set of edits for structures of the form $(X \times Y)^*$, written $\partial((X \times Y)^*)$, will be defined together with the list constructor $*$. Its elements will have the form $\mathsf{ins}(i)$ 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 ℓ^* will consist of two components—one for transporting edits from the left side to the right, written $(\ell^*).\!\Rrightarrow \in \partial(X \times Y)^* \to \partial(X \times Z)^*$,[1] and another for transporting edits from right to left, written $(\ell^*).\!\Lleftarrow \in \partial(X \times Z)^* \to \partial(X \times Y)^*$.

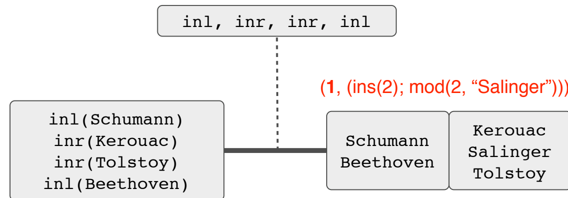
2.4.2 Complements

We sometimes need lenses to have a little more structure than this simple example suggests. To see why, consider defining a *partitioning* lens p between the sets $\partial((X+Y)^*)$ and $\partial(X^* \times Y^*)$. Figure 2.4 demonstrates the behavior of this lens. In part (a), we show the original replicas: on the left, a single list that intermingles authors and composers (with $\mathsf{inl}/\mathsf{inr}$ tags showing which is which), and

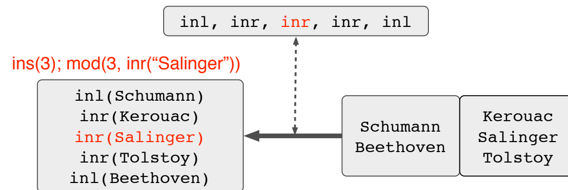
[1] The symbol $\Rrightarrow$ is pronounced "put an edit through the lens from left to right," or just "put right." It is the edit-analog of the *putr* function.



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



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



(c) the complement tells how to translate the index

Figure 2.4: A lens with complement.

on the right a pair of homogeneous (untagged) lists, one for authors and one for composers. Now consider an edit, as in (b), that inserts a new element somewhere in the author list on the right. It is clear that we should transport this into an insertion on the left replica, but where, exactly, should we insert it? If the $\Lleftarrow$ 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 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

$$p.\Rrightarrow \in \partial((X+Y)^*) \times C \to \partial(X^* \times Y^*) \times C$$

and

$$p.\Lleftarrow \in \partial(X^* \times Y^*) \times C \to \partial((X+Y)^*) \times C.$$

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

Note that the complement stores just the $\mathsf{inl}/\mathsf{inr}$ 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 replicas; 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 replicas.

2.4.3 Formal definition

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.*

Definition 2. *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. The monoid framework can also accommodate more abstract notions of edit. For example, the set of all total functions from a set X to itself forms a monoid, where the multiplication operation is function composition. This is essentially the form of edits considered by Stevens [24]. We mostly focus on the simple case where edit languages are free monoids. Laws can be added to the product and sum lens constructions, and possibly for lists and general containers as well.

Definition 3. *Given a monoid M and a set X, a* monoid action *on M and X is a partial function $\odot \in M \times X \rightharpoonup X$ satisfying two laws:*

$$\mathbf{1} \odot x = x$$

$$(m \cdot n) \odot x = m \odot (n \odot x)$$

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 replica states arising in practice. This requirement, corresponding to the *totality* property of previous presentations of lenses [6], is formalized and proven. 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:

Definition 4. *A* module *is a tuple $\langle X,\ init_X,\ \partial X,\ \odot_X \rangle$ comprising a set X, an element $init_X \in X$, a monoid ∂X, and a monoid action $\odot_X$ of ∂X on X.*

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.

[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.

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 the examples, 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 [11] 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.

Definition 5. *Given monoids M and N and a* complement set C, *a* stateful monoid homomorphism *from M to N over C is a function $h \in M \times C \to N \times C$ satisfying two laws:*

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

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

These are basically just the standard monoid homomorphism laws, except that h is given access to some internal state $c \in C$ that it uses (and updates) when mapping from M to N; in the second law, we must thread the state c' produced by the first h into the second use of h, and we demand that both the result and the effect on the state should be the same whether we send a composite element $m' \cdot m$ through h all at once or in two pieces.

The intended usage of an edit lens is as follows. There are two users, one holding an element of X the other one an element of Y, both referred to hereafter as *replicas*. Initially, they hold $init_X$ 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 replica 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 replica. In the process, the internal state c of the lens is updated to reflect the new correspondence between the two replicas.

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

Definition 6. *A* symmetric edit lens *between modules X and Y consists of a complement set C, a distinguished element $missing \in C$, two stateful monoid homomorphisms*

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

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

- $(init_X, missing, init_Y) \in K$;

- *if* $(x, c, y) \in K$ *and* $dx\ x$ *is defined and* $\Rrightarrow(dx, c) = (dy, c')$*, then* $dy\ y$ *is also defined and* $(dx\ x, c', dy\ y) \in K$*;*
- *if* $(x, c, y) \in K$ *and* $dy\ y$ *is defined and* $\Lleftarrow(dy, c) = (dx, c')$*, then* $dx\ x$ *is also defined and* $(dx\ x, c', dy\ y) \in K$*.*[3]

[3] One might consider a more general format with "creation" operations $\mathit{creater} \in X \to Y \times C$ and symmetrically *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 $\mathit{creater}(x, c) = \Rrightarrow(\mathrm{set}(x), c)$.

