Matching Lenses: Alignment and View Update

Davi M. J. Barbosa
École Polytechnique

Julien Cretin
École Polytechnique

Nate Foster
Princeton University

Michael Greenberg
University of Pennsylvania

Benjamin C. Pierce
University of Pennsylvania

Technical Report MS-CIS-10-01
Department of Computer and Information Science
University of Pennsylvania

January 8, 2010

Abstract

Bidirectional programming languages have been proposed as a practical approach to the *view update problem*. Programs in these languages, often called *lenses*, can be read in two ways—from left to right as functions mapping sources to views, and from right to left as functions mapping updated views back to updated sources. Lenses address the view update problem by making it possible to define a view and its associated update policy together.

One 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 edited view with corresponding pieces of the underlying source. Unfortunately, existing bidirectional languages are extremely limited in their treatment of alignment—they only support simple strategies that do not suffice for many examples of practical interest.

In this paper, we propose a novel framework of *matching lenses* that extends basic lenses with new mechanisms for calculating and using alignments. We enrich the types of lenses with "chunks" that identify the locations of data that should be re-aligned after updates, and we formulate refined behavioral laws that capture essential constraints on the handling of chunks. To demonstrate the utility of our approach, we develop a core language of matching lenses for string data, and we extend it with primitives for describing a number of useful alignment heuristics.

1. Introduction

View update is a classic problem in data management. Given a view and an update to the view, the problem concerns finding an update to the source that reflects the modification made to the view. Recently, we and others have made new progress on this old problem by adopting a linguistic approach—developing programming languages in which every expression denotes both a function mapping sources to views as well as one mapping updated views back to updated sources. These *bidirectional programming languages* provide an effective way for programmers to address instances of the view update problem by giving them a tool for defining a view and its associated update policy together.

A number of languages for describing bidirectional transformations, often called *lenses*, have been proposed [3, 4, 5, 8, 9, 12, 15, 16, 18, 19, 20, 21, 22, 23]. Formally, a *lens* l mapping between sets S of "sources", C of "complements", and V of "views" comprises four total functions

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

that obey the following "round-tripping" laws for every source s in S, complement c in C, and view v in V:

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

$$l.get\ (l.create\ v) = v \qquad \text{(CREATEGET)}$$

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

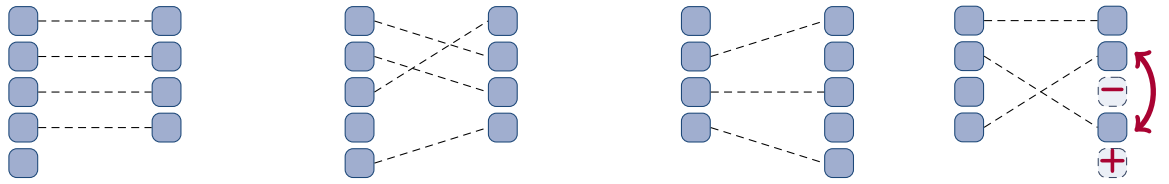
We write $S \overset{C}{\Longleftrightarrow} V$ for the set of all lenses between S, C, and V and we refer to them as *basic lenses* to distinguish them from the other kinds of lenses described in this paper. [1]

The *get* function computes a view from a source. The *res* (for "residue") function computes a complement—a structure that records (at least) the information from the source that is not reflected in the view. The other two functions handle updates: *put* takes an updated view and a complement and weaves them together to produce an updated source, while *create* handles the case where we need to propagate an update to a view but have no complement available. It builds a new source from the view directly, filling in any missing information with defaults. The round-tripping laws are closely related to conditions on view update translators that have been explored in the literature [1, 7, 13]. The PUTGET and CREATEGET laws require that updates to the view be propagated exactly to the source—i.e., given a modified view and a complement, the updated source produced by *put* must map back via *get* to the very same view, and similarly for *create*. The GETPUT law requires that *put* return the original source unmodified whenever the update to the view is a no-op.

In previous work [3], we identified *alignment* as a serious problem that comes up when lenses are used with ordered data. In general, the *get* component of a lens may hide some of the information in the source (formally, it may be a non-injective function), so the *put* function needs to reintegrate the information in the complement with the modified information in the view. When the source and view are ordered structures, and when the update to the view breaks the correspondence between pieces of the source and pieces of the view, doing this recombination correctly requires realigning the pieces of the source to reflect the the new correspondence with the pieces of the updated view. However, most proposals for lenses only have limited capabilities with respect to alignment—they work by position—so they are essentially useless for manipulating ordered structures.

To address this problem, we proposed extending lenses with a simple alignment mechanism based on keys. In a *dictionary lens*, the programmer identifies the reorderable *chunks* of the source and specifies how to compute a *key* for each chunk. The *put* function uses keys instead of positions to locate a source chunk for each piece of the view. Dictionary lenses work well in situations where chunks have stable keys, but they are not a general solution. If the

[1] Readers familiar with previous presentations of basic lenses will notice that we are providing *put* with an explicit complement structure instead of the original source. This is a minor change—we can recover the original definition by simply taking the set of complements to be S, the set of sources, and defining *res* to be the identity function on S.



(a) positional (b) minimizing edit distance (c) non-crossing (d) operation-based

Figure 1. Alignment strategies

source chunks do not have a natural key (e.g., because the chunks are blocks of otherwise unstructured text) or if keys may be edited, the simple alignment strategy that is "baked in" to dictionary lenses does not suffice. Moreover, the behavioral laws that dictionary lenses obey are rather weak. They fail to capture fundamental constraints on the handling of chunks, and they allow lenses with unintuitive behavior—e.g., lenses that do not translate reorderings on chunks in the view to reorderings on source chunks.

Our goal in this paper is to develop a completely generic alignment mechanism for lenses that overcomes these limitations and addresses the issue of ordered data once and for all. To this end, we propose a novel framework of *matching lenses* that decouples the handling of rigidly ordered and reorderable information from each other. This explicit separation of concerns provides a clean interface for supplying a lens with clear directives about how the source and view should be aligned, and it yields an extremely flexible framework that can be instantiated with arbitrary functions for computing alignments. Figure 1 depicts several common alignment strategies: (a) positional, (b) minimizing the total edit distance between matched chunks, (c) minimizing total edit distance but only considering "non-crossing" alignments (i.e., similar to longest common sequence), and (d) extracting an alignment from the actual update operation applied to the view. The matching lens framework accommodates each of these.

Operationally, we decouple the handling of rigidly ordered and reorderable information by splitting the representation of complements into two pieces: a *rigid complement* that records the source information that should be handled positionally and a *resource* that stores the information extracted from the reorderable chunks in the source. The resource provides a means of supplying a lens with explicit alignment information since rearranging its elements according to a given correspondence yields a pre-aligned structure in which every element matches up with the specified chunk in the view.

At the level of semantics, we enrich the types of matching lenses with explicit notions of what constitutes a reorderable "chunk," and we add new behavioral laws that capture essential constraints on the handling of chunks. These laws stipulate that matching lenses must carry chunks in the source through to chunks in the view, and vice versa. They can be used to derive other natural properties—e.g., that matching lenses translate reorderings to reorderings.

Finally, to demonstrate the utility of our framework, we develop a collection of primitives for defining matching lenses over string data. We work with strings, rather than richer structures such as trees or relations, primarily because strings are a simple setting for exploring foundational issues. However, strings are also pervasive in the real world, and we believe that our matching lens primitives will also be pragmatically useful.[2] We define coercions that convert between basic lenses and matching lenses, reinterpret the familiar regular operators (union, concatenation, and Kleene star) as matching lenses, and show that matching lenses are closed under composition. We also present primitives for specifying, combining, and tuning alignment functions in terms of "species", "tags", "keys", and "thresholds."

Our contributions can be summarized as follows:

1. We define a novel semantic space of *matching lenses* that enriches the types of lenses with chunks and adds new behavioral laws ensuring that chunks are handled correctly. Unlike dictionary lenses, which use a single

[2] Indeed, dictionary lenses, which also have strings as their data model, have been used in industry [18].

alignment strategy, matching lenses are a flexible framework that can be instantiated with many different strategies.

2. We define a core language of primitive matching lenses for transforming strings, and we prove (in the appendix) that each primitive is a well behaved matching lens.
3. We describe several alignment strategies we have developed in our implementation, and we show how these strategies can be tuned using notions of "species", "tags", "keys" and, "thresholds."

The next section introduces a simple example that illustrates the problems that arise with ordered data and the essential ingredients of our solution. Section 3 defines the semantic space of matching lenses and develops some of their essential properties. Section 4 defines a core language of matching lens primitives for string data. Section 5 presents primitives for describing and tuning alignment strategies. Section 6 describes extensions to the framework. Section 7 discusses related work. We discuss future work in Section 8.

2. Example

Let's start with a simple example that illustrates the complications that arise when lenses are used to manipulate ordered data. Suppose that the source structure is a string containing the nationality, bib number, team code, finishing time, and name for cyclists in a road race...

```
GBR 213 THR 6h42'45" CAVENDISH Mark
GER 74  CTT 6h42'45" HAUSSLER Heinrich
NOR 75  CTT 6h42'49" HUSHOVD Thor
AUS 182 QST 6h42'49" DAVIS Alan
ITA 177 LPR 6h42'49" PETACCHI Alessandro
```

...and the view is a string containing just the finish time, reformatted name, and expanded team name for each rider:

```
6h42'45" Mark Cavendish (Team Columbia-HTC)
6h42'45" Heinrich Haussler (Cervelo Test Team)
6h42'49" Thor Hushovd (Cervelo Test Team)
6h42'49" Alan Davis (Quick Step)
6h42'49" Alessandro Petacchi (LPR Brakes-Farnese Vini)
```

Figure 2 shows a (basic, non-matching) lens in the Boomerang language [10] that computes the view. The first few lines of the program define regular expressions that describe various pieces of the source—nationalities, bib numbers, team codes, etc. We write `(.)` for concatenation and use standard POSIX notation for union (`|`), repetition (`*`, `+`, `{1,3}`, etc.), and character sets (`[a-z]`, `[0-9]`, etc.).

For the sake of the example, we have written the top-level `riders` lens as the composition of two smaller lenses. In the first phase, the lens `l1` processes each line of the source. It deletes the nationality, copies the bib number, swaps the order of the team and finish time, rewrites the name from "`LAST First`" to "`First LAST`", and removes extra whitespace. The (`copy` E) primitive recognizes a string matching E in the source and copies it to the view and the (E `<->` u) primitive recognizes a string matching E in the source and adds the constant string u to the view. The lenses (`del` E) and (`ins` u) are abbreviations for (E `<->` `""`) and (`""` `<->` u) respectively. The overloaded operators `(.)`, `(|)`, and `(*)` denote union, concatenation, and iteration of lenses. The swap operator, written `(~)`, is like concatenation, but inverts the order of the strings in the view.

In the second phase, the lens `l2` takes a line of text produced by `l1` as output and processes it further. It deletes the bib number, copies the finish time, reformats the name from "`First LAST`" to "`First Last`", and expands the abbreviated team name, wrapping the result in parentheses and placing it before the rider's name. The `riders` lens is the sequential composition of `(lines l1)`, which iterates `l1` over a list of lines, and `(lines l2)`.

Returning to the example, suppose that we update the view by removing Cavendish, swapping the positions of Hushovd and Davis in the final classification, and correcting the spelling of Davis's first name (such adjustments to preliminary results are often made by race officials after consulting photos of the finish):

```
(* regular expressions *)
let L,U,N : regexp * regexp * regexp = [a-z], [A-Z], [0-9]
let NAT : regexp = U{3}
let BIB : regexp = ( N . "  " | N{2} . " " | N{3} )
let TEAM : regexp = U{3}
let TIME : regexp = N+ . "h" . N{2} . "'" . N{2} . "\""
let LN, FN : regexp * regexp = U+, ( U . L* )
let NAME : regexp = LN . " " . FN

(* helpers *)
let parens (l:lens) : lens = ins "(" . l . ins ")"
let lines (l:lens) : lens = (l . copy "\n")*

let nat1 : lens = del ( NAT . " " )
let bib1 : lens = copy BIB . del " "
let team1 : lens = copy TEAM . del " "
let time1 : lens = copy TIME . del " "
let name1 : lens = copy LN ~ ( copy " " ~ copy FN )

let bib2 : lens = del BIB
let time2 : lens = copy TIME
let team2 : lens =
  parens ( ...| "CTT" <-> "Cervelo Test Team" |... )
let name2 : lens =
  copy U . L* . copy " " .
  copy U . ( "A" <-> "a" |...| "Z" <-> "z" )* . ins " "

(* main lens *)
let l1 : lens = nat1 . bib1 . ( team1 ~ time1 ) . name1
let l2 : lens = bib2 . time2 . ( team2 ~ name2 )
let riders : lens = ( lines l1 ; lines l2 )
```

Figure 2. Boomerang source code for the riders lens

```
 6h42'45" Heinrich Haussler (Cervelo Test Team)
 6h42'49" Allan Davis (Quick Step)
 6h42'49" Thor Hushovd (Cervelo Test Team)
 6h42'49" Alessandro Petacchi (LPR Brakes-Farnese Vini)
```

Because the *get* function hides some of the information in the source—i.e., nationalities and bib numbers—the *put* function needs to reintegrate the hidden information from the source with the updated information in the view. However, if it restores the hidden information to lines by position the string it produces as a result will be wrong:

```
GBR 213 CTT 6h42'45" HAUSSLER Heinrich
GER 74  QST 6h42'49" DAVIS Allan
NOR 75  CTT 6h42'49" HUSHOVD Thor
AUS 182 LPR 6h42'49" PETACCHI Alessandro
```

Here, almost every line contains incorrect information (shaded in grey above): the line for Haussler has Cavendish's nationality and bib number, Davis's line has Haussler's information, and so on. The only line with the right

information is Hushovd's, but this is purely an accident—the update happened to preserve his overall third position in the list of riders, so the positional strategy restored his nationality and bib number from the right bit of the source.

Clearly, what we want is for the lens to match up lines in the source and view using some criteria other than position. For example, we could match up lines in the source and view that have similar names (but note that we cannot match them up *by* name, because we corrected the spelling of Davis's first name in the view). Using this strategy, on the same inputs as above, the *put* function would restore the nationality and bib number from each source line to the appropriate rider in the view:

```
GER 74  CTT 6h42'45" HAUSSLER Heinrich
AUS 182 QST 6h42'49" DAVIS Allan
NOR 75  CTT 6h42'49" HUSHOVD Thor
ITA 177 LPR 6h42'49" PETACCHI Alessandro
```

Here is a revised version of the `riders` lens written using the extensions proposed in this paper that behaves better:

```
let l1 : lens = nat1 . bib1 . ( team1 ~ time1 ) . name1
let l2 : lens = bib2 . time2 . ( team2 ~ key name2 )
let riders : lens =
  ( lines <set:l1> ; lines <set:l2> )
```

Compared to the initial version of the lens, we have made two changes (shaded in grey). First, we have indicated that each line of text processed by `l1` and `l2` should be treated as a reorderable "chunk" by enclosing those lenses in angle brackets. And second, we have specified that chunks should be aligned by matching up lines with corresponding names. The `key` combinator indicates that the name of each rider should be considered when computing an alignment while `set` indicates the overall alignment strategy to use: minimizing the sum of the edit distances between pairs of matched chunks and the lengths of unmatched chunks.

The next few sections explain how these features work in detail. The point of this example is that we can tackle instances of the view update problem involving ordered data by providing programmers with simple, compositional primitives for specifying alignment strategies in a lens program.

3. Semantics of Matching Lenses

In this section, we begin our technical development by defining the semantic space of matching lenses. Matching lenses are organized around a simple architecture in which a top-level lens handles the alignment of chunks and the processing of information outside of chunks and a lower-level basic lens handles the processing of information contained in chunks. For now, we will assume that chunks only appear at the top level and the same basic lens is used to process every chunk. We will see how to generalize this simple architecture with multiple basic lenses and nested chunks in Sections 5 and 6.

By themselves, matching lenses do not provide mechanisms for actually computing alignments between chunks in the source and view. Instead, they provide an interface for manipulating chunks that can be used to supply a lens with explicit information about how the chunks in the source should be aligned against the chunks in the view. Thus, matching lenses are a flexible framework that can be instantiated with many different alignment strategies. We describe some specific functions for computing alignments in Section 5.

3.1 Notation

Before we can define matching lenses precisely, we need to fix a few pieces of notation. We assume that the sets of sources and views come equipped with notions of what constitutes a reorderable chunk of information. When u is a structure containing chunks we write

- $|u|$ for the number of chunks in u,
- $locs(u)$ for the set of locations of chunks in u, where a location is a natural number and we number the chunks of u from 1 to $|u|$ in some canonical way,

- $u[n]$ for the chunk located at n in u, where $n \in \mathit{locs}(u)$,
- $u[n{:=}v]$ for the structure obtained from u by setting the chunk at n to v, where n in $\mathit{locs}(u)$ and v is a structure,
- and $\mathit{skel}(u)$ for the residual structure consisting of the parts of u that are not contained in any chunk.

