Quotient Lenses

J. Nathan Foster

University of Pennsylvania

Alexandre Pilkiewicz

École Polytechnique and INRIA

Benjamin C. Pierce

University of Pennsylvania

Abstract

There are now a number of *bidirectional programming languages*, where every program can be read both as a forward transformation mapping one data structure to another and as a reverse transformation mapping an edited output back to a correspondingly edited input. Besides parsimony—the two related transformations are described by just one expression—such languages are attractive because they promise strong behavioral laws about how the two transformations fit together—e.g., their composition is the identity function. It has repeatedly been observed, however, that such laws are actually a bit too strong: in practice, we do not want them "on the nose," but only up to some equivalence, allowing inessential details, such as whitespace, to be modified after a round trip. Some bidirectional languages loosen their laws in this way, but only for specific, baked-in equivalences.

In this work, we propose a general theory of *quotient lenses*—bidirectional transformations that are well behaved modulo equivalence relations controlled by the programmer. Semantically, quotient lenses are a natural refinement of *lenses*, which we have studied in previous work. At the level of syntax, we present a rich set of constructs for programming with *canonizers* and for quotienting lenses by canonizers. We track equivalences explicitly, with the type of every quotient lens specifying the equivalences it respects.

We have implemented quotient lenses as a refinement of the bidirectional string processing language Boomerang. We present a number of useful primitive canonizers for strings, and give a simple extension of Boomerang's regular-expression-based type system to statically typecheck quotient lenses. The resulting language is an expressive tool for transforming real-world, ad-hoc data formats. We demonstrate the power of our notation by developing an extended example based on the UniProt genome database format and illustrate the generality of our approach by showing how uses of quotienting in other bidirectional languages can be translated into our notation.

1. Introduction

"Good men must not obey the laws too well"

—R W Emerson

Most programs compute in a single direction, from input to output. But it is often useful to take a modified *output* and "compute backwards" to obtain a correspondingly modified *input*. For example, if we have a transformation mapping from a simple XML database format describing classical composers...

```
<composers>
  <composer>
    <name>Jean Sibelius</name>
    <years birth="1865" death="1956"/>
    <nationality>Finnish</nationality>
  </composer>
</composers>
```

... to comma-separated lines of ASCII...

```
Jean Sibelius, 1865-1956
```

... we may want to be able to edit the ASCII output (e.g., to correct the erroneous death date above) and push the change back into the original XML. The need for such *bidirectional transformations* arises in many diverse areas of computing, including data synchronizers (Foster et al. 2007a,b), parsers and pretty printers (Fisher and Gruber 2005; Eger 2005), marshallers and unmarshallers (Ramsey 2003; Kennedy 2004), structure editors (Hu et al. 2004), graphical user interfaces (Meertens 1998; Evers et al. 2006; Greenberg and Krishnamurthi 2007), software model transformations (Stevens 2007; Xiong et al. 2007), system configuration management (Lutterkort 2007), schema evolution (Miller et al. 2001; Cunha et al.; Berdaguer et al. 2007), and databases (Bancilhon and Spyratos 1981; Dayal and Bernstein 1982; Bohannon et al. 2006, etc.).

In previous work (Foster et al. 2007b; Bohannon et al. 2006, 2008), we have used the term *lens* to describe a bidirectional program. Formally, a lens l mapping between a set C of "concrete" structures and a set A of "abstract" ones comprises three functions:

$$l.get \ \in \ C \longrightarrow A$$
$$l.put \ \in \ A \longrightarrow C \longrightarrow C$$
$$l.create \ \in \ A \longrightarrow C$$

The *get* component is the forward transformation, a total function from C to A. The *put* component takes an old C and a modified A and yields a correspondingly modified C. The *create* component handles the special case where we want to compute a C from an A but we have no C to use as the "old value"; *create* uses defaults to fill in any information in C that is thrown away by the *get* function (such as the nationality of each composer in the example above). Every lens obeys the following "round-tripping" laws for every $c \in C$ and $a \in A$:

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

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

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

The first law states that the *put* function must restore all the information discarded by *get* when its arguments are an abstract structure and a concrete structure that generates the very same abstract structure. The second and third laws state that *put* and *create* must propagate all of the information contained in their abstract arguments to the concrete structure they produce. These laws express fundamental expectations about how the components of a lens should work together; they are closely related to classical conditions on correct *view update translation* developed in the database community (see Foster et al. 2007b). The set of all lenses mapping between C and A is written $C \Longleftrightarrow A$.

The naive way to build a lens is simply to write three separate functions (*get*, *put*, and *create*) in a general-purpose programming language, and check manually that they satisfy the lens laws. But this is unsatisfactory for all but the

simplest lenses: the three functions will be very redundant, since each of them will embody the structure of both C and A—a maintenance nightmare. A better alternative is to design a *bidirectional programming language* in which every expression can be read both from left to right (as a *get* function) and from right to left (as *put* or *create*). Besides avoiding redundancy, this approach permits us to carry out the proofs of the behavioral laws once and for all, by designing a type system in which—by construction—every well-typed expression denotes a well-behaved lens. Many such programming languages have been proposed (Foster et al. 2007b; Bohannon et al. 2006, 2008; Meertens 1998; Kennedy 2004; Benton 2005; Ramsey 2003; Hu et al. 2004; Matsuda et al. 2007; Brabrand et al. 2007; Kawanaka and Hosoya 2006; Fisher and Gruber 2005; Alimarine et al. 2005).