Chapter 3

Spreadsheets

Lenses keep two similar pieces of data consistent; as either one evolves, the lens finds analogous evolutions for the other. However, current lenses don't generalize smoothly to more than two pieces of data. Spreadsheets manage many pieces (cells) of data that are related to each other, but they are generally unidirectional: some cells are special automatically-updated cells, and the values in these cells are always computed by the system and cannot be changed by the user. Constraint propagation systems generalize spreadsheets to be many-directional when possible. However, current systems do not use old states of the system to guide the computation of new states; any system state which satisfies the given constraints is allowed.

The goal of the hyperlenses project is to merge the three systems, giving a way of maintaining constraints between many pieces of data that, when given an update to some part of the system, finds an "analogous" update to the rest of the system. Below we discuss criteria on which the success of the hyperlenses project can be judged.

In typical constraint propagation systems, there are variables and constraints. Constraints may involve any number of variables, and are simply relations on valuations of those variables. In the following, many of the relations we care about will be of the form

$$\{(x_1, \ldots, x_m, y_1, \ldots, y_n) \mid f(\overline{x}) = g(\overline{y})\}$$

and so we will simply write these as $f(\overline{x}) = g(\overline{y})$ when it is clear from context that a relation is expected.

3.1 Simple example

A user might draw up a vacation expenditures spreadsheet that looks like this:

Day	Travel	Lodging	Food	Total
1	750	120	45	915
2	30	120	18	168
3	0	120	150	270
4	15	120	30	165
5	750	0	15	765
Total	1545	480	258	2283

Along with the table, we would expect to see some constraints like

$$\begin{aligned} \text{Travel}_{\text{Total}} &= \text{Travel}_1 + \text{Travel}_2 + \text{Travel}_3 + \text{Travel}_4 + \text{Travel}_5 \\ \text{Total}_1 &= \text{Travel}_1 + \text{Lodging}_1 + \text{Food}_1 \end{aligned}$$

and so on, with ten constraints in all (one each for days 1, 2, 3, 4, 5, and Total, and one each for categories Travel, Lodging, Food, and Total). Here are some things a user might want to do with this setup:

- The user might go on another vacation, and want to make an estimate of how much he spent on food given his credit card balance at the end of the trip. To do this, he might update $\text{Total}_{\text{Total}}$ to his balance and look in the $\text{Food}_{\text{Total}}$ cell to get a guess.
- The user might like to plan a vacation to a certain location with a certain budget; then he could fix the $\text{Total}_{\text{Total}}$ cell and update the travel prices for the first and last day's plane tickets to get an estimate of how much he can spend on the various other days and categories while staying in his budget.

- Perhaps the user discovers that he is missing a category for entertainment and wants to add a new column, initially populated with zeros. He did not keep careful track of his daily spending for this on the last vacation, but he knows that in total he spent about $800 on entertainment, so he updates the new $\text{Entertainment}_{\text{Total}}$ cell to 800.

3.2 Goal statement

A hyperlens should be a generalization of both lenses and spreadsheets that supports high-level planning.

Hyperlenses should generalize lenses. It should be true that there is a behavior-preserving embedding of asymmetric, state-based lenses: that is, we can identify two variables in the hyperlens system whose values correspond to states of the asymmetric lens' two repositories, and the values in the hyperlens system evolve in the same way they would evolve when running the lens itself. Moreover, we demand that there be a "hyperlens composition" that preserves this property: the embedding of a composition of lenses is the composition of their embeddings.

A stretch goal is to generalize symmetric, state-based lenses in a similar way, and again to find a composition operator (potentially different from the previous one) that corresponds to symmetric lens composition.

Hyperlenses should generalize spreadsheets. It is unreasonable to demand that the hyperlens framework be capable of bidirectionalizing all spreadsheets in a reasonable way. However, on the class of spreadsheets that can be bidirectionalized, it should be the case that using the corresponding hyperlens as if it were a unidirectional spreadsheet produces the same answers as the original spreadsheet would.

A stretch goal is to generalize spreadsheets in the sense that any spreadsheet can be expressed as a (possibly still unidirectional) hyperlens with the same behavior.

Hyperlenses should support high-level planning. As with all bidirectional systems, there will be some updates which can be spread through the remainder of the system in many ways. Support for high-level planning means that there is some holistic language (that is, which does not require intimate knowledge of the structure of the term used to define the hyperlens) for expressing the relative desirability of the various coherent updates to the system. The system should then be able to compute the most desirable update. For example, being able to request that a particular variable be a sink for as much of the change as possible would satisfy this requirement.

3.3 Simplistic solutions

Some particularly simple regimes have already been explored. Four such regimes are discussed below, but it will be helpful to have a few conventions in place first.

Fix a linearly ordered set $\mathcal{N}$ of names and a universe U of values. Since we are dealing with partial functions, we will use the convention that $a = b$ whenever both a and b are undefined or whenever a and b are defined and identical. Likewise, $a \in b$ means that both a and b are undefined or they are both defined and a is a member of b.

Definition 7. *A* valuation *is a finite map from $\mathcal{N}$ to U.*

We generalize valuation application from names to finite sets of names using the linear ordering on $\mathcal{N}$: whenever $x_1 < \cdots < x_m$ is an increasing chain, $f(\{x_1, \ldots, x_m\}) = (f(x_1), \ldots, f(x_m))$.