To ensure that chunks can be freely reordered, we require the sets of sources and views be closed under the operation of replacing chunks by other chunks. Formally, when U is a set of structures with chunks (e.g., the set of sources or views) and U' is a set of ordinary structures, we say that U is *chunk compatible* with U' if and only if

- the chunks of every structure in U belong to U'—i.e., for every $u \in U$ 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 $u \in U$ and $n \in \mathit{locs}(u)$ and $u' \in U'$ we have that $u[n{:=}u'] \in U'$.

In a matching lenses, we separate the handling of rigidly ordered and reorderable source information. To represent the reorderable chunks of the source we use finite maps from locations to complements extracted from the source chunks. This representation makes it easy to align the information contained in source chunks with the chunks in the view—we can reorder complements so that they match up with chunks in the view and discard complements that do not match any chunk. When r is a finite map we write

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

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

- and $\{\!|\mathbb{N} \mapsto C|\!\}$ for the set of all finite maps from locations to complements in a set C.

To illustrate how these finite maps are used in a matching lens, consider a simple abstract example. Suppose that we start with a source s that the *get* function maps to a view v and the *res* function maps to a rigid complement c and a finite map r, called a resource (recall that matching lenses split the representation of rigidly ordered and reorderable source information). Also suppose that the chunks in s, v and r are in exact correspondence—i.e., the lens does not reorder chunks, so for every location n, the source chunk $s[n]$ at n maps to $v[n]$ in the view and $r(n)$ in the resource. Now suppose that we change the view to v' and compute—in some way—a correspondence g between v' and v, represented formally as a partial injective function on the locations of chunks in v'. Composing g and r as functions yields a new resource in which the complement for each chunk in s is lined up with the specified chunk in v'. Thus, to propagate the modification make to the view we simply need to *put* back v' with the rigid complement c and the pre-aligned resource $(r \circ g)$ using a lens that accesses source information for chunks through the resource. Developing a framework that captures the constraints on the handling of chunks and resources needed to ensure that this protocol for using lenses behaves as expected is the goal of this section.

3.2 Matching Lenses

We are now ready to define the semantic space of matching lenses precisely.

3.1 Definition [Matching Lens]: Let S and V be sets of structures with chunks, C a set of rigid complements, and k a basic lens with S chunk compatible with $k.S$ (i.e., the source type of k) and V chunk compatible with $k.V$ (i.e.,

the view type of k). A *matching lens* l on S, C, k, and V comprises four functions

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

obeying the following laws for every source $s \in S$, views $v \in V$ and $v' \in V$, rigid complement $c \in C$, and resource $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ (i.e., finite map from locations to appropriately-typed complements for k):

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

$$l.\mathit{get}\ (l.\mathit{create}\ v\ r) = v \qquad \text{(CREATEGET)}$$

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

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

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

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

$$\frac{n \in (\mathit{locs}(v) \cap \mathrm{dom}(r))}{(l.\mathit{create}\ v\ r)[n] = k.\mathit{put}\ v[n]\ (r(n))} \qquad \text{(CHUNKCREATE)}$$

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

$$\frac{n \in (\mathit{locs}(v) \setminus \mathrm{dom}(r))}{(l.\mathit{create}\ v\ r)[n] = k.\mathit{create}\ v[n]} \qquad \text{(NOCHUNKCREATE)}$$

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

$$\frac{\mathit{skel}(v) = \mathit{skel}(v')}{\mathit{skel}(l.\mathit{create}\ v\ r) = \mathit{skel}(l.\mathit{create}\ v'\ r')} \qquad \text{(SKELCREATE)}$$

We write $S \overset{C,k}{\Longleftrightarrow} V$ for the set of all matching lenses on S, C, k and V.

Note that we build k, the basic lens that processes chunks, into the semantics of matching lenses because the CHUNKPUT, NOCHUNKPUT, CHUNKCREATE, and NOCHUNKCREATE laws all mention it. For technical reasons, it is important that the same basic lens be used for each chunk—among other things, it ensures that a matching lens translates reorderings on the view to reorderings on the source.

The *get* function has the same type as in basic lenses. The *put* function, however, has a different type: it takes a rigid complement and a resource rather than a complement. The *res* extracts these structures from a source. The *create* function also has a different type—along with the view, it takes a resource as an argument. This makes it possible for matching lenses to restore source information to chunks that have been newly created. To create a source from a view "from scratch", we invoke *create* with the empty resource.

The PUTGET, CREATEGET, and GETPUT laws express the same fundamental constraints as the basic lens laws.

The GETCHUNKS and RESCHUNKS law capture straightforward constraints on the handling of chunks. They force matching lenses to maintain a one-to-one correspondence between the chunks in the source and view and the complements in the resource. Specifically, the GETCHUNKS law stipulates that each chunk in the source must be carried through to a chunk in the view. This rules out lenses that advertise the presence of chunks in the source but not in the view and vice versa. The RESCHUNKS law requires an analogous property for the resource generated by the *res* function from the source. Lenses that violate these these laws would cause problems with the protocol

for using alignments information with a matching lens described previously—rearranging the resource using an alignment computed for the view would not make sense if the underlying source had different chunks than the view. We do not state PUTCHUNKS and CREATECHUNKS laws because they can be derived from the other laws:

A.1 Lemma [PutChunks]: For every matching lens $l \in S \overset{C,k}{\Longleftrightarrow} V$, view $v \in V$, skeleton $c \in C$, and resource $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ we have $\mathit{locs}(l.\mathit{put}\ v\ (c, r)) = \mathit{locs}(v)$.

A.2 Lemma [CreateChunks]: For every matching lens $l \in S \overset{C,k}{\Longleftrightarrow} V$, view $v \in V$, and resource $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ we have $\mathit{locs}(l.\mathit{create}\ v\ r) = \mathit{locs}(v)$.

The next four laws are the essential matching lens laws—they ensure that the *put* and *create* functions use their resource arguments and the basic lens k correctly. The PUTCHUNKS law stipulates that the nth 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 complement associated to n in the resource. The CHUNKCREATE law is similar. The NOCHUNKPUT and NOCHUNKCREATE laws stipulate that the matching lens must use $k.\mathit{create}$ to produce the nth source instead of $k.\mathit{put}$ when the resource does not contain a complement for n.

The last two laws, SKELPUT and SKELCREATE, state that the skeleton of the sources produced by *put* and *create* must not depend on any of the chunks in the view or complements in the resource. This law is critical for ensuring that matching lenses translate reorderings on the view to reorderings on source chunks.

Compared to the basic lens laws, these laws have a low-level and operational feel—they spell out the precise handling of chunks and resources in detail. However, we can use them to derive higher-level, more declarative properties. For instance, we can use them to show that the *put* and *create* components of every matching lens translate reorderings on the chunks in the view to corresponding reorderings on the chunks in the source. We will write $\mathsf{Perms}(u)$ for the set of all permutations of the chunks in u and $(q\ u)$ for the structure obtained by reordering the chunks of u according to a permutation q. The next two lemmas follow directly from the matching lens laws:

A.3 Lemma [ReorderPut]: For every matching lens $l \in S \overset{C,k}{\Longleftrightarrow} V$, 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 $q\ (l.\mathit{put}\ v\ (c, r)) = l.\mathit{put}\ (q\ v)\ (c, r \circ q^{-1})$.

A.4 Lemma [ReorderCreate]: For every matching lens $l \in S \overset{C,k}{\Longleftrightarrow} V$, 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: $q\ (l.\mathit{create}\ v\ r) = l.\mathit{create}\ (q\ v)\ (r \circ q^{-1})$.

To illustrate the semantics of matching lenses, consider the coercion $\lfloor\cdot\rfloor$ (pronounced "lower"), which 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 realizes the procedure for using a matching lens described above, where we compute a correspondence between the chunks in the new and old views and use it to pre-align the resource before invoking the *put* function (it computes the alignment using a new function align, which is described below):

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

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

The typing rule in the top box can be read as a lemma asserting that if l is a matching lens at $S \overset{C,k}{\Longleftrightarrow} V$ then $\lfloor l \rfloor$ is a basic lens at $S \overset{S}{\Longleftrightarrow} V$. We state and prove this lemma explicitly as Lemma A.5 in the appendix.

The bottom box defines the components of $\lfloor l \rfloor$. The *get* function is identical to l.*get* and the *res* function simply uses the whole source as the complement. The *put* function takes a view v and a complement s as arguments. It first uses l.*res* to calculate a rigid complement c and resource r from s and then calculates a correspondence g between the locations of chunks in v and chunks in $(l.get\ s)$ using $align$. For now, we simply assume that $align$ is a fixed function that takes two views and computes a correspondence between their chunks—formally, a partial injective function on their locations. (We will describe mechanisms for specifying $align$ in Section 5). Next, it composes r and g as functions, which has the effect of pre-aligning the complements in r with the chunks in v as specified by 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. Note that $\lfloor \cdot \rfloor$ does not assume anything about the $align$ function except that it returns the identity alignment when its arguments are identical views. We use this property in the proof that $\lfloor l \rfloor$ obeys the GETPUT law.

4. Matching Lenses for String Data

Having defined the semantic space of matching lenses and developed a few of their main properties, we now turn our attention to syntax and develop a collection of combinators for describing matching lenses that operate on string data. We work with strings both because they expose all the complications related to alignment and because they are ubiquitous (so having a language for defining lenses on strings is useful). The primitives defined in this section are based on the basic and dictionary lens combinators we have studied in previous work [3, 12].

4.1 Notation

First, let us fix some notation for strings with chunks. Let Σ be a finite alphabet (e.g., ASCII). A language L is a subset of Σ^*. When L is non-empty, we write $choose(L)$ for an arbitrary representative of L. The symbol ϵ denotes the empty string and $(u{\cdot}v)$ denotes the concatenation of strings u and v. We lift concatenation to languages in the obvious 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 $L_1 {\cdot}^{!} L_2$, if 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.

We will define the types of our matching lens primitives using regular expressions decorated with annotations indicating the locations of chunks. The set of regular expressions is generated by the following grammar

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

where u ranges over arbitrary strings (including ϵ). 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]).

Now we will show how to add annotations to regular expressions to specify the locations of chunks. Let '$\langle$' and '$\rangle$' be fresh symbols that do not occur in Σ. The set of chunk-annotated regular expressions is generated by the following grammar

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

where $\mathcal{R}$ ranges over ordinary regular expressions. Observe that every ordinary regular expression is also a chunk-annotated regular expression and that chunks only appear at the top level. The denotation $[\![A]\!]$ of a chunk-annotated regular expression A is a language of chunk-annotated strings—i.e., strings over the extended alphabet $(\Sigma \cup \{\text{'}\langle\text{'}, \text{'}\rangle\text{'}\})^*$ where occurrences of '$\langle$' and '$\rangle$' are balanced and non-nested. We write $\lfloor \cdot \rfloor$ for the erasure function that maps chunk-annotated strings to ordinary strings (by removing '$\langle$' and '$\rangle$' characters and mapping every other character to itself) and we lift $\lfloor \cdot \rfloor$ to regular expressions and languages in the obvious way.

We will use languages of chunk-annotated strings to "read off" the locations of chunks in ordinary strings. Given a language of chunk-annotated strings L and an ordinary string u in the erasure of L, we calculate the number $|u|$ of chunks in u, the chunk $u[n]$ at n in u, and so on, by first "parsing" u into a chunk-annotated string using L, and then using the explicit chunks in the result to give meaning to each of the concepts involving chunks. For example, if L is the language of chunk-annotated strings described by the chunk-annotated regular expression $\langle(\text{'A'} \mid \ldots \mid \text{'Z'})\cdot(\text{'1'} \mid \ldots \mid \text{'9'})\rangle^*$ and u is "A1B2C3", then u parses into "$\langle$A1$\rangle\langle$B1$\rangle\langle$C1$\rangle$", so the number $|u|$ of chunks in u is 3, the second chunk $u[2]$ in u is "B2", and the string $u[2{:=}\text{"Z9"}]$ obtained by setting the second chunk in u to "Z9" is "A1Z9C3".

Obviously for this way of identifying chunks in ordinary strings to make sense, we need to be sure that every string has a unique parse into a chunk-annotated string using L. Not every language has this property—e.g., $(a{\cdot}b)$ has two different parses using the language $\{$"$\langle a\rangle b$", "$a\langle b\rangle$"$\}$. To rule out such chunk ambiguous languages, we will be careful to ensure that every language of chunk-annotated strings under discussion uniquely determines the chunks of strings in its erasure—i.e., whenever we introduce a chunk-annotated regular language L as the source or view type of a matching lens, we will make sure that $\lfloor\cdot\rfloor$ is bijective on L.

4.2 Primitives

With this notation in place, we are now ready to define matching lens primitives for strings.

Lift The first primitive lifts a basic string lens to a matching lens. This makes it possible to use basic lenses such as `copy` and `(<->)` in matching lens programs. As the source and view types of a basic lens are sets of ordinary strings, the lifted lens does not have chunks in its type 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 $\widehat{k}$ can be an arbitrary lens because the source and view types do not have chunks.

Match Another way to convert a basic lens to a matching lens is to place it within a chunk.

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

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

The $\langle k\rangle$ (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 $\square$ 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.*put* on the view and $r(1)$ if r is defined on 1 and k.*create* on the view otherwise. The *create* function is identical to *put*.

Concatenation The next three primitives build bigger lenses out of smaller ones using the regular operators. Together, these operators represent a core language that can be used to express many useful transformations on strings. The concatenation operator is the simplest:

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

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

The *get* function splits the source string into two smaller strings s_1 and s_2, applies the *get* functions of l_1 and l_2 to these strings, and concatenates the resulting strings. We write $(s_1 \cdot s_2)$ in the box above to indicate that s_1 and s_2 are strings in S_1 and S_2 that concatenate to $(s_1 \cdot s_2)$. Because the typing rule requires that the languages $\lfloor S_1 \rfloor$ and $\lfloor S_2 \rfloor$ of ordinary strings be unambiguously concatenable, s_1 and s_2 are unique. This also ensures that $(S_1 \cdot S_2)$ unambiguously determines chunks in the source.

The *res* function splits the source into smaller strings s_1 and s_2 and applies the *res* functions of l_1 and l_2 to these strings. This yields rigid complements c_1 and c_2 and resources r_1 and r_2. It merges the complements into a pair (c_1, c_2) and combines the resources into a single finite map $(r_1 +\!\!+ r_2)$. Because 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 belonging to $\{\!|\mathbb{N} \mapsto k.C|\!\}$. 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. The *create* function is similar. Both functions split the resource using $\textit{split}(n, r)$ and $|v_1|$ (the number of chunks of the first substring of the view). This yields two resources: one that behaves like r restricted to locations less than or equal to $|v_1|$ and another resource that behaves like like r shifted down by $|v_1|$. Splitting the resource in this way ensures that every complement that is aligned with a chunk in the view remains aligned with the same chunk in the corresponding portion of the resource and substring of the view. Formally, *split* is defined as follows:

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

Note that $\textit{split}(|r_1|, r_1 +\!\!+ r_2) = (r_1, r_2)$. This property is essential for ensuring the GETPUT law.

As discussed above, 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. We might be tempted to relax the condition and allow l_1 and l_2 to be defined over two different basic lenses, as long as those lenses had compatible complement types. Unfortunately,

this would lead to lenses with weaker properties. For example, consider the lens $(\langle k_1 \rangle \cdot \langle k_2 \rangle)$ where k_1 and k_2 are basic lenses defined as follows:

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

Invoking the *put* function of this lens on "aa" yields "ab" as a result (since k_1 and k_2 are "bijective" lenses, the rigid complement and resource arguments do not affect the evaluation of *put*). Now suppose that we swap the chunks of "aa". According to Lemma A.3, the *put* function should produce "ba"—i.e., the string obtained by swapping the chunks of "ab". But this is not what happens. Swapping the chunks of "aa" is a no-op, so *put* produces the same result as before. Thus, although it is tempting to allow matching lenses that use different lenses to process chunks, we don't do this, because it would require sacrificing natural properties such as Lemma A.3.

Kleene Star The Kleene star operator iterates a lens:

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

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

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. 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 $l.\mathit{create}$. This is the reason that *create* takes a resource as an argument—the resource may contain complements for chunks in the extra substrings.