Now comes the fly in the ointment. The story we've told so far is appealing... but not perfectly true! Most bidirectional languages for processing real-world data do *not* guarantee the behavioral laws we have given—or rather, they guarantee them only "modulo insignificant details." The nature of these details varies from one application to another; examples include whitespace, artifacts of representing graph or tree structures such as XML as text (order of XML attributes, etc.), escaping of atomic data (XML PCDATA, vCard and BibTeX values), ordering of fields in record-structured data (BibTeX fields, XML attributes), breaking of long lines in ASCII formats (RIS bibliographies, UniProtKB genomic data bases), and duplicated information (aggregated data, tables of contents).

To illustrate, consider the composers example again. The information about each composer could be larger than fits comfortably on a single line in the ASCII format, especially if the example were more complex. We might then want to relax the abstract schema so that a line could be broken (optionally) using a newline followed by at least one space, so that

```
Jean Sibelius,
  1865-1956
```

would be accepted as an equivalent, alternate presentation of the data in the original example. But now we have a problem: the PUTGET law is only be satisfied when the *put* function is injective in its first (A) parameter. But this means that

```
Jean Sibelius, 1865-1956
```

and

```
Jean Sibelius,
  1865-1956
```

must map to different XML trees—the presence or absence of linebreaks must be reflected in the concrete structure produced by *put*. We *could* construct a lens that does this—e.g., storing the line break inside the PCDATA string containing the composer name...

```
<composers>
  <composer>
    <name>Jean Sibelius
  </name>
    <years birth="1865" death="1956"/>
    <nationality>Finnish</nationality>
  </composer>
</composers>
```

... but this "solution" isn't especially attractive. For one thing, it places an unnatural demand on the XML representation (indeed, possibly an unsatisfiable demand—e.g., if the application that uses the XML data assumes that the PCDATA does not contain newlines). For another, writing the lens so that it handles and propagates linebreaks correctly is going to involve some extra work. And finally, this fiddly work and warping of the XML format is all for the purpose of maintaining information that we actually don't care about!

A better alternative is to relax the lens laws. There are several ways to do this.

1. We can be a bit informal, stating the laws in their present form and explaining that they "essentially hold" for our program, perhaps supporting this claim by giving some algorithmic description of how inessential details are
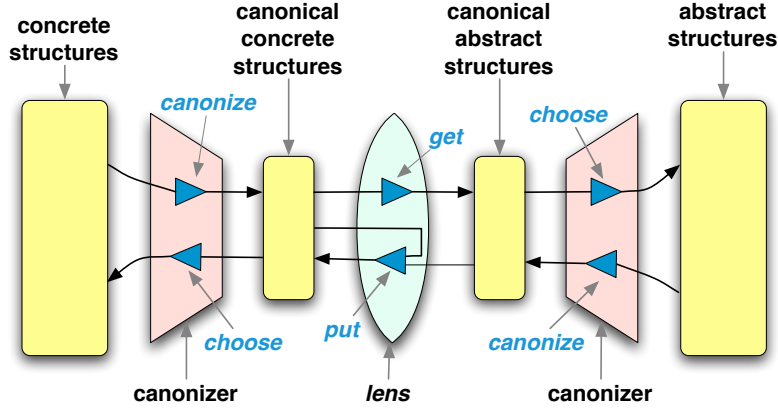
Figure 1. Architecture with "canonizers at the edges"

processed. For many purposes such informality may be perfectly acceptable, and several bidirectional languages adopt this strategy. For instance, biXid (Kawanaka and Hosoya 2006), a language for describing XML to XML conversions using pairs of intertwined regular tree grammars, provides no explicit guarantees about the round-trip behavior of the transformations its programs describe, but the clear intention is that they should be "morally bijective."

2. We can keep the same basic structures but claim weaker laws. For example, X (Hu et al. 2004) is a general-purpose bidirectional language with a *duplication* operator. This operator makes it possible to express many useful transformations—e.g., augmenting a document with a table of contents—but because the duplicated data is not preserved exactly on round-trips, the PUTGET law does not hold. Instead, X programs satisfy a "round-trip and a half" variant that is significantly weaker:

$$\frac{c' = \text{put } a \ c}{\text{put } (\text{get } c') \ c' = c'} \tag{PUTGETPUT}$$

This law still imposes some constraint on the behavior of lenses, but opens the door to a wide range of unintended behaviors. For example, a lens whose *put* component is a constant function *put a c = c′* is considered well behaved.[1]

3. We can give a more precise account of the situation—yielding better principles for programmers to reason about the behavior of their programs—by splitting bidirectional programs into a "core component" that is a lens in the strict sense plus "canonization" phases at the beginning and end that standardize the representation of whitespace (or whatever) and makes sure the pure lens part only has to work with a particular representative of each equivalence class. See Figure 1.