Definition 8. *A* constraint *is a finite set $n \subset \mathcal{N}$ together with a relation $R \subset U^{|n|}$.*

Definition 9. *A valuation f* satisfies *constraint (n, R) when $f(n) \in R$.*

Definition 10. *The* constraint system graph *induced by a set of constraints is the undirected bipartite graph whose nodes are drawn from $\mathcal{N}$ in one part and constraints in the other, and which has an edge $(v, (n, R))$ iff $v \in n$.*

In some systems we discuss below, each constraint will have a clear notion of "output" variable. In these systems it will be convenient to also have a directed variant of the constraint system graph.

Definition 11. *A* pointed constraint *is a pair $(v, (n, R))$ of a constraint and a designated variable $v \in n$.*

Definition 12. *The* directed constraint system graph *induced by a set of pointed constraints is a directed bipartite graph whose nodes are drawn from $\mathcal{N}$ in one part and pointed constraints in the other. It has an edge from v to $(v', (n, R))$ iff $v \neq v'$ and $v \in n$, and it has an edge from $(v', (n, R))$ to v iff $v = v'$.*

Definition 13. *A* root node *is a node in a directed graph with in-degree zero.*

Definition 14. *A node has* high in-degree *if its in-degree is greater than one.*

In a directed constraint system graph, a root node corresponds to a name that is not the output of any constraint.

3.3.1 Spreadsheets

One particularly simple regime is where there is a directed constraint system graph with no cycles, no nodes with high in-degree, and edits are made only to root nodes. This corresponds to how most well-known spreadsheets behave. However, the restriction that you may only edit root nodes is quite limiting: it essentially means that the system is unidirectional.

3.3.2 Tree topology

The lens frameworks discussed in 2 amount to having a constraint system graph with exactly two variables and exactly one constraint connecting them. You may only edit one of the variables at a time. Then, to propagate the change, you spread the change from the changed variable to the unchanged one. We can take the idea of propagating change from changed variables to untouched ones somewhat further: when the constraint system graph is a tree, an update to a single node may be propagated through the entire graph by treating the updated node as a root of the tree and updating the values of nodes in topologically-sorted order.

We leave the discussion here short because this solution is completely subsumed by the solution in Section 3.3.4.

3.3.3 Linear constraints

Consider this simple, non-tree-structured spreadsheet:

$$\begin{aligned} tax &= 0.08 * base \\ total &= base + tax \end{aligned}$$

Despite the mildly interesting structure of the constraint system graph, it is still definitely possible to bidirectionalize this spreadsheet.

Suppose each constraint in the spreadsheet is *linear*, that is, has the form

$$x = b + \sum_i c_i y_i$$

for some constants b and $\overline{c}$. Additionally, we take the topological constraints that the directed constraint system graph has no (directed) cycles or nodes with high in-degree. (Note that the two-equation spreadsheet above has cycles in its constraint system graph but not its directed constraint system graph.)

Under these assumptions, a simple argument shows that, given a cell in the spreadsheet, we can write an affine formula which maps the values of root cells to the value of the given cell. The argument goes by induction on the length of the longest path from a root cell to the given one, and proceeds by substituting in affine formulas for each non-root variable at each step. In fact, we can go a step farther: we can write affine transformations from root cells to any set of cells. If we manage to give a characterization of when these affine transformations can be bidirectionalized, then we will have given an account of how to handle the multi-update problem and relax the simple structure requirement in the parts of the spreadsheet where only affine formulae are used.

Thus, we can now frame our problem in another way: what is the right way to bidirectionalize an affine transformation? Accordingly, we will now step away from spreadsheets and frame our discussion in linear algebra terms.

A function $get \in \mathbb{R}^m \rightarrow \mathbb{R}^n$ is affine exactly when there is a matrix M of dimension $n \times m$ and vector $\mathbf{b} \in \mathbb{R}^n$ such that $get(\mathbf{x}) = M\mathbf{x} + \mathbf{b}$. Affine functions are surjective (and hence bidirectionalizable) just when M has rank n, so we assume this. When $m = n$, f is a bijection. The put function in this case is particularly boring, because it ignores the original source:

$$put(\mathbf{x}, \mathbf{y}) = M^{-1}(\mathbf{y} - \mathbf{b})$$

The more interesting case is when $m > n$, and where each $\mathbf{y}$ is therefore the image of a nontrivial subspace of $\mathbb{R}^m$. There are many heuristics one may choose to identify a particular point in this subspace; we choose the specification:

$$put(\mathbf{x}, \mathbf{y}) = \operatorname*{argmin}_{\mathbf{x}', get(\mathbf{x}')=\mathbf{y}} ||\mathbf{x}' - \mathbf{x}||$$

Lemma 1. *There exists an $m \times n$ matrix P such that*

$$put(\mathbf{x}, \mathbf{y}) = \mathbf{x} + P(\mathbf{y} - get(\mathbf{x}))$$

satisfies the specification above.

