Matching Lenses: Alignment and View Update

Davi M. J. Barbosa Julien Cretin Nate Foster Michael Greenberg Benjamin C. Pierce
École Polytechnique Princeton University University of Pennsylvania

Abstract

Bidirectional programming languages are a practical approach to the view update problem. Programs in these languages, called *lenses*, define both a view and an update policy—i.e., every program can be read as a function mapping sources to views as well as one mapping updated views back to updated sources.

One thorny issue that has not received sufficient attention in the design of bidirectional languages is *alignment*. In general, to correctly propagate an update to a view, a lens needs to match up the pieces of the view with the corresponding pieces of the underlying source, even after data has been inserted, deleted, or reordered. However, existing bidirectional languages either support only simple strategies that fail on many examples of practical interest, or else propose specific strategies that are baked deeply into the underlying theory.

We propose a general framework of *matching lenses* that parameterizes lenses over arbitrary heuristics for calculating alignments. We enrich the types of lenses with "chunks" identifying reorderable pieces of the source and view that should be re-aligned after an update, and we formulate behavioral laws that capture essential constraints on the handling of chunks. We develop a core language of matching lenses for strings, together with a set of "alignment combinators" that implement a variety of alignment strategies.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

General Terms Languages, Design, Theory

Keywords Bidirectional languages, lenses, alignment, view update problem, Boomerang

1. Introduction

The *view update problem* is a classic issue in data management [6]: given a view and an update to the view, how do we find a new source that accurately reflects the update? Recent work in the programming languages community has made progress on this old problem by developing new languages in which programs, called *lenses*, can be read *both* as view definitions *and* as update translators. This approach avoids code duplication and allows once-and-for-all proofs of round-tripping laws as corollaries of type soundness.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

Formally, a *basic lens* l mapping between sets of sources S and views V with respect to "complements" C comprises functions:

$$\begin{array}{rcl} l.get & \in & S \to V \\ l.res & \in & S \to C \\ l.put & \in & V \to C \to S \\ l.create & \in & V \to S \end{array}$$

The *get* function maps a source to a view. The *res* ("residue") function maps a source to a complement, a structure that records (at least) the information not reflected in the view—i.e., the information that needs to be "remembered" so that it can be mixed together with an updated view to produce an updated source. The other two functions handle updates: *put* takes a view and a complement and builds a new source, while *create* handles the special case where we need to map a view to a source but do not have a complement available. It builds a source from a view directly, filling in any missing information with defaults. We write $S \overset{C}{\Longleftrightarrow} V$ for the set of all basic lenses between S and V with respect to C.[1]

Basic lenses must obey the following laws for every source s, view v, and complement c:[2]

$$l.get\ (l.put\ v\ c) = v \qquad \text{(PUTGET)}$$

$$l.put\ (l.get\ s)\ (l.res\ s) = s \qquad \text{(GETPUT)}$$

These laws are closely related to the conditions on update translators that have been proposed in the database literature [1, 6, 13]. PUTGET ensures that updates to the view are translated "exactly"—i.e., that, given a view and a complement, the *put* function produces a source that *get* maps back to the very same view. GETPUT ensures a "stability" property for the source—i.e., it requires that the *put* function return the original source unmodified whenever the update to the view is a no-op. It also guarantees that the complement computed by *res* records all of the source information not reflected in the view.

Lenses have been studied extensively [3, 4, 9, 12, 17, 18, 19, 20, 21, 23] and applied in areas as diverse as user interfaces [19], structure editors [14], configuration management [17], software model transformations [7, 22, 25], pattern matching [24], data synchronization [8], and security [11]. See [5] for a survey.

However, one fundamental issue continues to hinder wide application of these ideas: *alignment*. In general, the *get* function may discard some of the information in the source, so the *put* function needs to recombine parts of the view with parts of the complement

[1] Readers familiar with lenses will see some small differences from previous formulations: the *put* function has type $V \to C \to S$ rather than $V \to S \to S$ and we assume that every lens has a *res* function that extracts a complement from a source. To recover the original presentation, we can take the set C to be S and let *res* be the identity function. The added precision that we get by breaking out a separate set of complements will be helpful in formulating the concepts we're working with here.

[2] Lenses also obey a CREATEGET law analogous to PUTGET. To save space, we elide this law and all other laws involving *create*. Complete definitions can be found in the long version (via the last author's web page).

to produce the updated source. When the source and view include *ordered* data (lists, strings, XML trees, etc.), doing this recombination correctly requires matching up the pieces of the updated view with the corresponding pieces of the complement. Consider a simple example where the source is a Wiki page

```
=Tour de France=
The Tour is held in July...
=Vuelta a Spain=
The Vuelta is held in September...
```

and the view is a string containing just the section headings:

```
Tour de France
Vuelta a Spain
```

If we change the view by replacing "`Spain`" with "`Espana`" and adding a line for the Giro, we would like the *put* function to take the new view

```
Giro d'Italia
Tour de France
Vuelta a Espana
```

together with the complement of the original source and build a new source reflecting the same updates. But if the lens uses a simple positional strategy—the only one available in most bidirectional languages—then the first line in the view will be matched up with the first section in the source, the second with the second, and so on. The result will be a mangled Wiki page

```
=Giro d'Italia=
The Tour is held in July...
=Tour de France=
The Vuelta is held in September...
=Vuelta a Espana=
```

in which the paragraph for the Tour appears underneath the heading for the Giro and the paragraph for the Vuelta appears underneath the heading for the Tour—a recipe for tragedy in the cycling world!

Existing bidirectional languages deal with the challenge of alignment in different ways. At the simple end of the spectrum, many languages ignore the issue entirely and use a straightforward positional strategy to match up pieces of the source and view [9, 18, 21, 23, 24]. This works in some cases—when the structures are unordered to begin with, or when they are sufficiently rigid that updates only need to modify information in-place, without changing its position—but fails in many others.

Other languages deal with alignment by adopting an operation-based approach [14, 19, 20, 25]—that is, rather than taking the *state* of the new view as an argument, the *put* function is told what update *operation* was applied to the view. Working with explicit operations gives the *put* function detailed information about the nature of the update applied to the view, which can help it calculate a good alignment. However, this solution is not fully general. The "update language" recognized by *put* functions is fixed—and usually simple, to keep the theory manageable—so complicated updates have to be broken down into several smaller ones. For example, many update languages support inserting and deleting items but not moving items from one location to another. To move an item in the view, we have to delete the item and re-insert it at the new location, causing the hidden information associated with the item to be lost.

Finally, a few systems align the pieces of the source and view using *keys*. For example, in our own earlier proposal for *dictionary lenses* [3], the programmer identifies the reorderable *chunks* in the source and specifies how to compute a key for each one. The *put* function uses keys rather than positions to locate a chunk for each piece of the view. This alignment strategy works well when chunks have stable keys, but it is also not a complete solution. In particular,

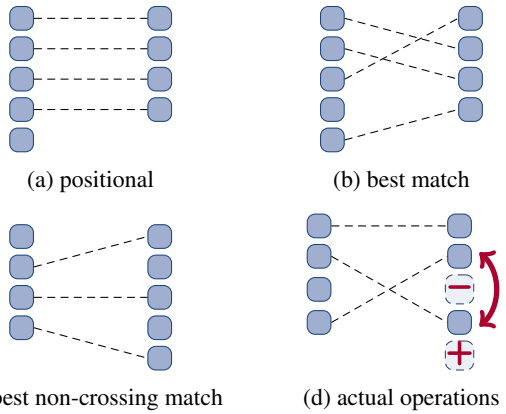

Figure 1. Alignment strategies

when the chunks do not have a natural key (e.g., because they are blocks of otherwise unstructured text) or when keys themselves can be edited in the view (as in the Wiki example above), the simple alignment strategy baked into dictionary lenses can lead to mangled or lost data. Similar limitations apply to *relational lenses* [4], which use functional dependencies to identify keys that can be used to perform database operations like join in an updatable fashion.

Our goal is a completely generic mechanism that overcomes the limitations of all these approaches and addresses the issue of alignment once and for all. To this end, we propose a new framework of *matching lenses* that separates the process of aligning the original and edited views from the process of weaving together the original source and the updated view to build an updated source. This separation yields a flexible framework that can be instantiated with arbitrary heuristic alignment strategies without complicating the basic theory. Figure 1 depicts some possible choices: (a) simple positional alignment; (b) "best match" alignment, which tries to match chunks without regard to ordering (good for set-like data where ordering is not critical); (c) a variant of best-match that only considers "non-crossing" matches, like the longest common subsequence heuristic used by `diff` (this can lose hidden data if the actual edit is a move, but in return it can use local context to guide matching and will often perform better on document-like data); and (d) using the actual edit operations performed by the user (assuming these are available) to calculate exactly the "intended match." The matching lens framework can accommodate all of these, and many others.

At the level of the mathematical semantics, we enrich lens types with annotations specifying what constitutes a reorderable "chunk," and we add behavioral laws that capture the essential constraints on the handling of chunks—e.g., these stipulate that lenses must carry chunks in the source through to chunks in the view and vice versa, and they guarantee that reorderings on the chunks in the view are translated to corresponding reorderings on the source.

Operationally, we make the separation of concerns described above explicit by splitting the complement into two pieces: a *rigid complement* that represents the source information that should be handled positionally and a *resource* that represents the information extracted from chunks. To supply a lens with a precise alignment directive, we simply rearrange the resource according to the directive and obtain a pre-aligned resource in which each piece of the source matches up with the specified piece of the view.