For example, in our earlier language for lenses on trees (Foster et al. 2007b), the end-to-end transformations on actual strings (e.g., concrete representations of XML trees in the filesystem) only obey the lens laws up to the equivalence induced by a *viewer*—a pair of functions mapping between strings and trees (e.g., an XML parser and printer). Similarly, XSugar (Brabrand et al. 2007), a language for converting between XML and ASCII, guarantees that its transformations are bijective modulo a fixed relation on input and output structures

---

[1] Later work by the same authors (the journal version of Hu et al. 2004 and Liu et al. 2007) excludes such examples by annotating data with "edit tags," ordering this data according to a "more edited than" relation, and adding a new law stipulating that a *put* followed by a *get* should yield a more edited abstract structure.

that is obtained by canonizing XML, data matching special "unordered" productions, and certain "ignorable" non-terminals.[2]

This approach is quite workable as long as the data formats and canonizers are generic (e.g., XML parsers and printers). However, for ad-hoc formats, such as textual genome databases, bibliographies, configuration files, etc., this approach rapidly becomes impractical because the two directions of the canonization transformation themselves become difficult to write and maintain. In particular the structure of the data is recapitulated, redundantly, in the lens and in each direction of the canonizer. In other words, we wind up back in the situation that lenses were designed to avoid! In our experience, these difficulties quickly become unmanageable for many formats of practical interest.

4. We can develop a more refined account of the whole semantic and syntactic framework allowing us to say, precisely and truthfully, that the lens laws hold of a given program *modulo a particular equivalence* that can be calculated from the program, with explicit constructs for defining and applying canonizers anywhere in a program, not just at the edges. This is the goal of this paper.

At the semantic level, the refinement is straightforward, as we show in Section 2. We enrich the types of lenses with equivalence relations—instead of $C \Longleftrightarrow A$, we write $C/{\sim_C} \Longleftrightarrow A/{\sim_A}$, where $\sim_C$ is an equivalence on C and $\sim_A$ is an equivalence on A, and we relax the lens laws accordingly. We call the structures inhabiting such types *quotient lenses*, or *q-lenses* for short. (When we need to distinguish them, we use the term *basic lenses* for the original, un-quotientied ones we have studied previously.)

The story is much more interesting on the syntactic side. Our goal is to discover basic principles for bidirectional languages that treat quotienting explicitly—principles that will be useful in other domains besides strings, such as bidirectional UML model transformation (Stevens 2007). We therefore begin our investigation of syntax in Section 3, in a completely generic setting that is independent of the particular domain of structures over which the lenses operate. We propose a general notion of *canonizers*, structures that map (bidirectionally!) between a set of structures and a set of normalized representatives; we then develop operations for *quotienting* a q-lens by a canonizer; and we show how lenses themselves can be converted into canonizers. A pleasant corollary of this last observation is that many of our core primitives can be used both as lenses as canonizers . Better yet, q-lens composition and quotienting may be composed freely with other operations such as concatenation and union, allowing more compact descriptions in which canonization is interleaved with other processing, instead of occurring only "at the edges" as in Figure 1.

It is important to ground this sort of language design in experiments with real-world examples. To support such experiments, in the later sections of the paper we extend Boomerang, a language for writing lenses on strings whose primitives are based on finite state transductions (Bohannon et al. 2008), with canonizers and quotient operators. We show how to reinterpret the core string lens combinators—lenses for copying and deleting data, sequential composition, and the rational operators union, concatenation, and Kleene-star—as q-lenses in Section 4, and we prove that the resulting structures are well behaved according to the more refined q-lens behavioral laws.[3] Section 5 introduces primitive q-lenses and canonizers for strings.

In Section 6, we discuss an unexpected side benefit of our design: we can use canonizers to simplify overly complex types, significantly mitigating the difficulties of programming with the extremely precise regular types that arise in Boomerang.

In Section 7 we consider typechecking algorithms for q-lenses in Boomerang. The challenge here is choosing a tractable syntactic presentation for the equivalence relations appearing in types. We first give a simple, coarse

---

[2] The XML canonization component is treated as a distinct "pre-processing" phase. Canonization of other ignorable data is interleaved with other processing; in this respect, XSugar can be regarded as a special case of the framework we are proposing here. We return to this comparison in more detail in Section 9.

[3] Readers familiar with Boomerang may recall that it is based not on simple string lenses as we have described them here but on *dictionary lenses*, which incorporate extra mechanisms for correctly handling ordered lists of records identified by keys. The semantics in (Bohannon et al. 2008) also relies on an equivalence relation for describing "reorderable chunks of data." The two equivalence relations are unrelated but compatible: dictionary lenses and q-lenses can be combined very straightforwardly; see Section 10.

representation, classifying each equivalence relation as either "the identity" or "something other than the identity." Surprisingly, this very simple analysis suffices for all of our examples. We then discuss a more precise technique where equivalence relations are represented by rational functions; this technique yields a more flexible type system, but appears prohibitively expensive to implement.

Section 8 demonstrates the utility of our language by describing some large q-lenses we have built, including a generic library of lenses for processing XML and an XML-to-ASCII converter for the UniProt genomic database format (Bairoch et al. 2005).

Finally, to illustrate the generality of our constructions, we show in Section 9 how a large subset of XSugar and many uses of quotienting in biXid and X can be translated into our notation.

