Boomerang: Resourceful Lenses for String Data

Aaron Bohannon
University of Pennsylvania

J. Nathan Foster
University of Pennsylvania

Benjamin C. Pierce
University of Pennsylvania

Alexandre Pilkiewicz
École Polytechnique

Alan Schmitt
INRIA Rhône-Alpes

Abstract

A *lens* is a bidirectional program. When read from left to right, it denotes an ordinary function that maps inputs to outputs. When read from right to left, it denotes an "update translator" that takes an input together with an updated output and produces a new input that reflects the update. Many variants of this idea have been explored in the literature, but none deal fully with *ordered* data. If, for example, an update changes the order of a list in the output, the items in the output list and the chunks of the input that generated them can be misaligned, leading to lost or corrupted data.

We attack this problem in the context of bidirectional transformations over strings, the primordial ordered data type. We first propose a collection of bidirectional *string lens combinators*, based on familiar operations on regular transducers (union, concatenation, Kleene-star) and with a type system based on regular expressions. We then design a new semantic space of *dictionary lenses*, enriching the lenses of Foster et al. (2007b) with support for two additional combinators for marking "reorderable chunks" and their keys. To demonstrate the effectiveness of these primitives, we describe the design and implementation of Boomerang, a full-blown *bidirectional programming language* with dictionary lenses at its core. We have used Boomerang to build transformers for complex real-world data formats including the SwissProt genomic database.

We formalize the essential property of *resourcefulness*—the correct use of keys to associate chunks in the input and output—by defining a refined semantic space of *quasi-oblivious lenses*. Several previously studied properties of lenses turn out to have compact characterizations in this space.

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

General Terms Languages, Design, Theory

Keywords Bidirectional languages, lenses, view update problem, regular string transducers, regular types

1. Introduction

"The art of progress is to preserve order amid change and to preserve change amid order."
—A N Whitehead

Most of the time, we use programs in just one direction, from input to output. But sometimes, having computed an output, we need to be able to *update* this output and then "calculate backwards" to find a correspondingly updated input. The problem of writing such bidirectional transformations arises in a multitude of domains, including data converters and synchronizers, parsers and pretty printers, picklers and unpicklers, structure editors, constraint maintainers for user interfaces, and, of course, in databases, where it is known as the view update problem. Our own study of bidirectional transformations is motivated by their application in a generic synchronization framework, called Harmony, where they are used to synchronize heterogeneous data formats against each other (Pierce et al. 2006; Foster et al. 2007a).

The naive way to write a bidirectional transformation is simply to write two separate functions in any language you like and check (by hand) that they fit together in some appropriate sense—e.g., that composing them yields the identity function. However, this approach is unsatisfying for all but the simplest examples. For one thing, verifying that the two functions fit together in this way requires intricate reasoning about their behaviors. Moreover, it creates a maintenance nightmare: both functions will embody the structure that the input and output schemas have in common, so changes to the schemas will require coordinated changes to both. (See the appendix for a concrete example.)

A better alternative is to design a notation in which both transformations can be described at the same time—i.e., a *bidirectional programming language*. In a bidirectional language, every expression, when read from left to right, denotes a function mapping inputs to outputs; when read from right to left, the same expression denotes a function mapping an updated output together with an original input to an appropriately updated version of the input. Not only does this eliminate code duplication; it also eliminates paper-and-pencil proofs that the two transformations fit together properly: we can design the language to guarantee it.

Many different bidirectional languages have been proposed, including constraint maintainers (Meertens 1998), pickler combinators (Kennedy 2004), embedding projection pairs (Benton 2005; Ramsey 2003), X/Inv (Hu et al. 2004), XSugar (Brabrand et al. 2005), biXid (Kawanaka and Hosoya 2006), PADS (Fisher and Gruber 2005), and bi-arrows (Alimarine et al. 2005). The design challenge for all these languages lies in striking a balance between expressiveness and reliability—making strong promises to programmers about the joint behavior of pairs of transformations and the conditions under which they can safely be used.

Lenses The language described in this paper is an extension of our previous presentation on lenses (Foster et al. 2007b)—called *basic lenses* here.[1] Among the bidirectional languages listed above,

[1] Readers familiar with the original presentation will notice some minor differences. First we handle situations where an element of C must be created from an element of A using a *create* function instead of enriching C with a special element Ω and using *put*. Second, as we are not considering lenses defined by recursion, we take the components of lenses to be total functions rather than defining lenses with *partial* components and establishing totality later. Finally, we take the behavioral laws as part of the fundamental

lenses are unique in their emphasis on strong guarantees on behavior and on compositional reasoning techniques for establishing those guarantees.

Formally, a basic lens l mapping between a set of inputs C ("concrete structures") and a set of outputs A ("abstract structures") comprises three functions

$$\begin{aligned} l.\mathit{get} &\in C \longrightarrow A \\ l.\mathit{put} &\in A \longrightarrow C \longrightarrow C \\ l.\mathit{create} &\in A \longrightarrow C \end{aligned}$$

obeying the following laws for every $c \in C$ and $a \in A$:

$$l.\mathit{put}\ (l.\mathit{get}\ c)\ c = c \qquad (\textsc{GetPut})$$

$$l.\mathit{get}\ (l.\mathit{put}\ a\ c) = a \qquad (\textsc{PutGet})$$

$$l.\mathit{get}\ (l.\mathit{create}\ a) = a \qquad (\textsc{CreateGet})$$

The set of basic lenses from C to A is written $C \Longleftrightarrow A$.

The *get* component of a lens may, in general, discard some of the information from the concrete structure while computing the abstract structure. The *put* component therefore takes as arguments not only an updated abstract structure but also the original concrete structure; it weaves the data from the abstract structure together with information from the concrete structure that was discarded by the *get* component, yielding an updated concrete structure. The *create* component is like *put* except that it only takes an A argument; it supplies defaults for the information discarded by *get* in situations where only the abstract structure is available.

Every lens obeys three laws. The first stipulates that the *put* function must restore all of the information discarded by the *get* if its arguments are an abstract structure and a concrete structure that generates the very same abstract structure; the second and third demand that the *put* and *create* functions propagate all of the information in the abstract structure back to the updated concrete structure. These laws are closely related to classical conditions on view update translators in the database literature (see Foster et al. (2007b) for a detailed comparison).

Motivations and Contributions In previous work, we designed lenses for trees (Foster et al. 2007b) and for relations (Bohannon et al. 2006); in this paper we address the special challenges that arise when *ordered* data is manipulated in bidirectional languages. Our goals are both foundational and pragmatic. Foundationally, we explore the correct treatment of ordered data, embodied abstractly as a new semantic law stipulating that the *put* function must align pieces of the concrete and abstract structures in a reasonable way, even when the update involves a reordering. Pragmatically, we investigate lenses on ordered data by developing a new language based around notions of chunks, keys, and dictionaries. To ground our investigation, we work within the tractable arena of string transformations. Strings already expose many fundamental issues concerning ordering, allowing us to grapple with these issues without the complications of a richer data model.

While primary focus is on exposing fundamental issues, we have also tried to design our experimental language, Boomerang, to be useful in practice. There is a lot of string data in the world—textual database formats (iCalendar, vCard, BibTeX, CSV), structured documents (LaTeX, Wiki, Markdown, Textile), scientific data (SwissProt, Genebank, FASTA), and simple XML formats (RSS, AJAX data) and microformats (JSON, YAML) whose schemas are non-recursive—and it is often convenient to manipulate this data directly, without first mapping it to more structured representations. Since most programmers are already familiar with regular languages, we hope that a bidirectional language for strings built around regular operations (i.e., finite state transducers) will have broad appeal.

definition of basic lenses, rather than first defining bare structures of appropriate type and then adding the laws—i.e., in the terminology of Foster et al. (2007b), basic lenses correspond to *well-behaved, total lenses*.

Our contributions can be summarized as follows:

1. We develop a set of *string lens combinators* with intuitive semantics and typing rules that ensure the lens laws, all based on familiar regular operators (union, concatenation, and Kleene-star).
2. We address a serious issue in bidirectional manipulation of ordered data—the need for lenses to be able to match up chunks of data in the concrete and abstract structures by key rather than by position, which we call *resourcefulness*—by adding two more combinators and interpreting all the combinators in an enriched semantic space of *dictionary lenses*.
3. We formalize a condition called *quasi-obliviousness* and use it to study properties of dictionary lenses. Some previously studied properties of basic lenses also have neat characterizations using this condition.
4. We sketch the design and implementation of *Boomerang*, a full-blown *bidirectional programming language* based on dictionary lenses, and describe some programs we have built for transforming real-world data structures such as SwissProt.

String Lenses To give a first taste of these ideas, let us consider a simple example where the concrete structures are newline-separated records, each with three comma-separated fields representing the name, dates, and nationality of a classical composer

```
"Jean Sibelius, 1865-1957, Finnish
 Aaron Copland, 1910-1990, American
 Benjamin Britten, 1913-1976, English"
```

and the abstract structures include just names and nationalities:

```
"Jean Sibelius, Finnish
 Aaron Copland, American
 Benjamin Britten, English"
```

Here is a string lens that implements this transformation:

```
let ALPHA = [A-Za-z ]+
let YEARS = [0-9]{4} . "-" . [0-9]{4}
let comp = copy ALPHA . copy ", "
         . del YEARS . del ", "
         . copy ALPHA

let comps = copy "" | comp . (copy "\n" . comp)*
```

The first two lines define ordinary regular expressions for alphabetical data and year ranges. We use standard POSIX notation for character sets (`[A-Za-z ]` and `[0-9]`) and repetition (`+` and `{4}`).

The lens that processes a single composer is `comp`; lists of composers are processed by `comps`. In the *get* direction, these lenses can be read as ordinary string transducers, written in regular expression style: `copy ALPHA` matches `ALPHA` in the concrete structure and copies it to the abstract structure, and `copy ", "` matches and copies a literal comma-space, while `del YEARS` matches `YEARS` in the concrete structure but adds nothing to the abstract structure. The union (`|`), concatenation (`.`), and iteration (`*`) operators work as expected. Thus, the *get* component of `comp` matches an entry for a single composer, consisting of a substring matching the regular expression `ALPHA`, followed by a comma and a space (all of which are copied to the output), followed by a string matching `YEARS` and another comma and space (which are not copied) and a final `ALPHA`. The *get* of `comps` matches either a completely empty concrete structure (which it copies to the output) or a newline-separated concatenation of entries, each of which is processed by `comp`.

The *put* component of `comps` restores the dates positionally: the name and nationality from the nth line in the abstract structure are combined with the years from the nth line in the concrete structure, using a default year range to handle cases where the abstract structure has more lines than the concrete one. We will see precisely how all this works in Section 2; for now, the important point is that the *put* component of `comps` operates by splitting both of its arguments into lines and invoking the *put* component of `comp` on the first line from the abstract structure together with the first line from the concrete structure, then the second line from the abstract structure together with the second line from the concrete structure, etc. For some updates—e.g., when entries have been edited and perhaps added at the end of the abstract structure but the order of lines has not changed—this policy does a good job. For example, if the update to the abstract structure replaces Britten's nationality with "British" and adds an entry for Tansman, the *put* function combines the new abstract structure

```
"Jean Sibelius, Finnish
 Aaron Copland, American
 Benjamin Britten, British
 Alexandre Tansman, Polish"
```

with the original concrete structure and yields

```
"Jean Sibelius, 1865-1957, Finnish
 Aaron Copland, 1910-1990, American
 Benjamin Britten, 1913-1976, British
 Alexandre Tansman, 0000-0000, Polish"
```

(the default year range `0000-0000` is generated by the `del` lens in `comp` from the regular expression `YEARS`).