Proof sketch. The intuition is that we wish to move as little as possible in source-space to match the move in target-space. This can be achieved by minimizing how far we move in the null space of M[1], since (exactly) these motions result in no motion in target-space. Choose a basis $\{\mathbf{x}_1, \ldots, \mathbf{x}_{m-n}\}$ for the null space of M. Then we will take:

$$B = \begin{bmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_{m-n}^\top \end{bmatrix}$$
$$P = \begin{bmatrix} M \\ B \end{bmatrix}^{-1} \begin{bmatrix} I_n \\ \mathbf{0}_{m-n,n} \end{bmatrix}$$

Turning the sketch into a proof involves arguing three things: that the square matrix in the definition of P is invertible; that P produces a put function that roundtrips; and that the put function produced by P produces minimal changes. The definition above was crafted so that (assuming for the moment that P exists) we have $MP = I$, which is used to show the roundtrip property, and $BP = \mathbf{0}$, which is used to show minimality.

3.3.4 Biased graph combination

In the three previous solutions, we have carefully restricted the allowed graph topologies and valuation updates so that there is at most one best strategy for reinstating constraints. This skirts one of the major goals of the hyperlens project: developing tools that embrace ambiguous updates and provide tools for disambiguating. In this section, we will explore a setting with some slight ambiguity and a way to give programmer control. We will also allow multi-update, and give a type system describing statically solvable updates. The high-level idea is to combine two small constraints on variable sets S and T into a single large constraint on variable set $S \cup T$ in a biased way: the programmer designates one of the two constraints as special, and whenever an update could be resolved by resolving the two constraints in either order, we choose to resolve the special constraint first.

Consider the spreadsheet given by these two schematic constraints:

$$o_1 = f(i_1, i_2)$$
$$o_2 = g(i_1, i_2)$$

Given an update to i_1, it may be possible to resolve these two constraints in either of the following two ways:

- First, update i_2 and o_1 to restore the f constraint. Then, update o_2 without modifying i_1 or i_2 (so as not to disrupt the already-restored f constraint) to restore the g constraint.
- Symmetrically, first update i_2 and o_2 to restore the g constraint, then update o_1 without modifying i_1 or i_2 to restore the f constraint.

[1] Recall that the null space of M is the set $\{x \mid Mx = \mathbf{0}\}$.

It is perfectly reasonable for these two plans to choose different updated values for i_2. One way to resolve the tie is to give either f or g higher priority.

Let us call the objects of this simple solution *multiway lenses*. We will give a definition, show how to combine multiway lenses (so that e.g. the above example would be the combination of one multiway lens for the f constraint and one for the g constraint), and briefly discuss some limitations of this approach.

Definition 15. *Given $N \subset \mathcal{N}$, a* multiway lens *$\ell \in \mathcal{M}(N)$ is a triple $(\mathcal{D}, put, K)$ where*

- *$\mathcal{D} \in 2^{2^N}$ is a predicate on sets of names called the* danger zone *of ℓ,*
- *$put \in 2^N \to (N \to U) \to (N \to U)$ is an update function which takes a set of names and a valuation on N and produces another valuation, and*
- *K is an invariant expressed as a predicate on valuations on N.*

The danger zone will be our "type system": updates to sets of variables that are in the danger zone are statically unsolvable. The first argument to the *put* function should be read as marking which variables have been updated. There are some well-formedness conditions on multiway lenses. First, we will want an analog of the lens framework's roundtrip laws. We will also ask that danger zones be sensible in the sense that updating more variables does not make the constraint system more solvable, and that the *put* function restrain itself to modifying parts of the valuation that are not inputs.

Definition 16. *A set D is* upwards-closed *if $i \in D$ and $N \subset \mathcal{N}$ implies $i \cup N \in D$.*

Definition 17. *A multiway lens ℓ is* well-formed *when it satisfies the following laws:*

- *$put(i, f) \in K$ whenever $i \notin \mathcal{D}$*
- *$put(i, f) = f$ whenever $f \in K$*
- *$\mathcal{D}$ is upwards-closed*
- *$put(i, f)|_i = f|_i$*

In the definition of multiway lens composition below, it will be convenient to extend *put* functions so that they act on valuations that cover extra variables by leaving the extra variables' values unchanged.

Definition 18. *Suppose we have $N \subset M \subset \mathcal{N}$, a lens $k \in \mathcal{M}(N)$, a set of names $i \subset M$, and f is a valuation on M. Then $focus(k, i, f)$ is a valuation on M:*

$$focus(k,i,f)(x) = \begin{cases} k.put(i \cap N, f|_N)(x) & x \in N \\ f(x) & x \notin N \end{cases}$$

Definition 19. *Suppose $k \in \mathcal{M}(N_k)$ and $\ell \in \mathcal{M}(N_\ell)$ and let $S = N_k \cap N_\ell$ be the set of shared variables. Then the composition $k \mid \ell \in \mathcal{M}(N_k \cup N_\ell)$ of k and ℓ has:*

$$\mathcal{D} = k.\mathcal{D} \cup \ell.\mathcal{D} \cup \{(d_k \cup d_\ell) \setminus S \mid d_k \in k.\mathcal{D}, d_\ell \in \ell.\mathcal{D}\}$$

$$put(i) = \begin{cases} focus(\ell, i \cup S) \circ focus(k, i) & i \notin k.\mathcal{D} \wedge i \cup S \notin \ell.\mathcal{D} \\ focus(k, i \cup S) \circ focus(\ell, i) & otherwise \end{cases}$$

$$K(f) = k.K(f|_{N_k}) \wedge \ell.K(f|_{N_\ell})$$

Lemma 2. *$k \mid \ell$ is well-formed whenever k and ℓ are.*

In the definition of $(k \mid \ell).put$, there are two clauses corresponding to running k first and running ℓ first, respectively. The side condition on the first clause—$i \notin k.\mathcal{D} \wedge i \cup S \notin \ell.\mathcal{D}$—itself has two parts that amount to checking that we can safely run $k.put$ and that we can treat all of the outputs of $k.put$ as inputs when running $\ell.put$, respectively. The bias alluded to above arises from the fact that the second clause is active only when the first clause fails; in cases where it is safe to run either k or ℓ first, we arbitrarily choose to run k first.