2. Foundations

We begin our technical development by describing the semantic space of q-lenses. This part of our account and the fundamental combinators described in Section 3 are both completely generic: we make no assumptions about the universes of concrete and abstract structures. Later we will instantiate these universes to strings and introduce additional q-lens combinators that work specifically with strings.

The definition of q-lenses is a straightforward refinement of basic lenses: we enrich the domain and codomain types with equivalences, loosen the lens laws appropriately, and add a few natural conditions on how the lens components and equivalence relations interact. Formally, let C and A be sets of concrete and abstract structures and let $\sim_C$ and $\sim_A$ be equivalence relations C and A. We write $C/\sim_C \Longleftrightarrow A/\sim_A$ for the set of q-lenses between C (modulo $\sim_C$) and A (modulo $\sim_A$). A q-lens l in this set has components with the same types as a basic lens but it is only required to obey the lens laws up to $\sim_C$ and $\sim_A$:

$$l.put\ (l.get\ c)\ c \sim_C c \qquad\qquad\qquad \text{(GETPUT)}$$

$$l.get\ (l.put\ a\ c) \sim_A a \qquad\qquad\qquad \text{(PUTGET)}$$

$$l.get\ (l.create\ a) \sim_A a \qquad\qquad\qquad \text{(CREATEGET)}$$

These relaxed laws are just the basic lens laws on the equivalence classes $C/\sim_C$ and $A/\sim_A$ (and when $\sim_C$ and $\sim_A$ are equality they revert to the basic lens laws precisely). However, while we want to reason about the behavior of q-lenses as if they worked on equivalence classes, their component functions actually work on representatives—i.e., members of the underlying sets of concrete and abstract structures: the type of get is $C \longrightarrow A$, not $C/\sim_C \longrightarrow A/\sim_A$. Thus, we need three additional laws stipulating that the functions respect $\sim_C$ and $\sim_A$.

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

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

$$\frac{a \sim_A a'}{l.create\ a \sim_C l.create\ a'} \qquad\qquad\qquad \text{(CREATEEQUIV)}$$

These laws ensure that the components of a q-lens treat equivalent structures equivalently; they play a critical role in (among other things) the proof that the composition operator defined below produces a well-formed q-lens.

3. Basic Combinators

Every basic lens can be lifted to a q-lens, with equality as the equivalence relation on both C and A.

$$\boxed{\frac{l \in C \Longleftrightarrow A}{lift\ l \in C/= \Longleftrightarrow A/=}}$$

The get, put, and $create$ components of $lift\ l$ are identical to those of l. This inference rule can be read as a lemma asserting that the lifted lens is a q-lens at the given type. The proofs of each property is trivial, since l is a basic lens satisfying the strict laws, and since the equivalence relations on both sides is the identity.
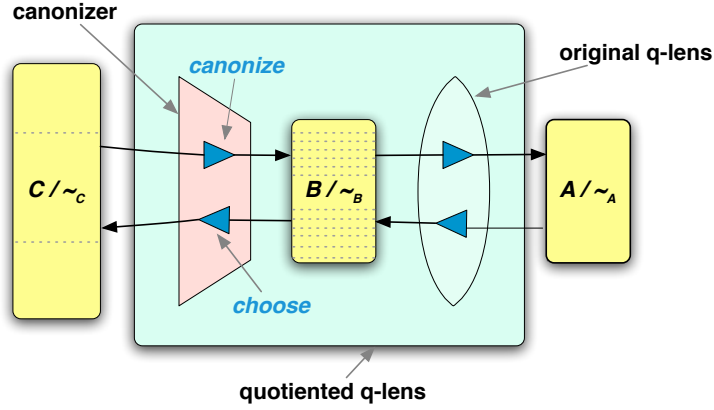
Figure 2. Adding a canonizer to a q-lens (on the left)

3.1 Lemma: $\textit{lift } l \in C/= \Longleftrightarrow A/=.$

Proof:

▶ **GetEquiv**: If $c = c'$ then $(\textit{lift } l).\textit{get } c = (\textit{lift } l).\textit{get } c'$ trivially.

▶ **PutEquiv**: If $a = a'$ and $c = c'$ then $(\textit{lift } l).\textit{put } a\ c = (\textit{lift } l).\textit{get } a'\ c'$ trivially.

▶ **CreatEquiv**: If $a = a'$ then $(\textit{lift } l).\textit{create } a = (\textit{lift } l).\textit{create } a'$ trivially.

▶ **GetPut**: Follows trivially by (the basic lens version of) GETPUT for l.

▶ **PutGet**: Follows trivially by (the basic lens version of) PUTGET for l.

▶ **CreateGet**: Follows trivially by (the basic lens version of) CREATEGET for l.

<div align="right">□</div>

Lifting basic lenses gives us q-lenses with equivalences that are finer than we may want. We need a mechanism for loosening up a q-lens, making it work on a larger domain and/or codomain with coarser equivalences. To this end, we introduce two new operators: *lquot*, which coarsens the domain by adding a canonizer on the left of a q-lens, and *rquot*, which coarsens the codomain by adding a canonizer on the right. Let us consider *lquot* first.