Problems with Order On other examples, however, the behavior of this *put* function is highly unsatisfactory. If the update to the abstract string breaks the positional association between lines in the concrete and abstract strings, the output will be mangled—e.g., when the update to the abstract string is a reordering, combining

```
"Jean Sibelius, Finnish
 Benjamin Britten, English
 Aaron Copland, American"
```

with the original concrete structure yields an output

```
"Jean Sibelius, 1865-1957, Finnish
 Benjamin Britten, 1910-1990, English
 Aaron Copland, 1913-1976, American"
```

where the year data has been taken from the entry for Copland and inserted into into the entry for Britten, and vice versa.

This is a serious problem, and a pervasive one: it is triggered whenever a lens whose *get* function discards information is iterated over an ordered list and the update to the abstract list breaks the association between elements in the concrete and abstract lists. It is a show-stopper for many of the applications we want to write.

What we want is for the *put* to align the entries in the concrete and abstract strings by matching up lines with identical name components. On the inputs above, this *put* function would produce

```
"Jean Sibelius, 1865-1957, Finnish
 Benjamin Britten, 1913-1976, English
 Aaron Copland, 1910-1990, American"
```

but neither basic lenses nor any other existing bidirectional language provides the means to achieve this effect.

Dictionary Lenses Our solution is to enrich lenses with a simple mechanism for tracking *provenance* (Cui and Widom 2003; Buneman et al. 2001, etc.). The idea is that the programmer should identify *chunks* of the concrete string and a *key* for each chunk. These induce an association between chunks and pieces of the abstract string, and this association can be used by *put* during the translation of updates to find the chunk corresponding to each piece of the abstract, even if the abstract pieces have been reordered. Operationally, we retool all our *put* functions to use this association by parsing the whole concrete string into a dictionary, where each concrete chunk is stored under its key, and then making this dictionary, rather than the string itself, available to the *put* function. We call these enriched structures *dictionary lenses*.

Here is a dictionary lens that gives us the desired behavior for the composers example:

```
let comp = key ALPHA . copy ", "
         . del (YEARS . ", ")
         . copy ALPHA

let comps = "" | <comp> . ("\n" . <comp>)*
```

The first change from the earlier version of this program is that the two occurrences of `comp` in `comps` are marked with angle brackets, indicating that these subexpressions are the reorderable chunks of information. The corresponding substring of the concrete structure at each occurrence (which is passed to the *put* of `comp`) is obtained not positionally but by matching keys. The second change is that the first `copy` at the beginning of `comp` has been replaced by the special primitive `key`. The lens `key ALPHA` has the same copying behavior as `copy ALPHA`, but it additionally specifies that the matched substring is to be used as the key of the chunk in which it appears—i.e., in this case, that the key of each composer's entry is their name. This choice means that we can both reorder the entries in the abstract structure and edit their nationalities, since the correspondence between chunks in the concrete and abstract structures is based just on names. We do not actually demand that the key of each chunk be unique—i.e., these "keys" are not required to be keys in the strict database sense. If several pieces of the abstract structure have the same key, they are matched by position.

Quasi-Obliviousness For the composers example, the behavior of the new dictionary lens is clearly preferable to that of the original basic lens: its *put* function has the effect of translating a reordering on the abstract string into a corresponding reordering on the concrete string, whereas the *put* function of the original lens works by position and produces a mangled result. We would like a characterization of this difference—i.e., a way of expressing the intuition that the second lens does something good, while the first does not.

To this end, we define a semantic space of lenses called *quasi-oblivious lenses*. Let l be a lens in $C \Longleftrightarrow A$ and let $\sim$ be an equivalence relation on C. We say that l is a quasi-oblivious lens with respect to $\sim$ if its *put* function ignores differences between equivalent concrete arguments.

We are primarily interested in lenses that are quasi-oblivious with respect to an equivalence relating concrete strings up to reorderings of chunks. It should be clear that a dictionary lens that operates on dictionaries in which the relative order of concrete lines is forgotten will be quasi-oblivious with respect to such an equivalence, while the analogous basic lens, which operates on lines positionally, is not. Using the above condition on *put*, we can derive intuitive properties for many such quasi-oblivious lenses—e.g., for the dictionary lens for composers above, we can show that updates to the abstract list consisting of reorderings are translated by the *put* as corresponding reorderings on the concrete list.

Lenses that are quasi-oblivious with respect to equivalences other than reordering are also interesting. Indeed, we can characterize some important special cases of basic lenses (*oblivious* and *very well behaved* lenses) in terms of quasi-obliviousness.

Boomerang Our theoretical development focuses on a small set of basic combinators. Of course, writing large programs entirely in terms of such low-level primitives would be tedious; we don't do this. Instead, we have implemented a full-blown programming language, called Boomerang, in which the combinators are embedded in a functional language, *à la* Algol-60. That is, a Boomerang program is a functional program over the base type "lens"; to apply it to string data, we first evaluate the functional program to produce a lens, and then apply this lens to the strings. This functional infrastructure can be used to abstract out common patterns as generic bidirectional libraries (e.g., for processing XML structures) that make higher-level programming quite convenient.

Boomerang also comes equipped with a type checker that infers lens types and checks the conditions needed to ensure that a dictionary lens satisfies the lens laws. The domain and codomain types for dictionary lenses are regular languages, so the analysis performed by this checker is very precise—a huge aid in writing correct lens programs.

Using Boomerang, we have developed several large lenses for processing a variety of data including vCard, CSV, and XML address books, BibTeX and RIS bibliographic databases, LaTeX documents, iTunes libraries, and databases of protein sequences represented in the ASCII SwissProt format and XML.

Outline Section 2 introduces notation and formally defines the basic string lenses used in the first example above. Syntax, semantics, and typing rules for dictionary lenses are given in Section 3. Section 4 defines the refined semantic space of quasi-oblivious lenses. Sections 5 and 6 describe Boomerang and our experiences building lenses for real-world data formats. Section 7 discusses related work. Section 8 describes extensions and ideas for future work. To save space, proofs are deferred to the full version, available as a technical report (Bohannon et al. 2007).

2. Basic String Lenses

Before presenting dictionary lenses, let us warm up by formalizing the language for basic lenses from the first example in the introduction. Let Σ be a fixed alphabet (e.g., ASCII). A language is a subset of Σ^*. Metavariables u, v, w range over strings in Σ^*, and ϵ denotes the empty string. The concatenation of two strings u and v is written $u{\cdot}v$; concatenation is lifted to languages L_1 and L_2 in the obvious way: $L_1{\cdot}L_2 = \{u{\cdot}v \mid u \in L_1 \text{ and } v \in L_2\}$. We write L^* to denote the iteration of L: i.e., $L^* = \bigcup_{n=0}^{\infty} L^n$ where L^n denotes the n-fold concatenation of L with itself.

The typing rules for some lenses require that for every string belonging to the concatenation of two languages, there be a unique way of splitting that string into two substrings belonging to the concatenated languages. Two languages L_1 and L_2 are unambiguously concatenable, written $L_1 {\cdot^!} L_2$, if for every u_1, v_1 in L_1 and u_2, 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 every $u_1, \ldots, u_m, v_1, \ldots, v_n, \in L$, if $u_1{\cdot}\cdots{\cdot}u_m = v_1{\cdot}\cdots{\cdot}v_n$ then $m = n$ and $u_i = v_i$ for every i from 1 to n.

2.1 Fact: It is decidable whether two regular languages L_1 and L_2 are unambiguously concatenable and whether a single language L is unambiguously iterable.

Proof sketch: Let M_1 and M_2 be DFAs accepting L_1 and L_2. Construct an NFA N_{12} for $L_1{\cdot}L_2$ using the standard Thompson construction. An NFA N is ambiguous iff there exists a string for which there are two distinct paths through N; ambiguity of NFAs can be decided by *squaring* (Berstel et al. 2005, Prop. 4.3): N is ambiguous iff there is a path through $N \times N$ that does not lie entirely on the diagonal. It is easy to show that L_1 and L_2 are unambiguously concatenable iff N_{12} is unambiguous. Unambiguous iteration is similar. □

Regular expressions are generated by the grammar

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

where u ranges over arbitrary strings (including ϵ). The notation $[\![E]\!]$ denotes the (non-empty) language described by $E \in \mathcal{R}$. The function $choose(E)$ picks an arbitrary element from $[\![E]\!]$.

With that notation in hand, we now define five combinators for building basic string lenses over regular languages. Recall that we write $l \in C \Longleftrightarrow A$ when l is a basic lens mapping between strings in C and A. Each basic lens expects to be applied to arguments in its domain/codomain—it is nonsensical to do otherwise. In our implementation, we perform a membership test on every string before supplying it to a lens. (We do this check just once, at the top level: internally, the typing rules guarantee that every sublens is provided with well-typed inputs.)

The simplest primitive, *copy* E, copies every string belonging to (the language denoted by) E from the concrete structure to the abstract structure, and conversely in the *put* direction. The components of *copy* are precisely defined in the box below. The second primitive lens, *const* E u v maps every string belonging to E to a constant string u. Its *put* function restores its concrete argument. It also takes as an argument a default v belonging to E, which is used by *create* when no concrete argument is available. Note that *const* satisfies PUTGET because its codomain is a singleton set.

The inference rules should be read as the statements of lemmas that each combinator is a basic lens at the given type.

$$\frac{E \in \mathcal{R}}{\textit{copy}\ E \in [\![E]\!] \Longleftrightarrow [\![E]\!]}
\qquad
\begin{array}{lcl}
\textit{get}\ c & = & c \\
\textit{put}\ a\ c & = & a \\
\textit{create}\ a & = & a
\end{array}$$

$$\frac{E \in \mathcal{R} \qquad u \in \Sigma^* \qquad v \in [\![E]\!]}{\textit{const}\ E\ u\ v \in [\![E]\!] \Longleftrightarrow \{u\}}
\qquad
\begin{array}{lcl}
\textit{get}\ c & = & u \\
\textit{put}\ a\ c & = & c \\
\textit{create}\ a & = & v
\end{array}$$

Several lenses can be expressed as syntactic sugar using *const*:

$$\begin{array}{rcl}
E \leftrightarrow u & = & \textit{const}\ E\ u\ (choose(E)) \\
\textit{del}\ E & = & E\ \leftrightarrow \epsilon \\
\textit{ins}\ u & = & \epsilon \leftrightarrow u
\end{array}$$

They behave as follows: $E \leftrightarrow u$ is like *const*, but uses an arbitrary element of E for *create*; the *get* function of *del* E deletes a concrete string belonging to E and restores the deleted string in the *put* direction; *ins* u inserts a fixed string u in the *get* direction and deletes u in the opposite direction.