Finally, we instantiate this abstract framework with primitive matching lenses and combinators for string data.[3] We give coercions that convert basic lenses to matching lenses and vice versa, and we show how to interpret each of the regular operators (union, concatenation, and Kleene star) as well as the composition operator on matching lenses. Finally, we describe primitives for specifying, combining, and tuning alignment strategies using notions of "species," "tags," "keys," and "thresholds." Our contributions can be summarized as follows:

1. We define a new semantic space of *matching lenses* that enriches the types of lenses with chunks and adds behavioral laws ensuring that chunks are handled correctly (Section 3).
2. We define a simple syntax for matching lenses over string data (Section 4), and we prove (in the long version of the paper) that each primitive is well behaved at its specified type.
3. We develop several ways of calculating alignments, using notions of "species", "tags", "keys," and "thresholds" (Section 5). Alignments are represented as concrete data structures, making a clean interface between the core lens behaviors described in Sections 3 and 4 and these alignment-generating algorithms.
4. We sketch extensions to the framework (Section 6) and discuss an implementation based on Boomerang [10] (Section 7).

Related and future work are discussed in Sections 8 and 9.

2. Example

Let's push the Wiki example a little further, to preview the essential ingredients of our solution. First, here is a Boomerang program that implements the original Wiki lens with positional alignment:

```
let NONSPECIAL : regexp = [^=\n]
let HEADING : regexp = [^=\n ] . NONSPECIAL*
let LINES : regexp = (NONSPECIAL+ . "\n")*
let PARAGRAPHS : regexp = LINES . ("\n" . LINES)*

let section : lens =
  "="<->"" .
  copy HEADING .
  ("=\n" . PARAGRAPHS)<->"\n"
let wiki : lens = section*
```

The first few lines define regular expressions describing "non-special" characters (everything except '`=`' and '`\n`'), section headings, lines of text, and paragraphs. The `section` lens processes one section of the Wiki source. The `copy` E lens recognizes a source string matching the regular expression E and copies it (in both directions). The *get* direction of the "replace by constant" lens E `<->` u recognizes a source string matching E but adds the fixed string u to the view; the put direction recognizes u and restores the original source from the complement. The concatenation operator `.` uses one sublens to process the beginning of its input and the other for the end. The `wiki` lens, defined using the Kleene star operator `*`, iterates the `section` lens to handle a list of sections.

This version of the Wiki lens uses a simple positional alignment strategy for matching up paragraphs in the source with lines in the view, leading to the unfortunate behavior described in the introduction. Here is a better version, written using the features developed in this paper, that uses section headings to locate the corresponding paragraphs from the source:

[3] We work concretely with strings, rather than richer structures like trees or graphs, but we use regular expression types to overlay tree structures onto strings. Indeed, our types are already expressive enough to describe arbitrary XML structures with non-recursive schemas.

```
let section : lens =
  "="<->"" .
  key (copy HEADING) .
  ("=\n" . PARAGRAPHS)<->"\n"
let wiki : lens = <best:section>*
```

We've made two changes. First, in the definition of the `wiki` lens, we have indicated that each section in the source should be treated as a reorderable "chunk" by enclosing the `section` lens in angle brackets. Second, we have specified the policy that should be used to align chunks using the annotations `key` and `best`. The `key` annotation indicates the portions of each chunk that should be taken into account when computing an alignment. The "alignment species" `best` selects the overall heuristic to use for computing a correspondence between chunks: one that minimizes the sum of the edit distances between the keys of matched chunks.

The point of this example is that we can provide programmers with simple, compositional primitives that allow them to specify rich alignment strategies directly in a lens program. In particular, if we put back the updated view

```
Giro d'Italia
Tour de France
Vuelta a Espana
```

with the complement from the original source, we obtain a new source in which paragraphs are restored to the appropriate sections:

```
=Giro d'Italia=
=Tour de France=
The Tour is held in July...
=Vuelta a Espana=
The Vuelta is held in September...
```

Readers familiar with dictionary lenses [3] will recall similar constructs for specifying chunks and alignment policies. Indeed, on the surface, matching lenses are designed to look like a straightforward generalization of dictionary lenses. Under the hood, the critical difference that makes the generalization work is that matching lenses make all alignment decisions in a *separate phase* that happens before the outermost *put* function is called, whereas dictionary lenses *interleave* alignment decisions with the operation of *put* functions. This untangling of mechanisms has several beneficial effects. First, it *modularizes* the framework, allowing us to use many different alignment strategies instead of just one. Second, it allows us to use *global alignment strategies* that optimize some distance metric over the whole structure; in particular, we can relax the assumption that keys are never edited, a major practical restriction of dictionary lenses. Third, it permits an elegant treatment of the *composition* operator (see Section 4), which we have found important in practical bidirectional programming but which doesn't interact nicely with dictionary lenses. Fourth, it clarifies the underlying *theory* by treating alignment algorithms, which are typically complex and heuristic, separately from the core language, which remains simple and generic. And finally, it avoids some arbitrary choices forced by the locality of alignment decisions in dictionary lenses—e.g., the left bias of concatenation and Kleene star.

3. Semantics

We begin our technical development by defining the semantic space of matching lenses, which are organized around a two-tier architecture: a top-level *matching lens* processes the information outside of chunks while a subordinate *basic lens* processes the chunks themselves. To simplify the presentation, we assume in this section that chunks only appear at the top level, that the same basic lens is used to process every chunk, and that lenses themselves do not reorder chunks. We relax each of these assumptions in Section 6.

Consider an example that illustrates how matching lenses work:

```
let k : lens = key (copy [A-Z]) . [a-z]<->""
let l : lens = <best:k> . (copy "," . <best:k>)*
```

The basic lens `k` copies an upper-case letter from source to view and deletes a lower-case letter, while the matching lens `l` iterates `k` over a non-empty list of comma-separated chunks. The behavior of `l`'s *get* component is straightforward—e.g., it maps "`Xx,Yy,Zz`" to "`X,Y,Z`". Its *put* function is more interesting: it restores the lower-case letters from source chunks by matching upper-case letters in the old and new views. For example, if we reorder the view and insert "`W`" in the middle, then *put* behaves as follows:

```
l.put "Z,Y,W,X" into "Xx,Yy,Zz" = "Zz,Yy,Wa,Xx"
```

The evaluation of `l`.*put* proceeds in several steps. First, it uses `l`.*res* to extract a complement from the source. In a matching lens, the complement is represented as two structures: a *rigid complement* c that contains the information outside of chunks and a *resource* r that contains the information within chunks.

$$c = (\Box, [(\text{“,”}, \Box), (\text{“,”}, \Box)]) \qquad r = \left\{\!\!\left\{ \begin{array}{l} 1 \mapsto \text{“Xx”} \\ 2 \mapsto \text{“Yy”} \\ 3 \mapsto \text{“Zz”} \end{array} \right\}\!\!\right\}$$

The rigid complement records the position of each chunk and the commas separating the chunks; its structure (a pair whose second component is a list of pairs) comes from the structure of the lens `l`. The resource records a mapping from chunk locations to chunk contents—i.e., $r(i)$ contains the contents of the ith chunk in s, while c has a $\Box$ at the corresponding location.

Next, the *put* function invokes an alignment function to compute a correspondence g between the locations of chunks in the new and old views, and composes this correspondence with the resource r to obtain a "pre-aligned" resource $(r \circ g)$:

$$g = \begin{array}{cc} \texttt{Z} & \\ \texttt{Y} & \texttt{X} \\ \texttt{W} & \texttt{Y} \\ \texttt{X} & \texttt{Z} \end{array} = \left\{\!\!\left\{ \begin{array}{l} 1 \mapsto 3 \\ 2 \mapsto 2 \\ 4 \mapsto 1 \end{array} \right\}\!\!\right\} \qquad (r \circ g) = \left\{\!\!\left\{ \begin{array}{l} 1 \mapsto \text{“Zz”} \\ 2 \mapsto \text{“Yy”} \\ 4 \mapsto \text{“Xx”} \end{array} \right\}\!\!\right\}$$

Finally, it runs `l`.*put* on the updated view, the rigid complement, and the pre-aligned resource. The overall effect is that each lower-case letter is restored to the chunk containing the appropriate upper-case letter. Notice that the third chunk, `W` is created with the default lower-case letter "`a`" because the pre-aligned resource $(r \circ g)$ is undefined on location 3.

3.1 Preliminaries

Before we can define matching lenses formally, we need some notation for chunks and resources.

Structures with Chunks The semantic space of matching lenses is generic—the source and view can be arbitrary—but we require that structures come equipped with a notion of what constitutes a *chunk*. When u is a (source or view) structure with chunks, we write

- $\mathit{chunks}(u)$ for the list of chunks in u,
- $|u|$ for the length of $\mathit{chunks}(u)$,
- $\mathit{locations}(u)$ for the set $\{1, ..., |u|\}$ of locations of chunks in u,
- $u[n]$ for the nth chunk in u, where n is in $\mathit{locations}(u)$,
- $u[n{:=}v]$ for the structure obtained by setting the nth chunk in u to v, and
- $\mathit{skel}(u)$ for the skeleton structure obtained by replacing each chunk in u with $\Box$,

and we require that the number of chunks $|u|$ be equal to the number of occurrences of $\Box$ in $\mathit{skel}(u)$. Examples of such structures abound, including conventional datatypes such as trees, lists, matrices, etc. Jay's "shapely types", which require that it be possible to divide structures into a *shape* and a list of *data* items with the arity of the shape equal to the number of items in the list, capture the same concept [15].