To illustrate the last few definitions, let us consider a simple example:

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

The lens `k` copies an upper-case letter from source to view and deletes a lower-case letter while `l` uses the match, concatenation, and Kleene-star lenses to iterate `k` over a non-empty list of comma-separated chunks (the Boomerang implementation automatically inserts coercions to lift basic lenses to matching lenses using $(\widehat{\cdot})$ and to convert the top-level matching lens to a basic lens using $\lfloor \cdot \rfloor$ when we invoke its `get` or `put` component with string arguments). The behavior of `l`.*get* is straightforward—e.g., it maps "`Xx,Yy,Zz`" to "`X,Y,Z`". However, `l`.*put* is

more sophisticated—it restores the lower-case letters from source chunks by matching up upper-case letters in the old and new views. For instance, if we insert "W" into the middle of the view, *put* behaves as follows:

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

Let us trace the evaluation of this example in detail. First, the $\lfloor \texttt{l} \rfloor$.*put* lens uses `l`.*res* to calculate a rigid complement c and resource r from the source string:

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

Next, it calculates a correspondence g between the chunks in the old view and the new view and composes this correspondence with r to obtain the pre-aligned resource (for the moment we are ignoring how the lens computes g—see Section 5):

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

Finally, it invokes `l`.*put* on the new view, c, and $(r \circ g)$. The effect is that the lower-case letters are restored to the chunk containing the corresponding upper-case letter. Note that the third chunk, `W` is created fresh because the pre-aligned resource $(r \circ g)$ is undefined on 3.

Union The final regular operator forms the union of two matching lenses:

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

$$\begin{array}{ll} \textit{get}\ s & = \begin{cases} l_1.\textit{get}\ s \text{ if } s \in \lfloor S_1 \rfloor \\ l_2.\textit{get}\ s \text{ if } s \in \lfloor S_2 \rfloor \end{cases} \\ \textit{res}\ s & = \begin{cases} \mathit{Inl}(l_1.\textit{res}\ s) \text{ if } s \in \lfloor S_1 \rfloor \\ \mathit{Inr}(l_2.\textit{res}\ s) \text{ if } s \in \lfloor S_2 \rfloor \end{cases} \\ \textit{put}\ v\ (c, r) & = \begin{cases} l_1.\textit{put}\ v\ (c_1, r) \text{ if } v \in \lfloor V_1 \rfloor \wedge c = \mathit{Inl}(c_1) \\ l_2.\textit{put}\ v\ (c_2, r) \text{ if } v \in \lfloor V_2 \rfloor \wedge c = \mathit{Inr}(c_2) \\ l_1.\textit{create}\ v\ r \quad \text{if } v \notin \lfloor V_2 \rfloor \wedge c = \mathit{Inl}(c_2) \\ l_2.\textit{create}\ v\ r \quad \text{if } v \notin \lfloor V_1 \rfloor \wedge c = \mathit{Inr}(c_1) \end{cases} \\ \textit{create}\ v\ r & = \begin{cases} l_1.\textit{create}\ v\ r \text{ if } v \in \lfloor V_1 \rfloor \\ l_2.\textit{create}\ v\ r \text{ if } v \notin \lfloor V_1 \rfloor \end{cases} \end{array}$$

The union lens is 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 $\mathit{Inl}(c)$ if the source belongs to $\lfloor S_1 \rfloor$ and $\mathit{Inr}(c)$ 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 $(\lfloor V_1 \rfloor \setminus V_2)$ but the complement is of the form $Inl(c)$. 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 $(V_1 \mid V_2)$ is chunk unambiguous—i.e., that strings in the intersection $(V_1 \cap V_2)$ have unique parses. It rules out language of chunk-annotated strings such as $(a{\cdot}\langle b\rangle \mid \langle a\rangle{\cdot}b)$.

Composition The composition operator puts two matching lenses in sequence:

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

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

Composition is especially interesting as a matching lens because it propagates alignment information through two phases of computation. The *get* function applies l_1.*get* and l_2.*get* in sequence. The *res* function applies l_1.*res* to the source s, yielding a rigid complement c_1 and resource r_1, and l_2.*res* to $(l_1.\textit{get } s)$, 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 is defined as follows:[3]

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

Note that we have the following equalities

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

so $(\textit{zip } r_1\ r_2)$ is defined on the same locations as $\mathrm{dom}(r_1)$ and $\mathrm{dom}(r_2)$.

The *put* function unzips the resource and applies l_2.*put* and l_1.*put* in that order. The $\textit{unzip}$ function on finite maps is defined as follows

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

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 the behavior of the composition lens, consider the following example:

```
let k1 : lens = del [0-9] . copy [A-Z] . copy [a-z]
let k2 : lens = del [A-Z] . key (copy [a-z])
let l : lens =
```

[3] The angle brackets distinguish these pairs from the rigid complements generated for the concatenation lens.

```
<k1> . (copy "," . <k1>)* ;
<k2> . (copy "," . <k2>)*
```

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

```
l.get "1Aa,2Bb,3Cc" = "a,b,c"
```

The resource produced by *res* represents the upper-case letter and number together, so the *put* function restores both to the appropriate chunk:

```
l.put "b,a" into "1Aa,2Bb,3Cc" = "2Bb,1Aa"
```

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 exclusively on the mechanisms of matching lenses, which extend basic lenses with a notion of chunks and provide an interface for supplying a lens with explicit directives about how source chunks should be aligned against the view. But we have not said where these directives come from.

In this section, we describe the strategies for computing alignments that we have implemented in the Boomerang language [10]. We describe three different alignment "species" and we present mechanisms for tuning alignment strategies using notions of "keys" and "thresholds". Because alignment is a fundamentally heuristic operation, the choice of a proper alignment function depends intimately on the details of the application at hand. One of the main strengths of the framework of matching lenses is its flexibility. Matching lenses can be instantiated with arbitrary alignment functions since the well-behavedness of lenses does not hinge on any special properties of the function used to align chunks: the only property we require is that it return the identity alignment when its arguments are identical. Thus, the mechanisms described in this section should not be taken as exhaustive; it would be easy to extend them with additional mechanisms if needed.

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

- **Positional:** The alignment matches up chunks by position. If one of the lists has more chunks than the other, the extra chunks at the end of the longer list do not match any chunk in the shorter list.
- **Set-like:** The alignment minimizes the sum of the total edit distances between pairs of matched chunks and the lengths of unmatched chunks.
- **Diff-like:** The alignment minimizes the same function as the set-like strategy, but only considers alignments without "crossing" edges. This heuristic can be computed efficiently using a variant of the standard algorithm for computing the longest common subsequence of two lists.

These species are illustrated in the following simple examples:

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

These examples also illustrate how Boomerang programmers indicate a species to use with chunks. The match combinator implemented in Boomerang actually takes two arguments: an annotation that specifies the alignment

species and a basic lens for chunks. (The shorthand <*l*> we have been using in examples desugars to <set:*l*>). When we coerce a matching lens to a basic lens using $\lfloor \cdot \rfloor$, it instantiates the *align* function using the species indicated in the annotation (recall that this coercion is automatically implemented by the Boomerang type checker when we invoke the `get` or `put` component of a matching lens). Boomerang's typechecker checks that the same annotation is used on every instance of the match combinator—e.g., it disallows `(<pos:l > . <dif:l>)`, which specifies two different species for chunks.

Keys Typically, we do not want to consider the entire contents of chunks when we compute alignments. Boomerang includes two primitives, `key` and `nokey`, that allow programmers to control the portions of each chunk that are used to compute alignments. Both of these combinators take a matching 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" a key for each chunks (just like we use annotations to "read off" the locations of chunks). When the *align* function computes an alignment for two lists of chunks, it uses the view type to extract the regions of each chunk that are marked as keys and ignores the rest of each chunk.

To illustrate the use of keys, consider a simple example:

```
let k = del [0-9] . copy [A-Z] . copy [a-z]
let l = <set:k> . (copy "," . <set:k>)*
l.put "Cc,Bb,Aa" into "1Aa,2Bb,3Cc" = "1Cc,2Bb,3Aa"
```

Although this program aligns chunks using the set-like species, it behaves positionally because the view type does not contain any key annotations—i.e., the key of every chunk is the empty string. The following revised version of has a key annotation

```
let k = del [0-9] . key (copy [A-Z]) . copy [a-z]
let l = <k> . (copy "," . <k>)*
```

so its *put* function matches up chunks using the upper-case letters in the view:

```
l.put "Cc,Bb,Aa" into "1Aa,2Bb,3Cc" = "3Cc,2Bb,1Aa"
```

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

```
l.put "Ca,Bb,Ac" into "1Aa,2Bb,3Cc" = "3Ca,2Bb,1Ac"
```

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

```
let k = key (del [0-9] . copy [A-Z] . nokey (copy [a-z]))
let l = <k> . (copy "," . <k>)*
```

The simple mechanisms for indicating keys provided by the `key` and `nokey` primitives suffice for many practical examples, but there are many ways that it could be extended. For example, we could provide programmers with mechanisms for generating unique keys or for building keys structured as tuples or records (rather than simply flattening the regions of each chunk marked as a key into a string). We plan to explore these ideas in future work.

Thresholds The set-like and diff-like 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 result in a minimal cost alignment. For instance, in the following program, where keys are three characters long

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

we might like the DBD and BBB2 chunks to not be aligned with each other. However, the set-like species aligns 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 achieve the behavior we want, we can add a threshold, as shown in the following example:

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

The sim species is similar to set, but takes an integer n as an argument. It minimizes the total edit distances between aligned chunks, like set, but it only aligns chunks whose longest common subsequence is at least $n\%$ of the lengths of their keys. (The set species actually desugars to (sim 0) and dictionary lenses can be simulated using (sim 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 diff species also supports thresholds. We often use diff with a threshold to align chunks containing of unstructured text.

6. Extensions

Our design for matching lenses is based on three assumptions:

1. the source and view only contain chunks at the top level,
2. the same lens is used to process every chunk, and
3. the lens does not reorder chunks.

However, it is often important to be able to use different lenses to process multiple kinds of chunks, to nest chunks within other chunks, and to reorder chunks in going from source to view. This section describes how we can extend the matching lens framework to accommodate each of these features.

6.1 Nested Chunks

Some sources contain reorderable information at several different levels of structure. For example, suppose that the source is a Wiki with three levels of structure: sections, subsections, and paragraphs,

```
=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 Tour is usually held in July...
=Classics=
The classics are one-day cycling races...
==Milan-San Remo==
Milan-San Remo is held in March...
==Paris-Roubaix==
Paris-Roubaix is held in mid-April...
```

and the view is a simplified list of section and subsection headings:

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

If we update the view by reordering the sections and adding some new subsections to each

```
Classics:
```

```
  Milan-San Remo
  Ronde van Vlaanderen
  Paris-Roubaix
Grand Tours:
  Giro d'Italia
  Tour de France
  Vuelta a Espana
```

we would the paragraphs to be restored to the appropriate section or subsection:

```
=Classics=
The classics are one-day cycling races...
==Milan-San Remo==
Milan-San Remo is held in March...
==Ronde van Vlaanderen==
==Paris-Roubaix==
Paris-Roubaix is held in mid-April...
=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 Tour is usually held in July...
==Vuelta a Espana==
```

To do this, we need a lens that aligns chunks at several levels of structure, not just at the top-level. Using the lower combinator we can convert a matching lens for the nested chunks to a basic lens and use it to process the nested chunks as in the following program:

```
let HEADING : regexp = ([^=\n]* - ( " " . [^]* ))
let TEXT : regexp = (([^=\n] . [^\n]* )? . "\n")*
let paragraphs : lens = del (TEXT . ("\n" . TEXT)* )
let subsection : lens =
  ins "  " . del "==" . key HEADING . del "==" .
  copy "\n" .
  paragraphs
let section : lens =
  del "=" . key HEADING . del "=" . ins ":" .
  copy "\n" .
  paragraphs . lower < set : subsection >*
let wiki : lens =
  < set : section >*
```

The `paragraph` lens deletes blocks of text separated by double newline characters. The `subsection` lens inserts two space characters as indentation, copies the subsection heading, and deletes the paragraphs that follow. The `section` lens copies the heading, inserts a colon character, deletes the paragraphs that follow, and then processes a list of subsections. The top-level `wiki` lens processes a list of sections.

The main thing to notice about this program is that we can use `lower` to build matching lenses that process chunks using other matching lenses even though the match combinator takes a basic lens as an argument. Lenses constructed in this way align chunks in strict nested fashion—e.g., in this example, the top-level chunks for sections are first aligned against other sections and the nested chunks for subsections within each section are aligned next.

6.2 Tags

In other applications, we need to use several different basic lenses to process chunks. For example, suppose that we wanted to build a version of the `wiki` lens that aligns subsections and sections separately. Why would we want this? Observe that the nested alignments computed by the `wiki` lens just described never align subsections in different sections. Thus, if we update the view by moving the heading for the "`Paris-Roubaix`" subsection from the classics section to the grand tours section

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

the paragraph under the Paris-Roubaix subsection will be lost when we invoke the *put* function:

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

The reason for this behavior is that the alignment follows the nesting structure of the document.

In this example, it would be better to align section and subsections separately instead of following the structure of the document. To do this, we need to generalize matching lenses to allow multiple kinds of chunks in the same program. Here is a revised version of the `wiki` lens written using "tags" that has the behavior we want:

```
let section : lens =
  del "=" . key HEADING . del "=" . ins ":" .
  copy "\n" .
  paragraphs
let wiki : lens =
  ( < tag "section" set : section > .
    < tag "subsection" set : subsection >* )*
```

Rather than having nested chunks, this lens has two chunks at the top level—one for sections and another for subsections. The `tag` primitive gives distinct names to these chunks and indicates that the two kinds of chunks should be handled separately by the lens. On the same inputs as above, the *put* function produces a new source

```
=Classics=
The classics are one-day cycling races...
==Milan-San Remo==
Milan-San Remo is held in March...
=Grand Tours=
The grand tours are major cycling races...
==Paris-Roubaix==
```

```
Paris-Roubaix is held in mid-April...
==Giro d'Italia==
The Giro is usually held in May and June...
==Tour de France==
The Tour is usually held in July...
```

where the paragraph under the Paris-Roubaix subsection is correctly restored from the source. Extending matching lenses with tags is simple—we generalize each of our structures with an extra level of indirection for tags. For example, we change the type of resources to finite maps from tags to locations to complements and we compute alignments by tag.

6.3 Swap

All of the matching lenses we have seen so far map chunks in the source through to the same chunks in the view and vice versa, but in some applications we need matching lenses that reorder chunks. The swap operator, written $(l_1 \sim l_2)$, behaves like the concatenation lens, but swaps the order of strings in the view. Adding swap as a matching lens complicates the story significantly because it makes it possible to construct lenses that reorder chunks. Lenses that reorder chunks break the protocol for using matching lenses where we pre-align the resource using a correspondence computed for the view. They also cause problems with the sequential composition operator—in general, the two lenses will reorder the chunks in different ways, so it will not make sense to simply zip the resources generated by each lens together together and align them against the view.

To recover the behavior we want in the presence of primitives that reorder chunks, we need to keep track of the permutation on chunks that is computed by the lens. Therefore, we add a new component to every matching lens

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

that computes the permutation on chunks realized by the *get* function.

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 only chunk, and the concatenation operator merges the permutations returned by its sublenses in the obvious way.

We also need to generalize the CHUNKPUT, CHUNKCREATE, NOCHUNKPUT, and NOCHUNKCREATE laws using *perm*—the old versions do not hold for lenses that permute the order of chunks in going from source to view:

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

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

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

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

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 are similar generalizations of the previous versions.

Composition Using *perm*, we can define a better version of the sequential composition operator that uses the permutation on chunks computed in each phase:

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

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

The *res* function applies the inverse of the permutation computed by l_2 from the intermediate view before it zips the resources computed by l_1 and l_2 together. This puts the resource computed by l_1 into the "view order" of l_2. Likewise, the *put* function applies the inverse of the permutation computed by l_2 to put the resource r_1 into the view order of l_1.

Swap The swap lens is defined as follows:

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

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

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 $(\mathbin{**})$ operator

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

which behaves similarly to the $(++)$ operator for resources.

7. 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 a law

$$\frac{s \sim s'}{l.put\ v\ s = l.put\ v\ s'} \qquad \text{(EQUIVPUT)}$$

that forces the *put* function to be "oblivious" to certain features of sources, which are characterized by an equivalence relation $\sim$. If we instantiate $\sim$ with an equivalence that relates strings differing only in the relative order of chunks having different keys we get some constraints on *put*—e.g., forbidding lenses that operate positionally—but they are weaker than the conditions stated in the matching lens laws. For example, Lemma A.3 does not hold for dictionary lenses because the type system does not explicitly 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 does not work well with general reorderings.

Relational view update translators often use constraints expressed in the schema 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 [17]. Matching lenses also use a notion of keys for alignment, but they permit the correspondence between chunks to be computed using an arbitrary heuristics.