The next three combinators build bigger lenses from smaller ones using regular operators. Concatenation is simplest:

$$\frac{\begin{array}{cc} C_1{\cdot^!}C_2 & A_1{\cdot^!}A_2 \\ l_1 \in C_1 \Longleftrightarrow A_1 & l_2 \in C_2 \Longleftrightarrow A_2 \end{array}}{l_1{\cdot}l_2 \in C_1{\cdot}C_2 \Longleftrightarrow A_1{\cdot}A_2}$$

$$\begin{array}{lcl}
\textit{get}\ (c_1{\cdot}c_2) & = & (l_1.\textit{get}\ c_1){\cdot}(l_2.\textit{get}\ c_2) \\
\textit{put}\ (a_1{\cdot}a_2)\ (c_1{\cdot}c_2) & = & (l_1.\textit{put}\ a_1\ c_1){\cdot}(l_2.\textit{put}\ a_2\ c_2) \\
\textit{create}\ (a_1{\cdot}a_2) & = & (l_1.\textit{create}\ a_1){\cdot}(l_2.\textit{create}\ a_2)
\end{array}$$

The notation $c_1{\cdot}c_2$ used in the definition of concatenation assumes that c_1 and c_2 are members of C_1 and C_2 respectively; we use this convention silently in the rest of the paper.

The typing rule for concatenation requires that the concrete domains and the abstract codomains each be unambiguously concatenable. This condition is essential for ensuring that the components of the lens are well-defined functions and that the whole lens is well behaved. As an example of what would go wrong without these conditions, consider the (ill-typed) lens l_{ambig}, defined as $(\texttt{a} \leftrightarrow \texttt{a} \mid \texttt{aa} \leftrightarrow \texttt{aa}){\cdot}(\texttt{a} \leftrightarrow \texttt{b} \mid \texttt{aa} \leftrightarrow \texttt{b})$ ("|" is defined formally below). The *get* component is not well defined since, according to the above specification, l_{ambig}.*get* $\texttt{aaa} = \texttt{ab}$ if we split $\texttt{aaa}$ into $\texttt{a}{\cdot}\texttt{aa}$ and l_{ambig}.*get* $\texttt{aaa} = \texttt{aab}$ if we split it into $\texttt{aa}{\cdot}\texttt{a}$. This issue could be side-stepped using a fixed policy for choosing among multiple parses (e.g., with a shortest match policy, $l_{ambig}.\mathit{get}$ `aaa` = `ab`). However, doing so would not always give us a lens that satisfies the lens laws; intuitively, just because one direction uses a given match policy does not mean that the string it produces will split the same way using the same policy in the other direction. Consider l_{bogus} defined as $k{\cdot}k$ where k is (`a` $\leftrightarrow$ `bb` | `aa` $\leftrightarrow$ `a` | `b` $\leftrightarrow$ `b` | `ba` $\leftrightarrow$ `ba`). Then using the shortest match policy we have $l_{bogus}.\mathit{get}$ `aaa` equals ($k.\mathit{get}$ `a`)$\cdot$($k.\mathit{get}$ `aa`), which is `bba`, but $l_{bogus}.\mathit{put}$ `bba` `aaa` equals ($k.\mathit{put}$ `b` `a`)$\cdot$($k.\mathit{put}$ `ba` `aa`), which is `bba`. That is, the GETPUT law fails. For these reasons, we require that each pair of C_1 and C_2 and A_1 and A_2 be unambiguously concatenable.

The Kleene-star combinator is similar:

$$\boxed{\begin{array}{l}
\dfrac{l \in C \Longleftrightarrow A \qquad C^{!*} \qquad A^{!*}}{l* \;\in\; C* \Longleftrightarrow A*} \\[2ex]
\begin{array}{lcl}
\mathit{get}\ (c_1 \cdots c_n) & = & (l.\mathit{get}\ c_1) \cdots (l.\mathit{get}\ c_n) \\
\mathit{put}\ (a_1 \cdots a_n)\ (c_1 \cdots c_m) & = & c'_1 \cdots c'_n \\
\quad \text{where } c'_i = \begin{cases} l.\mathit{put}\ a_i\ c_i & i \in \{1, ..., \min(m,n)\} \\ l.\mathit{create}\ a_i & i \in \{m+1, ..., n\} \end{cases} & & \\
\mathit{create}\ (a_1 \cdots a_n) & = & (l.\mathit{create}\ a_1) \cdots (l.\mathit{create}\ a_n)
\end{array}
\end{array}}$$

Note that the *put* component of $l*$ calls the *put* of l on elements of A and C having the same index in their respective lists. This is the root of the undesirable behavior in the example in the introduction.[2] Also note that it must handle cases where the number of As is not equal to the number of Cs. Since the number of As produced by the *get* of $l*$ equals the number of Cs in its argument, the result of the *put* function must have exactly as many Cs as there are As in its abstract string—otherwise, PUTGET would not hold. When there are more Cs than As, the lens simply ignores the extra Cs. When there are more As, it must put them back into the concrete domain, but it has no concrete argument to use. It uses $l.\mathit{create}$ to process these extra pieces.

The final combinator forms the union of two lenses:

$$\boxed{\begin{array}{l}
\dfrac{\begin{array}{c} C_1 \cap C_2 = \emptyset \\ l_1 \in C_1 \Longleftrightarrow A_1 \qquad l_2 \in C_2 \Longleftrightarrow A_2 \end{array}}{l_1 \mid l_2 \;\in\; C_1 \cup C_2 \Longleftrightarrow A_1 \cup A_2} \\[2ex]
\mathit{get}\ c \;=\; \begin{cases} l_1.\mathit{get}\ c & \text{if } c \in C_1 \\ l_2.\mathit{get}\ c & \text{if } c \in C_2 \end{cases} \\[2ex]
\mathit{put}\ a\ c \;=\; \begin{cases} l_1.\mathit{put}\ a\ c & \text{if } c \in C_1 \wedge a \in A_1 \\ l_2.\mathit{put}\ a\ c & \text{if } c \in C_2 \wedge a \in A_2 \\ l_1.\mathit{create}\ a & \text{if } c \in C_2 \wedge a \in A_1 \setminus A_2 \\ l_2.\mathit{create}\ a & \text{if } c \in C_1 \wedge a \in A_2 \setminus A_1 \end{cases} \\[2ex]
\mathit{create}\ a \;=\; \begin{cases} l_1.\mathit{create}\ a & \text{if } a \in A_1 \\ l_2.\mathit{create}\ a & \text{if } a \in A_2 \setminus A_1 \end{cases}
\end{array}}$$

The typing rule forces C_1 and C_2 to be disjoint. Like the ambiguity condition in the rule for concatenation, this condition is essential to ensure that the lens is well defined. The abstract codomains, however, may overlap. In the *put* direction, when the abstract argument belongs to $A_1 \cap A_2$, the union lens uses the concrete argument to select a branch. In the *create* function, since no concrete argument is available, it just uses l_1. (This choice is arbitrary, but is not a limitation: to use l_2 by default, the programmer writes $l_2 \mid l_1$. It does mean, though, that union is not commutative.)

[2] We cannot, however, repair the problem just by fixing Kleene-star; the same issues come up with concatenation.

3. Dictionary Lenses

Now that we've seen basic string lenses, let us define lenses that deal more flexibly with ordering. We will accomplish this by adding two new primitives, *match* (written with angle brackets in the concrete syntax of Boomerang) and *key*, and interpreting these new primitives and the primitives defined in the previous section in a refined space called *dictionary lenses*.

The main difference between basic and dictionary lenses is that their *put* components operate on different types of structures—strings and dictionaries respectively. Dictionary lenses also include two new components: *parse*, which is used to build a dictionary out of a concrete string, and *key*, which is used to compute the key of data that goes into a dictionary.

In a dictionary lens, the work of the basic lens *put* function is divided into two phases. In the first, the concrete string is given to *parse*, which splits it into two parts: a collection of concrete chunks organized as a dictionary and a *skeleton* representing the parts of the string outside of the chunks. In the second, the *put* function uses the abstract string, the skeleton, and the dictionary to build an updated concrete string (and a new dictionary). These phases occur in strict sequence: given a dictionary lens l, an abstract string a, and a concrete string c, we first *parse* c, which yields a skeleton s and dictionary d; these are then passed, together with a, to l's *put* function, which walks recursively over the structure of s and a, threading d through l's sublenses and pulling chunks out as needed.

To streamline the exposition, we start with the definition of dictionary lenses, which relies on several notions we have not seen yet—skeleton sets S, the set of keys K, dictionary type specifications L, dictionary types $D(L)$, and an infix operation $++$ that appends dictionary values. These are defined below.

A dictionary lens from C to A with skeleton type S and dictionary type specification L has components

$$\begin{array}{rcl}
l.\mathit{get} & \in & C \longrightarrow A \\
l.\mathit{parse} & \in & C \longrightarrow S \times D(L) \\
l.\mathit{key} & \in & A \longrightarrow K \\
l.\mathit{create} & \in & A \longrightarrow D(L) \longrightarrow C \times D(L) \\
l.\mathit{put} & \in & A \longrightarrow S \times D(L) \longrightarrow C \times D(L)
\end{array}$$

obeying the following behavioral laws:[3]

$$\frac{s, d' = l.\mathit{parse}\ c \qquad d \in D(L)}{l.\mathit{put}\ (l.\mathit{get}\ c)\ (s, (d' ++ d)) = c, d} \qquad \text{(GETPUT)}$$

$$\frac{c, d' = l.\mathit{put}\ a\ (s, d)}{l.\mathit{get}\ c = a} \qquad \text{(PUTGET)}$$

$$\frac{c, d' = l.\mathit{create}\ a\ d}{l.\mathit{get}\ c = a} \qquad \text{(CREATEGET)}$$

We write $C \overset{S,L}{\Longleftrightarrow} A$ for the set of dictionary lenses with this type.

Both *create* and *put* consume dictionaries. We thread dictionaries through calls to these functions in a functional style that simulates a global, mutable dictionary, and remove entries as they are used, so that the next lookup of the same key finds the (positionally) next chunk from the concrete string. The *put* function takes a skeleton argument, whereas the *create* function does not. The skeleton, when available, represents the original concrete string with the chunks removed and provides enough information to reconstruct the original concrete string in cases where GETPUT requires it.

[3] In law GETPUT, the extra dictionary d shows that all and only the chunks originating from c are used by the *put* function.

To see how the components of a dictionary lens fit together, let us see how to build a basic lens $\bar{l}$ from a dictionary lens l:

$$\begin{array}{lcl} \bar{l}.\mathit{get}\ c & = & l.\mathit{get}\ c \\ \bar{l}.\mathit{put}\ a\ c & = & \pi_1(l.\mathit{put}\ a\ (l.\mathit{parse}\ c)) \\ \bar{l}.\mathit{create}\ a & = & \pi_1(l.\mathit{create}\ a\ \{\}) \end{array}$$

This definition embodies the strict phase separation between *parse* and *put* discussed above. It is easy to show that the dictionary lens laws guarantee the original laws for basic lenses built this way.