Suppose l is a q-lens from $B/\sim_B$ to $A/\sim_A$, where $\sim_B$ is a relatively fine equivalence (e.g., B could be some set of "canonical strings" with no extraneous whitespace and $\sim_B$ could be equality). We want to construct a new q-lens whose domain is some larger set C (e.g., the same set of strings with more whitespace in various places) with a relatively coarse equivalence $\sim_C$ (relating pairs of strings that differ only in whitespace). To get back and forth between C and B, we need two functions: one (called *canonize*) from C to B, which maps each element to its "canonical representative" (e.g., by throwing away extra whitespace) and another (*choose*) from B to C that maps each canonical representative to some element in its inverse image under *canonize* (for example, the identity function, or perhaps a pretty printer that adds whitespace according to some layout convention). The *canonize* and *choose* functions together are called a canonizer; see Figure 2.

Clearly, a canonizer is a bit like a lens (minus the *put* component); the difference is that we impose a weaker law. Formally, let C and B be sets and $\sim_B$ an equivalence relation on B.[4] A canonizer q from C to $B/\sim_B$ comprises two functions

$$q.\textit{canonize} \ \in \ C \longrightarrow B$$
$$q.\textit{choose} \ \in \ B \longrightarrow C$$

such that, for every $b \in B$:

---

[4] We name the equivalence on B explicitly because, when we put the canonizer together with a q-lens using *lquot*, the equivalences on B need to match. We do not need to mention the equivalence on C because it is going to be calculated later (by the typing rule for *lquot*).

$$q.canonize\ (q.choose\ b) \sim_B b \qquad\qquad\qquad \text{(RECANONIZE)}$$

That is, *canonize* is a left inverse of *choose* modulo $\sim_B$. The set of all canonizers from C to $B/\sim_B$ is written $C \leftrightarrow\!\!\!\rightarrow B/\sim_B$.

Now *lquot* takes as arguments a canonizer q and a q-lens l and yields a new q-lens where l is coarsened on the left using q.

$$
\begin{array}{c}
q \in C \leftrightarrow\!\!\!\rightarrow B/\sim_B \qquad l \in B/\sim_B \Longleftrightarrow A/\sim_A \\
\underline{\quad c \sim_C c' \text{ iff } q.canonize\ c \sim_B q.canonize\ c' \quad} \\
lquot\ q\ l \in C/\sim_C \Longleftrightarrow A/\sim_A
\end{array}
$$

$$
\begin{array}{ll}
get\ c & = l.get\ (q.canonize\ c) \\
put\ a\ c & = q.choose\ (l.put\ a\ (q.canonize\ c)) \\
create\ a & = q.choose\ (l.create\ a)
\end{array}
$$

The concrete argument to the *get* function is first canonized to an element of B using $q.canonize$ and then mapped to an A by $l.get$. Similarly, the abstract argument to the *create* function is first mapped to a B using $l.create$, which is then transformed to a C using $q.choose$. The equivalence $\sim_C$ is the relation induced by $q.canonize$ and $\sim_B$—i.e., two elements of C are equivalent if $q.canonize$ maps them to equivalent elements of B.

3.2 Lemma: $lquot\ q\ l \in C/\sim_C \Longleftrightarrow A/\sim_A$.

Proof:

▶ **GetEquiv**: Let $c, c' \in C$ with $c \sim_C c'$. By the definition of $\sim_C$ we have

$$q.canonize\ c \sim_B q.canonize\ c'$$

We calculate as follows

$$
\begin{array}{ll}
(lquot\ q\ l).get\ c & \\
= l.get\ (q.canonize\ c) & \\
\sim_A l.get\ (q.canonize\ c') & \text{by GETEQUIV for } l \\
= (lquot\ q\ l).get\ c' &
\end{array}
$$

to obtain the required equivalence.

▶ **PutEquiv**: Let $a, a' \in A$ and $c, c' \in C$ with $a \sim_A a'$ and $c \sim_C c'$. We will prove that

$$(lquot\ q\ l).put\ a\ c \sim_C (lquot\ q\ l).put\ a'\ c'$$

by showing that

$$q.canonize\ ((lquot\ q\ l).put\ a\ c) \sim_B q.canonize\ ((lquot\ q\ l).put\ a'\ c')$$

As in the previous case, by the definition of $\sim_C$ we have

$$q.canonize\ c \sim_B q.canonize\ c'$$

Using this fact, we calculate as follows

$$
\begin{array}{ll}
q.canonize\ ((lquot\ q\ l).put\ a\ c) & \\
= q.canonize\ (q.choose\ (l.put\ a\ (q.canonize\ c))) & \\
\sim_B l.put\ a\ (q.canonize\ c) & \text{by RECANONIZE for } q \\
\sim_B l.put\ a'\ (q.canonize\ c') & \text{by PUTEQUIV for } l \\
\sim_B q.canonize\ (q.choose\ (l.put\ a'\ (q.canonize\ c'))) & \text{by RECANONIZE for } q \\
= q.canonize\ ((lquot\ q\ l).put\ a'\ c') &
\end{array}
$$

(Note that in this proof, and throughout the rest of the paper, we silently use elementary facts about equivalence relations, such as the transitivity of $\sim_B$ in the reasoning above.) Using this equivalence, and the definition of $\sim_C$, we then obtain the required equivalence.