Chunk Compatibility To ensure that chunks can be freely reordered, we require that the sets of sources and views must be closed under the operation of replacing chunks by other chunks. Formally, we say that a set of structures with chunks U is *chunk compatible* with an ordinary set of structures U' if and only if

- the chunks of every structure in U belong to U'—i.e., for every u in U and n in $\mathit{locations}(u)$ we have $u[n]$ in U', and
- membership in U is preserved when we replace arbitrary chunks with elements of U'—i.e., for every u in U, n in $\mathit{locations}(u)$, and u' in U' we have $u[n{:=}u']$ in U.

Resources We represent resources using finite maps from locations to basic lens complements. This makes it easy to re-align a resource—we simply apply a (possibly lossy) reordering to the map. We write

- $\{\!|\}\!|$ for the totally undefined map,
- $\{\!|n \mapsto c|\}\!|$ for the singleton map that associates the location n to the basic lens complement c and is otherwise undefined,
- $r(n)$ for the basic lens complement that the finite map r associates to n,
- $\mathrm{dom}(r)$ for the domain of the finite map r,
- $|r|$ for the largest element of $\{0\} \cup \mathrm{dom}(r)$,
- $(r_1 \mathbin{+\!\!+} r_2)$ for the finite map that behaves like the finite map r_1 on locations in $\mathrm{dom}(r_1)$ and like the finite map r_2 with locations shifted up by $|r_1|$ otherwise,

$$(r_1 \mathbin{+\!\!+} r_2)(n) \triangleq \begin{cases} r_1(n) & \text{if } n \leq |r_1| \\ r_2(n - |r_1|) & \text{otherwise,} \end{cases}$$

  and

- $\{\!|\mathbb{N} \mapsto C|\}\!|$ for the set of all finite maps from locations to elements of C, where C is a set of basic lens complements.

3.2 Matching Lenses

Let S and V be sets of structures with chunks, C a set of structures ("rigid complements"), and k a basic lens in $S_k \overset{C_k}{\Longleftrightarrow} V_k$ such that S is chunk compatible with S_k and V is chunk compatible with V_k. Also let *align* be a function that takes the list of chunks for the new and old views and computes a correspondence between them, represented formally as a partial injective function from old locations to new locations. A *matching lens* l on S, C, k, and V comprises four functions

$$\begin{array}{rcl} l.\mathit{get} & \in & S \to V \\ l.\mathit{res} & \in & S \to C \times \{\!|\mathbb{N} \mapsto C_k|\}\!| \\ l.\mathit{put} & \in & V \to C \times \{\!|\mathbb{N} \mapsto C_k|\}\!| \to S \\ l.\mathit{create} & \in & V \to \{\!|\mathbb{N} \mapsto C_k|\}\!| \to S \end{array}$$

obeying the laws in Figure 2 (to be explained below). We write $S \overset{C,k}{\Longleftrightarrow} V$ for the set of all matching lenses between S and V with respect to C and k.

The *get* function has the same type as in basic lenses. The *put* function takes a view together with a rigid complement and a resource as arguments, while the *res* function extracts a rigid complement and a resource from a source. The *create* function

$$l.get\ (l.put\ v\ (c,r)) = v \qquad \text{(PUTGET)}$$

$$l.put\ (l.get\ s)\ (l.res\ s) = s \qquad \text{(GETPUT)}$$

$$locations(s) = locations(l.get\ s) \qquad \text{(GETCHUNKS)}$$

$$\frac{c, r = l.res\ s}{locations(s) = \mathrm{dom}(r)} \qquad \text{(RESCHUNKS)}$$

$$\frac{n \in (locations(v) \cap \mathrm{dom}(r))}{(l.put\ v\ (c,r))[n] = k.put\ v[n]\ (r(n))} \qquad \text{(CHUNKPUT)}$$

$$\frac{n \in (locations(v) \setminus \mathrm{dom}(r))}{(l.put\ v\ (c,r))[n] = k.create\ v[n]} \qquad \text{(NOCHUNKPUT)}$$

$$\frac{skel(v) = skel(v')}{skel(l.put\ v\ (c,r)) = skel(l.put\ v'\ (c,r'))} \qquad \text{(SKELPUT)}$$

Figure 2. Matching lens laws

takes just a view and a resource; this makes it possible for matching lenses to restore information to chunks whose rigid complement is newly created—e.g., the last chunk in the example from the beginning of this section, which contains "`X`". To create a source from scratch, we invoke *create* with the empty resource $\{\!|\}$.

The PUTGET and GETPUT laws in Figure 2 express the same fundamental conditions as the corresponding basic lens laws; the remaining laws capture essential constraints on the handling of chunks. The GETCHUNKS law stipulates that the *get* function must carry each chunk in the source to a chunk in the view; the RESCHUNKS law imposes an analogous constraint on the resource generated by *res*. (We do not state PUTCHUNKS as a law because it can be derived from the other laws.)

The CHUNKPUT and NOCHUNKPUT laws are the most important. They ensure that the *put* function uses its resource argument correctly. The CHUNKPUT law stipulates that the nth chunk in the source produced by *put* must be identical to the structure produced by applying $k.put$ to the nth chunk in the view and the complement associated to n in the resource (if the resource contains a complement for n). For instance, in the example above, it stipulates that the second chunk in the source obtained by putting back the updated view "`Z,Y,W,X`" using the pre-aligned resource $(r \circ g)$ must be equal to the result obtained by applying `k`.*put* to "`Z,Y,W,X`"[2] and $(r \circ g)(2)$—i.e., to "`Y`" and "`Yy`". The NOCHUNKPUT law is similar, but handles the case where the resource does not contain a complement for n. For example, it stipulates that the third chunk in the source must be equal to the result obtained by applying `k`.*create* to "`Z,Y,W,X`"[3]—i.e., "`W`".

The last law, SKELPUT, states that the skeleton of the sources produced by *put* must not depend on any of the chunks in the view or complements in the resource. Among other things, this law is critical for ensuring that matching lenses translate reorderings on the view to reorderings on the source.

Compared to the basic lens laws, these laws have a somewhat low-level and operational feel, spelling out the handling of chunks and resources in quite a bit of detail. Other axiomatizations of matching lenses are possible (see Section 9). We chose these laws because they express conditions that are readily verified using simple, local checks. In addition, we can use them to derive higher-level properties. For instance, we can show that the *put* function translates reorderings on the chunks in the view to corresponding reorderings on the chunks in the source. We write $\mathsf{Perms}(u)$ for the set of permutations of chunks in u and $\circlearrowleft_q\ u$ for the structure obtained by reordering the chunks of u according to a permutation q. The next lemma follows directly from the matching lens laws:

3.1 Lemma [ReorderPut]: Let l be a matching lens in $S \overset{C,k}{\Longleftrightarrow} V$. For every view v in V, rigid complement c in C, resource r in $\{\!|\mathbb{N} \mapsto k.C|\}$, and permutation q in $\mathsf{Perms}(v)$, we have $\circlearrowleft_q\ (l.put\ v\ (c,r)) = l.put\ (\circlearrowleft_q\ v)\ (c, r \circ q^{-1})$.

Lowering To complete the discussion of semantics, we define a coercion $\lfloor\cdot\rfloor$ (pronounced "lower") that takes a matching lens l in $S \overset{C,k}{\Longleftrightarrow} V$ and packages it up with the interface of a basic lens in $S \overset{S}{\Longleftrightarrow} V$. This coercion performs the steps needed to actually *use* the *put* component of a matching lens, as described in the example at the start of this section. It turns out that we only need a single constraint on the alignment function *align* to ensure well-behavedness of the lens resulting from $\lfloor\cdot\rfloor$: when presented with identical lists of chunks, it must yield the identity alignment.

$$\frac{l \in S \overset{C,k}{\Longleftrightarrow} V}{\lfloor l \rfloor \in S \overset{S}{\Longleftrightarrow} V}$$

$$\begin{array}{ll} get\ s & = l.get\ s \\ res\ s & = s \\ put\ v\ s & = l.put\ v\ (c,\ r \circ g) \\ & \quad \text{where } (c,r) = l.res\ s \\ & \quad\quad \text{and } g = align(chunks(v), chunks(l.get\ s)) \\ create\ v & = l.create\ v\ \{\!|\}\end{array}$$

The typing rule in the top box can be read as a lemma asserting that, if l is a matching lens in $S \overset{C,k}{\Longleftrightarrow} V$, then $\lfloor l \rfloor$ is a basic lens in $S \overset{S}{\Longleftrightarrow} V$. A proof appears in the long version of this paper.

The bottom box defines the components of $\lfloor l \rfloor$. The *get* function is just $l.get$. The *res* function uses the whole source as the basic lens complement. The *put* function takes a (possibly updated) view v and a basic lens complement s as arguments. It first uses $l.res$ to calculate a rigid complement c and a resource r from s, and then uses *align* to calculate a correspondence g between the locations of chunks in the updated view v and chunks in the original view $l.get\ s$. Next, it composes r and g as functions, which has the effect of rearranging the complements in the resource r according to the alignment g. To finish the job, the *put* function passes v, c, and $(r \circ g)$ to $l.put$, which produces the new source. The basic *create* function invokes $l.create$ with the view and the empty resource.

4. Matching Lenses for Strings

Having defined the semantic space of matching lenses, we now turn our attention to syntax. This section defines a collection of matching lens primitives for strings, based on the basic and dictionary lenses for strings that we have studied previously [3, 12].