3.1 Theorem: If $l \in C \overset{S,L}{\Longleftrightarrow} A$ then $\bar{l} \in C \Longleftrightarrow A$.

We now give the definitions deferred previously. We write $h :: t$ for the list with head h and tail t, $\mathsf{List}(X)$ for the set of lists of X and, and $l_1 @ l_2$ for the concatenation of lists l_1 and l_2. We write $X \times Y$ for the set of pairs $\{(x,y) \mid x \in X \text{ and } y \in Y\}$. We take the set of skeletons $\mathcal{S}$ to be the smallest set closed under these operations that includes plain strings and a distinguished atom $\square$, which is used to mark the locations of chunks in skeletons. Formally, $\mathcal{S} = \bigcup_{n=0}^{\infty} S_n$, where $S_0 = \Sigma^* \cup \{\square\}$ and $S_{i+1} = S_i \cup (S_i \times S_i) \cup \mathsf{List}(S_i)$. We define K, the set of keys, to be just Σ^* (richer structures are also possible; see Section 8).

As chunks may be nested within chunks (by nesting the *match* combinator), the type of dictionaries is recursive. A dictionary is a total function from keys to lists of pairs, each consisting of a skeleton and another dictionary. Formally, the set of dictionaries is defined recursively on the structure of a list of sets of skeletons $L \in \mathsf{List}(\mathcal{P}(\mathcal{S}))$ specifying the skeletons that may appear at each level, as follows:

$$\begin{array}{rcl} D([]) & = & K \longrightarrow \{[]\} \\ D(S :: L) & = & K \longrightarrow \mathsf{List}(S \times D(L)) \end{array}$$

We write $\{\}$ for the dictionary that maps every k to the empty list. Let d be a dictionary, k a key, and v a skeleton-dictionary pair list of appropriate type. The update of a dictionary, written $d[k \leftarrow v]$, is defined as

$$d[k \leftarrow v](k') = \begin{cases} d(k') & \text{if } k' \neq k \\ v & \text{if } k' = k \end{cases}$$

We write $\{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\}$ for $\{\}[k_1 \leftarrow v_1]\cdots[k_n \leftarrow v_n]$. The concatenation of two dictionaries d_1 and d_2, written $d_1 +\!\!+\, d_2$, is defined using list concatenation as follows: $(d_1 +\!\!+\, d_2)(k) = d_1(k) @ d_2(k)$. Dictionaries are accessed using a partial function *lookup* that takes a key k and a dictionary d as arguments. When it finds a matching value, *lookup* returns the value found and the dictionary that remains after deleting that value.

$$\mathit{lookup}(k,d) = \begin{cases} e, d[k \leftarrow l] & \text{if } d(k) = e :: l \\ \text{undefined} & \text{otherwise} \end{cases}$$

We now reinterpret each combinator from the previous section as a dictionary lens and give the definitions of the new combinators *key* and *match*. The *key* combinator is nearly identical to *copy*, except that the *key* component of *copy* is a constant function (returning ϵ), while the *key* component of *key* returns the abstract string.

$$\begin{array}{c} E \in \mathcal{R} \qquad L \in \mathsf{List}(\mathcal{P}(\mathcal{S})) \\ \hline \mathit{copy}\ E \ \in\ [\![E]\!] \overset{[\![E]\!],L}{\Longleftrightarrow} [\![E]\!] \end{array}$$

$$\begin{array}{lcl} \mathit{get}\ c & = & c \\ \mathit{parse}\ c & = & c, \{\} \\ \mathit{key}\ a & = & \epsilon \\ \mathit{create}\ a\ d & = & a, d \\ \mathit{put}\ a\ (s,d) & = & a, d \end{array}$$

$$\begin{array}{c} E \in \mathcal{R} \qquad L \in \mathsf{List}(\mathcal{P}(\mathcal{S})) \\ \hline \mathit{key}\ E \ \in\ [\![E]\!] \overset{[\![E]\!],L}{\Longleftrightarrow} [\![E]\!] \end{array}$$

$$\begin{array}{lcl} \mathit{get}\ c & = & c \\ \mathit{parse}\ c & = & c, \{\} \\ \mathit{key}\ a & = & a \\ \mathit{create}\ a\ d & = & a, d \\ \mathit{put}\ a\ (s,d) & = & a, d \end{array}$$

The refined definition of *const* is also straightforward.

$$\begin{array}{c} E \in \mathcal{R} \qquad u \in \Sigma^* \qquad v \in [\![E]\!] \qquad L \in \mathsf{List}(\mathcal{P}(\mathcal{S})) \\ \hline \mathit{const}\ E\ u\ v \ \in\ [\![E]\!] \overset{[\![E]\!],L}{\Longleftrightarrow} \{u\} \end{array}$$

$$\begin{array}{lcl} \mathit{get}\ c & = & u \\ \mathit{parse}\ c & = & c, \{\} \\ \mathit{key}\ a & = & \epsilon \\ \mathit{create}\ u\ d & = & v, d \\ \mathit{put}\ u\ (s,d) & = & s, d \end{array}$$

Concatenation is similar to string lenses, but *create* and *put* thread the dictionary through the corresponding sublens functions.

$$\begin{array}{c} l_1 \ \in\ C_1 \overset{S_1,L}{\Longleftrightarrow} A_1 \quad C_1 \cdot^! C_2 \\ l_2 \ \in\ C_2 \overset{S_2,L}{\Longleftrightarrow} A_2 \quad A_1 \cdot^! A_2 \\ \hline l_1 \cdot l_2 \ \in\ C_1 \cdot C_2 \overset{S_1 \times S_2,L}{\Longleftrightarrow} A_1 \cdot A_2 \end{array}$$

$$\begin{array}{lcl} \mathit{get}\ c_1 \cdot c_2 & = & (l_1.\mathit{get}\ c_1) \cdot (l_2.\mathit{get}\ c_2) \\ \mathit{parse}\ c_1 \cdot c_2 & = & (s_1, s_2), d_1 +\!\!+\, d_2 \\ \quad \text{where } s_i, d_i = l_i.\mathit{parse}\ c_i & & \\ \mathit{key}\ a_1 \cdot a_2 & = & l_1.\mathit{key}\ a_1 \cdot l_2.\mathit{key}\ a_2 \\ \mathit{create}\ a_1 \cdot a_2\ d_1 & = & c_1 \cdot c_2, d_3 \\ \quad \text{where } c_i, d_{i+1} = l_i.\mathit{create}\ a_i\ d_i & & \\ \mathit{put}\ a_1 \cdot a_2\ ((s_1, s_2), d_1) & = & c_1 \cdot c_2, d_3 \\ \quad \text{where } c_i, d_{i+1} = l_i.\mathit{put}\ a_i\ (s_i, d_i) & & \end{array}$$

Lens concatenation is associative, modulo coercion to basic lenses: even though the skeleton structure of a lens differentiates $(l_1 \cdot l_2) \cdot l_3$ from $l_1 \cdot (l_2 \cdot l_3)$, we have $\overline{(l_1 \cdot l_2) \cdot l_3} = \overline{l_1 \cdot (l_2 \cdot l_3)}$. We implicitly associate lens concatenation (and the corresponding set-theoretic operations) to the left.

To illustrate, consider the following dictionary lens:

$$\begin{array}{ll} & l_\$ = \mathit{key}\ \mathtt{x}^* \cdot \mathit{del}\ \mathtt{y}^* \cdot \mathit{copy}\ (\mathtt{z}^* \cdot \$) \\ \text{with} & l_\$ \in \mathtt{x}^* \cdot \mathtt{y}^* \cdot \mathtt{z}^* \cdot \$ \overset{S,[]}{\Longleftrightarrow} \mathtt{x}^* \cdot \mathtt{z}^* \cdot \$ \\ \text{and} & S = \mathtt{x}^* \times \mathtt{y}^* \times \mathtt{z}^* \cdot \$. \end{array}$$

The *parse* function of $l_\$$ breaks apart a string according to the structure of the concrete domain:

$$l_\$.\mathit{parse}\ \mathtt{xxyz\$} = (\mathtt{xx}, \mathtt{y}, \mathtt{z\$}), \{\} +\!\!+ \{\} +\!\!+ \{\}$$

(The dictionary is empty because none of the sublenses use the *match* operator.) The *key* function returns the `xx...x` prefix of an abstract string. The other components of this lens induce the same functions as in the basic lens semantics.

The iteration combinator is analogous to the concatenation operator. Its *parse* function builds a concatenated dictionary and its *put* and *create* functions thread their dictionary argument (from left to right) through the corresponding sublens functions.

$$\begin{array}{c} l \ \in\ C \overset{S,L}{\Longleftrightarrow} A \qquad C^{!*} \qquad A^{!*} \\ \hline l^* \ \in\ C^* \overset{\mathsf{List}(S),L}{\Longleftrightarrow} A^* \end{array}$$

$$\begin{array}{ll} \mathit{get}\ c_1 \cdots c_n & = (l_1.\mathit{get}\ c_1) \cdots (l.\mathit{get}\ c_n) \\ \mathit{parse}\ c_1 \cdots c_n & = [s_1, ..., s_n], d_1 +\!\!+ \cdots +\!\!+\, d_n \\ \quad \text{where } s_i, d_i = l.\mathit{parse}\ c_i & \\ \mathit{key}\ a_1 \cdots a_n & = l.\mathit{key}\ a_1 \cdots l.\mathit{key}\ a_n \\ \mathit{create}\ a_1 \cdots a_n\ d_1 & = (c_1 \cdots c_n), d_{n+1} \\ \quad \text{where } c_i, d_{i+1} = l.\mathit{create}\ a_i\ d_i & \\ \mathit{put}\ a_1 \cdots a_n\ ([s_1, ..., s_m], d_1) & = (c_1 \cdots c_n), d_{n+1} \\ \quad \text{where } c_i, d_{i+1} = \begin{cases} l.\mathit{put}\ a_i\ (s_i, d_i) \\ \quad i \in \{1, ..., \min(m,n)\} \\ l.\mathit{create}\ a_i\ d_i \\ \quad i \in \{m+1, ..., n\} \end{cases} & \end{array}$$

The most interesting new combinator is *match*. Its *get* component passes off control to the sublens l. The *put* component matches up its abstract argument with a corresponding item in the dictionary and supplies both to the *put* function of l.

$$\frac{l \;\in\; C \overset{S,L}{\Longleftrightarrow} A}{\langle l\rangle \;\in\; C \overset{\{\Box\},S::L}{\Longleftrightarrow} A}$$

$$\begin{array}{lcl}
\textit{get } c & = & l.\textit{get } c \\
\textit{parse } c & = & \Box, \{l.\textit{key } (l.\textit{get } c) \mapsto [l.\textit{parse } c]\} \\
\textit{key } a & = & l.\textit{key } a \\
\textit{create } a\ d & = & \begin{cases} \pi_1(l.\textit{put } a(s_a,d_a)), d' \\ \quad \text{if } (s_a,d_a), d' = \textit{lookup}(l.\textit{key } a, d) \\ \pi_1(l.\textit{create } a\ \{\}), d \\ \quad \text{if } \textit{lookup}(l.\textit{key } a, d) \text{ undefined} \end{cases} \\
\textit{put } a\ (\Box, d) & = & \begin{cases} \pi_1(l.\textit{put } a\ (s_a,d_a)), d' \\ \quad \text{if } (s_a,d_a), d' = \textit{lookup}(l.\textit{key } a, d) \\ \pi_1(l.\textit{create } a\ \{\}), d \\ \quad \text{if } \textit{lookup}(l.\textit{key } a, d) \text{ undefined} \end{cases}
\end{array}$$