The danger zone defined for composition ensures that it is safe to run one of the two sub-multiway lenses first. Returning to our example, suppose the base lenses ℓ_f and ℓ_g for the f and g constraints had danger zones saying only that we cannot update all three variables involved at once, that is:

$$\ell_f.\mathcal{D} = \{\{i_1, i_2, o_1\}\}$$
$$\ell_g.\mathcal{D} = \{\{i_1, i_2, o_2\}\}$$

Then, the danger zone for $\ell_f \mid \ell_g$ would be

$$\{\{i_1, i_2, o_1\}, \{i_1, i_2, o_2\}, \{o_1, o_2\}\}$$

which says that an update is dangerous if it's dangerous for either of the sub-multiway lenses *or* if it changes both outputs at once. In the latter case, it would not be safe to run either sub-multiway lens first, since this would over-constrain the other sub-multiway lens.

Multiway lenses are an attractively simple formalism, but they do have some serious drawbacks. For example, it is clear by design that composition is not commutative. A more subtle and important failing is that composition is not *associative*. This is because the composition discussed here treats its components as black boxes: it will run each of its components as a chunk. As a result, we find that $(k \mid \ell) \mid m$ will always run m either first or last—never in between k and ℓ—and will therefore sometimes fail where $k \mid (\ell \mid m)$ can succeed by choosing one of these m-in-the-middle orderings.

3.4 Design axes

A full solution could reasonably build on either the restrictive constraints of the linear solution or the restrictive topology of the tree-structured solution. Because it seems difficult to extend the restrictive constraints solution sufficiently to achieve our top-level goal of embedding all asymmetric, state-based lenses, the approach of extending the restrictive topology solution seems more promising. Below, we discuss some of the difficulties that should be addressed by a successful generalization.

There is a distinction between the dynamic and static semantics of a constraint system graph. Unless specified otherwise, all discussion is of the static semantics.

Definition 20. *Semantics:*

Dynamically ambiguous *means the current instantiation of the graph's constraints and requested updates have multiple satisfying valuations.*

Dynamically unsolvable *means the current instantiation of the graph's constraints and requested updates have no satisfying valuations.*

Statically ambiguous *means the graph can be instantiated with constraints and requested updates which have multiple satisfying valuations.*

Statically unsolvable *means the graph can be instantiated with constraints and requested updates which have no satisfying valuation.*

For an example of the difference between dynamic and static, consider the very simple spreadsheet:

$$x = y$$
$$x = z$$

The requested update $\{y = 3, z = 3\}$ is dynamically solvable because we can choose $x = 3$ to get a complete, consistent valuation. On the other hand, $\{y = 3, z = 4\}$ is dynamically unsolvable. Because there are requested values for y and z that are dynamically unsolvable, the requested update $\{y = 3, z = 3\}$ is statically unsolvable.

3.4.1 Sources of ambiguity

Intra-constraint ambiguity

Consider the very simple constraint system which has only one constraint, $z = x + y$. Giving a value for z gives us a classical "underconstrained system": there are infinitely many choices for x and y that satisfy this constraint. For example, we might choose to keep y and only update x, we might choose to increase x and y by the same summand, we might ignore the old values of x and y altogether and make them both be particular fractions of z, we might attempt to preserve the product $x * y$, etc. In our simple example, when we update the grand total, one reasonable choice would be to scale all the summands by the same factor the grand total was scaled by.

More abstractly, we might wish to have some runtime control over how constraint solutions are being chosen in case there is ambiguity.

Desiderata 1. *Have programmer-level control over the resolution of individual constraints.*

Desiderata 2. *Have high-level control over the resolution of individual constraints.*

Cycles and inter-constraint ambiguity

Above, we discussed the possibility of constraint system graphs with cycles in them. We observed that in such situations, it may be that no ordering of the constraints' methods may result in a consistent state; however, there are also situations where many orderings each result in a consistent state – and indeed, the chosen consistent states may even differ. As a very simple example, consider this system that has some seemingly redundant variables:

$$z_1 = x + y$$
$$z_2 = x + y$$

We will assume that each constraint either allows us to update z_i alone given x and y or allows us to update z_i and y together. The first constraint uses the update policy

$$(z_1', y') = \left(z_1 + \frac{x' - x}{2}, y - \frac{x' - x}{2}\right)$$

which spreads half the change to each variable, while the second constraint uses the update policy

$$(z_2', y') = \left(z_2 + \frac{x' - x}{3}, y - \frac{2(x' - x)}{3}\right)$$

which spreads only a third of the change to z_1 and the rest to y.

Suppose we start from the all-zero valuation and then update x to 6. There are (at least) two reasonable update plans that guarantee consistency: update z_1 and y together to 3 and -3, respectively, then update z_2 to 3, or the symmetric plan that updates z_2 and y together to 2 and -4, then updates z_1 to 2.