Alignment issues also come up when bidirectional transformations are used for software model transformations. Some systems offer "traceability links" that can be used for alignment [6, 22].

8. 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 instantiating the framework of matching lenses in other settings besides strings and exploring implementation issues including algebraic optimization and lenses for streaming data. We are also interested in exploring alignment mechanisms based on data provenance.

Acknowledgments We are grateful to Zack Ives, Alexandre Pilkiewicz, Val Tannen, Philip Wadler, and Steve Zdancewic for helpful comments on an earlier draft of the paper. 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. To appear. Manuscript available from http://www-igm.univ-mlv.fr/~berstel/LivreCodes/.

[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] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4–5):385–406, 2008. Short version in DBPL '05.

[6] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT '09: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, pages 260–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] 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.

[8] Kathleen Fisher and Robert Gruber. PADS: a domain-specific language for processing ad hoc data. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Chicago, IL*, pages 295–304, 2005.

[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 bi-directional transformations. In *Partial Evaluation and Program Manipulation (PEPM)*, pages 178–189, 2004. Long version to appear in HOSC.

[15] 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.

[16] Shinya Kawanaka and Haruo Hosoya. bixid: a bidirectional transformation language for XML. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Portland, Oregon*, pages 201–214, 2006.

[17] 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.

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

[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] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *International Workshop Graph-Theoretic Concepts in Computer Science, Herrsching, Germany*, volume 903 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[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.

This appendix contains proofs for each of the results described in our technical development, including well-behavedness proofs for each primitive lens and lens combinator.

A. Matching Lens Proofs

A.1 Lemma [PutChunks]: For every matching lens $l \in S \overset{C,k}{\Longleftrightarrow} V$, view $v \in V$, skeleton $c \in C$, and resource $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ we have $\mathit{locs}(l.\mathit{put}\ v\ (c, r)) = \mathit{locs}(v)$.

Proof: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a matching lens, $v \in V$ a view, $c \in C$ a skeleton, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource. We calculate as follows

$$\begin{array}{lll} & \mathit{locs}(l.\mathit{put}\ v\ (c, r)) & \\ = & \mathit{locs}(l.\mathit{get}\ (l.\mathit{put}\ v\ (c, r))) & \text{by GETCHUNKS for } l \\ = & \mathit{locs}(v) & \text{by PUTGET for } l \end{array}$$

and obtain the required equality. □

A.2 Lemma [CreateChunks]: For every matching lens $l \in S \overset{C,k}{\Longleftrightarrow} V$, view $v \in V$, and resource $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ we have $\mathit{locs}(l.\mathit{create}\ v\ r) = \mathit{locs}(v)$.

Proof: Similar to the proof of Lemma A.1. □

A.3 Lemma [ReorderPut]: For every matching lens $l \in S \overset{C,k}{\Longleftrightarrow} V$, 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 $\circlearrowleft\ (l.\mathit{put}\ v\ (c, r)) = l.\mathit{put}\ (\circlearrowleft\ v)\ (c, r \circ q^{-1})$.

Proof: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a matching lens, $v \in \lfloor V \rfloor$ a view, $c \in C$ a rigid complement, $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource, and $q \in \mathsf{Perms}(v)$ a permutation on the chunks in v such that $\mathrm{dom}(r) = (\mathit{locs}(v))$. To shorten the proof, define structures s_1 and s_2 as follows:

$$s_1 \triangleq l.\mathit{put}\ (\circlearrowleft\ v)\ (c, r \circ q^{-1}) \qquad s_2 \triangleq \circlearrowleft\ (l.\mathit{put}\ v\ (c, r))$$

First, we demonstrate that the sets of locations in s_1 and s_2 are identical, by calculating as follows:

$$\begin{array}{lll} \mathit{locs}(s_1) = \mathit{locs}(l.\mathit{put}\ (\circlearrowleft\ v)\ (c, r \circ q^{-1})) & & \text{by definition } s_1 \\ = \mathit{locs}(\circlearrowleft\ v) & & \text{by Lemma A.1 for } l \\ = \mathit{locs}(v) & & \text{by definition } \mathit{locs} \text{ and } \circlearrowleft \\ = \mathit{locs}(l.\mathit{put}\ v\ (c, r)) & & \text{by Lemma A.1 for } l \\ = \mathit{locs}(\circlearrowleft\ (l.\mathit{put}\ v\ (c, r))) & & \text{by definition } \mathit{locs} \text{ and } \circlearrowleft \\ = \mathit{locs}(s_2) & & \text{by definition of } s_2 \end{array}$$

Next, we show that for every location $x \in \mathit{locs}(s_1)$ the chunk at x in s_1 is identical to the chunk at x in s_2. Let $x \in \mathit{locs}(s_1)$ be a location. We analyze two cases.

Case $q^{-1}(x) \in \mathrm{dom}(r)$**:** We calculate as follows

$$\begin{array}{lll} s_1[x] = l.\mathit{put}\ (\circlearrowleft\ v)\ (c, r \circ q^{-1})[x] & & \text{by definition } s_1 \\ = k.\mathit{put}\ ((\circlearrowleft\ v)[x])\ ((r \circ q^{-1})(x)) & & \text{by CHUNKPUT for } l \\ = k.\mathit{put}\ (v[q^{-1}(x)])\ (r(q^{-1}(x))) & & \text{by definition } \circlearrowleft \text{ and } [\cdot] \\ = (l.\mathit{put}\ v\ (c, r))[q^{-1}(x)] & & \text{by CHUNKPUT for } l \\ = (\circlearrowleft\ (l.\mathit{put}\ v\ (c, r)))[x] & & \text{by definition } \circlearrowleft \text{ and } [\cdot] \\ = s_2[x] & & \text{by definition } s_2 \end{array}$$

and obtain the required equality.

Case $q^{-1}(x) \notin \mathrm{dom}(r)$**:** Similar to the previous case, using NOCHUNKPUT instead of CHUNKPUT.

Finally, we prove that $skel(s_1) = skel(s_2)$. Observe that $skel(v) = skel(\circlearrowleft\, v)$. Using this fact, we calculate as follows:

$$\begin{aligned}
skel(s_1) &= skel(k.put\ (\circlearrowleft\, v)\ (c, r \circ q^{-1})) && \text{by definition } s_1 \\
&= skel(l.put\ v\ (c, r)) && \text{by SKELPUT for } l \\
&= skel(\circlearrowleft\ (l.put\ v\ (c, r))) && \text{by definition } skel \text{ and } \circlearrowleft \\
&= skel(s_2) && \text{by definition } s_2
\end{aligned}$$

Putting all these facts together we have $s_1 = s_2$, which completes the proof. □

A.4 Lemma [ReorderCreate]: For every matching lens $l \in S \overset{C,k}{\Longleftrightarrow} V$, 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: $\circlearrowleft\ (l.create\ v\ r) = l.create\ (\circlearrowleft\, v)\ (r \circ q^{-1})$.

Proof: Similar to the proof of Lemma A.3, using CHUNKCREATE and NOCHUNKCREATE instead of CHUNKPUT and NOCHUNKPUT. □

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

A.5 Lemma: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a matching lens. Then $\lfloor l \rfloor$ is a basic lens in $S \overset{S}{\Longleftrightarrow} V$.

Proof:

▶ **GetPut**: Let $s \in S$. We calculate as follows

$$\begin{aligned}
&\lfloor l \rfloor.put\ (\lfloor l \rfloor.get\ s)\ (\lfloor l \rfloor.res\ s) \\
&= \lfloor l \rfloor.put\ (l.get\ s)\ s && \text{by definition } \lfloor l \rfloor.get \text{ and } \lfloor l \rfloor.res \\
&= l.put\ (l.get\ s)\ (c, r \circ g) && \text{by definition } \lfloor l \rfloor.put \\
&\qquad \text{where } c, r = l.res\ s \\
&\qquad\quad \text{and } g = align(l.get\ s, l.get\ s) \\
&= l.put\ (l.get\ s)\ (c, r) && \text{by GETCHUNKS and RESCHUNKS} \\
&&& \text{and as } align(l.get\ s, l.get\ s) = id \\
&= l.put\ (l.get\ s)\ (l.res\ s) && \text{by definition } (c, r) \\
&= s && \text{by GETPUT for } l
\end{aligned}$$

and obtain the required equality.

▶ **PutGet**: Let $v \in V$ and $s \in S$. We calculate as follows

$$\begin{aligned}
&\lfloor l \rfloor.get\ (\lfloor l \rfloor.put\ v\ s) \\
&= \lfloor l \rfloor.get\ (l.put\ v\ (c, r \circ g)) && \text{by definition } \lfloor l \rfloor.put \\
&\qquad \text{where } c, r = l.res\ s \\
&\qquad\quad \text{and } g = align(v, l.get\ s) \\
&= l.get\ (l.put\ v\ (c, r \circ g)) && \text{by definition } \lfloor l \rfloor.get \\
&= v && \text{by PUTGET for } l
\end{aligned}$$

and obtain the required equality.

▶ **CreateGet**: Let $v \in V$. We calculate as follows

$$\begin{aligned}
&\lfloor l \rfloor.get\ (\lfloor l \rfloor.create\ v) \\
&= \lfloor l \rfloor.get\ (l.create\ v\ \{\!|\}\!|) && \text{by definition } \lfloor l \rfloor.create \\
&= l.get\ (l.create\ v\ \{\!|\}\!|) && \text{by definition } \lfloor l \rfloor.put \\
&= v && \text{by CREATEGET for } l
\end{aligned}$$

and obtain the required equality, which completes the proof. □

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

A.6 Lemma: Let $k \in S \overset{C}{\Longleftrightarrow} V$ and $k' \in S' \overset{C'}{\Longleftrightarrow} V'$ be basic lenses. Then $\widehat{k}$ is a matching lens in $S \overset{C,k'}{\Longleftrightarrow} V$.

Proof:

► **GetPut**: Let $s \in \lfloor S \rfloor$ be a string. As S is a language of ordinary strings, we have that $s \in S$. Using this fact, we calculate as follows

$$\begin{array}{ll} & \widehat{k}.\mathit{put}\ (\widehat{k}.\mathit{get}\ s)\ (\widehat{k}.\mathit{res}\ s) \\ = & \widehat{k}.\mathit{put}\ (k.\mathit{get}\ s)\ (k.\mathit{res}\ s, \{\!|\}\!|) \quad \text{by definition of } \widehat{k}.\mathit{get} \text{ and } \widehat{k}.\mathit{res} \\ = & k.\mathit{put}\ (k.\mathit{get}\ s)\ (k.\mathit{res}\ s) \quad \text{by definition of } \widehat{k}.\mathit{put} \\ = & s \quad \text{by GETPUT for } k \end{array}$$

and obtain the required equality.

► **PutGet**: Let $v \in \lfloor V \rfloor$ be a string, $c \in C$ a rigid complement, $r \in \{\!|\mathbb{N} \mapsto k'.C|\}\!|$ a resource. As V is a language of ordinary strings, we have that $v \in V$. Using this fact, we calculate as follows

$$\begin{array}{ll} & \widehat{k}.\mathit{get}\ (\widehat{k}.\mathit{put}\ v\ (c, r)) \\ = & \widehat{k}.\mathit{get}\ (k.\mathit{put}\ v\ c) \quad \text{by definition } \widehat{k}.\mathit{put} \\ = & k.\mathit{get}\ (k.\mathit{put}\ v\ c) \quad \text{by definition } \widehat{k}.\mathit{get} \\ = & v \quad \text{by PUTGET for } k \end{array}$$

and obtain the required equality.

► **CreateGet**: Let $v \in \lfloor V \rfloor$ be a string and $r \in \{\!|\mathbb{N} \mapsto k'.C|\}\!|$ a resource. As V is a language of ordinary strings, we have that $v \in V$. Using this fact, we calculate as follows

$$\begin{array}{ll} & \widehat{k}.\mathit{get}\ (\widehat{k}.\mathit{create}\ v\ r) \\ = & \widehat{k}.\mathit{get}\ (k.\mathit{create}\ v) \quad \text{by definition } \widehat{k}.\mathit{create} \\ = & k.\mathit{get}\ (k.\mathit{create}\ v) \quad \text{by definition } \widehat{k}.\mathit{get} \\ = & v \quad \text{by CREATEGET for } k \end{array}$$

and obtain the required equality.

► **GetChunks**: Let $s \in \lfloor S \rfloor$. We calculate as follows

$$\begin{array}{ll} \mathit{locs}(s) = \emptyset & \text{as } S \text{ is a language of ordinary strings} \\ = \mathit{locs}(\widehat{k}.\mathit{get}\ s) & \text{as } V \text{ is a language of ordinary strings} \end{array}$$

and obtain the required equality.

► **ResChunks**: Let $s \in \lfloor S \rfloor$ be a string, $c \in C$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k'.C|\}\!|$ a resource such that $(c, r) = \widehat{k}.\mathit{res}\ s$. By the definition of $\widehat{k}.\mathit{res}$ we have that $r = \{\!|\}\!|$. Using this fact, we calculate as follows

$$\begin{array}{ll} \mathit{locs}(s) = \emptyset & \text{as } S \text{ is a language of ordinary strings} \\ = \mathrm{dom}(r) & \text{as } r = \{\!|\}\!| \end{array}$$

and obtain the required equality.

► **ChunkPut**: Vacuously holds. Suppose, for a contradiction, that there exists a string $v \in \lfloor V \rfloor$, a resource $r \in \{\!|\mathbb{N} \mapsto k'.C|\}\!|$, and a location $x \in (\mathit{locs}(v) \cap \mathrm{dom}(r))$. As V is a language of ordinary strings, we have $\mathit{locs}(v) = \emptyset$, which contradicts $x \in \mathit{locs}(v)$.

▶ **ChunkCreate**: Vacuously holds by the same argument as the proof for CHUNKPUT.

▶ **NoChunkPut**: Vacuously holds by the same argument as the proof for CHUNKPUT.

▶ **NoChunkCreate**: Vacuously holds by the same argument as the proof for CHUNKPUT.

▶ **SkelPut**: Let $v \in \lfloor V \rfloor$ and $v' \in \lfloor V \rfloor$ be strings, $c \in C$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k'.C|\!\}$ and $r' \in \{\!|\mathbb{N} \mapsto k'.C|\!\}$ resources such that $skel(v) = skel(v')$. As V is a language of ordinary strings, we have that $v = v'$. Using this fact, we calculate as follows:

$$\begin{array}{ll} & skel(\widehat{k}.put\ v\ (c, r)) \\ = & skel(k.put\ v\ c) & \text{by definition } \widehat{k}.put \\ = & skel(k.put\ v'\ c) & \text{as } v = v' \\ = & skel(\widehat{k}.put\ v'\ (c, r')) & \text{by definition } \widehat{k}.put \end{array}$$

and obtain the required equality.

▶ **SkelCreate**: Similar to the proof for SKELPUT. □

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

A.7 Lemma: Let $k \in S \overset{C}{\Longleftrightarrow} V$ be a basic lens. Then $\langle k \rangle$ is a matching lens in $\langle S \rangle \overset{\{\Box\},k}{\Longleftrightarrow} \langle V \rangle$.

Proof:

▶ **GetPut**: Let $s \in \lfloor \langle S \rangle \rfloor$. By the semantics of chunk-annotated regular expressions we also have $s \in S$. Using this fact, we calculate as follows

$$\begin{array}{ll} & \langle k \rangle.put\ (\langle k \rangle.get\ s)\ (\langle k \rangle.res\ s) \\ = & \langle k \rangle.put\ (k.get\ s)\ (\Box, \{\!|1 \mapsto k.res\ s|\!\}) & \text{by definition } \langle k \rangle.get \text{ and } \langle k \rangle.res \\ = & k.put\ (k.get\ s)\ (\{\!|1 \mapsto k.res\ s|\!\}(1)) & \text{by definition } \langle k \rangle.put \text{ with } 1 \in \mathrm{dom}(\{\!|1 \mapsto k.res\ s|\!\}) \\ = & k.put\ (k.get\ s)\ (k.res\ s) & \text{by definition finite map application} \\ = & s & \text{by GETPUT for } k \end{array}$$

and obtain the required equality.

▶ **PutGet**: Let $v \in \lfloor \langle V \rangle \rfloor$ be a string, $\Box \in \{\Box\}$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource. By the semantics of chunk-annotated regular expressions we also have $v \in V$. Using this fact, we calculate as follows

$$\begin{array}{ll} & \langle k \rangle.get\ (\langle k \rangle.put\ v\ (\Box, r)) \\ = & \begin{cases} k.get\ (k.put\ v\ r(1)) & \text{if } 1 \in \mathrm{dom}(r) \\ k.get\ (k.create\ v) & \text{otherwise} \end{cases} & \text{by the definition of } \langle k \rangle.get \text{ and } \langle k \rangle.put \\ = & v & \text{by PUTGET or CREATEGET for } l \end{array}$$

and obtain the required equality.