To illustrate the operation of *match*, consider the lens $\langle l_\$\rangle$*. It has the same *get* behavior as $l_\$$*, but its *put* function restores the ys to each chunk using the association induced by keys rather than by position. Let us calculate the result produced by the following application of the derived *put* function:

$$\overline{\langle l_\$\rangle^*}.\textit{put}\ \texttt{xxzzz\$x\$}\ \texttt{xyz\$xxyyzz\$}$$

Here, the update to the abstract string has swapped the order of the chunks and changed the number of zs in each chunk. The *parse* function produces a dictionary structure that associates (the parse of) each chunk to its key:

$$\begin{array}{l}\langle l_\$\rangle^*.\textit{parse}\ \texttt{xyz\$xxyyzz\$} \\ \quad = [\Box,\Box], \left\{ \begin{array}{l} \texttt{x} \mapsto [((\texttt{x},\texttt{y},\texttt{z\$}),\{\})], \\ \texttt{xx} \mapsto [((\texttt{xx},\texttt{yy},\texttt{zz\$}),\{\})] \end{array} \right\}\end{array}$$

Each step invokes the *put* of the match lens, which locates a concrete chunk from the dictionary and invokes the *put* of $l_\$$. The final result illustrates the "resourcefulness" of this lens:

$$\overline{\langle l_\$\rangle^*}.\textit{put}\ \texttt{xxzzz\$x\$}\ \texttt{xyz\$xxyyzz\$} = \texttt{xxyyzzz\$xy\$}$$

By contrast, the *put* component of the basic lens $l_\$$* is not resourceful—it restores the ys to each chunk by position:

$$l_\$^*.\textit{put}\ \texttt{xxzzz\$x\$}\ \texttt{xyz\$xxyyzz\$} = \texttt{xxyzzz\$xyy\$}$$

The final operator forms the union of two dictionary lenses:

$$\frac{\begin{array}{c} l_1 \;\in\; C_1 \overset{S_1,L}{\Longleftrightarrow} A_1 \\ l_2 \;\in\; C_2 \overset{S_2,L}{\Longleftrightarrow} A_2 \\ C_1 \cap C_2 = \emptyset \quad S_1 \cap S_2 = \emptyset \end{array}}{l_1 \mid l_2 \;\in\; C_1 \cup C_2 \overset{S_1 \cup S_2, L}{\Longleftrightarrow} A_1 \cup A_2}$$

$$\begin{array}{lcl}
\textit{get } c & = & \begin{cases} l_1.\textit{get } c & \text{if } c \in C_1 \\ l_2.\textit{get } c & \text{if } c \in C_2 \end{cases} \\
\textit{parse } c & = & \begin{cases} l_1.\textit{parse } c & \text{if } c \in C_1 \\ l_2.\textit{parse } c & \text{if } c \in C_2 \end{cases} \\
\textit{key } a & = & \begin{cases} l_1.\textit{key } a & \text{if } a \in A_1 \\ l_2.\textit{key } a & \text{if } a \in A_2 \backslash A_1 \end{cases} \\
\textit{create } a\ d & = & \begin{cases} l_1.\textit{create } a\ d & \text{if } a \in A_1 \\ l_2.\textit{create } a\ d & \text{if } a \in A_2 \backslash A_2 \end{cases} \\
\textit{put } a\ (s,d) & = & \begin{cases} l_1.\textit{put } a\ (s,d) & \text{if } a,s \in A_1 \times S_1 \\ l_2.\textit{put } a\ (s,d) & \text{if } a,s \in A_2 \times S_2 \\ l_1.\textit{create } a\ d & \text{if } a,s \in (A_1 \backslash A_2) \times S_2 \\ l_2.\textit{create } a\ d & \text{if } a,s \in (A_2 \backslash A_1) \times S_1 \end{cases}
\end{array}$$

This definition is analogous to the union operator for basic string lenses. Because the *put* function takes a skeleton and dictionary rather than a concrete string (as the basic lens *put* does), the last two cases select a branch using the skeleton value. The typing rule ensures that skeleton domains are disjoint so that this choice is well-defined. The union combinator is associative, but not commutative (for the same reason that the basic lens is not).

One interesting difference from the basic lens is that the *create* function takes a dictionary argument, which can be used to transfer information from one branch to the other. The following example illustrates why this is useful. Define $l_{\$\$} = \langle l_\$\rangle \mid \langle l_\$\rangle \cdot \langle l_\$\rangle$. The typing rules give us the following facts:

$$\begin{array}{c} l_{\$\$} \in E_C \mid E_C \cdot E_C \overset{\{\Box,(\Box,\Box)\},[S]}{\Longleftrightarrow} E_A \mid E_A \cdot E_A, \\ \text{where} \quad E_C \;=\; \texttt{x}^*\cdot\texttt{y}^*\cdot\texttt{z}^*\cdot\$ \qquad E_A \;=\; \texttt{x}^*\cdot\texttt{z}^*\cdot\$ \\ S \;=\; \texttt{x}^* \times \texttt{y}^* \times \texttt{z}^*\cdot\$. \end{array}$$

Now consider $c_1, c_2 \in E_C$ and $a_1, a_2 \in E_A$, where $a_i = l_\$.\textit{get}\ c_i$. We have $l_{\$\$}.\textit{get}\ c_1 \cdot c_2 = a_1 \cdot a_2$. A natural way for the *put* function to reflect an update of $a_1 \cdot a_2$ to a_2 on the concrete string would be to produce c_2 as the result. However, since the update involves crossing from one branch of the union to the other, the basic lens version cannot achieve this behavior—crossing branches always triggers a *create* with defaults. For example, with $c_1 = \texttt{xyz\$}$, $c_2 = \texttt{xxyyzz\$}$, $a_1 = \texttt{xz\$}$, and $a_2 = \texttt{xxzz\$}$, we have

$$(\overline{l_\$} \mid \overline{l_\$} \cdot \overline{l_\$}).\textit{put}\ \texttt{xxzz\$}\ \texttt{xyz\$xxyyzz\$} = \texttt{xxzz\$}.$$

The dictionary lens version, however, is capable of carrying information from the concrete string via its dictionary, even when the update changes which branch is selected. On the same example, we have

$$\overline{l_{\$\$}}.\textit{put}\ \texttt{xxzz\$}\ \texttt{xyz\$xxyyzz\$} = \texttt{xxyyzz\$},$$

as we might have hoped.

4. Quasi-Obliviousness

As the examples above demonstrate, dictionary lenses can be written to work well in situations where the updates to abstract strings involve reordering. In particular, the dictionary lens version of the composers lens in the introduction behaves well with respect to reordering, while the original basic lens version does not. In this section, we develop a refinement of the semantic space of basic lenses that makes such comparisons precise. We first define a space of *quasi-oblivious* lenses and show how it can be used to derive intuitive properties of lenses operating on ordered data. We then show how it can be used more generally to succinctly characterize two important special cases of basic lenses—oblivious and very well behaved lenses.

Quasi-obliviousness is an extensional property of lenses—i.e., a property of the way they transform entire abstract and concrete structures. When discussing it, there is no need to mention internal structures like skeletons and dictionaries. We therefore return in this section to the simpler vocabulary of basic lenses, keeping in mind that a dictionary lens l can be converted into a basic lens $\overline{l}$ as described in Section 3.

Let l be a basic lens from C to A and let $\sim$ be an equivalence relation on C. Then l is *quasi-oblivious* with respect to $\sim$ if it obeys the following law for every $c, c' \in C$ and $a \in A$:

$$\frac{c \sim c'}{l.\textit{put}\ a\ c = l.\textit{put}\ a\ c'} \qquad (\text{EQUIVPUT})$$

Note that the law has equality rather than $\sim$ in the conclusion; this is because the *put* must propagate all of the information contained in a to satisfy PUTGET. In particular, the order of chunks in the result of the *put* is forced by their order in a.

Like the basic lens laws, EQUIVPUT is a simple condition that guides the design of lenses by specifying what effect they must have in specific cases where the correct behavior is clear. One way to understand its effect is to notice how it extends the range of situations to which the GETPUT law applies—GETPUT only constrains the behavior of the *put* on the unique abstract string generated from a concrete string by *get*; with EQUIVPUT, it must have the same behavior on the entire equivalence class.

Here is an example demonstrating how EQUIVPUT and GETPUT can be used together to derive an useful property of the *put* component of a lens, without any additional knowledge of how *put* operates. Let C and A be regular languages and suppose that we can identify the *chunks* of every string in C and the *key* of each chunk. For example, in the composers lens, the chunks are the lines and the keys are the names of the composers. These chunks and keys induce an equivalence relation on C where two strings c and c' are equivalent if they can be obtained from each other by a *key-respecting reordering* of chunks—i.e., by repeatedly swapping chunks such that the relative ordering of chunks with the same key is preserved. Write $\sim$ for this equivalence relation. Now let l be a quasi-oblivious lens with respect to $\sim$ and suppose that the notions of chunks and keys on C are carried by the *get* function to A in a natural way, and that every key-respecting reordering on A can be generated by applying the *get* function to a correspondingly reordered string in C. (This is the case with our dictionary lens in the composer example: chunks in the abstract codomain are lines, the composer names are preserved by the *get* function, and the order of the abstract lines after a *get* is the same as the order of the lines in the concrete structure.) Consider an arbitrary concrete string c, an abstract string $a = \mathit{get}\ c$, and an updated abstract string a' that is obtained by reordering chunks in a. Let us calculate the result of applying *put* to a' and c. By the above hypothesis, since a' was obtained by reordering the chunks in a, it is equal to the *get* of c' for some c' obtained from c by the corresponding reordering of chunks. By the GETPUT law, applying the *put* function to a' and c' is c'; by EQUIVPUT, applying the *put* function to a' and c also yields c'. Thus, quasi-obliviousness lets us derive an intuitive result: the *put* function translates reorderings of chunks in the abstract string as corresponding reorderings on the concrete string.

The EQUIVPUT law is useful both as a constraint on the design of lens primitives (in particular, dictionary lenses are designed with this principle in mind, for an equivalence based on reordering chunks) and as a guide for developing intuitions. Quasi-obliviousness does not, however, provide a complete specification of the correct handling of ordering in bidirectional languages. For example, it does not say what happens when the update to the abstract string deletes chunks or edits the value of a key. To capture such cases, one could formulate a condition stipulating that the *put* function must align chunks by key. Specifying this condition, however, requires talking about the sublens that operates on the chunks, which implies a syntactic representation of lenses analogous to dictionary lenses. We thus prefer to only consider the extensional law, even though it provides guarantees in fewer situations.