Desiderata 3. *Provide high-level control over ambiguous cycles.*

3.4.2 Sources of insolubility

Cycles

Suppose we have three variables, x, and y, and z, and three constraints, one on each pair of variables. We will allow ourselves to assume we also have a collection of methods for each individual constraint that can take an update to one of the variables and produce a value of the other variable that satisfies the constraint. The question now becomes: can we take an update to one variable, say, x, and produce updates to the other two that reinstate all three constraints?

The naive approach, where we compute y from our assumed method that reinstates the $\{x, y\}$ constraint and z from our assumed method that reinstates the $\{x, z\}$ constraint doesn't necessarily work, since there is no guarantee that the y and z computed this way satisfy the $\{y, z\}$ constraint.

Consider our simple example above: there are two "paths" in the constraint graph from the $\text{Total}_{\text{Total}}$ node to the Travel_1 node, namely via Total_1 and via $\text{Travel}_{\text{Total}}$. What we would be asking for is a guarantee that, for example, the way we choose to spread an update over the category totals and thereafter over the individual cells is compatible with the way we choose to spread an update over the day totals and thereafter over the individual cells. In the case of our simple example, we could certainly achieve this using arithmetic facts, but in more complicated examples the way forward is less clear.

Desiderata 4. *Handle dynamically solvable cycles.*

Multiple update

Many constraint propagation systems support the update of multiple variables simultaneously. As discussed in our simple running example, making a vacation plan on a budget might involve setting the grand total and the travel costs all at once. This is distinct from setting them one at a time, since we want the system to guarantee that all three values can coexist, whereas when we set them one at a time each update may disrupt the values of the other two.

Desiderata 5. *Identify solvable multiple updates.*

3.4.3 Other difficulties

Algebraic structure

Experience with programming in other lens languages has shown that sequential composition is a key feature for modularity. Hyperlens composition may not necessarily be sequential, but there should be some tools for developing hyperlenses in a modular way. For this to make sense, we expect that the programmer will wish for composition to be associative (so that how modules are combined does not matter) and may even wish for composition to be commutative (so that the order modules are defined does not matter).

Desiderata 6. *Provide an associative, commutative composition operation.*

Efficiency

We would like the hyperlens project to produce a framework that is usable for very large data sets, even when there is a lot of ambiguity. It is very easy to design constraint system graph topologies where certain updates produce an enormous number of reasonable full update plans; a good solution to the problem should be able to not only choose one of these plans, but also do so in an efficient way.

Desiderata 7. *Select an update plan in polynomial time in the size of the hyperlens.*

Desiderata 8. *Reinstate coherence in a single pass: for each constraint, execute one method at most once.*

Syntax

Any sensible formalism can be instantiated; for example, a sensible hyperlens formalism should include operations that correspond to some of the basic computations done with spreadsheets: some arithmetic, perhaps some simple string operations, aggregations, composition, and so on.

Desiderata 9. *Provide a syntax for basic spreadsheet programming.*

Inter-constraint coordination

It would be nice if the hyperlens associated with

$$x = a + b$$
$$y = x + c$$

behaved “similarly” to the hyperlens associated with

$$x = b + c$$
$$y = a + x$$

in the sense that an update to y in either system resulted in the same updates to a, b, and c. This is nice from a language design point of view because it means you need not introduce separate + functions for each arity, and is nice from a usability point of view because it means that there is no price to pay for modularity: you can split up your code into whatever units make sense to you and get the same program out.

Desiderata 10. *Allow constraints to interact during system update.*

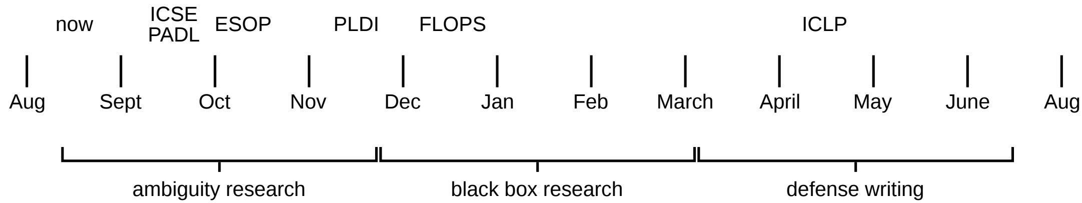


Figure 3.1: Proposed research timeline

3.5 Timeline

Currently, the most promising approach is to use local constraint propagation-style update planning, but add some features to give programmer control of constraint resolution order in situations where the order is ambiguous. One major unfinished piece of work is inventing and evaluating such schemes for providing programmer control. I expect there to be a tradeoff involved between flexibility (how much control the programmer has) and simplicity (how expert the programmer must be to achieve his goals).

Having simple, high-level control is a major goal of the project; nevertheless, I propose to begin with a serious investigation of very flexible, low-level control schemes, with the hopes that (1) a solid understanding of the behavior of low-level controls will guide the design of high-level schemes, and (2) that low-level schemes may serve as a "compilation target" for high-level schemes—and an escape hatch when the less flexible schemes produce undesirable update plans.

The other major unfinished work involves investigating how to incorporate high quality, special-purpose constraint solvers. For example, the linear constraint solver proposed in 3.3.3 has many desirable properties, and would be a convenient component in a larger system; as would dynamic solvers for statically unsolvable cycles.