4.1 Notation

Let Σ be a finite alphabet (e.g., ASCII). The ϵ symbol denotes the empty string and $(u{\cdot}v)$ denotes the concatenation of strings u and v. A language L is a subset of Σ^*; concatenation is lifted to languages in the usual way: $L_1{\cdot}L_2 \triangleq \{u{\cdot}v \mid u \in L_1 \text{ and } v \in L_2\}$. The iteration of a language L is $L^* \triangleq \bigcup_{n=0}^{\infty} L^n$, where L^n denotes the n-fold concatenation of L with itself.

Many of our definitions require that every string in the concatenation of two languages have a unique factorization into smaller strings belonging to the languages being concatenated.

Two languages L_1 and L_2 are *unambiguously concatenable*, written $L_1 \cdot^! L_2$, provided that, for all strings u_1 and v_1 in L_1 and u_2 and v_2 in L_2, if $(u_1 \cdot u_2) = (v_1 \cdot v_2)$ then $u_1 = v_1$ and $u_2 = v_2$. Similarly, a language L is *unambiguously iterable*, written $L^{!*}$, if for all strings u_1 to u_m and v_1 to v_n in L, if $(u_1 \cdots u_m) = (v_1 \cdots v_n)$ then $m = n$ and $u_i = v_i$ for i from 1 to n.

The set of regular expressions is generated by the grammar

$$\mathcal{R} ::= \emptyset \mid u \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R} | \mathcal{R} \mid \mathcal{R}^*$$

where u ranges over strings. The denotation $[\![E]\!]$ of a regular expression E is a regular language. Regular languages are closed under the boolean operators and have many decidable properties including emptiness, inclusion, and equivalence. It is also decidable whether two regular languages are unambiguously concatenable and whether a single regular language is unambiguously iterable (see [2, Prop. 4.1.3]).

4.2 Types

The types of our primitives are given by regular languages of strings decorated with annotations that indicate the locations of chunks. Let '$\langle$' and '$\rangle$' be fresh symbols not in Σ. The set of *chunk-annotated regular expressions* is generated by the grammar:

$$\mathcal{A} \ ::= \ \mathcal{R} \ \mid \ \langle \mathcal{R} \rangle \ \mid \ \mathcal{A} \,|\, \mathcal{A} \ \mid \ \mathcal{A} \cdot \mathcal{A} \ \mid \ \mathcal{A}^*$$

Note that every ordinary regular expression is also a chunk-annotated regular expression and that chunks are not nested. The denotation $[\![A]\!]$ of a chunk-annotated regular expression A is a language of chunk-annotated strings—i.e., a set of strings over the extended alphabet $\Sigma \cup \{\text{'}\langle\text{'}, \text{'}\rangle\text{'}\}$ in which occurrences of '$\langle$' and '$\rangle$' are balanced and not nested. We use these annotations to determine the number $|u|$ of chunks in u, the chunk $u[n]$ at n in u, and so on. For example, if u is "$\langle$A1$\rangle\langle$B2$\rangle\langle$C3$\rangle$", then $|u|$ is 3, $u[2]$ is "B2", and $u[2{:=}$"Z9"$]$ is "$\langle$A1$\rangle\langle$Z9$\rangle\langle$C3$\rangle$".

Although our primitives formally manipulate chunk-annotated strings, we can also use them to process ordinary strings—indeed, this is how we most often use them! Let $\lfloor \cdot \rfloor$ be the function that maps chunk-annotated strings to ordinary strings (by removing '$\langle$' and '$\rangle$' characters and mapping every other character to itself), and lift $\lfloor \cdot \rfloor$ to languages in the obvious way. We say that a language of chunk-annotated strings L is *chunk unambiguous* if and only if L is isomorphic to $\lfloor L \rfloor$. Not all languages are chunk unambiguous—e.g., in $\{$"$\langle a \rangle b$", "$a\langle b \rangle$"$\}$ the ordinary string "ab" corresponds to two different chunk-annotated strings—but for languages that are, we can get back and forth between ordinary strings and chunk-annotated strings unambiguously. Using the isomorphism between a chunk-unambiguous language and its erasure, we can view our matching lens primitives as acting either on chunk-annotated strings or on ordinary strings. To use a component of a lens to process an ordinary string, we first "parse" the input string, then apply the lens function to the resulting chunk-annotated string, and finally erase the annotations in the chunk-annotated output string. Moreover, for languages given by chunk-annotated regular expressions, implementing parse and erase functions is straightforward. In each of the typing rules below, we will be careful to ensure that the source and view types are chunk unambiguous.

4.3 Primitives

The first two primitives convert basic lenses into matching lenses.

Lift It should be clear that matching lenses generalize basic lenses. The *lift* operator witnesses this fact, and makes it possible to use basic lenses like `copy` and `<->` as matching lenses. As the source and view are ordinary strings, the lifted lens does not have chunks so it satisfies the new matching lens laws vacuously.

$$\frac{k' \in S' \overset{C'}{\Longleftrightarrow} V' \qquad k \in S \overset{C}{\Longleftrightarrow} V}{\widehat{k} \in S \overset{C,k'}{\Longleftrightarrow} V}$$

$$\begin{array}{ll} \mathit{get}\ s & = k.\mathit{get}\ s \\ \mathit{res}\ s & = k.\mathit{res}\ s, \{\!|\}\!\} \\ \mathit{put}\ v\ (c, r) & = k.\mathit{put}\ v\ c \\ \mathit{create}\ v\ r & = k.\mathit{create}\ v \end{array}$$

Note that the basic lens k' mentioned in the type of $\widehat{k}$ is arbitrary.

Match Another way to lift a basic lens is to place it in a chunk:

$$\frac{k \in S \overset{C}{\Longleftrightarrow} V}{\langle k \rangle \in \langle S \rangle \overset{\{\Box\},k}{\Longleftrightarrow} \langle V \rangle}$$

$$\begin{array}{ll} \mathit{get}\ s & = k.\mathit{get}\ s \\ \mathit{res}\ s & = \Box, \{\!|1 \mapsto k.\mathit{res}\ s|\}\!\} \\ \mathit{put}\ v\ (\Box, r) & = \begin{cases} k.\mathit{put}\ v\ (r(1)) & \text{if } 1 \in \mathrm{dom}(r) \\ k.\mathit{create}\ v & \text{otherwise} \end{cases} \\ \mathit{create}\ v\ r & = \begin{cases} k.\mathit{put}\ v\ (r(1)) & \text{if } 1 \in \mathrm{dom}(r) \\ k.\mathit{create}\ v & \text{otherwise} \end{cases} \end{array}$$

The lens $\langle k \rangle$ (pronounced "match k") is the essential matching lens. It uses the basic lens k to process strings in both directions, treating the entire source as a reorderable chunk. The *get* component of $\langle k \rangle$ simply passes off control to the basic lens k. The *res* function takes a source s and produces $\Box$ as the rigid complement and $\{\!|1 \mapsto k.\mathit{res}\ s|\}\!\}$ as the resource. The *put* function accesses the complement through its resource argument: it invokes $k.\mathit{put}$ on the view and $r(1)$ if r is defined on 1 and $k.\mathit{create}$ on the view otherwise. The *create* function is identical. In examples, we often specify the global *align* parameter as an argument to match—e.g., we write `<best:k>` to indicate that chunks should be aligned using the "best" heuristic. The typechecker verifies that all occurrences of match use the same heuristic—see Section 5.

Concatenation These regular operators represent a core language that can be used to express many useful transformations on strings. Concatenation is simplest:

$$\frac{\begin{array}{ll} l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1 & \lfloor S_1 \rfloor \cdot^! \lfloor S_2 \rfloor \\ l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2 & \lfloor V_1 \rfloor \cdot^! \lfloor V_2 \rfloor \end{array}}{l_1 \cdot l_2 \in (S_1 \cdot S_2) \overset{(C_1 \times C_2),k}{\Longleftrightarrow} (V_1 \cdot V_2)}$$

$$\begin{array}{ll} \mathit{get}\ (s_1 \cdot s_2) & = (l_1.\mathit{get}\ s_1) \cdot (l_2.\mathit{get}\ s_2) \\ \mathit{res}\ (s_1 \cdot s_2) & = (c_1, c_2), (r_1 +\!\!+ r_2) \\ & \quad \text{where } c_1, r_1 = l_1.\mathit{res}\ s_1 \\ & \quad \text{and } c_2, r_2 = l_2.\mathit{res}\ s_2 \\ \mathit{put}\ (v_1 \cdot v_2)\ (c, r) & = (l_1.\mathit{put}\ v_1\ (c_1, r_1)) \cdot (l_2.\mathit{put}\ v_2\ (c_2, r_2)) \\ & \quad \text{where } c_1, c_2 = c \\ & \quad \text{and } r_1, r_2 = \mathit{split}(|v_1|, r) \\ \mathit{create}\ (v_1 \cdot v_2)\ r & = (l_1.\mathit{create}\ v_1\ r_1) \cdot (l_2.\mathit{create}\ v_2\ r_2) \\ & \quad \text{where } r_1, r_2 = \mathit{split}(|v_1|, r) \end{array}$$

The *get* function splits the source into s_1 and s_2, applies the *get* functions of l_1 and l_2 to these strings, and concatenates the results. We write $s_1 \cdot s_2$ in patterns to indicate that s_1 and s_2 are strings in S_1 and S_2 that concatenate to $s_1 \cdot s_2$. The typing rule requires that the concatenation of $\lfloor S_1 \rfloor$ and $\lfloor S_2 \rfloor$ be unambiguous, so s_1 and s_2 are unique. Also, as S_1 and S_2 are chunk unambiguous, this condition also ensures that $S_1 \cdot S_2$ is chunk unambiguous.