By design, each dictionary lens is quasi-oblivious with respect to an equivalence relation that can be read off from its syntax. The equivalence identifies strings up to key-respecting reorderings of chunks, where chunks are denoted by the occurrences of angle brackets, and keys by the sections each chunk marked using the `key` combinator. To see that every dictionary lens is quasi-oblivious with respect to this equivalence, observe that *parse* maps strings that are equivalent in this sense to identical skeletons and dictionaries, and recall that the *put* function for a dictionary lens (when viewed as a basic lens) wraps an invocation of *parse*, and of *put*, which operates on this skeleton and dictionary directly. It follows that *put* behaves the same on equivalent concrete strings.

Returning to the composers example, we can see that why the basic lens is bad and the dictionary lens is good: the equivalence the programmer had in mind for *both* versions was the one that can be read off from the second one—every line is a chunk, and the relative order of lines with different names should not affect how dates are restored by the *put* function. The first version of the lens, which operates positionally, is not quasi-oblivious with respect to this equivalence.

So far, we have focused on equivalence relations which are key-respecting reorderings of chunks. More generally, we can consider arbitrary equivalences on C. In the rest of this section, we investigate some properties of this more general view of quasi-oblivious lenses.

For a given basic lens l, there are, in general, many equivalence relations $\sim$ such that l is an quasi-oblivious lens with respect to $\sim$. We write $\mathcal{C}l(\sim)$ for the set of equivalence classes (i.e., subsets of the concrete domain) of $\sim$. Every lens l is trivially quasi-oblivious with respect to equality, the finest equivalence relation on C, and the relation $\sim_{max}$, defined as the coarsest equivalence for which l satisfies EQUIVPUT ($c{\sim_{max}}c'$ iff $\forall a.\ \mathit{put}\ a\ c = \mathit{put}\ a\ c'$). Between equality and the coarsest equivalence, there is a lattice of equivalence relations.

Given an equivalence relation, every concrete element c may be characterized by the data preserved in the abstract codomain and the rest of the data shared by every other view of the equivalence class containing c. That is, given $C_i \in \mathcal{C}l(\sim)$ and an abstract view a, there is at most one view c such that $c \in C_i$ and $l.\mathit{get}\ c = a$. Conversely, if two different concrete views map to the same a, then they must belong to different equivalence classes.

In the original lens paper (Foster et al. 2007b), two special classes of lenses are discussed. A lens $l \in C \Longleftrightarrow A$ is called *oblivious* if its *put* function ignores its concrete argument completely. A lens l is *very well behaved* if the effect of two *put*s in sequence just has the effect of the second—i.e., if $l.\mathit{put}\ a\ (l.\mathit{put}\ a'\ c) = l.\mathit{put}\ a\ c$ for every a, a', and c. (Very well behavedness is a strong condition and imposing it on all lenses would prevent writing many useful transformations. For example, note that neither variant of the composers lens is very well behaved: if we remove a composer and add the same composer back immediately after, the birth and death dates will be the default ones instead of the original ones. This may be seen as unfortunate, but the alternative is disallowing deletions!)

Interestingly, both of these conditions can be formulated in terms of $\sim_{max}$. A lens l is oblivious iff the coarsest relation $\sim_{max}$ satisfying EQUIVPUT is the total relation on C. Moreover, l is very well behaved iff $\forall C_i \in \mathcal{C}l(\sim_{max}).\ l.\mathit{get}\ C_i = A$. This condition puts the abstract codomain in a bijection with each equivalence class of $\sim$ and forces the operation of the *put* function to use the information in the abstract and concrete structures as follows: the concrete structure identifies an equivalence class C_i; the information contained in the abstract structure determines an element of C_i. This turns out also to be equivalent to the classical notion of view update translation under "constant complement" (Bancilhon and Spyratos 1981).

5. Boomerang

Programming with combinators alone is low-level and tedious. To make lens programing more convenient, we have implemented a high-level programming language called *Boomerang* on top of our core primitives.

Boomerang's architecture is simple: dictionary lens combinators are embedded in a simply typed functional language (we use the syntactic conventions of OCaml) built over the base types `string`, `regexp`, and `lens`. The language has all of the usual constructs: functions and `let`-definitions,[4] as well as constants for using dictionary lenses with the interface of a basic lens (as described in Section 3):

```
get : lens -> string -> string
put : lens -> string -> string -> string
create : lens -> string -> string
```

Evaluation in Boomerang is logically divided into two levels, in the style of Algol 60. At the first level, expressions are evaluated using the standard strategy of a call-by-value λ-calculus. This, in turn, may trigger the assembly (and type checking!) of a new dictionary lens value. The run-time representation of a dictionary lens value is a record of functions (representing the *get*, *parse*, *key*, *create*, and *put* components) and several finite-state automata (representing the concrete, abstract, skeleton, and dictionary components of the type); when a lens is built, the type checker checks the conditions mentioned in the typing rules using operations on these automata.

Using libraries and user-defined functions, it is possible to assemble large combinator programs quite rapidly. For example, the following user-defined function encapsulates the low-level details of escaping characters in XML. It takes a regular expression `excl` of excluded characters, and yields a lens mapping between raw and escaped PCDATA characters:

```
let xml_esc (excl:regexp) =
  copy ([^&<>\n] - excl)
  | ">" <-> "&gt;"
  | "<" <-> "&lt;"
  | "&" <-> "&amp;"
```

(When `xml_esc` is applied, the value passed for `excl` typically contains the "separators" of fields in the format; these are used by the type checker, e.g., to verify unambiguous iteration.)

Similarly, the next two functions handle the details of processing atomic values and entire fields in BibTeX and RIS-formatted bibliographic databases. They are defined in a context where `ws`, `quot_str`, `brac_str`, and `bare_str` are bound to the lenses used to process whitespace, quoted strings, strings enclosed in curly braces, and bare strings respectively.

```
let val (ld:string) (r:regexp) (rd:string) =
  del (ws . "=" . ws . ld) .
  copy r .
  del (rd . ws . "," . ws . "\n")

let field (bibtex:string) (ris:string) =
  let quot_val = val "\""  quot_str "\"" in
  let brac_val = val "{" brac_str "}" in
  let bare_val = val "" bare_str "" in
  let any_val = quot_val | brac_val | bare_val in
  ws . bibtex <-> ris . any_val . ins "\n"
```

The `val` function is used to tidy BibTeX values; when it is applied to a left delimiter string `ld`, a regular expression describing the value `r`, and a right delimiter string `rd`, it produces a dictionary lens that strips out the "=" character, whitespace, and delimiters. The `field` function takes as arguments strings representing the name of a BibTeX field (e.g. `title`) and the corresponding RIS field (`T1`) and produces a dictionary lens that maps between entire key-value pairs in each format.

The most significant challenges in implementing Boomerang come from the heavy use of regular expressions in its type system. Since the types of dictionary lenses involve regular languages, Boomerang's type checker needs to be able to decide equivalence, inclusion, and emptiness of regular languages, which are all standard. However, standard automata libraries do not provide operations for deciding unambiguous concatenation and iteration, so we implemented a custom automata library for Boomerang. Our library uses well-known techniques to optimize the representation of transition relations, and to recognize several fast paths in automata constructions. Even with these optimizations, as several operations use product constructions, the memory requirements can be significant. In our experience, performance is good enough for examples of realistic size, but we plan to investigate further optimizations in the future.

Because the type analysis performed by the dictionary lens type checker is so precise, many subtle errors—overlapping unions, ambiguous concatenations, etc.—are detected early. Boomerang supports explicit programmer annotations of dictionary lens types, written in the usual way as `let e : (C <-> A)`. It also has mechanisms for printing out inferred types and generating counterexamples when type checking fails. We have found all these features incredibly helpful for writing, testing, and debugging large lens programs.[5]

6. Experience

We have built Boomerang lenses for a variety of real-world formats, including an address book lens that maps between vCard, CSV, and XML; a lens that maps BibTeX and RIS bibliographic databases; and lenses for calculating simple ASCII views of LaTeX documents and iTunes libraries represented in XML as Apple Plists. Our largest Boomerang program converts between protein sequence databases represented in ASCII using the SwissProt format and XML documents conforming to the UniProtKB schema. For example, the following snippet of a SwissProt entry

```
OS   Solanum melongena (Eggplant) (Aubergine).
OC   Eukaryota; Viridiplantae.
OX   NCBI_TaxID=4111;
```

is mapped to a corresponding UniProtKB XML value:

```
<name type="scientific">Solanum melongena</name>
<name type="common">Eggplant</name>
<name type="synonym">Aubergine</name>
<dbReference type="NCBI Taxonomy" key="1" id="4111"/>
<lineage>
  <taxon>Eukaryota</taxon>
  <taxon>Viridiplantae</taxon>
</lineage>
```

Like many textual database formats, SwissProt databases are lists of entries consisting of tagged lines; our lens follows this structure. Entries are processed by the *match* combinator as distinct chunks, so that the information discarded by the *get* (e.g., metadata about each entry's creation date) can be restored correctly when updates involve reorderings. The identifier line provides a natural key. Other lines are processed using lenses specifically written for their data (of course, we factor out common code when possible). Most of these consist of simple filtering and reformatting (and swapping—see Section 8), and are therefore straightforward to write as dictionary lens combinators.

Interestingly, as we were developing this lens, the Boomerang type checker uncovered a subtle ambiguity in one of the lines that stems from the use of both "`,`" and "`;`" as separators. Some implicit conventions not mentioned in the specification avoid this ambiguity in practice (and we were able to revise our code to reflect these conventions). The precision of Boomerang's type system makes it a very effective tool for debugging specifications!

[4] Although it is semantically straightforward to define lenses by recursion (see Foster et al. (2007b)), Boomerang does not support recursive definitions as it would then be possible to define lenses with context-free types.

[5] And small ones! All the lenses and examples typeset in a typewriter font in this document were checked and run within the Boomerang system.

7. Related Work

Basic lenses were the starting point for this work. The original paper (Foster et al. 2007b) includes an extensive survey of the connections between basic lenses and the view update problem in the database literature. Basic lenses for relational structures, using primitives based on relational algebra, have also been developed (Bohannon et al. 2006). The combinators for tree lenses described in the first lens paper can be used to write lenses for lists encoded as trees, and all of the problems with ordered data described in the present work arise in that setting too. (These problems do not come up in the relational setting, since the structures handled there are unordered.)