I propose the following timings:

- Low-level control schemes: 2 months
- High-level control schemes: 1 month
- Incorporating black boxes for linear and cyclic solvers: 1 month
- Extending the linear solver to nonlinear functions: 2 months
- Defense writing: 3 months

This proposed timeline is also shown in Figure 3.1, which includes some possible publication targets as well as one month of slack time for preparing publications.

Chapter 4

Related work

4.1 Symmetric lenses

There is a large literature on lenses and related approaches to propagating updates between connected structures. We discuss only the most closely related work here; good general surveys of the area can be found in [3, 9]. Connections to the literature on *view update* in databases are surveyed in [7]. A short version of this paper is available in [11].

The first symmetric approach to update propagation was proposed by Meertens [15] and followed up in the context of model-driven design by Stevens [23], Diskin [4], and Xiong, et al [27]. Meertens suggests modeling synchronization between two sets X and Y by a *consistency relation* $R \subseteq X \times Y$ and two *consistency maintainers* $\triangleleft : X \times Y \rightarrow X$ and $\triangleright : X \times Y \rightarrow Y$ such that $(x \triangleleft y)\ R\ y$ and $x\ R\ (x \triangleright y)$ always hold, and such that $x\ R\ y$ implies $x \triangleleft y = x$ and $x \triangleright y = y$.

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

We can transform any constraint maintainer into a symmetric lens as follows: take the relation R itself (viewed as a set of pairs) as the complement, and define $putl(x', (x, y)) = (x' \triangleright y, (x', x' \triangleright y))$ and similarly for *putr*. If we compose such a symmetric lens with its opposite we obtain $R \times R^{op}$ as the complement and, for example, $putr(x', ((x_1, y_1), (y_2, x_2))) = (x_2 \triangleleft (x' \triangleright y_1), ((x', x' \triangleright y_1), (x' \triangleright y_1, x_2 \triangleleft (x' \triangleright y_1))))$. For Meertens' counterexample, we would have complements of the form $((G_1, w_1), (w_2, G_2))$, with $w_1 \in L(G_1)$ and $w_2 \in L(G_2)$; "*putr*"-ing a new grammar G'_1 through the composed lens yields the complement $((G'_1, w'_1), (w'_1, G'_2))$, where w'_1 is w_1 if $w_1 \in L(G_1)$ and some default otherwise, and where $G'_2 = G_2$ if $w'_1 \in L(G_2)$ and $S \rightarrow w'_1$ (where S is the start state) otherwise. We observe that there is a property of lenses analogous to Meertens' requirement that $x\ R\ y$ implies $x \triangleleft y = x$. This property is not necessarily preserved by composition, and in particular the lens described above for synchronizing languages does not have it. Meertens recommends using a *chain* of consistency maintainers in such a situation to achieve a similar effect; however, the properties of such chains have not been explored.

For asymmetric lenses, a number of alternative choices of behavioral laws have been explored. Some of these are strictly weaker than ours; for example, a number of papers from a community of researchers based in Tokyo replace the PutGet law with a somewhat looser PutGetPut law, permitting a broader range of useful behaviors for lenses that duplicate information. It would be interesting to see what kind of categorical structures arise from these choices. The proposal by Matsuda et al. [14] is particularly interesting because it also employs the idea of complements. Conversely, stronger laws can be imagined, such as the PutPut law discussed by Foster et al. [7] and the more refined variants in [8].

A different foundation for defining lenses by recursion was explored by Foster et al. [7], using standard tools from domain theory to define monotonicity and continuity for lens combinators

parametrized on other lenses. The main drawback of this approach is that the required (manual) proofs that such recursive lenses are total tend to be somewhat intricate. By contrast, we expect that our initial-algebra approach can be equipped with automatic proofs of totality (that is, choices of the weight function w) in many cases of interest.

4.2 Edit lenses

The most closely related attempt at developing a theory of update propagation is [5] by Diskin et al. Their starting point is the observation (also discussed in [2]) that discovery of edits should be decoupled from their propagation. They thus propose a formalism, *sd-lenses*, for the propagation of edits across synchronized data structures, bearing some similarities with our edit lenses. The replicas, which we model as modules, are there modeled as categories (presented as reflexive graphs). Thus, for any two states x, x' there is a set of edits $X(x, x')$. An sd-lens then comprises two reflexive graphs X, Y and for any $x \in X$ and $y \in Y$ a set $C(x, y)$ of "correspondences" which roughly correspond to our complements. Forward and backward operations similar to our $\Lleftarrow$ and $\Rrightarrow$ then complete the picture. No concrete examples are given of sd-lenses, no composition, no notion of equivalence, and no combinators for constructing sd-lenses; the focus of the paper is rather on the discovery of suitable axioms, such as invertibility and undoability of edits, and a generalization of *hippocraticness* in the sense of Stevens [23]. They also develop a comparison with the state-based framework. In our opinion, the separation of edits and correspondences according to the states that they apply to or relate has two important disadvantages. First, in our examples, it is often the case that one and the same edit applies to more than one state and can be meaningfully propagated (and more compactly represented) as such. For example, while many of the container edits tend to only work for a particular shape, they are completely polymorphic in the contents of the container. Second, the fact that state sets are already categories suggests that a category of sd-lenses would be 2-categorical in flavor, entailing extra technical difficulties such as coherence conditions.

Meertens's seminal paper on *constraint maintainers* [15] discusses a form of containers for lists equipped with a notion of edits similar to our edit language for lists, but does not develop a general theory of edit-transforming constraint maintainers.