The *res* function applies $l_1.res$ to s_1 and $l_2.res$ to s_2, yielding rigid complements c_1 and c_2 and resources r_1 and r_2. It merges the rigid complements into a pair (c_1, c_2) and the resources into a finite map $(r_1 +\!\!+ r_2)$. As the same basic lens k is mentioned in the types of both l_1 and l_2, the resources r_1, r_2, and $(r_1 +\!\!+ r_2)$ are all finite maps in $\{\!|\mathbb{N} \mapsto C_k|\!\}$.[4] This ensures that we can freely reorder the resource and pass arbitrary portions of it to l_1 and l_2.

The *put* function splits each of the view, rigid complement, and resource in two, applies the *put* functions of l_1 and l_2 to the corresponding pieces of each, and concatenates the results. It splits the resource using *split*, which yields a resource that behaves like r on locations less than or equal to $|v_1|$ and one that behaves like r shifted down by $|v_1|$ on locations greater than $|v_1|$. Formally, *split* is defined as follows:

$$(\pi_1(split(n,r)))(m) = \begin{cases} r(m) & \text{if } m \le n \text{ and } m \in \mathrm{dom}(r) \\ \text{undefined} & \text{otherwise} \end{cases}$$
$$(\pi_2(split(n,r)))(m) = \begin{cases} r(m+n) & \text{if } (m+n) \in \mathrm{dom}(r) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Splitting resources in this way ensures that a complement aligned with a chunk in the view remains aligned with the same chunk in the corresponding substring of the view. The proof of GETPUT uses the equality $split(|r_1|, r_1 +\!\!+ r_2) = (r_1, r_2)$.

The typing rule requires that l_1 and l_2 be defined over the same basic lens k, which ensures that the resource $(r_1 +\!\!+ r_2)$ has a uniform type. It is tempting to relax this condition and allow l_1 and l_2 to be defined over different basic lenses, as long as those lenses have compatible complements. Unfortunately, this would require accepting weaker properties. For example, consider $\langle k_1 \rangle \cdot \langle k_2 \rangle$, where k_1 and k_2 are defined as follows:

$$k_1 \triangleq (\mathtt{a} \leftrightarrow \mathtt{b} \,|\, \mathtt{b} \leftrightarrow \mathtt{a} \,|\, \mathtt{c} \leftrightarrow \mathtt{c}) \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \overset{()}{\Longleftrightarrow} \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$$
$$k_2 \triangleq (\mathtt{a} \leftrightarrow \mathtt{b} \,|\, \mathtt{b} \leftrightarrow \mathtt{c} \,|\, \mathtt{c} \leftrightarrow \mathtt{a}) \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \overset{()}{\Longleftrightarrow} \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$$

Invoking *put* on "bc" yields "ab" as a result.[5] Swapping the chunks of "bc" gives "cb". According to Lemma 3.1, the *put* function should produce "ba"—i.e., the string obtained by swapping the chunks of "ab". But this is not what happens: invoking *put* on "cb" yields "ca". Thus, although it is tempting to allow matching lenses to use different lenses to process chunks, we do not allow it.

Kleene Star The Kleene star operator iterates a lens:

$$\frac{\lfloor S \rfloor^{!*} \qquad \lfloor V \rfloor^{!*} \qquad l \in S \overset{C,k}{\Longleftrightarrow} V}{l^* \in S^* \overset{(C\ \mathrm{list}),k}{\Longleftrightarrow} V^*}$$

$$\begin{aligned}
&get\,(s_1 \cdots s_n) &&= (l.get\,s_1) \cdots (l.get\,s_n) \\
&res\,(s_1 \cdots s_n) &&= [c_1, \ldots, c_n], (r_1 +\!\!+ \ldots +\!\!+ r_n) \\
&\quad \text{where } c_i, r_i = l.res\ s_i \text{ for } i \in \{1, \ldots, n\} \\
&put\,(v_1 \cdots v_n)\,(c, r) &&= s'_1 \cdots s'_n \\
&\quad \text{where } s'_i = \begin{cases} l.put\ v_i\ (c_i, r_i) & i \in \{1, \ldots, \min(n, m)\} \\ l.create\ v_i\ r_i & i \in \{m+1, \ldots, n\} \end{cases} \\
&\qquad \text{and } [c_1, \cdots, c_m] = c \\
&\qquad \text{and } r'_0 = r \\
&\qquad \text{and } r_i, r'_i = split(|v_i|, r'_{(i-1)}) \text{ for } i \in \{1, \ldots, n\} \\
&create\,(v_1 \cdots v_n)\,r &&= (l.create\,v_1\,r_1) \cdots (l.create\,v_n\,r_n) \\
&\quad \text{where } r'_0 = r \\
&\qquad \text{and } r_i, r'_i = split(|v_i|, r'_{(i-1)}) \text{ for } i \in \{1, \ldots, n\}
\end{aligned}$$

[4] Recall that C_k is the set of basic lens complements for k.

[5] As k_1 and k_2 are "bijective", the rigid complement and resource do not affect the evaluation of *put*.

The *get* and *res* components of the Kleene star lens are straightforward generalizations of the corresponding components of the concatenation lens. The *put* function, however is different. Because it must be a total function, it needs to handle situations where the number of substrings of the view is different than the number of items in the list of rigid complements—i.e., chunks have been added to or removed from the view. When there are more rigid complements than substrings of the view, the lens simply discards the extra complements. When there are more substrings than rigid complements, it processes the extra substrings using $l.create$. This is the reason that *create* takes a resource as an argument—the resource often has entries for the extra chunks (especially if the Kleene star lens appears embedded in an instance of the lower combinator, which pre-aligns the resource against the updated view before it invokes *put*).

Union The final regular operator is union:

$$\frac{l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1 \qquad \lfloor S_1 \rfloor \cap \lfloor S_2 \rfloor = \emptyset \qquad l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2 \qquad \lfloor V_1 \rfloor \cap \lfloor V_2 \rfloor \subseteq \lfloor V_1 \cap V_2 \rfloor}{l_1 \,|\, l_2 \in (S_1 \cup S_2) \overset{(C_1+C_2),k}{\Longleftrightarrow} (V_1 \cup V_2)}$$

$$\begin{aligned}
&get\ s &&= \begin{cases} l_1.get\ s \text{ if } s \in \lfloor S_1 \rfloor \\ l_2.get\ s \text{ if } s \in \lfloor S_2 \rfloor \end{cases} \\
&res\ s &&= \begin{cases} Inl(c_1), r_1 \text{ if } s \in \lfloor S_1 \rfloor \\ Inr(c_2), r_2 \text{ if } s \in \lfloor S_2 \rfloor \end{cases} \\
&\quad \text{where } c_1, r_1 = l_1.res\ s_1 \\
&\qquad \text{and } c_2, r_2 = l_2.res\ s_2 \\
&put\ v\ (c, r) &&= \begin{cases} l_1.put\ v\ (c_1, r) \text{ if } v \in \lfloor V_1 \rfloor \wedge c = Inl(c_1) \\ l_2.put\ v\ (c_2, r) \text{ if } v \in \lfloor V_2 \rfloor \wedge c = Inr(c_2) \\ l_1.create\ v\ r \quad \text{if } v \notin \lfloor V_2 \rfloor \wedge c = Inr(c_2) \\ l_2.create\ v\ r \quad \text{if } v \notin \lfloor V_1 \rfloor \wedge c = Inl(c_1) \end{cases} \\
&create\ v\ r &&= \begin{cases} l_1.create\ v\ r \text{ if } v \in \lfloor V_1 \rfloor \\ l_2.create\ v\ r \text{ if } v \notin \lfloor V_1 \rfloor \end{cases}
\end{aligned}$$

The union lens behaves like a bidirectional conditional operator. The *get* function selects $l_1.get$ or $l_2.get$ by testing whether the source string belongs to $\lfloor S_1 \rfloor$ or $\lfloor S_2 \rfloor$. The typing rule requires that these types be disjoint, so this choice is deterministic.

The *res* function also selects $l_1.res$ or $l_2.res$ by testing the source string. It places the resulting rigid complement in a tagged sum, producing $Inl(c)$ if the source belongs to $\lfloor S_1 \rfloor$ and $Inr(c)$ if it belongs to $\lfloor S_2 \rfloor$. It does not tag the resource—because l_1 and l_2 are defined over the same basic lens k for chunks, we can safely pass a resource computed by $l_1.res$ to $l_2.put$ and vice versa.

The *put* function is slightly more complicated, because the typing rule allows the view types to overlap. It tries to select one of $l_1.put$ or $l_2.put$ using the view and uses the rigid complement disambiguate cases where the view belongs to both $\lfloor V_1 \rfloor$ and $\lfloor V_2 \rfloor$. The *create* function is similar. Note that because *put* is a total function, it needs to handle cases where the view belongs to $(\lfloor V_1 \rfloor \setminus \lfloor V_2 \rfloor)$ but the complement is of the form $Inr(c)$. To satisfy the PUTGET law, it must invoke one of l_1's component functions, but it cannot invoke $l_1.put$ because the rigid complement c does not necessarily belong to C_1. It discards c and uses $l_1.create$ instead.

The side condition $(\lfloor V_1 \rfloor \cap \lfloor V_2 \rfloor) \subseteq \lfloor V_1 \cap V_2 \rfloor$ in the typing rule for union ensures that $(V_1 \cup V_2)$ is chunk unambiguous—i.e., that strings in the intersection $(V_1 \cap V_2)$ have unique parses. It rules out languages of chunk-annotated strings such as $\{$"$\langle a \rangle b$", "$a \langle b \rangle$"$\}$.