▶ **CreateGet**: Similar to the proof for PUTGET.

▶ **GetChunks**: Let $s \in \lfloor \langle S \rangle \rfloor$. We calculate as follows

$$\begin{array}{lll} locs(s) = \{1\} & & \text{by definition } locs \\ \quad = locs(\langle k \rangle.get\ s) & & \text{by definition } locs \end{array}$$

and obtain the required equality.

▶ **ResChunks**: Let $s \in \lfloor\langle S\rangle\rfloor$ be a string, $\square \in \{\square\}$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource such that $(\square, r) = \langle k\rangle.\mathit{res}\ s$. We calculate as follows

$$\begin{array}{rll} \mathrm{dom}(r) & = \mathrm{dom}(\{\!|1 \mapsto k.\mathit{res}\ s|\!\}) & \text{by definition } \langle k\rangle.\mathit{res} \text{ with } r = \langle k\rangle.\mathit{res}\ s \\ & = \{1\} & \text{by definition } \mathrm{dom} \\ & = \mathit{locs}(s) & \text{by definition } \mathit{locs} \end{array}$$

and obtain the required equality.

▶ **ChunkPut**: Let $v \in \lfloor\langle V\rangle\rfloor$ be a string, $\square \in \{\square\}$ a rigid complement, $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource, and $x \in (\mathit{locs}(v) \cap \mathrm{dom}(r))$ a location. As $\mathit{locs}(v) = \{1\}$ we must have that $x = 1$ and $1 \in \mathrm{dom}(r)$. Using these facts and definitions, we calculate as follows

$$\begin{array}{rll} & (\langle k\rangle.\mathit{put}\ v\ (\square, r))[x] & \\ = & \langle k\rangle.\mathit{put}\ v\ (c, r) & \text{by definition } [\cdot] \text{ and } x = 1 \\ = & k.\mathit{put}\ v\ (r(1)) & \text{by definition } \langle k\rangle.\mathit{put} \text{ and as } 1 \in \mathrm{dom}(r) \\ = & k.\mathit{put}\ (v[x])\ (r(x)) & \text{by definition } [\cdot] \text{ and } x = 1 \end{array}$$

and obtain the required equality.

▶ **ChunkCreate**: Similar to the proof of CHUNKPUT.

▶ **NoChunkPut**: Let $v \in \lfloor\langle V\rangle\rfloor$ be a string, $\square \in \{\square\}$ a rigid complement, $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource, and $x \in (\mathit{locs}(v) \setminus \mathrm{dom}(r))$ a location. As $\mathit{locs}(v) = \{1\}$ we must have that $x = 1$ and $1 \notin \mathrm{dom}(r)$. Using these facts and definitions, we calculate as follows

$$\begin{array}{rll} & (\langle k\rangle.\mathit{put}\ v\ (\square, r))[x] & \\ = & \langle k\rangle.\mathit{put}\ v\ (c, r) & \text{by definition } [\cdot] \text{ and } x = 1 \\ = & k.\mathit{create}\ v & \text{by definition } \langle k\rangle.\mathit{put} \text{ and as } 1 \notin \mathrm{dom}(r) \\ = & k.\mathit{create}\ (v[x]) & \text{by definition } [\cdot] \text{ and } x = 1 \end{array}$$

and obtain the required equality.

▶ **NoChunkCreate**: Similar to the proof for NOCHUNKPUT.

▶ **SkelPut**: Let $v \in \lfloor\langle V\rangle\rfloor$ and $v' \in \lfloor\langle V\rangle\rfloor$ be strings, $\square \in \{\square\}$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ and $r' \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ resources such that $\mathit{skel}(v) = \mathit{skel}(v)$. We calculate as follows

$$\begin{array}{rll} \mathit{skel}(\langle k\rangle.\mathit{put}\ v\ (\square, r)) & = \square & \text{by definition } \mathit{skel} \\ & = \mathit{skel}(\langle k\rangle.\mathit{put}\ v'\ (\square, r')) & \text{by definition } \mathit{skel} \end{array}$$

and obtain the required equality.

▶ **SkelCreate**: Similar to the proof for SKELPUT, which completes the proof. □

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

A.8 Lemma: Let $l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2$ be matching lenses such that $\lfloor S_1\rfloor \cdot^! \lfloor S_2\rfloor$ and $\lfloor V_1\rfloor \cdot^! \lfloor V_2\rfloor$. Then $(l_1 \cdot l_2)$ is a matching lens in $(S_1 \cdot S_2) \overset{(C_1 \times C_2),k}{\Longleftrightarrow} (V_1 \cdot V_2)$.

Proof:

► **GetPut**: Let $s \in \lfloor S_1{\cdot}S_2 \rfloor$. As $\lfloor S_1 \rfloor {\cdot}^{!} \lfloor S_2 \rfloor$ there exist unique strings $s_1 \in \lfloor S_1 \rfloor$ and $s_2 \in \lfloor S_2 \rfloor$ such that $s = (s_1{\cdot}s_2)$. Using this fact, we calculate as follows

$$
\begin{array}{ll}
\quad (l_1{\cdot}l_2).\mathit{put}\ ((l_1{\cdot}l_2).\mathit{get}\ s)\ ((l_1{\cdot}l_2).\mathit{res}\ s) & \\
= (l_1{\cdot}l_2).\mathit{put}\ ((l_1{\cdot}l_2).\mathit{get}\ (s_1{\cdot}s_2))\ ((l_1{\cdot}l_2).\mathit{res}\ (s_1{\cdot}s_2)) & \text{by definition } s_1 \text{ and } s_2 \\
= (l_1{\cdot}l_2).\mathit{put}\ ((l_1.\mathit{get}\ s_1){\cdot}(l_2.\mathit{get}\ s_2))\ ((c_1, c_2), r_1 +\!\!+ r_2) & \text{by definition } (l_1{\cdot}l_2).\mathit{get} \text{ and } (l_1{\cdot}l_2).\mathit{res} \\
\qquad \text{where } c_1, r_1 = l_1.\mathit{res}\ s_1 & \\
\qquad\quad \text{and } c_2, r_2 = l_2.\mathit{res}\ s_2 & \\
= (l_1.\mathit{put}\ (l_1.\mathit{get}\ s_1)\ (c_1, r_1')){\cdot}(l_2.\mathit{put}\ (l_2.\mathit{get}\ s_2)\ (c_2, r_2')) & \text{by definition } (l_1{\cdot}l_2).\mathit{put} \text{ with } \lfloor V_1 \rfloor {\cdot}^{!} \lfloor V_2 \rfloor \\
\qquad \text{where } r_1', r_2' = \mathit{split}(|l.\mathit{get}\ s_1|, r_1 +\!\!+ r_2) & \text{and } \mathrm{cod}(l_1.\mathit{get}) = \lfloor V_1 \rfloor \text{ and } \mathrm{cod}(l_2.\mathit{get}) = \lfloor V_2 \rfloor \\
= (l_1.\mathit{put}\ (l_1.\mathit{get}\ s_1)\ (c_1, r_1)){\cdot}(l_2.\mathit{put}\ (l_2.\mathit{get}\ s_2)\ (c_2, r_2)) & \text{by GetChunks and ResChunks for } l_1 \\
& \text{and definition } \mathit{split} \\
= (s_1{\cdot}s_2) & \text{by GetPut for } l_1 \text{ and } l_2 \\
= s & \text{by definition } s_1 \text{ and } s_2
\end{array}
$$

and obtain the required equality.

► **PutGet**: Let $v \in \lfloor V_1{\cdot}V_2 \rfloor$ and $(c_1, c_2) \in (C_1 \times C_2)$ and $r \in \{\!|\mathbb{N} \mapsto k.C'|\!\}$. As $\lfloor V_1 \rfloor {\cdot}^{!} \lfloor V_2 \rfloor$ there exist unique strings $v_1 \in \lfloor V_1 \rfloor$ and $v_2 \in \lfloor V_2 \rfloor$ such that $v = v_1{\cdot}v_2$. Using this fact, we calculate as follows

$$
\begin{array}{ll}
\quad (l_1{\cdot}l_2).\mathit{get}\ ((l_1{\cdot}l_2).\mathit{put}\ v\ ((c_1, c_2), r)) & \\
= (l_1{\cdot}l_2).\mathit{get}\ ((l_1{\cdot}l_2).\mathit{put}\ (v_1{\cdot}v_2)\ ((c_1, c_2), r)) & \text{by definition } v_1 \text{ and } v_2 \\
= (l_1{\cdot}l_2).\mathit{get}\ ((l_1.\mathit{put}\ v_1\ (c_1, r_1)){\cdot}(l_2.\mathit{put}\ v_2\ (c_2, r_2))) & \text{by definition } (l_1{\cdot}l_2).\mathit{put} \\
\qquad \text{where } r_1, r_2 = \mathit{split}(|v_1|, r) & \\
= (l_1.\mathit{get}\ (l_1.\mathit{put}\ v_1\ (c_1, r_1))){\cdot}(l_2.\mathit{get}\ (l_2.\mathit{put}\ v_2\ (c_2, r_2))) & \text{by definition } (l_1{\cdot}l_2).\mathit{get} \text{ with } \lfloor S_1 \rfloor {\cdot}^{!} \lfloor S_2 \rfloor \\
& \text{and } \mathrm{cod}(l_1.\mathit{put}) = \lfloor S_1 \rfloor \text{ and } \mathrm{cod}(l_2.\mathit{put}) = \lfloor S_2 \rfloor \\
= (v_1{\cdot}v_2) & \text{by PutGet for } l_1 \text{ and } l_2 \\
= v & \text{by definition } v_1 \text{ and } v_2
\end{array}
$$

and obtain the required equality.

► **CreateGet**: Similar to the proof for PutGet, using CreateGet for l_1 and l_2 instead of PutGet.

► **GetChunks**: Let $s \in \lfloor S_1{\cdot}S_2 \rfloor$. As $\lfloor S_1 \rfloor {\cdot}^{!} \lfloor S_2 \rfloor$ there exist unique strings $s_1 \in \lfloor S_1 \rfloor$ and $s_2 \in \lfloor S_2 \rfloor$ such that $s = s_1{\cdot}s_2$. Using this fact, we calculate as follows

$$
\begin{array}{ll}
\quad \mathit{locs}(s) & \\
= \mathit{locs}(s_1{\cdot}s_2) & \text{by definition } s_1 \text{ and } s_2 \\
= \{1, \ldots, (|s_1| + |s_2|)\} & \text{by definition } \mathit{locs} \\
= \{1, \ldots, (|l_1.\mathit{get}\ s_1| + |l_2.\mathit{get}\ s_2|)\} & \text{by GetChunks for } l_1 \text{ and } l_2 \\
= \mathit{locs}((l_1.\mathit{get}\ s_1){\cdot}(l_2.\mathit{get}\ s_2)) & \text{by definition } \mathit{locs} \\
= \mathit{locs}((l_1{\cdot}l_2).\mathit{get}\ (s_1{\cdot}s_2)) & \text{by definition } (l_1{\cdot}l_2).\mathit{get} \\
= \mathit{locs}((l_1{\cdot}l_2).\mathit{get}\ s) & \text{by definition } s_1 \text{ and } s_2
\end{array}
$$

and obtain the required equality.

► **ResChunks**: Let $s \in \lfloor S_1{\cdot}S_2 \rfloor$ be a string, $(c_1, c_2) \in (C_1 \times C_2)$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource with $((c_1, c_2), r) = (l_1{\cdot}l_2).\mathit{res}\ s$. As $\lfloor S_1 \rfloor {\cdot}^{!} \lfloor S_2 \rfloor$ there exist unique strings $s_1 \in \lfloor S_1 \rfloor$ and $s_2 \in \lfloor S_2 \rfloor$ such

that $s = (s_1 \cdot s_2)$. Using this fact, we calculate as follows

$$
\begin{array}{lll}
& \mathrm{dom}(r) & \\
= & \mathrm{dom}(r_1 \mathbin{+\!\!+} r_2) & \text{by definition } r \text{ and } (l_1 \cdot l_2).\mathit{res} \\
& \quad \text{where } r_1, c_1 = l_1.\mathit{res}\ s_1 & \\
& \quad\ \ \text{and } r_2, c_2 = l_2.\mathit{res}\ s_2 & \\
= & \mathrm{dom}(r_1) \cup \{i + \max(\mathrm{dom}(r_1)) \mid i \in \mathrm{dom}(r_2)\} & \text{by definition } (\mathbin{+\!\!+}) \text{ and } \mathrm{dom} \\
= & (\mathit{locs}(s_1)) \cup \{i + \max(\mathit{locs}(s_1)) \mid i \in (\mathit{locs}(s_2))\} & \text{by RESCHUNKS for } l_1 \text{ and } l_2 \\
= & \{1, \ldots, (|s_1| + |s_2|)\} & \text{by definition } |\cdot| \\
= & \mathit{locs}(s_1 \cdot s_2) & \text{by definition } \mathit{locs} \\
= & \mathit{locs}(s) & \text{by definition } s_1 \text{ and } s_2
\end{array}
$$

and obtain the required equality.

▶ **ChunkPut**: Let $v \in \lfloor V_1 \cdot V_2 \rfloor$ and $(c_1, c_2) \in (C_1 \times C_2)$ and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ and $x \in (\mathit{locs}(v) \cap \mathrm{dom}(r))$. As $\lfloor V_1 \rfloor \cdot^! \lfloor V_2 \rfloor$ there exist unique strings $v_1 \in \lfloor V_1 \rfloor$ and $v_2 \in \lfloor V_2 \rfloor$ such that $v = (v_1 \cdot v_2)$. We analyze two cases.

Case $x \in \mathit{locs}(v_1)$**:** We calculate as follows

$$
\begin{array}{lll}
& ((l_1 \cdot l_2).\mathit{put}\ v\ ((c_1, c_2), r))[x] & \\
= & ((l_1 \cdot l_2).\mathit{put}\ (v_1 \cdot v_2)\ ((c_1, c_2), r))[x] & \text{by definition } v_1 \text{ and } v_2 \\
= & ((l_1.\mathit{put}\ v_1\ (c_1, r_1)) \cdot (l_2.\mathit{put}\ v_2\ (c_2, r_2)))[x] & \text{by definition } (l_1 \cdot l_2).\mathit{put} \\
& \quad \text{where } r_1, r_2 = \mathit{split}(|v_1|, r) & \\
= & (l_1.\mathit{put}\ v_1\ (c_1, r_1))[x] & \text{by Lemma A.1 and definition } [\cdot] \\
= & k.\mathit{put}\ (v_1[x])\ (r_1(x)) & \text{by CHUNKPUT for } l_1 \\
= & k.\mathit{put}\ ((v_1 \cdot v_2)[x])\ ((r_1 \mathbin{+\!\!+} r_2)(x)) & \text{by definition } [\cdot] \text{ and } (\mathbin{+\!\!+}) \text{ and finite map application} \\
= & k.\mathit{put}\ (v[x])\ (r(x)) & \text{by definition } \mathit{split} \text{ and } v_1 \text{ and } v_2 \text{ and } r_1 \text{ and } r_2
\end{array}
$$

and obtain the required equality.

Case $x \notin (\mathit{locs}(v_1))$**:** Similar to the previous case, using CHUNKPUT for l_2 instead of l_1.

▶ **ChunkCreate**: Similar to the proof for CHUNKPUT.

▶ **NoChunkPut**: Similar to the proof for CHUNKPUT.

▶ **NoChunkCreate**: Similar to the proof for CHUNKPUT.

