Lens Extensions

Daniel Wagner

A Dissertation Proposal

in

Computer Science

2013

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, etc. So designing tools that support the creation and maintenance of these transformations is important.

Lots of nice things have been done (e.g. the asymmetric state-based lens model and its instantiation in Boomerang), but there are many niches where we might want more targeted support. For decentralized applications, neither repository is canonical and you want a way to deal with that. For big data, you want small descriptions of changes. Some data structures aren't most naturally represented as strings, but as trees, relations, grids, graphs, or other structures instead.

My thesis will be an exploration of these issues and what can be done to deal with them; the major unfinished work that I will be proposing is grid-structured data.

Chapter 2

Finished work

Is the blank page intentional?

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:

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

The *get* component is the forward transformation, a total function from X to Y. The *put* component takes an old X and a modified Y and yields a correspondingly modified X. 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. 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 record and insert a brand new one (which happens to have similar data to the old record), or

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 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, one such language might be, "spread as much of the change as possible to such-and-such a variable".

?

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 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 N to U.*

We generalize valuation application from names to finite sets of names using the linear ordering on 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 $n \subset N$ 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 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 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: 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.

3.3.3 Linear constraints

Here is a simple spreadsheet which nevertheless fails the tree-structured property described previously:

$$tax = 0.08 * base$$
$$total = base + tax$$

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

Fonts?

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

for some constants b and $\mathbf{c}$. 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 $get \in \mathbb{R}^m \to \mathbb{R}^n$ 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 N such that*

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

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, since (exactly) these motions result in no motion in target-space. ?

Take a basis $\{\mathbf{x}_1, \ldots, \mathbf{x}_{m-n}\}$ for the null space of M. Then we will take:

$$B = \begin{bmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_{m-n}^\top \end{bmatrix}$$

$$N = \begin{bmatrix} M \\ B \end{bmatrix}^{-1} \begin{bmatrix} I_n \\ \mathbf{0}_{m-n,n} \end{bmatrix}$$

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

3.3.4 Non-associative composition

3.4 Design axes

A full solution could reasonably build on either the restrictive constraints or the restrictive topology solutions given above. 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 extension.

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 15. *Semantics:*

Dynamically ambiguous *means the current set of constraints and requested updates have multiple satisfying valuations.*

Dynamically unsolvable *means the current set of constraints and requested updates have no satisfying valuations.*

?

Statically ambiguous *means there is a set of requested updates and a set of constraints whose graph is the current one which is dynamically ambiguous.*

Statically unsolvable *means there is a set of requested updates and a set of constraints whose graph the current one which is dynamically 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

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

Desiderata 7. *Reinstate coherence in a single pass: for each constraint, execute one method at most once.*

Desiderata 8. *Provide a syntax for basic spreadsheet programming.*

Inter-constraint coordination

It would be nice if the hyperlens associated with

$$\begin{aligned} x &= a + b \\ y &= x + c \end{aligned}$$

behaved "similarly" to the hyperlens associated with

$$\begin{aligned} x &= b + c \\ y &= a + x \end{aligned}$$

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 9. *Allow constraints to interact during system update.*

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 used 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.