▶ **CreatEquiv**: Analogous to the previous case (using CREATEEQUIV for l instead of PUTEQUIV).

▶ **GetPut**: Let $c \in C$. We will prove that

$$(lquot\ q\ l).put\ ((lquot\ q\ l).get\ c)\ c \sim_C c$$

by showing that

$$q.canonize\ ((lquot\ q\ l).put\ ((lquot\ q\ l).get\ c)\ c) \sim_B q.canonize\ c$$

We calculate as follows

$$
\begin{aligned}
&q.canonize\ ((lquot\ q\ l).put\ ((lquot\ q\ l).get\ c)\ c) \\
&= q.canonize\ (q.choose\ (l.put\ (l.get\ (q.canonize\ c))\ (q.canonize\ c))) \\
&\sim_B l.put\ (l.get\ (q.canonize\ c))\ (q.canonize\ c) && \text{by RECANONIZE for } q \\
&\sim_B q.canonize\ c && \text{by GETPUT for } l
\end{aligned}
$$

and obtain the required equivalence.

▶ **PutGet**: Let $a \in A$ and $c \in C$. By RECANONIZE for q we have

$$q.canonize\ (q.choose\ (l.put\ a\ (q.canonize\ c))) \sim_B l.put\ a\ (q.canonize\ c)$$

Using this fact, we calculate as follows:

$$
\begin{aligned}
&(lquot\ q\ l).get\ ((lquot\ q\ l).put\ a\ c) \\
&= l.get\ (q.canonize\ (q.choose\ (l.put\ a\ (q.canonize\ c)))) \\
&\sim_A l.get\ (l.put\ a\ (q.canonize\ c)) && \text{by GETEQUIV for } l \\
&\sim_A a && \text{by PUTGET for } l
\end{aligned}
$$

and obtain the required equivalence.

▶ **CreateGet**: Analogous to the previous case (using CREATEGET for l instead of PUTGET). $\qquad\square$

The *rquot* operator is symmetric; it quotients $l \in C/\sim_C \Longleftrightarrow B/\sim_B$ on the right, using a canonizer from A to $B/\sim_B$. One interesting difference is that its canonizer argument is applied in the opposite direction, compared to *lquot*—i.e., if we think of a canonizer as a weak form of lens, then *lquot* is essentially just lens composition, while *rquot* is a sort of "head to head" composition that would not make sense with lenses.

$$
\begin{array}{c}
\dfrac{l \in C/\sim_C \Longleftrightarrow B/\sim_B \qquad q \in A \leftrightarrowtail B/\sim_B \qquad a \sim_A a' \text{ iff } q.canonize\ a \sim_B q.canonize\ a'}{rquot\ l\ q \in C/\sim_C \Longleftrightarrow A/\sim_A} \\[2em]
\begin{aligned}
get\ c &= q.choose\ (l.get\ c) \\
put\ a\ c &= l.put\ (q.canonize\ a)\ c \\
create\ a &= l.create\ (q.canonize\ a)
\end{aligned}
\end{array}
$$

3.3 Lemma: $rquot\ l\ q \in C/\sim_C \Longleftrightarrow A/\sim_A$.

Proof:

▶ **GetEquiv**: Let $c, c' \in C$ with $c \sim_C c'$. We will prove that

$$(rquot\ l\ q).get\ c \sim_A (rquot\ l\ q).get\ c'$$

by showing that

$$q.canonize\ ((rquot\ l\ q).get\ c) \sim_B q.canonize\ ((rquot\ l\ q).get\ c')$$

We calculate as follows

$$
\begin{aligned}
&q.canonize\ ((rquot\ l\ q).get\ c) \\
&= q.canonize\ (q.choose\ (l.get\ c)) \\
&\sim_B l.get\ c && \text{by RECANONIZE for } q \\
&\sim_B l.get\ c' && \text{by GETEQUIV for } l \\
&\sim_B q.canonize\ (q.choose\ (l.get\ c')) && \text{by RECANONIZE for } q \\
&= q.canonize\ ((rquot\ l\ q).get\ c')
\end{aligned}
$$

and obtain the required equivalence.

▶ **PutEquiv**: Let $a, a \in A$ and $c, c' \in C$ with $a \sim_A a'$ and $c \sim_C c'$. By the definition of $\sim_A$, we have

$$q.canonize\ a \sim_B q.canonize\ a'$$

Using this fact, we calculate as follows

$$
\begin{aligned}
&(rquot\ l\ q).put\ a\ c \\
&= l.put\ (q.canonize\ a)\ c \\
&\sim_C l.put\ (q.canonize\ a')\ c' && \text{by PUTEQUIV for } l \\
&= (rquot\ l\ q).put\ a'\ c'
\end{aligned}
$$

and obtain the required equivalence.

▶ **CreatEquiv**: Analogous to the previous case (using CREATEEQUIV for l instead of PUTEQUIV).