▶ **SkelPut**: Let $v \in \lfloor V_1 \cdot V_2 \rfloor$ and $v' \in \lfloor V_1 \cdot V_2 \rfloor$ be strings, $(c_1, c_2) \in (C_1 \times C_2)$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ and $r' \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ resources such that $\mathit{skel}(v) = \mathit{skel}(v')$. As $\lfloor V_1 \rfloor \cdot^! \lfloor V_2 \rfloor$ there exist unique strings $v_1 \in \lfloor V_1 \rfloor$ and $v_2 \in \lfloor V_2 \rfloor$ and $v_1' \in \lfloor V_1 \rfloor$ and $v_2' \in \lfloor V_2 \rfloor$ such that $v = (v_1 \cdot v_2)$ and $v' = (v_1' \cdot v_2')$. Moreover, using the definition of skel we have that $\mathit{skel}(v_1) = \mathit{skel}(v_1')$ and $\mathit{skel}(v_2) = \mathit{skel}(v_2')$. Using these facts and definitions, we calculate as follows

$$
\begin{array}{lll}
& \mathit{skel}((l_1 \cdot l_2).\mathit{put}\ v\ (c_1, c_2, r)) & \\
= & \mathit{skel}((l_1 \cdot l_2).\mathit{put}\ (v_1 \cdot v_2)\ ((c_1, c_2), r)) & \text{by definition } v_1 \text{ and } v_2 \\
= & \mathit{skel}(l_1.\mathit{put}\ v_1\ (c_1, r_1)) \cdot (l_2.\mathit{put}\ v_2\ (c_2, r_2)) & \text{by definition } (l_1 \cdot l_2).\mathit{put} \\
& \quad \text{where } r_1, r_2 = \mathit{split}(|v_1|, r) & \\
= & \mathit{skel}(l_1.\mathit{put}\ v_1\ (c_1, r_1)) \cdot \mathit{skel}(l_2.\mathit{put}\ v_2\ (c_2, r_2)) & \text{by definition } \mathit{skel} \\
= & \mathit{skel}(l_1.\mathit{put}\ v_1'\ (c_1, r_1')) \cdot \mathit{skel}(l_2.\mathit{put}\ v_2'\ (c_2, r_2')) & \text{by SKELPUT for } l_1 \text{ and } l_2 \\
& \quad \text{where } r_1', r_2' = \mathit{split}(|v_1'|, r') & \\
= & \mathit{skel}(l_1.\mathit{put}\ v_1'\ (c_1, r_1')) \cdot (l_2.\mathit{put}\ v_2'\ (c_2, r_2')) & \text{by definition } \mathit{skel} \\
= & \mathit{skel}((l_1 \cdot l_2).\mathit{put}\ (v_1' \cdot v_2')\ ((c_1, c_2), r')) & \text{by definition } (l_1 \cdot l_2).\mathit{put} \text{ and } r_1' \text{ and } r_2' \\
= & \mathit{skel}((l_1 \cdot l_2).\mathit{put}\ v'\ ((c_1, c_2), r)) & \text{by definition } v_1' \text{ and } v_2'
\end{array}
$$

and obtain the required equality.

▶ **SkelCreate**: Similar to the proof for SKELPUT, which completes the proof. □

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

A.9 Lemma: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a matching lens such that $\lfloor S \rfloor^{!*}$ and $\lfloor V \rfloor^{!*}$. Then l^* is a matching lens in $S^* \overset{K\ list,R}{\Longleftrightarrow} V^*$.

Proof:

▶ **GetPut**: Let $s \in \lfloor S^* \rfloor$. As $\lfloor S \rfloor^{!*}$ there exist unique strings $s_1 \in \lfloor S \rfloor$ to $s_n \in \lfloor S \rfloor$ such that $s = (s_1 \cdots s_n)$. To shorten the proof, let $(c_i, r_i) = l.\mathit{res}\ s_i$ for $i \in \{1, \ldots, n\}$ and $r = (r_1 +\!\!+ \ldots +\!\!+ r_n)$. Also let $r''_0 = r$ and $(r'_i, r''_i) = \mathit{split}(|l.\mathit{get}\ s_i|,)r''_i$ for $i \in \{1, \ldots, n\}$. Using these facts and definitions, we calculate as follows

$$\begin{array}{ll}
\quad l^*.\mathit{put}\ (l^*.\mathit{gets})\ (l^*.\mathit{res}\ s) & \\
= l^*.\mathit{put}\ (l^*.\mathit{get}(s_1 \cdots s_n))\ (l^*.\mathit{res}\ (s_1 \cdots s_n)) & \text{by definition } s_1 \text{ to } s_n \\
= l^*.\mathit{put}\ ((l.\mathit{get}\ s_1) \cdots (l.\mathit{get}\ s_n))\ ([c_1, \ldots, c_n], r) & \text{by definition } l^*.\mathit{get} \text{ and } l^*.\mathit{res} \\
= (l.\mathit{put}\ (l.\mathit{get}\ s_1)\ (c_1, r'_1)) \cdots (l.\mathit{put}\ (l.\mathit{get}\ s_n)\ (c_n, r'_n)) & \text{by definition } l^*.\mathit{put} \text{ with } \lfloor V \rfloor^{!*} \text{ and } \mathrm{cod}(l.\mathit{get}) = \lfloor V \rfloor \\
= (l.\mathit{put}\ (l.\mathit{get}\ s_1)\ (c_1, r_1)) \cdots (l.\mathit{put}\ (l.\mathit{get}\ s_n)\ (c_n, r_n)) & \text{by GetChunks and ResChunks for } l \\
& \text{and definition } \mathit{split} \\
= (s_1 \cdots s_n) & \text{by GetPut for } l \\
= s & \text{by definition } s_1 \text{ to } s_n
\end{array}$$

and obtain the required equality.

▶ **PutGet**: Let $v \in \lfloor V^* \rfloor$ be a string, $[c_1, \ldots, c_m] \in C$ list a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource. As $\lfloor V \rfloor^{!*}$ there exist unique strings $v_1 \in \lfloor V \rfloor$ to $v_n \in \lfloor V \rfloor$ such that $v = (v_1 \cdots v_n)$. Let $r'_0 = r$ and $(r_i, r'_i) = \mathit{split}(|v_i|, r'_{(i-1)})$ for $i \in \{1, \ldots, n\}$. Using these facts and definitions, we calculate as follows

$$\begin{array}{ll}
\quad l^*.\mathit{get}\ (l^*.\mathit{put}\ v\ ([c_1, \ldots, c_m], r)) & \\
= l^*.\mathit{get}\ (l^*.\mathit{put}\ (v_1 \cdots v_n)\ ([c_1, \ldots, c_m], r)) & \text{by definition of } v_1 \text{ to } v_n \\
= l^*.\mathit{get}\ (s'_1 \cdots s'_n) & \text{by the definition of } l^*.\mathit{put} \\
\qquad \text{where } s'_i = \begin{cases} l.\mathit{put}\ v_i\ (c_i, r_i) & i \in \{1, \ldots, \min(n, m)\} \\ l.\mathit{create}\ v_i\ r_i & i \in \{m+1, \ldots, n\} \end{cases} & \\
= (l.\mathit{get}\ s'_1) \cdots (l.\mathit{get}\ s'_n) & \text{by the definition of } l^*.\mathit{get} \text{ with } \lfloor V \rfloor^{!*} \\
& \text{and } \mathrm{cod}(l.\mathit{put}) = \mathrm{cod}(l.\mathit{create}) = \lfloor V \rfloor \\
= (v_1 \cdots v_n) & \text{by PutGet and CreateGet for } l \\
= v & \text{by the definition of } v_1 \text{ to } v_n
\end{array}$$

and obtain the required equality.

▶ **CreateGet**: Similar to the proof for PutGet, using CreateGet for l.

▶ **GetChunks**: Let $s \in \lfloor S^* \rfloor$. As $\lfloor S \rfloor^{!*}$ there exist unique strings $s_1 \in \lfloor S \rfloor$ to $s_n \in \lfloor S \rfloor$ such that $s = (s_1 \cdots s_n)$. Using these facts, we calculate as follows

$$\begin{array}{ll}
\quad \mathit{locs}(s) & \\
= \mathit{locs}(s_1 \cdots s_n) & \text{by definition } s_1 \text{ to } s_n \\
= \{1, \ldots, \textstyle\sum_{i=1}^{n} |s_i|\} & \text{by definition } \mathit{locs} \\
= \{1, \ldots, \textstyle\sum_{i=1}^{n} |l.\mathit{get}\ s_i|\} & \text{by GetChunks for } l \\
= \mathit{locs}((l.\mathit{get}\ s_1) \cdots (l.\mathit{get}\ s_n)) & \text{by definition } \mathit{locs} \text{ with } \lfloor V \rfloor^{!*} \text{ and } \mathrm{cod}(l.\mathit{get}) = \lfloor V \rfloor \\
= \mathit{locs}(l^*.\mathit{get}\ (s_1 \cdots s_n)) & \text{by definition } l^*.\mathit{get} \\
= \mathit{locs}(l^*.\mathit{get}\ s) & \text{by definition } s_1 \text{ to } s_n
\end{array}$$

and obtain the required equality.

▶ **ResChunks**: Let $s \in \lfloor S^* \rfloor$ be a string, $c \in (C\ list)$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource such that $c, r) = l^*.res\ s$. As $S^{!*}$ there exist unique strings $s_1 \in \lfloor S \rfloor$ to $s_n \in \lfloor S \rfloor$ such that $s = (s_1 \cdots s_n)$. To shorten the proof, let $(c_i, r_i) = l.res\ s_i$ for $i \in \{1, \ldots, n\}$. Using these fact and definitions, we calculate as follows

$$
\begin{array}{ll}
\quad \mathrm{dom}(r) & \\
= \mathrm{dom}(r_1 +\!\!+ \ldots +\!\!+ r_n) & \text{by definition } l^*.res \\
= \bigcup_{i=1}^{n} \{j + \sum_{k=1}^{(i-1)} \max(\mathrm{dom}(r_k)) \mid j \in \mathrm{dom}(r_i)\} & \text{by definition } (+\!\!+) \text{ and } \mathrm{dom} \\
= \bigcup_{i=1}^{n} \{j + \sum_{k=1}^{(i-1)} \max(locs(l.get\ x_k)) \mid j \in locs(l.get\ s_i)\} & \text{by ResChunks for } l \\
= \{1, \ldots, \sum i = 1^n\ |l.get\ s_i|\} & \text{by definition } |\cdot| \\
= locs((l.get\ s_1) \cdots (l.get\ s_n)) & \text{by definition } locs \\
= locs(l^*.get\ (s_1 \cdots s_n)) & \text{by definition } l^*.get \\
= locs(l^*.get\ s) & \text{by definition } s_1 \text{ to } s_n
\end{array}
$$

and obtain the required equality.

▶ **ChunkPut**: Let $v \in \lfloor V^* \rfloor$ be a string, $[c_1, \ldots, c_m] \in (C\ list)$ a rigid complement, $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource, and $x \in (locs(v) \cap \mathrm{dom}(r))$ a location. As $\lfloor V \rfloor^{!*}$ there exist unique strings $v_1 \in \lfloor V \rfloor$ to $v_n \in \lfloor V \rfloor$ such that $v = (v_1 \cdots v_n)$. To shorten the proof, let $r'_0 = r$ and $(r_i, r'_i) = split(|v_i|, r'_{(i-1)})$ for $i \in \{1, \ldots, n\}$. We analyze several cases.

Case $x \in locs(v_1)$**:** We calculate as follows

$$
\begin{array}{ll}
\quad (l^*.put\ v\ ([c_1, \ldots, c_m], r))[x] & \\
= (l^*.put\ (v_1 \cdots v_n)\ ([c_1, \ldots, c_m], r))[x] & \text{by definition } v_1 \text{ to } v_n \\
= (s'_1 \cdots s'_n)[x] & \text{by definition } l^*.put \\
\qquad \text{where } s'_i = \begin{cases} l.put\ v_i\ (c_i, r_i) & i \in \{1, \ldots, \min(n, m)\} \\ l.create\ v_i\ r_i & i \in \{m+1, \ldots, n\} \end{cases} & \\
= s'_1[x] & \text{by Lemmas A.1 and A.2 and definition } [\cdot] \\
= k.put\ v_1\ (r_1(x)) & \text{by ChunkPut and ChunkCreate for } l \\
= k.put\ (v_1 \cdots v_n)[x]\ ((r_1 +\!\!+ \ldots +\!\!+ r_n)(x)) & \text{by definition } [\cdot] \text{ and } (+\!\!+) \\
& \text{and finite map application} \\
= k.put\ v[x]\ (r(x)) & \text{by definition } split \text{ and } v_1 \text{ to } v_n \text{ and } r_1 \text{ to } r_n
\end{array}
$$

and obtain the required equality.

Case $x \notin locs(v_1)$**:** Similar to the previous case.

▶ **ChunkCreate**: Similar to the proof for ChunkPut.

▶ **NoChunkPut**: Similar to the proof for ChunkPut.

▶ **NoChunkCreate**: Similar to the proof for ChunkPut.

▶ **SkelPut**: Let $v \in \lfloor V^* \rfloor$ and $v' \in \lfloor V^* \rfloor$ be strings, $[c_1, \ldots, c_m] \in (C\ list)$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ and $r' \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ resources such that $skel(v) = skel(v')$. As $\lfloor V \rfloor^{!*}$ there exist unique strings $v_1 \in \lfloor V \rfloor$ to $v_n \in \lfloor V \rfloor$ and $v'_1 \in \lfloor V \rfloor$ to $v'_n \in \lfloor V \rfloor$ such that $v = (v_1 \cdots v_n)$ and $v' = (v_1 \cdots v'_o)$. Moreover, by the definition of $skel$ we have that $n = o$ and $skel(v_i) = skel(v'_i)$ for $i \in \{1, \ldots, n\}$. To shorten the proof, let

$$
\begin{array}{llll}
r'_0 = r & (r_i, r'_i) = split(|v_i|, r'_{(i-1)}) & \text{for } i \in \{1, \ldots, n\} \\
r''_0 = r' & (r''_i, r'''_i) = split(|v'_i|, r'''_{(i-1)}) & \text{for } i \in \{1, \ldots, o\}
\end{array}
$$

Using these facts and definitions, we calculate as follows

$$
\begin{array}{ll}
\quad \mathit{skel}(l^*.\mathit{put}\ v\ ([c_1,\ldots,c_m],r)) & \\
= \mathit{skel}(l^*.\mathit{put}\ (v_1 \cdots v_n)\ ([c_1,\ldots,c_m],r)) & \text{by definition } v_1 \text{ to } v_n \\
= \mathit{skel}(s'_1 \cdots s'_n) & \text{by the definition of } l^*.\mathit{put} \\
\qquad \text{where } s'_i = \begin{cases} l.\mathit{put}\ v_i\ (c_i,r_i) & i \in \{1,\ldots,\min(n,m)\} \\ l.\mathit{create}\ v_i\ r_i & i \in \{m+1,\ldots,n\} \end{cases} & \\
= (\mathit{skel}(s'_1)) \cdots (\mathit{skel}(s'_n)) & \text{by the definition of } \mathit{skel} \\
= (\mathit{skel}(s''_1)) \cdots (\mathit{skel}(s''_n)) & \text{by SKELPUT and SKELCREATE for } l \\
\qquad \text{where } s''_i = \begin{cases} l.\mathit{put}\ v'_i\ (c_i,r''_i) & i \in \{1,\ldots,\min(n,m)\} \\ l.\mathit{create}\ v'_i\ r''_i & i \in \{m+1,\ldots,n\} \end{cases} & \\
= \mathit{skel}(s''_1 \cdots s''_n) & \text{by the definition of } \mathit{skel} \\
= \mathit{skel}(l^*.\mathit{put}\ (v'_1 \cdots v'_n)\ ([c_1,\ldots,c_m],r')) & \text{by definition } l^*.\mathit{put} \text{ and } r''_1 \text{ to } r''_n \\
= \mathit{skel}(l^*.\mathit{put}\ v'\ ([c_1,\ldots,c_m],r')) & \text{by definition } v'_1 \text{ to } v'_n
\end{array}
$$

and obtain the required equality.

► **SkelCreate**: Similar to the proof for SKELPUT. □

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

A.10 Lemma: Let $l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2$ be matching lenses such that $\lfloor S_1 \rfloor \cap \lfloor S_2 \rfloor = \emptyset$ and $\lfloor V_1 \rfloor \cap \lfloor V_2 \rfloor \subseteq \lfloor V_1 \cap V_2 \rfloor$. Then $(l_1 \mid l_2)$ is a matching lens in $(S_1 \cup S_2) \overset{(C_1+C_2),k}{\Longleftrightarrow} (V_1 \cup V_2)$.

Proof:

► **GetPut**: Let $s \in \lfloor S_1 \cup S_2 \rfloor$. We analyze two cases.

Case $s \in \lfloor S_1 \rfloor$**:** We calculate as follows

$$
\begin{array}{ll}
\quad (l_1 \mid l_2).\mathit{put}\ ((l_1 \mid l_2).\mathit{get}\ s)\ ((l_1 \mid l_2).\mathit{res}\ s) & \\
= (l_1 \mid l_2).\mathit{put}\ (l_1.\mathit{get}\ s)\ (\mathit{Inl}(c_1),r) & \text{by definition } (l_1 \mid l_2).\mathit{get} \text{ and } (l_1 \mid l_2).\mathit{res} \text{ with } s \in \lfloor S_1 \rfloor \\
\qquad \text{where } c_1, r = l.\mathit{res}\ s & \\
= l_1.\mathit{put}\ (l_1.\mathit{get}\ s)\ (c_1,r) & \text{by the definition of } (l_1 \mid l_2).\mathit{put} \text{ with } \mathrm{cod}(l.\mathit{get}) = \lfloor V_1 \rfloor \\
= l_1.\mathit{put}\ (l_1.\mathit{get}\ s)\ (l_1.\mathit{res}\ s) & \text{by the definition of } (c_1,r) \\
= s & \text{by PUTGET for } l_1
\end{array}
$$

and obtain the required equality, which finishes the case.