Meertens's formal treatment of *constraint maintainers* for user interfaces (Meertens 1998, Section 5.3) recognizes the problem we are dealing with in this paper when operating on lists, and proposes a solution for the special case of "small updates" specified by edit operations, using a network of constraints between list entries. The idea of using constraints between concrete and abstract structures is related to our use of keys in dictionary lenses, but handling updates by translating edit operations represents a significant departure from the approach used in lenses, where "updates" are not given as operations, but by the updated value itself. The treatment of ordering for lists and trees in the bidirectional languages X and Inv (Hu et al. 2004; Mu et al. 2004), comes closest to handling the sorts of "resourceful updating" situations that motivate this work. Their approach is based on Meertens's ideas. As in his proposal, updates to lists in X are performed using edit operations. But rather than maintaining a correspondence between elements of concrete and abstract lists, the semantics of the edit operation is a function yielding a tagged value indicating which modification was performed by the edit. The structure editor described in (Hu et al. 2004) based on X does handle single *insert* and *delete* operations correctly by propagating these modification tags locally in lists. However, the *move* operation is implemented as a *delete* followed by an *insert*. This means that the association between the location of the moved element in the concrete and abstract lists is not maintained, and so moved data is populated with default values at the point of insertion; e.g., our composers example is not handled correctly.

There is a large body of work on bidirectional languages for situations in which round-trips are intended to be bijective modulo "ignorable information" (such as whitespace). XSugar (Brabrand et al. 2005) is a bidirectional language that targets the special case when one structure is an XML document and the other is a string. Transformations are specified using pairs of intertwined grammars. A similar bidirectional language, biXid (Kawanaka and Hosoya 2006), operates just on XML data. The PADS system (Fisher and Gruber 2005) makes it possible to generate a data type, parser, and pretty printer for an ad-hoc data formats from a single, declarative description. PADS comes with a rich collection of primitives for handling a wide variety of data including characters, strings, fixed-with integers, floating point values, separated lists, etc. Kennedy's combinators (2004) describe pickler and unpicklers. Benton (2005) and Ramsey (2003) both describe systems for mapping between run-time values in a host language and values manipulated by an embedded interpreter. In all of these systems, as round-trips are intended to be essentially bijective, the problems with reordering that our dictionary lenses are designed to solve do not come up.

JT (Ennals and Gay 2007) synchronizes programs written in different high level languages, such as C and Jekyll, an extension of C with features from ML. JT relies on a notion of distance to decide how to propagate modifications, allowing the detection of non local edits such as the swap of two functions. The synchronization seems to work well in many cases but there is no claim that the semantics of the synchronized programs are the same.

Recently, Stevens (2007) has applied the ideas of basic lenses in the context of *model transformations* (leaving aside issues of ordering, for the moment, though this is a goal for future work).

Our lens combinators are based on finite-state transducers, which were first formulated as multitape automata by Scott and Rabin (1959). Languages based on finite-state automata have been developed, largely in the area of natural language processing; the collection edited by Roche and Schabes gives a survey (1996). Mechanized checking for string processing languages that, like Boomerang, have type systems based on regular automata have also been studied (Tabuchi et al. 2002).

8. Extensions and Future Work

This final section presents some extensions to the basic design described in previous sections—including both ideas we have already implemented in Boomerang and ones we leave for future work. We discuss a range of topics including additional combinators, implementation optimizations, and stronger semantic constraints.

We first consider extensions to the set of operators, starting with *sequential composition*. Extending the grammar of skeletons with a new form of pairs, $\langle S_1, S_2 \rangle$, and writing $X \otimes Y$ for $\{\langle x, y \rangle \mid x \in X, y \in Y\}$, we can define the sequential composition of l_1 and l_2 as follows.

$$\boxed{\begin{array}{l}
\dfrac{l_1 \in C \overset{S_1,L}{\Longleftrightarrow} B \qquad l_2 \in B \overset{S_2,L}{\Longleftrightarrow} A}{l_1 \,;\, l_2 \in C \overset{S_1 \otimes S_2, L}{\Longleftrightarrow} A} \\[2ex]
\begin{array}{lcl}
\textit{get } c & = & l_2.\textit{get } (l_1.\textit{get } c) \\
\textit{parse } c & = & \langle s_1, s_2 \rangle, (d_2 +\!\!+\, d_1) \\
\quad \text{where } s_1, d_1 = l_1.\textit{parse } c & & \\
\quad \text{and } s_2, d_2 = l_2.\textit{parse } (l_1.\textit{get } c) & & \\
\textit{key } a & = & l_2.\textit{key } a \\
\textit{create } a\ d & = & c, d_2 \\
\quad \text{where } b, d_1 = l_2.\textit{create } a\ d & & \\
\quad \text{and } c, d_2 = l_1.\textit{create } b\ d_2 & & \\
\textit{put } a\ (\langle s_1, s_2 \rangle, d) & = & c, d_2 \\
\quad \text{where } b, d_1 = l_2.\textit{put } a\ (s_2, d) & & \\
\quad \text{and } c, d_2 = l_1.\textit{put } b\ (s_1, d_2) & &
\end{array}
\end{array}}$$

Sequential composition is very useful in practice when some preprocessing of data is needed so that every chunk with the same information actually belongs to the same regular language. For instance, suppose that the concrete language is a concatenation of substrings belonging to $\mathtt{x}^* \cdot \mathtt{y}^* \cdot \$ \cdot \mathtt{z}^*$ followed by a separator "#" and then a concatenation of substrings belonging to $\mathtt{x}^* \cdot \mathtt{y}^* \cdot \mathtt{z}^* \cdot \$$. If we want to ignore the position of the symbol \$ and process these substrings uniformly as chunks, allowing reordering to occur freely between any chunk, we can use the following lens:

$$\begin{array}{l}
l_{\#} = \\
(\textit{copy } (\mathtt{x}^*{\cdot}\mathtt{y}^*) \cdot \textit{del } \$ \cdot \textit{copy } \mathtt{z}^* \cdot \textit{ins } \$)^* \cdot \textit{copy } \# \cdot \\
\quad (\textit{copy } (\mathtt{x}^*{\cdot}\mathtt{y}^*{\cdot}\mathtt{z}^*{\cdot}\$))^*; \\
\langle l_{\$} \rangle^* \cdot \textit{copy } \# \cdot \langle l_{\$} \rangle^*
\end{array}$$

With $c = \mathtt{xy\$zxxyy\$zz\#zzz\$xxxyyy\$}$, we have

$$\begin{array}{rcl}
l_{\#}.\textit{get } c & = & \mathtt{xz\$xxzz\$\#zzz\$xxx\$} \\
\overline{l_{\#}}.\textit{put } \mathtt{xxxzzz\$\#xxzz\$}\ c & = & \mathtt{xxxyyy\$zzz\#xxyyzz\$}
\end{array}$$

as desired. However, the interactions of sequential composition with parsing and dictionaries are somewhat tricky because, in general, each lens in a composite can have its own notion of chunk. Thus, we leave a full investigation of the composition operator as future work.

The *get* components of the string lens combinators we have described—*copy* and *const* closed under the regular operators and composition—are all expressible as standard one-way finite state transducers. This class contains many useful transformations, powerful enough to express a large collection of examples, but has a fundamental limitation: the restriction to finite state means that it is impossible for a lens to "remember" arbitrary amounts of data. For example, with the basic combinators, we cannot write a variant of the composers example where the order of the name and nationality are inverted in the view:

```
"Finnish, Jean Sibelius
 American, Aaron Copland"
```

Lifting this restriction poses no semantic problems, and the actual set of dictionary lenses implemented in Boomerang includes primitives for swapping and sorting arbitrary data. For example, the combinator *swap* $l_1\ l_2$ swaps the views computed by the *get* functions—i.e., $c_1{\cdot}c_2$ maps to $(l_2.\mathit{get}\ c_2){\cdot}(l_1.\mathit{get}\ c_1)$—and unswaps the results computed by the *put* functions.

$$\frac{\begin{array}{c} l_1 \in C_1 \overset{S_1,L}{\Longleftrightarrow} A_1 \quad C_1 \cdot^! C_2 \\ l_2 \in C_2 \overset{S_2,L}{\Longleftrightarrow} A_2 \quad A_2 \cdot^! A_1 \end{array}}{\mathit{swap}\ l_1\ l_2 \in C_1{\cdot}C_2 \overset{S_1 \times S_2, L}{\Longleftrightarrow} A_2{\cdot}A_1}$$

$$\begin{array}{lcl}
\mathit{get}\ c_1{\cdot}c_2 & = & (l_2.\mathit{get}\ c_2){\cdot}(l_1.\mathit{get}\ c_1) \\
\mathit{parse}\ c_1{\cdot}c_2 & = & (s_1, s_2), d_2 +\!\!+\ d_1 \\
\quad \text{where } s_i, d_i = l_i.\mathit{parse}\ c_i & & \\
\mathit{key}\ a_2{\cdot}a_1 & = & l_2.\mathit{key}\ a_2 \cdot l_1.\mathit{key}\ a_1 \\
\mathit{create}\ a_2{\cdot}a_1\ d_1 & = & c_1{\cdot}c_2, d_3 \\
\quad \text{where } c_2, d_2 = l_2.\mathit{create}\ a_2\ d_1 & & \\
\quad \text{and } c_1, d_3 = l_1.\mathit{create}\ a_1\ d_2 & & \\
\mathit{put}\ a_2{\cdot}a_1\ ((s_1, s_2), d_1) & = & c_1{\cdot}c_2, d_3 \\
\quad \text{where } c_2, d_2 = l_2.\mathit{put}\ a_2\ (s_2, d_1) & & \\
\quad \text{where } c_1, d_3 = l_1.\mathit{put}\ a_1\ (s_1, d_2) & &
\end{array}$$

A lens that computes the above transformation for a single composer is the following:

```
swap (key ALPHA)
     (del (", " . YEARS . ", ") . ALPHA . ins ", ")
```

Swap and other related primitives are used critically in the lenses for real-world data that we have built, including the BibTeX and SwissProt lenses. Building on swap and union, we have also defined lenses for handling simple forms of sorting: e.g., where the concrete domain is the interleaving of a list of regular languages and the abstract codomain has the languages in list order. A related generalization of the iteration combinator yields another kind of sorting. It partitions a sequence by moving elements that do not belong to a given regular expression to the end.

We have also implemented a generalized version of the *match* combinator. The lens described in Section 3 uses a single dictionary structure to match concrete chunks with pieces of the abstract string having the same key. This matching is performed globally across the entire concrete string. Nested match combinators provide one way to limit the scope of matching, but in other situations, it is convenient to use matchable regions with several distinct sorts, and to have the matching of chunks with different sorts kept separate. In our implementation, we generalize the design described above by fixing a set T of string "tags", and associating each $\langle l \rangle$ combinator to a tag. The type D of dictionaries is similarly generalized from finite maps from keys K to lists, to finite maps from tags T to K to lists. This allows us to express transformations in which different pieces of the input and view are matched globally, using different dictionaries.

Thus far, we have assumed that keys are generated by concatenating the non-empty substrings generated by the `key` combinator. In some situations it is useful to ignore parts of the key (e.g., with nested matches), or to add a fixed string to a key (e.g., to separate keys that are concatenated). Boomerang includes primitives to do these operations on keys. More generally, our basic design could also be extended so that keys are represented by richer structures—e.g., strings, sets, and lists—and add combinators for transforming between different kinds of keys. For example, the iteration combinator could return the list of keys of chunks instead of their flat concatenation, and this list could be further transformed into a set, if order is not important, or concatenated into a string.