Composition The composition operator puts two matching lenses in sequence:

$$\frac{l_1 \in S \overset{C_1,k_1}{\Longleftrightarrow} U \qquad l_2 \in U \overset{C_2,k_2}{\Longleftrightarrow} V}{l_1;l_2 \in S \overset{(C_1 \otimes C_2),(k_1;k_2)}{\Longleftrightarrow} V}$$

$$\begin{array}{ll}
\mathit{get}\ s & = l_2.\mathit{get}\ (l_1.\mathit{get}\ s) \\
\mathit{res}\ s & = \langle c_1, c_2 \rangle, \mathit{zip}\ r_1\ r_2 \\
 & \quad \text{where } c_1, r_1 = l_1.\mathit{res}\ s \\
 & \quad \text{and } c_2, r_2 = l_2.\mathit{res}\ (l_1.\mathit{get}\ s) \\
\mathit{put}\ v\ (\langle c_1, c_2 \rangle, r) & = l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (c_2, r_2))\ (c_1, r_1) \\
 & \quad \text{where } r_1, r_2 = \mathit{unzip}\ r \\
\mathit{create}\ v\ r & = l_1.\mathit{create}\ (l_2.\mathit{create}\ v\ r_2)\ r_1 \\
 & \quad \text{where } r_1, r_2 = \mathit{unzip}\ r
\end{array}$$

This operator is especially interesting as a matching lens because it handles alignment in two sequential phases of computation. Composition provides strong evidence that our design for matching lenses is robust. Unlike the composition operator defined in our previous work on dictionary lenses, whose behavior was often unpredictable, the constraints imposed by the matching lens laws lead naturally to a definition of an operator whose behavior is intuitive.

The *get* function applies $l_1.\mathit{get}$ and $l_2.\mathit{get}$ in sequence. The *res* function applies $l_1.\mathit{res}$ to the source s, yielding a rigid complement c_1 and resource r_1, and $l_2.\mathit{res}$ to $l_1.\mathit{get}\ s$, yielding c_2 and r_2. It merges the rigid complements into a pair $\langle c_1, c_2 \rangle$ and combines the resources by zipping them together, where the *zip* function takes a C_1-resource and a C_2-resource to a $C_1 \otimes C_2$-resource as follows:[6]

$$(\mathit{zip}\ r_1\ r_2)(m) = \begin{cases} \langle r_1(m), r_2(m) \rangle & \text{if } m \in \mathrm{dom}(r_1) \cap \mathrm{dom}(r_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that we have the following equalities

$$\begin{array}{rcll}
\mathrm{dom}(r_1) & = & \mathit{locations}(s) & \text{by RESCHUNKS for } l_1 \\
 & = & \mathit{locations}(l.\mathit{get}\ s) & \text{by GETCHUNKS for } l_1 \\
 & = & \mathrm{dom}(r_2) & \text{by RESCHUNKS for } l_2
\end{array}$$

so $\mathit{zip}\ r_1\ r_2$ is defined on all locations in $\mathrm{dom}(r_1)$ and $\mathrm{dom}(r_2)$.

The *put* function unzips the resource and applies $l_2.\mathit{put}$ and $l_1.\mathit{put}$ in that order. The unzip function is defined by

$$(\pi_i(\mathit{unzip}\ r))(m) = \begin{cases} c_i & \text{if } r(m) = \langle c_1, c_2 \rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $i \in \{1, 2\}$. Because the zipped resource represents the resources generated by l_1 and l_2 together, rearranging the resource has the effect of pre-aligning the resources for both phases of computation. To illustrate, consider the following example:

```
let k1 : lens =
  copy [A-Z]. copy [a-z] . [0-9]<->""
let k2 : lens = [A-Z]<->"" . key (copy [a-z])
let l : lens =
  <best:k1> . (copy "," . <best:k1>)* ;
  <best:k2> . (copy "," . <best:k2>)*
```

The *get* function takes a non-empty list of comma-separated chunks containing an upper-case letter, a lower-case letter, and a number, and deletes the number in the first phase and the upper-case letter in the second phase:

```
l.get "Aa1,Bb2,Cc3" = "a,b,c"
```

The resource produced by *res* represents the upper-case letter and number together, so even though the alignment is only calculated against the final view, the effect after applying the alignment to the resource is that the *put* function restores information from both sequential phases to the appropriate chunk:

```
l.put "b,a" into "Aa1,Bb2,Cc3" = "Bb2,Aa1"
```

The typing rule for the composition lens requires that the view type of l_1 be identical to the source type of l_2. In particular, it requires that the chunks in these types must be identical. Intuitively, this makes sense—the only way that the *put* function can reasonably translate alignments on the view back through both phases of computation to the source is if the chunks in the types of each lens agree. However, in some situations, it is useful to compose lenses that have identical erased types but different notions of chunks—e.g., one lens does not have any chunks, while the other lens does have chunks. To do this "asymmetric" form of composition, we can convert both lenses to basic lenses using $\lfloor \cdot \rfloor$, which forgets the chunks in the source and view, and compose them as basic lenses.

5. Alignments

So far, our discussion has focused on the core mechanisms of matching lenses—extending basic lenses with chunks and developing an interface for supplying lenses with explicit alignment directives. But we have not said where these alignment directives come from! In this section, we describe the primitives for specifying alignments implemented in our extension of the Boomerang language. We describe three alignment "species" and show how alignments can be tuned using "keys" and "thresholds". Because alignment is a fundamentally heuristic operation, the choice of an alignment function depends intimately on the details of the application at hand. One of the main strengths of the matching lens framework is its flexibility. Matching lenses can be instantiated with arbitrary alignment functions since well-behavedness does not hinge on any special properties of the function used to align chunks: the only property we require is that it returns the identity alignment when its arguments are identical. Thus, the functions described in this section are not exhaustive; it would be easy to add new primitives as needed.

Species Boomerang currently supports three different alignment "species", depicted graphically in Figure 1 (a-c):

- **Positional:** The alignment matches chunks by position. If one list contains more chunks, the extras at the end of the longer list are not matched with any chunk in the other list.
- **Best match:** The alignment minimizes the sum of the total edit distances between matched chunks and the lengths of unmatched chunks.
- **Best non-crossing match:** The alignment minimizes the same heuristic as in best match, but only considers alignments with "non-crossing" edges. This heuristic can be computed efficiently using a variant of the standard algorithm for computing longest common subsequence.

For example:

```
let l : lens = key [A-Z] . [0-9]<->""
<pos:l>*.put  "BCA" into "A1B2C3" = "B1C2A3"
<best:l>*.put "BCA" into "A1B2C3" = "B2C3A1"
<nonx:l>*.put "BCA" into "A1B2C3" = "B2C3A0"
```

When we convert a matching lens to a basic lens using the lower coercion, $\lfloor \cdot \rfloor$, the *align* function is instantiated using the species indicated in the match combinator. The Boomerang system checks that the same annotation is used on every instance of the match combinator—e.g., it disallows `(<pos:l> . <nonx:l>)`, which uses two different species.

[6] The angle brackets and type operator $\otimes$ distinguish these pairs from the ordinary pairs generated as rigid complements for the concatenation lens.

Keys Typically we only want to consider a part of each chunk when we compute an alignment. Boomerang includes two primitives, `key` and `nokey`, that provide a way for programmers to control the portion of each chunk that is used to compute an alignment. These combinators take a basic lens as an argument but they do not change the *get*/*put* behavior of the lens they enclose. Instead, they add extra annotations to the view type that we use to "read off" the key for chunks (just as we use annotations on regular expressions to "read off" the locations of chunks). When the *align* function computes an alignment for two lists of chunks, it first uses the view type to extract the regions of each chunk marked as keys and then computes an alignment. To illustrate the use of keys, consider a simple example:

```
let k : lens =
  copy [A-Z] . copy [a-z] . [0-9]<->""
let l : lens = <best:k>*
l.put "CcBbAa" into "Aa1Bb2Cc3" = "Cc1Bb2Aa3"
```

This program uses the `best` species, but behaves positionally because the view type does not contain any key annotations—i.e., the key of every chunk is the empty string. By adding a key annotation we obtain a lens whose *put* function matches up chunks using the upper-case letters in the view:

```
let k : lens =
  key(copy [A-Z]) . copy [a-z] . [0-9]<->""
let l : lens = <best:k>*
l.put "CcBbAa" into "Aa1Bb2Cc3" = "Cc3Bb2Aa1"
```

Note that lower-case letters, which are not marked as a part of the key, do not affect the alignment:

```
l.put "CaBbAc" into "Aa1Bb2Cc3" = "Ca3Bb2Ac1"
```

The `nokey` primitive is dual to `key`—it removes the key annotation on the view type. We can write an equivalent version of the previous lens using `nokey`:

```
let k : lens =
  key(copy [A-Z] . nokey(copy [a-z]) . [0-9]<->"")
```

These simple mechanisms for indicating keys suffice for many practical examples, but they could be extended in several ways. For example, we could provide programmers with ways to generate unique keys or build keys structured as tuples or records (rather than flattening the portion of each chunk marked as a key into a string). We plan to explore these ideas in future work.