Case $s \in S_2$**:** Symmetric to the previous case, using l_2 instead of l_1.

► **PutGet**: Let $v \in \lfloor V_1 \cup V_2 \rfloor$ and $c \in (C_1 + C_2)$ and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$. We analyze several cases.

Case $v \in \lfloor V_1 \rfloor$ **and** $c = \mathit{Inl}(c_1)$**:** We calculate as follows

$$
\begin{array}{ll}
\quad (l_1 \mid l_2).\mathit{get}\ ((l_1 \mid l_2).\mathit{put}\ v\ (c,r)) & \\
= (l_1 \mid l_2).\mathit{get}\ (l_1.\mathit{put}\ v\ (c_1,r)) & \text{by definition } (l_1 \mid l_2).\mathit{put} \text{ and the assumptions of the case} \\
= l_1.\mathit{get}\ (l_1.\mathit{put}\ v\ (c_1,r)) & \text{by definition } (l_1 \mid l_2).\mathit{get} \text{ with } \mathrm{cod}(l_1.\mathit{put}) = \lfloor S_1 \rfloor \\
= v & \text{by PUTGET for } l
\end{array}
$$

and obtain the required equality.

Case $v \in \lfloor V_2 \rfloor$ **and** $c = \mathit{Inr}(c_2)$**:** Symmetric to the previous case, using l_2 instead of l_1.
Case $v \notin \lfloor V_2 \rfloor$ **and** $c = \mathit{Inr}(c_2)$**:** We calculate as follows

$$\begin{aligned}
&(l_1 \mid l_2).\mathit{get}\ ((l_1 \mid l_2).\mathit{put}\ v\ (c, r)) \\
&= (l_1 \mid l_2).\mathit{get}\ (l_1.\mathit{create}\ v\ r) && \text{by definition } (l_1 \mid l_2).\mathit{put} \text{ and the assumptions of the case} \\
&= l_1.\mathit{get}\ (l_1.\mathit{create}\ v\ r) && \text{by definition } (l_1 \mid l_2).\mathit{get} \text{ with } \mathrm{cod}(l_1.\mathit{create}) = \lfloor S_1 \rfloor \\
&= v && \text{by CREATEGET for } l
\end{aligned}$$

and obtain the required equality.
Case $v \notin \lfloor V_1 \rfloor$ **and** $c = \mathit{Inl}(c_1)$**:** Symmetric to the previous case, using l_2 instead of l_1.

▶ **CreateGet**: Similar to the proof for PUTGET.

▶ **GetChunks**: Let $s \in \lfloor S_1 \cup S_2 \rfloor$. As $(\lfloor V_1 \rfloor \cap \lfloor V_2 \rfloor) \subseteq \lfloor V_1 \cap V_2 \rfloor$, for every $v \in \lfloor V_1 \cap V_2 \rfloor$ we have that the set of chunks of v specified by V_1 and by V_2 are identical. We analyze two cases.

Case $s \in \lfloor S_1 \rfloor$**:** We calculate as follows

$$\begin{aligned}
\mathit{locs}(s) &= \mathit{locs}(l_1.\mathit{get}\ s) && \text{by GETCHUNKS for } l_1 \\
&= \mathit{locs}((l_1 \mid l_2).\mathit{get}\ s) && \text{by definition } (l_1 \mid l_2).\mathit{get} \text{ with } s \in \lfloor S_1 \rfloor
\end{aligned}$$

and obtain the required equality.
Case $s \in \lfloor S_2 \rfloor$**:** Symmetric to the previous case, using l_2 instead of l_1.

▶ **ResChunks**: Let $s \in \lfloor S_1 \cup S_2 \rfloor$ be a string, $c \in (C_1 + C_2)$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource such that $(c, r) = (l_1 \mid l_2).\mathit{res}\ s$. We analyze two cases.

Case $s \in \lfloor S_1 \rfloor$**:** By the assumption of the case and the definition of $(l_1 \mid l_2).\mathit{res}$ we have that $c = \mathit{Inl}(c_1)$ where $c_1, r = l_1.\mathit{res}\ s$. The required equality, $\mathit{locs}(s) = \mathrm{dom}(r)$, is immediate by RESCHUNKS for l_1.
Case $s \in \lfloor S_1 \rfloor$**:** Symmetric to the previous case, using l_2 instead of l_1.

▶ **ChunkPut**: Let $v \in \lfloor V_1 \cup V_2 \rfloor$ be a string, $k \in (C_1 + C_2)$ a rigid complement, $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource, and $x \in (\mathit{locs}(v) \cap \mathrm{dom}(r))$ a location. We analyze several cases.

Case $v \in \lfloor V_1 \rfloor$ **and** $c = \mathit{Inl}(c_1)$**:** As $(\lfloor V_1 \rfloor \cap \lfloor V_2 \rfloor) \subseteq \lfloor V_1 \cap V_2 \rfloor$, we have that x is also a location of a chunk as specified by V_1. Using this fact, we calculate as follows

$$\begin{aligned}
&(l_1 \mid l_2).\mathit{put}\ v\ (c, r)[x] \\
&= l_1.\mathit{put}\ v\ (c_1, r)[x] && \text{by definition } (l_1 \mid l_2).\mathit{put} \text{ with } v \in \lfloor V_1 \rfloor \text{ and } c = \mathit{Inl}(c_1) \\
&= k.\mathit{put}\ (v[x])\ (r(x)) && \text{by CHUNKPUT for } l_1
\end{aligned}$$

and obtain the required equality.
Case $v \in \lfloor V_2 \rfloor$ **and** $c = \mathit{Inr}(c_2)$**:** Symmetric to the previous case, using l_2 instead of l_2.
Case $v \notin \lfloor V_2 \rfloor$ **and** $c = \mathit{Inr}(c_2)$**:** We calculate as follows.

$$\begin{aligned}
&(l_1 \mid l_2).\mathit{put}\ v\ (c, r)[x] \\
&= l_1.\mathit{create}\ v\ r[x] && \text{by definition } (l_1 \mid l_2).\mathit{put} \text{ with } v \notin \lfloor V_2 \rfloor \text{ and } c = \mathit{Inr}(c_2) \\
&= k.\mathit{put}\ (v[x])\ (r(x)) && \text{by CHUNKCREATE for } l_1
\end{aligned}$$

and obtain the required equality.
Case $v \notin \lfloor V_1 \rfloor$ **and** $c = \mathit{Inl}(c_1)$**:** Symmetric to the previous case, using l_1 instead of l_2.

▶ **ChunkCreate**: Similar to the proof for CHUNKPUT.

▶ **NoChunkPut**: Similar to the proof for CHUNKPUT.

▶ **NoChunkCreate**: Similar to the proof for CHUNKPUT.

▶ **SkelPut**: Let $v \in \lfloor V_1 \cup V_2 \rfloor$ and $v' \in \lfloor V_1 \cup V_2 \rfloor$ be strings, $c \in (C_1 + C_2)$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ and $r' \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ resources such that $\mathit{skel}(v) = \mathit{skel}(v')$. We analyze several cases.

Case $v \in \lfloor V_1 \rfloor$ **and** $v' \in \lfloor V_1 \rfloor$ **and** $c = \mathit{Inl}(c_1)$**:** Using the assumptions of the case, we calculate as follows

$$\begin{array}{lll} & \mathit{skel}((l_1 \mid l_2).\mathit{put}\ v\ (c, r)) & \\ = & \mathit{skel}(l_1.\mathit{put}\ v\ (c_1, r)) & \text{by definition } (l_1 \mid l_2).\mathit{put} \text{ with } v \in \lfloor V_1 \rfloor \text{ and } c = \mathit{Inl}(c_1) \\ = & \mathit{skel}(l_1.\mathit{put}\ v'\ (c_1, r')) & \text{by SKELPUT for } l_1 \\ = & \mathit{skel}((l_1 \mid l_2).\mathit{put}\ v'\ (c, r')) & \text{by definition } (l_1 \mid l_2).\mathit{put} \text{ with } v' \in \lfloor V_1 \rfloor \text{ and } c = \mathit{Inl}(c_1) \end{array}$$

and obtain the required equality.

Case $v \in \lfloor V_2 \rfloor$ **and** $v' \in \lfloor V_2 \rfloor$ **and** $c = \mathit{Inr}(c_2)$**:** Symmetric to the previous case, using l_2 instead of l_1.

Case $v \in \lfloor V_1 \rfloor$ **and** $v' \in \lfloor V_1 \rfloor$ **and** $c = \mathit{Inr}(c_2)$**:** Similar to the first case, using SKELCREATE instead of SKELPUT

Case $v \in \lfloor V_2 \rfloor$ **and** $v' \in \lfloor V_2 \rfloor$ **and** $c = \mathit{Inl}(c_1)$**:** Similar to the first case, using SKELCREATE instead of SKELPUT.

Case $v \in \lfloor V_1 \rfloor$ **and** $v' \notin \lfloor V_1 \rfloor$**:** Can't happen. As $\mathit{skel}(v) = \mathit{skel}(v')$, we have the sets of locations $\mathit{locs}(v)$ and $\mathit{locs}(v')$ are identical. Let v'' be the string obtained from v by setting the chunk at every location in $\mathit{locs}(v)$ to the corresponding chunk in v'. By construction, we have $v' = v'$. By chunk compatibility we also have $v'' \in \lfloor V_1 \rfloor$. However, by the assumptions of the case, we have $v' \notin \lfloor V_1 \rfloor$, which is a contradiction.

Case $v \in \lfloor V_2 \rfloor$ **and** $v' \notin \lfloor V_2 \rfloor$**:** Symmetric to the previous case.

► **SkelCreate**: Similar to the proof for SKELPUT, which completes the proof. □

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

A.11 Lemma: Let $l_1 \in S \xleftrightarrow{C_1,k_1} U$ and $l_2 \in U \xleftrightarrow{C_2,k_2} V$ be matching lenses. Then $(l_1;l_2)$ is a matching lens in $S \xleftrightarrow{(C_1 \otimes C_2),(k_1;k_2)} V$.

Proof:

► **GetPut**: Let $s \in \lfloor S \rfloor$. We calculate as follows

$$\begin{array}{lll} & (l_1;l_2).\mathit{put}\ ((l_1;l_2).\mathit{get}\ s)\ ((l_1;l_2).\mathit{res}\ s) & \\ = & (l_1;l_2).\mathit{put}\ (l_2.\mathit{get}\ (l_1.\mathit{get}\ s))\ (\langle c_1, c_2 \rangle, \mathit{zip}\ r_1\ r_2) & \text{by definition } (l_1;l_2).\mathit{get} \text{ and } (l_1;l_2).\mathit{res} \\ & \quad \text{where } c_1, r_1 = l_1.\mathit{res}\ s & \\ & \quad\ \text{and } c_2, r_2 = l_2.\mathit{res}\ (l_1.\mathit{get}\ s) & \\ = & l_1.\mathit{put}\ (l_2.\mathit{put}\ (l_2.\mathit{get}\ (l_1.\mathit{get}\ s))\ (c_2, r_2'))\ (c_1, r_1') & \text{by definition } (l_1;l_2).\mathit{put} \\ & \quad \text{with } r_1', r_2' = \mathit{unzip}\ (\mathit{zip}\ r_1\ r_2) & \\ = & l_1.\mathit{put}\ (l_2.\mathit{put}\ (l_2.\mathit{get}\ (l_1.\mathit{get}\ s))\ (c_2, r_2))\ (c_1, r_1) & \text{as } \mathit{unzip}(\mathit{zip}\ r_1\ r_2) = r_1, r_2 \\ = & l_1.\mathit{put}\ (l_2.\mathit{put}\ (l_2.\mathit{get}\ (l_1.\mathit{get}\ s))\ (l_2.\mathit{res}\ (l_1.\mathit{get}\ s)))\ (l_1.\mathit{res}\ s) & \text{by definition } (c_1, r_1) \text{ and } (c_2, r_2) \\ = & l_1.\mathit{put}\ (l_1.\mathit{get}\ s)\ (l_1.\mathit{res}\ s) & \text{by GETPUT for } l_2 \\ = & s & \text{by GETPUT for } l_1 \end{array}$$

and obtain the required equality.

► **PutGet**: Let $v \in V$ and $\langle c_1, c_2 \rangle \in (C_1 \otimes C_2)$ and $r \in \{\!|\mathbb{N} \mapsto (k_1;k_2).C|\!\}$. We calculate as follows

$$\begin{array}{lll} & (l_1;l_2).\mathit{get}\ ((l_1;l_2).\mathit{put}\ v\ (\langle c_1, c_2 \rangle, r)) & \\ = & (l_1;l_2).\mathit{get}\ (l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (c_2, r_2))\ (c_1, r_1)) & \text{by definition } (l_1;l_2).\mathit{put} \\ & \quad \text{where } r_1, r_2 = \mathit{unzip}\ r & \\ = & l_2.\mathit{get}\ (l_1.\mathit{get}\ (l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (c_2, r_2))\ (c_1, r_1))) & \text{by definition } (l_1;l_2).\mathit{get} \\ = & l_2.\mathit{get}\ (l_2.\mathit{put}\ v\ (c_2, r_2)) & \text{by PUTGET for } l_1 \\ = & v & \text{by PUTGET for } l_2 \end{array}$$

and obtain the required equality.

▶ **CreateGet**: Similar to the proof for PUTGET, using CREATEGET instead of PUTGET.

▶ **GetChunks**: Let $s \in \lfloor S \rfloor$. We calculate as follows

$$\begin{array}{rll} \mathit{locs}(s) & = \mathit{locs}(l_1.\mathit{get}\ s) & \text{by GETCHUNKS for } l_1 \\ & = \mathit{locs}(l_2.\mathit{get}\ (l_1.\mathit{get}\ s)) & \text{by GETCHUNKS for } l_2 \\ & = \mathit{locs}((l_1;l_2).\mathit{get}\ s) & \text{by definition } (l_1;l_2).\mathit{get} \end{array}$$

and obtain the required equality.

▶ **ResChunks**: Let $s \in \lfloor S \rfloor$ be a string, $(c_1, c_2) \in (C_1 \otimes C_2)$, a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto (k_1;k_2).C|\!\}$ a resource with $(c, r) = (l_1;l_2).\mathit{res}\ s$. The proof goes in three steps.

First, we show that the set of locations in s is equal to the domain of the resource computed from s using $l_1.\mathit{res}$.

$$\begin{array}{rll} \mathit{locs}(s) & = \mathrm{dom}(r_1) & \text{by RESCHUNKS for } l_1 \\ & \quad \text{where } c_1, r_2 = l_1.\mathit{res}\ s & \end{array}$$

Next, we show that the set of locations in s is equal to the domain of the resource computed from $(l_1.\mathit{get}\ s)$ using $l_2.\mathit{res}$.

$$\begin{array}{rll} \mathit{locs}(s) & = \mathit{locs}(l_1.\mathit{get}\ s) & \text{by GETCHUNKS for } l_1 \\ & \quad \mathrm{dom}(r_2) & \text{by RESCHUNKS for } l_2 \\ & \quad \text{where } c_2, r_2 = l_2.\mathit{res}\ (l_1.\mathit{get}\ s) & \end{array}$$

Finally, using these facts, we calculate as follows

$$\begin{array}{rll} & \mathit{locs}(s) & \\ = & \mathrm{dom}(\mathit{zip}\ r_1\ r_2) & \text{by definition } \mathit{zip} \text{ with } \mathrm{dom}(r_1) = \mathit{locs}(s) = \mathrm{dom}(r_2) \\ = & \mathrm{dom}(r) & \text{by definition } (l_1;l_2).\mathit{res} \end{array}$$

and obtain the required equality.

▶ **ChunkPut**: Let $v \in \lfloor V \rfloor$ be a string, $\langle c_1, c_2 \rangle \in (C_1 \otimes C_2)$ a rigid complement, $r \in \{\!|\mathbb{N} \mapsto (k_1;k_2).C|\!\}$ a resource, and $x \in (\mathit{locs}(v) \cap \mathrm{dom}(r))$ a location. We calculate as follows

$$\begin{array}{rll} & ((l_1;l_2).\mathit{put}\ v\ (\langle c_1, c_2 \rangle, r))[x] & \\ = & (l_1.\mathit{put}\ (l_2.\mathit{put}\ (c_2, r_2))\ (c_1, r_1))[x] & \text{by definition } (l_1;l_2).\mathit{put} \\ & \text{where } r_1, r_2 = \mathit{unzip}\ r & \\ = & k_1.\mathit{put}\ ((l_2.\mathit{put}\ v\ (c_2, r_2))[x])\ (r_1(x)) & \text{by CHUNKPUT for } l_1 \\ = & k_1.\mathit{put}\ (k_2.\mathit{put}\ (v[x])\ (r_2(x)))\ (r_1(x)) & \text{by CHUNKPUT for } l_2 \\ = & (k_1;k_2).\mathit{put}\ (v[x])\ \langle r_1(x), r_2(x) \rangle & \text{by definition of } (k_1;k_2).\mathit{put} \\ = & (k_1;k_2).\mathit{put}\ (v[x])\ (r(x)) & \text{by definition } (r_1, r_2) \text{ and } \mathit{unzip} \end{array}$$

and obtain the required equality.