Many languages where transformations are bijective modulo "ignorable information"—e.g., whitespace and minor formatting details—have been proposed. We are investigating the generalization of this idea to lenses in the domain of strings. The *get* and *put* functions may discard ignorable information, and the lens laws are only required modulo the same information. As an example, we have implemented a variant of the *const* lens whose *get* function produces a constant string, but whose *put* function accepts a regular language. This variant is used in the SwissProt lens to indicate that whitespace occurring in XML documents can be ignored.

We now turn the discussion to ongoing work concerning the efficiency of our implementation. Our Boomerang interpreter is based on an NFA representation of finite automata. These are used to decide the side conditions in the typing rules of dictionary lenses, and for operations such as splitting a string belonging to unambiguous concatenations into unique substrings. The performance of this implementation is good enough for examples of realistic size, but we would like to engineer an optimized implementation in the future. One possibility is to compile regular expressions to DFAs using derivatives (Brzozowski 1964). Although DFAs can be exponentially larger than NFAs, string matching is much faster, and they can be constructed lazily. The main challenge is developing efficient techniques for deciding non-ambiguity directly.

Another extension we would like to explore is streaming string lenses. This is motivated by large examples such as SwissProt, where the size of the concrete string is on the order of 1GB! We would like to develop a variant of the iteration combinator whose *get* function processes elements one at a time, rather than operating on a string representing the whole sequence. Similarly, the *put* function would operate on elements of the abstract string one at a time. Of course, the *put* function also needs a dictionary that, in the current design, represents the entire concrete string. To optimize the memory requirements, we are also investigating an extension in which only the minimum information needed to satisfy GetPut is kept in skeletons and dictionaries. For example, *copy* could produce a trivial skeleton rather than copying the entire string (which is ignored by *put*). Interestingly, these minimal "complements" to the *get* function, would result in *parse* functions that calculate the coarsest equivalence satisfying PutEquiv. Thus, they may provide insights into semantic properties of lenses such as very well behavedness.

Semantics is another area of ongoing work. The EquivPut law is stated with respect to an equivalence relation on the concrete domain $\sim_C$. As observed, this equivalence arises naturally from a dictionary lens. Moreover, $\sim_C$ also induces an equivalence on the abstract codomain by taking the image of $\sim_C$ under *get*. However, our lenses do not guarantee that the induced equivalence is expressible solely in terms of chunks and keys in the abstract codomain. We are investigating an extension of dictionary lenses that gives rise to an explicit equivalence on the abstract codomain $\sim_A$. With such an equivalence in hand, we would then like to ensure that our lenses translate equivalence-preserving updates to equivalence-preserving updates. The first step is to check that our lenses satisfy the law

$$\frac{l.\mathit{get}\ c \sim_A a'}{\exists c'.c \sim_C c' \wedge l.\mathit{get}\ c' = a'} \qquad (\text{EquivExists})$$

which asserts, in the case where the equivalences are based on key-respecting reorderings of chunks, that every reordering of chunks in an abstract string can be realized as a corresponding reordering of chunks in the concrete domain. Combining this with EQUIVPUT, we can prove a derived rule

$$\frac{l.get\ c \sim_A a'}{l.put\ a'\ c \sim_C c} \qquad (\text{GETPUTEQUIV})$$

which states that reorderings on A *are*, in fact, translated as reorderings on C.

Finally, we plan on investigating resourceful and quasi-oblivious lenses for trees, relations, and graphs.

Acknowledgments

We are grateful to Peter Buneman, Perdita Stevens, Stijn VanSummeren, Val Tannen, James Cheney, Ravi Chugh, and the members of the Penn PLClub, CLunch, and Database Groups for stimulating discussions, and to Michael Greenberg for helpful comments on an earlier draft of the paper. Our work is supported by the National Science Foundation under grant IIS-0534592 *Linguistic Foundations for XML View Update*. Nathan Foster is also supported by an NSF Graduate Research Fellowship.

References

Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: Arrows for invertible programming. In *ACM SIGPLAN Workshop on Haskell*, pages 86–97, 2005.

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

Nick Benton. Embedded interpreters. *Journal of Functional Programming*, 15(4):503–542, 2005.

Jean Berstel, Dominique Perrin, and Christophe Reutenauer. *Codes and Automata*. 2005. Manuscript available from `http://www-igm.univ-mlv.fr/~berstel/LivreCodes/`.

Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. Technical Report MS-CIS-07-15, Dept. of CIS University of Pennsylvania, November 2007.

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. In *Database Programming Languages (DBPL), Trondheim, Norway*, volume 3774 of *Lecture Notes in Computer Science*, pages 27–41. Springer-Verlag, August 2005.

Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *International Conference on Database Theory (ICDT), London, UK*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2001.

Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. *VLDB Journal*, 12(1):41–58, 2003.

Robert Ennals and David Gay. Multi-language synchronization. In *European Symposium on Programming (ESOP), Braga, Portugal*, 2007. To appear.

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.

J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 73(4):669–689, June 2007a. Extended abstract in *Database Programming Languages (DBPL)* 2005.

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

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.

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.

Andrew J. Kennedy. Functional pearl: Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.

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

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

Benjamin C. Pierce et al. Harmony: A synchronization framework for heterogeneous tree-structured data, 2006. `http://www.seas.upenn.edu/~harmony/`.

M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

Norman Ramsey. Embedding an interpreted language using higher-order functions and types. In *ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators (IVME), San Diego, CA*, pages 6–14, 2003.

Emmanuel Roche and Yves Schabes, editors. *Finite-State Language Processing*. MIT Press, 1996.

Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *MODELS*, 2007.

Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. Regular expression types for strings in a text processing language. In *Workshop on Types in Programming (TIP), Dagstuhl, Germany*, volume 75 of *Electronic Notes in Theoretical Computer Science*, pages 95–113, 2002.

A. Manual Lenses

In this appendix we entertain a heretical proposition—that bidirectional languages might not be worth the trouble. Might it not be simpler just to write the *get* and *put* components as separate functions in a general-purpose programming language?[6] To evaluate this proposition (and ultimately reject it), we consider OCaml definitions of the *get* and *put* functions for the composers example from the introduction, and then step through the reasoning needed to verify that they are total and obey the lens laws.

The `get` function is written as follows:

```
let get c =
  let comps_c = split '\n' c in
  let comps_a = List.map
    (fun ci ->
      let [n;y;c] = split ',' ci in
```

[6] Of course, there are many semantically valid lenses whose *get* and *put* functions can *only* be expressed in a general-purpose language—or, equivalently, by adding new primitives to Boomerang. Whether or not our choice to focus on finite-state string transductions hits a "sweet spot" between expressiveness and tractable reasoning is a different question—one whose answer will require more experience with additional real-world examples.

```
        join ',' [n;c])
      comps_c in
    join '\n' comps_a
```

It splits the input into a list of lines, maps a function over this list that retains the name and country from each line, and joins the resulting list. The corresponding `put` function is:

```
let put a c =
  let a_s,c_s = split '\n' a, split '\n' c in
  let c_s_assoc = List.map
    (fun ci -> let [n;y;c] = (split ',' ci) in (n,y))
    c_s in
  let cs' = List.fold_left
    (fun (acc, assoc) ai ->
       let [n;c] = split ',' ai in
       let yi = assoc_find n assoc " 0000-0000" in
       let assoc' = assoc_remove n assoc in
       ((join ',' [n;yi;c])::acc, assoc'))
    ([], c_s_assoc) a_s in
  join '\n' (List.rev (fst cs'))
```

This splits the concrete and abstract arguments into lists of lines, and then constructs an association list from the concrete list in which every name is paired with the corresponding year range. Next, it folds down the abstract list, locates a year range from the association list (using a default when none is found), and concatenates the name, dates, and nationality together. Finally, it joins the resulting list, yielding the new concrete view.

These two functions have the same behavior as the second lens from the introduction. To finish the job we need to prove that each function is total and that the pair satisfies the lens laws. Demonstrating totality is not difficult, although we need to say what type they are total at. They are not total functions on the set of all strings—e.g., when `get` is applied to

```
"Jean\nSibelius, 1865-1957, Finnish"
```

neither line yields a list of length three when split by `','` which triggers a `Match_failure` exception. To prevent such failures, we can wrap the bare functions with code that checks that the arguments match the regular expressions

```
[A-Za-z ]+, [0-9]{4}-[0-9]{4}, [A-Za-z ]+
```

and:

```
[A-Za-z ]+, [A-Za-z ]+
```

With this modification, checking totality is straightforward.

To verify the GETPUT law, we have to consider an arbitrary concrete string together with the unique abstract string produced from it by the `get` function, and show that applying `put` to these arguments yields the original concrete string. We can do this in three steps. First, we check that the string obtained by joining `comps_a` produced at the end of `get` splits into the same list at the start of `put`. Second, we check that each step of the `List.fold_left` locates the correct year range for the composer. This requires a few additional steps of reasoning about the value in the `assoc` accumulator as it is threaded through the fold (in particular, if the list contains repeated names, then we must verify that the corresponding year ranges are restored positionally). Finally, we check that the order of elements in the updated concrete list is the same as in the original concrete list. Checking PUTGET is similar but simpler, since the year ranges produced by `put` do not matter—they are discarded by `get`. Checking EQUIVPUT is also straightforward since the `put` function only uses the concrete string via the association list it constructs, and every concrete list containing the same names and dates maps to the same association list.

By contrast, the Boomerang version of the same lens (written here with explicit regular expressions)

```
let comp = key [A-Za-z ]+ . copy ", "
         . del ([0-9]{4} . "-" . [0-9]{4} . ", ")
         . copy [A-Za-z ]+

let comps = "" | <comp> . ("\n" . <comp>)*
```

consists of a single phrase which is only a little more complicated than the regular expressions that we had to add to the OCaml program to ensure totality. Moreover, types are inferred automatically, and well-typedness implies the lens laws.

All this is nice. But the *real* benefits of using a bidirectional language become apparent when the lens evolves. Suppose that, for some reason, we decide that the character used to separate the fields in each line should be "`!`". Changing the Boomerang program requires two local changes—one for each occurrence of a `","`. The `get` and `put` functions and inferred types all change automatically. By contrast, the OCaml functions and regular expressions have eight occurrences of "`,`", and these are scattered across two functions and two regular expressions! Moreover, the totality and lens law proofs must each be rechecked by hand. In particular, we need to verify that changing the separator does not introduce an ambiguity that breaks the property that `split` and `join` are inverses.

Of course, this change could have been made simpler by defining the separator as a constant. But now suppose we need to change the concrete format to:

```
Jean Sibelius: 1865-1957, Finnish
```

The lens program only requires one change. The OCaml functions, however, require significant modifications because a single invocation of `split` for each line is no longer enough. Instead, we have to split each line by "`:`", and then again by "`,`". Making this change requires touching several lines of code—and correspondingly deep revisions to the proofs of totality and the lens laws. At this point, the urge to cut corners—make changes to the code but skip fixing the proofs (if indeed they were written out in the first place)—will be strong.

Thus, even for this nearly trivial example, a bi-directional language is a much more attractive option. The low-level, two-function approach is surprisingly difficult to get right in the first place and even more difficult to imagine maintaining.