▶ **GetPut**: Let $c \in C$. We calculate as follows

$$
\begin{aligned}
&(rquot\ l\ q).put\ ((rquot\ l\ q).get\ c)\ c \\
&= l.put\ (q.canonize\ (q.choose\ (l.get\ c)))\ c \\
&\sim_C l.put\ (l.get\ c)\ c && \text{by RECANONIZE for } q \text{ and PUTEQUIV for } l \\
&\sim_C c && \text{by GETPUT for } l
\end{aligned}
$$

and obtain the required equivalence.

▶ **PutGet**: Let $a \in A$ and $c \in C$. We will show that

$$(rquot\ l\ q).get\ ((rquot\ l\ q).put\ a\ c) \sim_A a$$

by showing that

$$q.canonize\ ((rquot\ l\ q).get\ ((rquot\ l\ q).put\ a\ c)) \sim_B q.canonize\ a$$

We calculate as follows

$$
\begin{aligned}
&q.canonize\ ((rquot\ l\ q).get\ ((rquot\ l\ q).put\ a\ c)) \\
&= q.canonize\ (q.choose\ (l.get\ (l.put\ (q.canonize\ a)\ c))) \\
&\sim_B l.get\ (l.put\ (q.canonize\ a)\ c) && \text{By RECANONIZE for } q \\
&\sim_B (q.canonize\ a) && \text{By PUTGET for } l
\end{aligned}
$$

and obtain the desired equivalence.

▶ **CreateGet**: Analogous to the previous case (using CREATEGET for l instead of PUTGET). $\qquad\square$

The *lquot* and *rquot* operators allow us to quotient a q-lens repeatedly on either side, which has the effect of composing canonizers. We do this often in q-lens programming—stacking up several canonizers, each of which canonizes a distinct aspect of the concrete or abstract structures. In composing canonizers like this, the following typing rule, which allows the equivalence relation component of a canonizer's type to be coarsened, is often useful:

$$\frac{q \in A \leftrightarrow\!\!\!\shortmid\!\!\!\rightarrow B/{\sim_B} \qquad \sim_B \text{ refines } \sim_{B'}}{q \in A \leftrightarrow\!\!\!\shortmid\!\!\!\rightarrow B/{\sim_{B'}}}$$

3.4 Lemma: $q \in A \leftrightarrow\!\!\!\shortmid\!\!\!\rightarrow B/{\sim_{B'}}$.

Proof: Let $b \in B$. As $q \in A \leftrightarrow\!\!\!\shortmid\!\!\!\rightarrow B/{\sim_B}$ we have that $q.canonize\ (q.choose\ b) \sim_B b$. Since $\sim_B$ refines $\sim_{B'}$ we immediately have that $q.canonize\ (q.choose\ b) \sim_{B'} b$, as required. $\qquad\square$

The next combinator gives us a different kind of composition—of q-lenses themselves.

$$\frac{l \in C/{\sim_C} \Longleftrightarrow B/{\sim_B} \qquad k \in B/{\sim_B} \Longleftrightarrow A/{\sim_A}}{l\,;k \in C/{\sim_C} \Longleftrightarrow A/{\sim_A}}$$

$$
\begin{aligned}
get\ c \quad &= k.get\ (l.get\ c) \\
put\ a\ c \quad &= l.put\ (k.put\ a\ (l.get\ c))\ c \\
create\ a &= l.create\ (k.create\ a)
\end{aligned}
$$

The typing rule demands that the intermediate type B have the same equivalence relation $\sim_B$ on both sides. To see what goes wrong if this condition is dropped, consider

$$
\begin{aligned}
l_1 \quad &= \quad id \quad \in \quad \{a\}/{=} \Longleftrightarrow \{a\}/{=} \\
l_2 \quad &= \quad id \quad \in \quad \{a,b\}/{=} \Longleftrightarrow \{a,b\}/{=}
\end{aligned}
$$

where *id* is the identity lens (whose *get* and *create* components are identity functions and whose *put* component is the identity on its abstract argument) and $q \in \{a,b\} \leftrightarrow\!\!\!\shortmid\!\!\!\rightarrow \{a\}/{=}$ defined by

$$
\begin{aligned}
q.canonize\ x \quad &= \quad a \\
q.choose\ a \quad &= \quad a.
\end{aligned}
$$

If we now take $l = (rquot\ l_1\ q)\,;l_2$ (where the equivalence on the left is the total relation on $\{a,b\}$, which is strictly coarser than equality, the relation on the right), then the CREATEGET law fails:

$$
\begin{aligned}
&\quad l.get\ (l.create\ b) \\
&= \quad l.get\ a \\
&= \quad l_2.get\ (q.choose\ (l_1.get\ a)) \\
&= \quad a \\
&\neq \quad b
\end{aligned}
$$

Conversely, if we take $l = l_2\,;(lquot\ q\ l_1)$, (where the left equivalence is equality and the right equivalence is the total relation on $\{a,b\}$), then the GETPUT law fails, since $a = l.get\ b$ but

$$
\begin{aligned}
&\quad l.put\ a\ b \\
&= \quad l_2.put\ ((lquot\ q\ l_1).put\ a\ (l_2.get\ b))\ b \\
&= \quad l_2.put\ (q.choose\ (l_1.put\ a\ (q.canonize\ (l_2.get\ b))))\ b \\
&= \quad l_2.put\ a\ b \\
&= \quad a \\
&\neq \quad b.
\end{aligned}
$$

This requirement raises an interesting implementation issue: to statically type the composition operator, we must be able to check whether two equivalence relations are identical; see Section 7.