▶ **ChunkCreate**: Similar to the proof for CHUNKPUT.

▶ **NoChunkPut**: Similar to the proof for CHUNKPUT.

▶ **NoChunkCreate**: Similar to the proof for CHUNKPUT.

▶ **SkelPut**: Let $v \in \lfloor V \rfloor$ and $v' \in \lfloor V \rfloor$ be strings $\langle c_1, c_2 \rangle \in C_1 \otimes C_2$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ and $r' \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ resources such that $\mathit{skel}(v) = \mathit{skel}(v')$. To shorten the proof, let r_1 and r_2 and r'_1 and r'_2 be resources, and u and u' be strings defined as follows:

$$\begin{array}{rcl} r_1, r_2 & = & \mathit{unzip}\ r \\ r'_1, r'_2 & = & \mathit{unzip}\ r' \\ u & = & l_2.\mathit{put}\ v\ (c_2, r_2) \\ u' & = & l_2.\mathit{put}\ v'\ (c_2, r'_2) \end{array}$$

Observe that $skel(u) = skel(u')$ by SKELPUT for l_2. Using these facts and definitions, we calculate as follows

$$\begin{array}{ll}
& skel((l_1;l_2).put\ v\ (\langle c_1, c_2\rangle, r)) \\
= & skel(l_1.put\ (l_2.put\ v\ (c_2, r_2))\ (c_1, r_1)) & \text{by definition } (l_1;l_2).put \\
= & skel(l_1.put\ u\ (c_1, r_1)) & \text{by definition } u \\
= & skel(l_1.put\ u'\ (c_1, r_1')) & \text{by SKELPUT for } l_1 \\
= & skel(l_1.put\ (l_2.put\ v'\ (c_2, r_2'))\ (c_1, r_1')) & \text{by definition } u' \\
= & skel((l_1;l_2).put\ v'\ (\langle c_2, c_2\rangle, r')) & \text{by definition } (l_1;l_2).put
\end{array}$$

and obtain the required equality.

► **SkelCreate**: Similar to the proof for SKELPUT, which finishes the proof. □

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

A.12 Lemma: Let $l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2$ be lenses with $\lfloor S_1\rfloor{\cdot}^!\lfloor S_2\rfloor$ and $\lfloor V_1\rfloor{\cdot}^!\lfloor V_2\rfloor$. Then $(l_1 \sim l_2)$ is a matching lens in $(S_1{\cdot}S_2) \overset{(C_2\times C_1),k}{\Longleftrightarrow} (V_2{\cdot}V_1)$.

Proof:

► **GetPut**: Let $s \in \lfloor S_1{\cdot}S_2\rfloor$. As $\lfloor S_1\rfloor{\cdot}^!\lfloor S_2\rfloor$ there exist unique strings $s_1 \in \lfloor S_1\rfloor$ and $s_2 \in \lfloor S_2\rfloor$ such that $s = (s_1{\cdot}s_2)$. Using this fact, we calculate as follows

$$\begin{array}{ll}
& (l_1 \sim l_2).put\ ((l_1 \sim l_2).get\ s)\ ((l_1 \sim l_2).res\ s) \\
= & (l_1 \sim l_2).put\ ((l_1 \sim l_2).get\ (s_1{\cdot}s_2))\ ((l_1 \sim l_2).res\ (s_1{\cdot}s_2)) & \text{by definition } s_1 \text{ and } s_2 \\
= & (l_1 \sim l_2).put\ ((l_2.get\ s_2){\cdot}(l_1.get\ s_1))\ ((c_2, c_1), r_2 +\!\!+ r_1) & \text{by definition } (l_1 \sim l_2).get \text{ and } (l_1 \sim l_2).res \\
& \quad \text{where } c_1, r_1 = l_1.res\ s_1 & \\
& \quad\quad \text{and } c_2, r_2 = l_2.res\ s_2 & \\
= & (l_1.put\ (l_1.get\ s_1)\ (c_1, r_1')){\cdot}(l_2.put\ (l_2.get\ s_2)\ (c_2, r_2')) & \text{by definition } (l_1 \sim l_2).put \text{ with } \lfloor V_2\rfloor{\cdot}^!\lfloor V_1\rfloor \\
& \quad \text{where } r_1', r_2' = split(|v_2|, r_2 +\!\!+ r_1) & \text{and } \mathrm{cod}(l_2.get) = \lfloor V_2\rfloor \text{ and } \mathrm{cod}(l_1.get) = \lfloor V_1\rfloor \\
= & (l_1.put\ (l_1.get\ s_1)\ (c_1, r_1)){\cdot}(l_2.put\ (l_2.get\ s_2)\ (c_2, r_2)) & \text{by GETCHUNKS and RESCHUNKS for } l_1 \\
& & \text{and definition } split \\
= & (l_1.put\ (l_1.get\ s_1)\ (l_1.res\ s_1)){\cdot}(l_2.put\ (l_2.get\ s_2)\ (l_2.res\ s_2)) & \text{by definition } (c_1, r_1) \text{ and } (c_2, r_2) \\
= & (s_1{\cdot}s_2) & \text{by GETPUT for } l_1 \text{ and } l_2 \\
= & s & \text{by definition } s_1 \text{ and } s_2
\end{array}$$

and obtain the required equality.

► **PutGet**: Let $v \in \lfloor V_1{\cdot}V_2\rfloor$ and $(c_2, c_1) \in (C_2 \times C_1)$ and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$. As $\lfloor V_2\rfloor{\cdot}^!\lfloor V_1\rfloor$ there exist unique strings $v_2 \in \lfloor V_2\rfloor$ and $v_1 \in \lfloor V_1\rfloor$ such that $v = (v_2{\cdot}v_1)$. Using this fact, we calculate as follows

$$\begin{array}{ll}
& (l_1 \sim l_2).get\ ((l_1 \sim l_2).put\ v\ ((c_2, c_1), r)) \\
= & (l_1 \sim l_2).get\ ((l_1 \sim l_2).put\ (v_2{\cdot}v_1)\ ((c_2, c_1), r)) & \text{by definition } v_2 \text{ and } v_1 \\
= & (l_1 \sim l_2).get\ ((l_1.put\ v_1\ (c_1, r_1)){\cdot}(l_2.put\ v_2\ (c_2, r_2))) & \text{by definition } (l_1 \sim l_2).put \\
& \quad \text{where } r_2, r_1 = split(|v_2|, r) & \\
= & (l_2.get\ (l_2.put\ v_2\ (c_2, r_2))){\cdot}(l_1.get\ (l_1.put\ v_1\ (c_1, r_1))) & \text{by definition } (l_1 \sim l_2).get \text{ and } \lfloor S_1\rfloor{\cdot}^!\lfloor S_2\rfloor \\
& & \text{with } \mathrm{cod}(l_1.put) = \lfloor S_1\rfloor \text{ and } \mathrm{cod}(l_2.put) = \lfloor S_2\rfloor \\
= & (v_2{\cdot}v_1) & \text{by PUTGET for } l_2 \text{ and } l_1 \\
= & v & \text{by definition } v_2 \text{ and } v_1
\end{array}$$

and obtain the required equality.

▶ **CreateGet**: Similar to the proof for PUTGET, using CREATEGET for l_1 and l_2 instead of PUTGET.

▶ **GetChunks**: Let $s \in \lfloor S_1 \cdot S_2 \rfloor$. As $\lfloor S_1 \rfloor \cdot^! \lfloor S_2 \rfloor$ there exist unique strings $s_1 \in \lfloor S_1 \rfloor$ and $s_2 \in \lfloor S_2 \rfloor$ such that $s = s_1 \cdot s_2$. Using this fact, we calculate as follows

$$
\begin{array}{ll}
& \mathit{locs}(s) \\
= & \mathit{locs}(s_1 \cdot s_2) & \text{by definition } s_1 \text{ and } s_2 \\
= & \{1, \ldots, (|s_1| + |s_2|)\} & \text{by definition } \mathit{locs} \\
= & \{1, \ldots, (|l_1.\mathit{get}\ s_1| + |l_2.\mathit{get}\ s_2|)\} & \text{by GETCHUNKS for } l_1 \text{ and } l_2 \\
= & \mathit{locs}((l_2.\mathit{get}\ s_2) \cdot (l_1.\mathit{get}\ s_1)) & \text{by definition } \mathit{locs} \\
= & \mathit{locs}((l_1 \sim l_2).\mathit{get}\ (s_1 \cdot s_2)) & \text{by definition } (l_1 \sim l_2).\mathit{get} \\
= & \mathit{locs}((l_1 \sim l_2).\mathit{get}\ s) & \text{by definition } s_1 \text{ and } s_2
\end{array}
$$

and obtain the required equality.

▶ **ResChunks**: Let $s \in \lfloor S_1 \cdot S_2 \rfloor$ be a string, $(c_1, c_2) \in (C_1 \times C_2)$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource with $((c_1, c_2), r) = (l_1 \sim l_2).\mathit{res}\ s$. As $\lfloor S_1 \rfloor \cdot^! \lfloor S_2 \rfloor$ there exist unique strings $s_1 \in \lfloor S_1 \rfloor$ and $s_2 \in \lfloor S_2 \rfloor$ such that $s = (s_1 \cdot s_2)$. Using this fact, we calculate as follows

$$
\begin{array}{ll}
& \mathrm{dom}(r) \\
= & \mathrm{dom}(r_2 +\!\!+ r_1) & \text{by definition } r \text{ and } (l_1 \sim l_2).\mathit{res} \\
& \quad \text{where } r_1, c_1 = l_1.\mathit{res}\ s_1 \\
& \quad\ \ \text{and } r_2, c_2 = l_2.\mathit{res}\ s_2 \\
= & \mathrm{dom}(r_2) \cup \{i + \max(\mathrm{dom}(r_2)) \mid i \in \mathrm{dom}(r_1)\} & \text{by definition } (+\!\!+) \text{ and } \mathrm{dom} \\
= & \mathit{locs}(s_2) \cup \{i + \max(\mathit{locs}(s_2)) \mid i \in \mathit{locs}(s_1)\} & \text{by RESCHUNKS for } l_2 \text{ and } l_1 \\
= & \{1, \ldots, (|s_2| + |s_1|)\} & \text{by definition } |\cdot| \\
= & \mathit{locs}(s_1 \cdot s_2) & \text{by definition } \mathit{locs} \\
= & \mathit{locs}(s) & \text{by definition } s_1 \text{ and } s_2
\end{array}
$$

and obtain the required equality.

▶ **ChunkPut**: Let $v \in \lfloor V_2 \cdot V_1 \rfloor$ and $(c_2, c_1) \in (C_2 \times C_1)$ and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ and $x \in (\mathit{locs}(v) \cap \mathrm{dom}(r))$. As $\lfloor V_2 \rfloor \cdot^! \lfloor V_1 \rfloor$ there exist unique strings $v_2 \in \lfloor V_2 \rfloor$ and $v_1 \in \lfloor V_1 \rfloor$ such that $v = (v_2 \cdot v_1)$. To shorten the proof, define the following resources and permutations:

$$
\begin{array}{rcl}
r_2, r_1 & = & \mathit{split}(|v_2|, r) \\
q_2 & = & l_2.\mathit{perm}\ (l_2.\mathit{put}\ v_2\ (c_2, r_2)) \\
q_1 & = & l_1.\mathit{perm}\ (l_1.\mathit{put}\ v_1\ (c_1, r_1)) \\
q & = & (q_2 *\!\!* q_1)
\end{array}
$$

We analyze two cases.

Case $x \in \mathit{locs}(v_2)$**:** Let y be a location satisfying $q(y) = x$. From $x \in \mathit{locs}(v_2)$ and the definition of $(*\!\!*)$, we have that $y > n_1$ and $q(y) = q_2(y - n_1)$ where $n_1 = |l.\mathit{put}\ v_1\ (c_1, r_1)|$. Using these facts, we calculate as follows:

$$
\begin{array}{ll}
& ((l_1 \sim l_2).\mathit{put}\ v\ ((c_2, c_1), r))[y] \\
= & ((l_1 \sim l_2).\mathit{put}\ (v_2 \cdot v_1)\ ((c_2, c_1), r))[y] & \text{by definition } v_2 \text{ and } v_1 \\
= & ((l_1.\mathit{put}\ v_1\ (c_1, r_1)) \cdot (l_2.\mathit{put}\ v_2\ (c_2, r_2)))[y] & \text{by definition } (l_1 \sim l_2).\mathit{put} \\
& \quad \text{where } r_2, r_2 = \mathit{split}(|v_2|, r) \\
= & l_2.\mathit{put}\ v_2\ (c_2, r_2)[y - n_1] & \text{by definition } [\cdot] \\
= & k.\mathit{put}\ v_2[x]\ (r_2(x)) & \text{by CHUNKPUT for } l_2 \\
= & k.\mathit{put}\ (v_1 \cdot v_2)[x]\ ((r_2 +\!\!+ r_1)(x)) & \text{by definition } [\cdot] \text{ and } (+\!\!+) \text{ and finite map application} \\
= & k.\mathit{put}\ (v[x])\ (r(x)) & \text{by definition } \mathit{split} \text{ and } v_1 \text{ and } v_2 \text{ and } r_1 \text{ and } r_2
\end{array}
$$

and obtain the required equality.

Case $x \notin locs(v_2)$**:** Similar to the previous case.

► **ChunkCreate**: Similar to the proof for CHUNKPUT, using CHUNKCREATE for l_1 and l_2 instead of CHUNKPUT.

► **SkelPut**: Let $v \in \lfloor V_2 \cdot V_1 \rfloor$ and $v' \in \lfloor V_2 \cdot V_1 \rfloor$ be strings, $(c_1, c_2) \in (C_1 \times C_2)$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ and $r' \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ resources such that $skel(v) = skel(v')$. As $\lfloor V_2 \rfloor \cdot^! \lfloor V_1 \rfloor$ there exist unique strings $v_2 \in \lfloor V_2 \rfloor$ and $v_1 \in \lfloor V_1 \rfloor$ and $v_2' \in \lfloor V_2 \rfloor$ and $v_1' \in \lfloor V_1 \rfloor$ such that $v = (v_2 \cdot v_1)$ and $v' = (v_2' \cdot v_1')$. Moreover, from the definition of $skel$ we have that $skel(v_2) = skel(v_2')$ and $skel(v_1) = skel(v_1')$. Using these facts and definitions, we calculate as follows

$$
\begin{aligned}
& skel((l_1 \sim l_2).put\ v\ (c_1, c_2, r)) \\
&= skel((l_1 \sim l_2).put\ (v_2 \cdot v_1)\ ((c_1, c_2), r)) && \text{by definition } v_2 \text{ and } v_1 \\
&= skel(l_1.put\ v_1\ (c_1, r_1)) \cdot (l_2.put\ v_2\ (c_2, r_2)) && \text{by definition } (l_1 \sim l_2).put \\
&\qquad \text{where } r_1, r_2 = split(|v_2|, r) \\
&= skel(l_1.put\ v_1\ (c_1, r_1)) \cdot skel(l_2.put\ v_2\ (c_2, r_2)) && \text{by definition } skel \\
&= skel(l_1.put\ v_1'\ (c_1, r_1')) \cdot skel(l_2.put\ v_2'\ (c_2, r_2')) && \text{by SKELPUT for } l_1 \text{ and } l_2 \\
&\qquad \text{where } r_1', r_2' = split(|v_1'|, r') \\
&= skel(l_1.put\ v_1'\ (c_1, r_1')) \cdot (l_2.put\ v_2'\ (c_2, r_2')) && \text{by definition } skel \\
&= skel((l_1 \sim l_2).put\ (v_2' \cdot v_1')\ ((c_1, c_2), r')) && \text{by definition } (l_1 \sim l_2).put \text{ and } r_1' \text{ and } r_2' \\
&= skel((l_1 \sim l_2).put\ v'\ ((c_1, c_2), r)) && \text{by definition } v_2' \text{ and } v_1'
\end{aligned}
$$

and obtain the required equality.

► **SkelCreate**: Similar to the proof for SKELPUT, which finishes the proof. □