A long series of papers from the group at the University of Tokyo [10, 12, 17, 18, 28, etc.] deal with the alignment issue using an approach that might be characterized as a hybrid of state-based and edit-based. Lenses work with whole states, but these states are internally annotated with tags showing where edits have been applied—e.g., marking inserted or deleted elements of lists. Barbosa et al.'s *matching lenses* [2] offer another approach to dealing with issues of alignment in the framework of pure state-based lenses.

4.3 Spreadsheets

The spreadsheet model proposed here draws significant inspiration from the field of constraint programming systems, which dates back to at least the Sketchpad drawing system of 1963 [25]. A good survey of the huge body of work done in this area is given in [26]. In constraint programming languages, programs typically include a series of declarations defining what valuations are valid and invalid; the language runtime is then tasked with finding a valuation which satisfies the constraints. Our proposed work would maintain this broad framework, but extend it by providing more control over which of many possible valuations may be chosen. In particular, our proposal is

to model information both about satisfying valuations (as is done in previous systems) *and* about the evolution of valuations. To achieve this, the modules responsible for re-instating individual declared constraints must be given data about both the old satisfying valuation and the desired new valuation. Additionally, we propose an investigation of methods for separately specifying whether a valuation is valid and how desirable a valuation is.

There is a chain of work on DeltaBlue, a particular constraint programming system, which adds the ability to rank constraints, and break low-ranked constraints during the constraint solution stage [19, 20, 21]. This gives one way of influencing ambiguous solutions: add low-ranked constraints expressing the desirable properties of your valuation. When there are multiple valuations possible on the high-ranked constraints, the low-ranked ones may be used to choose between them. We believe some properties of "desirability" are not naturally representable as constraints, especially when "desirable" means "the new valuation is close to the old one in this way".

The algebraic properties of constraint propagation systems have been explored somewhat [13]. Järvi et al. discuss the model we intend to use as a starting point, and show that it can be decomposed into a structure with an associative, commutative composition and an algorithm which traverses this structure for update planning without losing efficiency. However, because determinism is not a goal for them, they make no efforts to resolve the ambiguity that can arise from the existence of multiple update plans.

There are several systems with proposed solutions that are not based on constraint propagation systems. For example, a GUI resembling a spreadsheet is discussed in [22]. Numerical constraints – including constraints representing assignments of values to particular cells – are shipped out to Mathematica for solution. A significant advantage of this approach is that it can take advantage of the significant solving power of Mathematica. However, this methodology is restricted to data types known by Mathematica; does not provide old cell values to use when updating the spreadsheet; provides little control over which of many satisfying valuations are chosen; and does not attempt to make any guarantees about totality.

Another such system is Tiresias, which extends Datalog with bidirectional capabilities for numeric computations [16]. They give a variant of Datalog that allows for nondeterministic predicates (that is, where tuples may or may not appear) and show how to pick a deterministic instantiation of those predicates that satisfies a Datalog query. The choice of instantiation can also be guided by an objective function to be maximized or minimized. This seems to be a very promising approach, but does have a few important limitations: first, there is a topological constraint (the Datalog query must be stratified); and second, it is not clear that the approach can be easily generalized beyond the small collection of arithmetic predicates that it currently supports.

Bibliography

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

[2] Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: Alignment and view update. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, September 2010.

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

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

[5] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. Technical Report GSDLAB-TR 2011-05-03, University of Waterloo, May 2011.

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

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

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

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

[10] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, September 2010.

[11] Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas*, January 2011.

[12] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bi-directional transformations. In *Partial Evaluation and Program Manipulation (PEPM)*, pages 178–189, 2004. Extended version in *Higher Order and Symbolic Computation*, Volume 21, Issue 1-2, June 2008.

[13] Jaakko Järvi, Magne Haveraaen, John Freeman, and Mat Marcus. Expressing multi-way dataflow constraint systems as a commutative monoid makes many of their properties obvious. In *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming*, pages 25–32. ACM, 2012.

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

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

[16] Alexandra Meliou and Dan Suciu. Tiresias: the database oracle for how-to queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 337–348. ACM, 2012.

[17] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, pages 2–20, November 2004.

[18] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In *Seventh International Conference on Mathematics of Program Construction (MPC)*, 2004.

[19] Michael Sannella. Analyzing and debugging hierarchies of multi-way local propagation constraints. In *Principles and Practice of Constraint Programming*, pages 63–77. Springer, 1994.

[20] Michael Sannella. Skyblue: a multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 137–146. ACM, 1994.

[21] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software: Practice and Experience*, 23(5):529–566, 1993.

[22] Marc Stadelmann. A spreadsheet based on constraints. In *Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 217–224. ACM, 1993.

[23] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS), Nashville, TN*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2007.

[24] Perdita Stevens. Towards an algebraic theory of bidirectional transformations. In *Graph Transformations: 4th International Conference, Icgt 2008, Leicester, United Kingdom, September 7-13, 2008, Proceedings*, page 1. Springer, 2008.

[25] Ivan E Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop*, pages 6–329. ACM, 1964.

[26] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1-2):139–168, 1996.

[27] Y. Xiong, H. Song, Z. Hu, and M. Takeichi. Supporting parallel updates with bidirectional model transformations. *Theory and Practice of Model Transformations*, pages 213–228, 2009.

[28] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *IEEE/ACM International Conference on Automated Software Engineering (ASE), Atlanta, GA*, pages 164–173, 2007.