Thresholds The `best` and `nonx` species compute alignments by minimizing the sum of the total edit distances between matched chunks and the lengths of unmatched chunks. In some applications, it is important to *not* match up chunks that are "too different," even if aligning those chunks would produce a minimal cost alignment. For instance, in the following example, where keys are three characters long

```
let k : lens = key [A-Z]{3} . [0-9]<->""
let l : lens = <best:k> . (copy ";" . <best:k>)*
l.put "DBD;CCC;AAA" into
      "AAA1;BBB2;CCC3" =
      "DBD2;CCC3;AAA1"
```

we might prefer to not align the `DBD` and `BBB2` chunks with each other. The `best` species does align them because the cost of a two-character edit is less than the six-character edit of deleting `BBB` from the view and adding `DBD`. To obtain the desired behavior, we can add a threshold annotation:

```
let l : lens =
  <best 50:k> . (copy ";" . <best 50:k>)*
l.put "DBD;CCC;AAA" into
      "AAA1;BBB2;CCC3" =
      "DBD0;CCC3;AAA1"
```

The `best` species takes an optional integer n as an argument. When supplied with such an integer, it minimizes the total edit distances between aligned chunks, but it only aligns chunks whose longest common subsequence is at least $n\%$ of the lengths of their keys. (The strict key-based alignment used in dictionary lenses can be simulated using `best 100`.) The revised version of the `l` lens does not align `DBD` with `BBB2` because the longest common subsequence computed from their keys does not meet the threshold. The `nonx` species also supports thresholds. We often use `nonx` with a threshold to align chunks containing totally unstructured text.

6. Extensions

To streamline the discussion, our presentation of matching lenses in the preceding sections has made three important assumptions: (1) chunks only appear at the top level, (2) the same basic lens processes every chunk, and (3) the lens does not reorder chunks in going from source to view. Of course, in many applications, it is important to be able to nest chunks, to use different basic lenses to process chunks, and to reorder chunks. This section describes how we can extend the matching lens framework to accommodate these features. Each of these extensions is implemented in Boomerang.

6.1 Nested Chunks

To handle sources with reorderable information at several different levels, it is often useful to nest chunks inside each other. For example, suppose that we want to extend our Wiki lens to handle several levels of nested structure: sections, subsections, and paragraphs. So the *get* function will map the source

```
=Grand Tours=
The grand tours are major cycling races...
==Giro d'Italia==
The Giro is usually held in May and June...
=Classics=
The classics are one-day cycling races...
==Milan-San Remo==
The Spring classic is held in March...
```

to a view that contains just section and subsection headings:

```
Grand Tours
  Giro d'Italia
Classics
  Milan-San Remo
```

If we modify the view by reordering sections and adding new subsections,

```
Classics
  Milan-San Remo
  Paris-Roubaix
Grand Tours
  Giro d'Italia
  Tour de France
```

we would like paragraphs to be restored to the appropriate section or subsection.

We can build a matching lens that has chunks at multiple levels of structure using the `lower` combinator, which converts a matching lens to a basic lens:

```
let subsection : lens =
  "=="<->"  " .
  key (copy HEADING) .
  ("==\n" . PARAGRAPHS)<->"\n"
```

```
let section : lens =
  "="<->"" .
  key (copy HEADING) .
  ("=\n" . PARAGRAPHS)<->"\n" .
  lower <best:subsection>*
let wiki : lens = <best:section>*
```

The `subsection` lens inserts two characters of indentation, copies the heading, and deletes any paragraphs that follow. The `section` lens copies the heading, deletes the paragraphs that follow, and then uses `lower` to convert the matching lens that processes the list of subsection chunks into a basic lens. The top-level `wiki` lens uses the `section` lens to process a list of section chunks. If we *put* back the updated view into the original source, we get an updated source where paragraphs are restored appropriately:

```
=Classics=
The classics are one-day cycling races...
==Milan-San Remo==
The Spring classic is held in March...
==Paris-Roubaix==
=Grand Tours=
The grand tours are major cycling races...
==Giro d'Italia==
The Giro is usually held in May and June...
==Tour de France==
```

The main thing to notice about this program is that we can use `lower` to build matching lenses that process nested chunks. Lenses built in this way align chunks in strict nested fashion—e.g., in this example, the top-level `wiki` lens aligns the section chunks and then aligns the nested chunks for subsections within each section.

6.2 Multiple Lenses

We can also build matching lenses that use different basic lenses to process chunks. Returning to our running example, suppose that we wanted a version of the `wiki` lens in which subsections and sections are aligned separately. Why would we want this? Observe that the lens described in the previous section never aligns subsections that appear in different sections. This means that if we move a subsection from one section to another

```
Classics
Grand Tours
  Giro d'Italia
  Milan-San Remo
```

the paragraph under that subsection will be lost when we *put* the result back into the original source

```
=Classics=
The classics are one-day cycling races...
=Grand Tours=
The grand tours are major cycling races...
==Giro d'Italia==
The Giro is usually held in May and June...
==Milan-San Remo==
```

because the alignment strictly follows the nesting structure of the document.

We can build a lens that aligns section and subsections separately by using two different kinds of chunks, as in the following program, written using "tags":

```
let section : lens =
  "="<->"" .
  key (copy HEADING) .
  ("=\n" . PARAGRAPHS)<->"\n"
let wiki : lens =
    ( < tag "section" best : section > .
      < tag "subsection" best : subsection >* )*
```

This version of the `wiki` lens has two chunks at the top level—one for sections and another for subsections. The `tag` primitives assigns a distinct name to each kind of chunk, where each tag is associated with a different basic lens. On the same inputs as above, the *put* function of this lens produces a new source

```
=Classics=
The classics are one-day cycling races...
=Grand Tours=
The grand tours are major cycling races...
==Giro d'Italia==
The Giro is usually held in May and June...
==Milan-San Remo==
The Spring classic is held in March...
```

where the paragraph under the Milan–San Remo subsection is restored from the source. To extend matching lenses with tags we simply generalize each of our structures with an extra level of indirection—e.g., we change the type of resources from finite maps from locations to complements to finite maps from tags to locations to complements. When we align chunks, we compute a separate alignment for each tag.

6.3 Reordering Chunks

Some applications require matching lenses that reorder chunks in going from source to view. The swap operator ($l_1 \sim l_2$) is similar to concatenation, but inverts the order of the strings in the view. Adding swap as a primitive breaks the procedure for using a matching lens implemented by the $\lfloor\cdot\rfloor$ coercion described in Section 3 where we pre-align the resource using a correspondence computed between the old and new view. It also causes problems with the sequential composition operator—in general, the lenses being composed may reorder the source chunks differently, so it does not make sense to simply zip the resources generated by each lens together and align the result against the view.

To recover the behavior we want, we need to extend matching lenses with another function that keeps track of the permutation on chunks computed by the lens:

$$l.\mathit{perm} \in \Pi\, s : \lfloor S \rfloor.\ \mathsf{Perms}(\mathit{locations}(s))$$

It is straightforward to add *perm* to each of the lenses we have seen so far—e.g., the lift primitive returns the empty permutation, match returns the identity permutation on its single chunk, the concatenation operator merges the permutations returned by its sublenses in the obvious way, and so on. We also need the ChunkPut and NoChunkPut laws to use *perm*—the old versions are no longer valid for lenses that reorder chunks:

$$\frac{n \in (\mathit{locations}(v) \cap \mathrm{dom}(r)) \quad (l.\mathit{perm}\,(l.\mathit{put}\,v\,(c,r)))(m) = n}{(l.\mathit{put}\,v\,(c,r))[m] = k.\mathit{put}\,v[n]\,(r(n))} \quad \text{(ChunkPut)}$$

$$\frac{n \in (\mathit{locations}(v) \setminus \mathrm{dom}(r)) \quad (l.\mathit{perm}\,(l.\mathit{put}\,v\,(c,r)))(m) = n}{(l.\mathit{put}\,v\,(c,r))[m] = k.\mathit{create}\,v[n]} \quad \text{(NoChunkPut)}$$

These laws generalize the laws given in Section 3. The ChunkPut law stipulates that the mth chunk in the source produced by *put* must be identical to the structure produced by applying $k.\mathit{put}$ to the nth chunk in the view and the element $r(n)$ in the resource, where the permutation computed by the *perm* function on the source maps m to n. The other laws generalize similarly.

Composition Using *perm*, we can refine sequential composition operator to use the permutation on chunks computed in each phase:

$$\frac{l_1 \in S \xleftrightarrow{C_1,k_1} U \qquad l_2 \in U \xleftrightarrow{C_2,k_2} V}{(l_1;l_2) \in S \xleftrightarrow{(C_1 \otimes C_2),(k_1;k_2)} V}$$

$$\begin{array}{ll}
\mathit{get}\ s & = l_2.\mathit{get}\ (l_1.\mathit{get}\ s) \\
\mathit{res}\ s & = \langle c_1, c_2\rangle, \mathit{zip}\ (r_1 \circ p_2^{-1})\ r_2 \\
 & \quad \text{where } c_1, r_1 = l_1.\mathit{res}\ s \\
 & \quad\quad \text{and } c_2, r_2 = l_2.\mathit{res}\ (l_1.\mathit{get}\ s) \\
 & \quad\quad \text{and } p_2 = l_2.\mathit{perm}\ (l_1.\mathit{get}\ s) \\
\mathit{perm}\ s & = (l_2.\mathit{perm}\ (l_1.\mathit{get}\ s)) \circ (l_1.\mathit{perm}\ s) \\
\mathit{put}\ v\ (\langle c_1, c_2\rangle, r) & = l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (c_2, r_2))\ (c_1, r_1 \circ p_2) \\
 & \quad \text{where } r_1, r_2 = \mathit{unzip}\ r \\
 & \quad\quad \text{and } p_2 = l_2.\mathit{perm}\ (l_2.\mathit{put}\ v\ (c_2, r_2)) \\
\mathit{create}\ v\ r & = l_1.\mathit{create}\ (l_2.\mathit{create}\ v\ r_2)\ (r_1 \circ p_2) \\
 & \quad \text{where } r_1, r_2 = \mathit{unzip}\ r \\
 & \quad\quad \text{and } p_2 = l_2.\mathit{perm}\ (l_2.\mathit{create}\ v\ r_2)
\end{array}$$

The *res* function applies the inverse of the permutation computed by l_2 on the intermediate view to the resource computed by l_1, which puts it into the "view order" of l_2. Likewise, the *put* function puts the r_1 resource back into the view order of l_1.

Swap The swap lens is defined as follows:

$$\frac{\begin{array}{ll} l_1 \in S_1 \xleftrightarrow{C_1,k} V_1 & \lfloor S_1 \rfloor \cdot^! \lfloor S_2 \rfloor \\ l_2 \in S_2 \xleftrightarrow{C_2,k} V_2 & \lfloor V_2 \rfloor \cdot^! \lfloor V_1 \rfloor \end{array}}{l_1 \sim l_2 \in (S_1 \cdot S_2) \xleftrightarrow{(C_2 \times C_1),k} (V_2 \cdot V_1)}$$

$$\begin{array}{ll}
\mathit{get}\ (s_1 \cdot s_2) & = (l_2.\mathit{get}\ s_2) \cdot (l_1.\mathit{get}\ s_1) \\
\mathit{res}\ (s_1 \cdot s_2) & = (c_2, c_1), (r_2 +\!\!+ r_1) \\
 & \quad \text{where } c_1, r_1 = l_1.\mathit{res}\ s_1 \\
 & \quad\quad \text{and } c_2, r_2 = l_2.\mathit{res}\ s_2 \\
\mathit{perm}\ (s_1 \cdot s_2) & = (l_2.\mathit{perm}\ s_2) \ast\ast (l_1.\mathit{perm}\ s_1) \\
\mathit{put}\ (v_2 \cdot v_1)\ (c, r) & = (l_1.\mathit{put}\ v_1\ (c_1, r_1)) \cdot (l_2.\mathit{put}\ v_2\ (c_2, r_2)) \\
 & \quad \text{where } c_2, c_1 = c \\
 & \quad\quad \text{and } r_2, r_1 = \mathit{split}(|v_2|, r) \\
\mathit{create}\ (v_2 \cdot v_1)\ r & = (l_1.\mathit{create}\ v_1\ r_1) \cdot (l_2.\mathit{create}\ v_2\ r_2) \\
 & \quad \text{where } r_2, r_1 = \mathit{split}(|v_2|, r)
\end{array}$$

Like the concatenation lens, the *get* component of swap splits the source string in two and applies $l_1.\mathit{get}$ and $l_2.\mathit{get}$ to the resulting substrings. However, before it concatenates the results, it swaps their order. The *res*, *put*, and *create* functions are similar. The *perm* component of swap combines permutations using the $(\ast\ast)$ operator

$$(q_2 \ast\ast q_1)(m) = \begin{cases} q_1(m) + |q_2| & \text{if } m < |q_1| \\ q_2(m - |q_1|) & \text{otherwise} \end{cases}$$

which behaves like the $(+\!+)$ operator for resources.

7. Implementation

To test the expressiveness and usability of our framework, we have extended the Boomerang implementation with the string matching lenses discussed in Section 4, with all the alignment strategies described in Section 5 and the extensions in Section 6. Typechecking is decidable—indeed, it can be made quite efficient using standard regular-expression algorithms over an extended alphabet $\Sigma \cup \{\text{`}\langle\text{'}, \text{`}\rangle\text{'}\}$, where the extra characters are used for checking side conditions involving chunk annotations. Alignment strategies are implemented using a straightforward algorithm for the positional strategy, a diff-like least-common-subsequence algorithm for the non-crossing best matching, and a version of the Hungarian algorithm for the best-match strategy. We have developed several small applications, including lenses for structured documents, Wikis, and literate Coq sources.

8. Related Work

This paper extends our previous work on lenses [3, 4, 9, 11, 12] with new mechanisms for specifying and using alignments. The original paper on lenses [9] includes an extensive survey of relevant threads from the database and programming languages literature. We focus here on the most closely related work.

Matching lenses grew out of the dictionary lenses we proposed previously [3], but they differ in several important ways. First, dictionary lenses are based on a single alignment mechanism—"by keys"—whereas matching lenses provide a generic framework for using alignments in lenses that can be instantiated with arbitrary functions. Second, the semantic laws that govern the behavior of dictionary lenses express much weaker constraints than the matching lens laws, which specify the handling of chunks directly and in detail. Specifically, dictionary lenses obey an EQUIVPUT law that forces the *put* function to be "oblivious" to certain features of sources characterized by an equivalence relation $\sim$. By picking $\sim$ to be an equivalence that relates strings differing only in the relative order of chunks with different keys we get some constraints on *put*—e.g., it forbids lenses that operate positionally—but these constraints are weaker than the conditions stated in the matching lens laws. For example, Lemma 3.1 does not hold for dictionary lenses because the type system does not explicitly keep track of chunks.

Much of the previous work on view update assumes that the user will modify the view using special operations in some "update language", and, often, these update operations can be used to infer an intended alignment. For example, in Meertens's work on constraint maintainers for user interfaces [19] users manipulate lists using "small updates" for which it is easy to maintain the correspondence between source and view items. Similarly, the bidirectional languages X and Inv [14, 20] assume that edit operations are applied to the data to yield annotated values that indicate whether a value was newly created or deleted. Their languages handle single insertions and deletions but not general reorderings.

Many relational view update translators use schemas to guide the selection of a source update. For example Keller identifies criteria for view update translators requiring that the key of each source item appears in the view [16]. Matching lenses also use a notion of keys for alignment, but they permit the correspondence between chunks to be computed using arbitrary heuristics.

Alignment issues also come up in software model transformations. Some systems offer "traceability links" that can be used for alignment [5, 22].

9. Conclusions and Future Work

Matching lenses provide a general solution to the problems that come up when updatable views are defined over ordered structures. Decoupling the handling of rigidly ordered and reorderable information yields a flexible framework that can be instantiated with arbitrary heuristics for alignment.

Our work can be extended in several directions. We are interested in exploring other axiomatizations of matching lenses. One idea, originally suggested by Alexandre Pilkiewicz, is to replace the current laws with laws stated in terms of a lens on skeletons and a basic lens mapped on the list of chunks. This would provide a more elegant description of the semantics of matching lenses. However, we believe it would make it more complicated to verify operators such as concatenation. We are interested in instantiating the framework of matching lenses in other settings besides strings and exploring implementation issues, including algebraic optimization and lenses for streaming data.

Acknowledgments We wish to thank Zack Ives, Alexandre Pilkiewicz, Val Tannen, Philip Wadler, Steve Zdancewic, and the anonymous ICFP reviewers for helpful comments. Our work is supported by the National Science Foundation under grants IIS-0534592 *Linguistic Foundations for XML View Update*, and CT-0716469 *Manifest Security*.

References

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

[2] Jean Berstel, Dominique Perrin, and Christophe Reutenauer. *Codes and Automata*. Cambridge University Press, 2009.

[3] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, CA*, pages 407–419, January 2008.

[4] Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS), Chicago, TL*, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

[5] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. GRACE meeting notes, state of the art, and outlook. In *International Conference on Model Transformations (ICMT), Zurich, Switzerland*, pages 260–283, June 2009. Invited paper.

[6] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, September 1982.

[7] Zinovy Diskin. Algebraic models for bidirectional model synchronization. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS), Toulouse, France*, pages 21–36, September 2008.

[8] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 73(4), June 2007. Short version in DBPL '05.

[9] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007.

[10] J. Nathan Foster and Benjamin C. Pierce. *Boomerang Programmer's Manual*, 2009. Available from `http://www.seas.upenn.edu/~harmony/`.

[11] J. Nathan Foster, Benjamin C. Pierce, and Steve Zdancewic. Updatable security views. In *IEEE Computer Security Foundations Symposium (CSF), Port Jefferson, NY*, pages 60–74, July 2009.

[12] J. Nathan Foster, Alexandre Pilkiewcz, and Benjamin C. Pierce. Quotient lenses. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, BC*, pages 383–395, September 2008.

[13] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems (TODS)*, 13(4):486–524, 1988.

[14] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1–2), June 2008.

[15] C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In *Proceedings of the European Symposium on Programming (ESOP), London, UK*, pages 302–316, 1994.

[16] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of Fourth Annual ACM Symposium on Principles of Database Systems (PODS)*, pages 154–163, march 1985. Portland, Oregon.

[17] David Lutterkort. Augeas–A configuration API. In *Linux Symposium, Ottawa, ON*, pages 47–56, 2008.

[18] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Freiburg, Germany*, pages 47–58, 2007.

[19] Lambert Meertens. Designing constraint maintainers for user interaction, 1998. Manuscript, available from `ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps`.

[20] 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.

[21] Hugo Pacheco and Alcino Cunha. Generic point-free lenses. In *International Conference on Mathematics of Program Construction (MPC), Québec City, QC*, 2010. To appear.

[22] 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.

[23] Janis Voigtländer. Bidirectionalization for free! In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Savannah, GA*, pages 165–176, January 2009.

[24] Meng Wang, Jeremy Gibbons, Kazutaka Matsuda, and Zhenjiang Hu. Gradual refinement: Blending pattern matching with data abstraction. In *International Conference on Mathematics of Program Construction (MPC), Québec City, QC*, 2010. To appear.

[25] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), Amsterdam, Netherlands*, pages 315–324, 2009.