3.5 Lemma: $l \mathbin{;} k \in C/\sim_C \Longleftrightarrow A/\sim_A$.

Proof:

▶ **GetEquiv**: Let $c, c' \in C$ with $c \sim_C c'$. We have the following equivalences

$$
\begin{aligned}
&(l \mathbin{;} k).get\ c \\
&= k.get\ (l.get\ c) \\
&\sim_A k.get\ (l.get\ c') \quad \text{by GetEquiv for } l \text{ and } k \\
&= (l \mathbin{;} k).get\ c'
\end{aligned}
$$

as required.

▶ **PutEquiv**: Let $a, a \in A$ and $c, c' \in C$ with $a \sim_A a'$ and $c \sim_C c'$. We have the following equivalences

$$
\begin{aligned}
&(l \mathbin{;} k).put\ a\ c \\
&= l.put\ (k.put\ a\ (l.get\ c))\ c \\
&\sim_C l.put\ (k.put\ a'\ (l.get\ c'))\ c' \quad \text{by GetEquiv for } l \text{ and PutEquiv for } l \text{ and } k \\
&= (l \mathbin{;} k).put\ a'\ c'
\end{aligned}
$$

as required.

▶ **CreatEquiv**: Analogous to the previous case (using CreateEquiv for l and k).

▶ **GetPut**: Let $c \in C$. We calculate as follows

$$
\begin{aligned}
&(l \mathbin{;} k).put\ ((l \mathbin{;} k).get\ c)\ c \\
&= l.put\ (k.put\ (k.get\ (l.get\ c))\ (l.get\ c))\ c \\
&\sim_C l.put\ (l.get\ c)\ c \qquad\qquad\qquad\qquad \text{by GetPut for } k \text{ and PutEquiv for } l \\
&\sim_C c \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{by GetPut for } l
\end{aligned}
$$

and obtain the required equivalence.

▶ **PutGet**: Let $a \in A$ and $c \in C$. We calculate as follows

$$
\begin{aligned}
&(l \mathbin{;} k).get\ ((l \mathbin{;} k).put\ a\ c) \\
&= k.get\ (l.get\ (l.put\ (k.put\ a\ (l.get\ c))\ c)) \\
&\sim_A k.get\ (k.put\ a\ (l.get\ c)) \qquad\qquad \text{by PutGet for } l \text{ and GetEquiv for } k \\
&\sim_A a \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{by PutGet for } k
\end{aligned}
$$

and obtain the required equivalence.

▶ **CreateGet**: Analogous to the previous case, using CreateGet for l and k. □

So far, we have seen how to lift basic lenses to q-lenses, how to coarsen the equivalence relations in their types using canonizers, and how to compose them. We have not, however, discussed where canonizers themselves come from. Of course, we can always define canonizers as primitives—this is essentially the approach used in previous "canonizers at the edges"-style proposals, where the set of parsers and pretty printers is fixed. But we can do better: we can build a canonizer out of the *get* and *create* components of an arbitrary lens—indeed, of an arbitrary q-lens!

$$
\frac{l \in C/\sim_C \Longleftrightarrow B/\sim_B}{canonizer\ l \in C \leftrightarrow\!\!\!\!\!\triangleright\!\!\!\!\triangleright B/\sim_B}
$$

$$
\begin{aligned}
canonize\ c &= l.get\ c \\
choose\ b &= \quad l.create\ b
\end{aligned}
$$

3.6 Lemma: *canonizer* $l \in C \leftrightarrow\!\!\triangleright\!\!\rightarrow B/\sim_B$.

Proof: Let $b \in B$. We have

$$
\begin{aligned}
&(canonizer\ l).canonize\ ((canonizer\ l).choose\ b) \\
&= l.get\ (l.create\ b) \\
&\sim_B b \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{by CREATEGET for } l
\end{aligned}
$$

as required. □

Building canonizers from lenses gives us a pleasantly parsimonious design, allowing us to define canonizers using whatever generic or domain-specific primitives are already available on lenses (e.g., in our implementation, primitives for copying, deletion, etc., as well as the rational operators—concatenation, iteration, union, etc.—described in Section 4). A composition operator on canonizers can be derived from the quotienting operators (on the identity lens, *copy*, defined in Section 5). We state a simple version here, whose type can be derived straightforwardly from the types of *lquot* and *copy*.

$$
\boxed{
\begin{aligned}
(q_1; q_2) &\triangleq lquot\ q_1\ (quot\ q_2\ (copy\ A)) \\
&\forall q_1 \in C \leftrightarrow\!\!\triangleright\!\!\rightarrow B/=. \\
&\forall q_2 \in B \leftrightarrow\!\!\triangleright\!\!\rightarrow A/=. \\
&(q_1; q_2) \in C \leftrightarrow\!\!\triangleright\!\!\rightarrow B/\sim_B
\end{aligned}
}
$$

In general, the equivalence on B need not be the identity, but must refine the equivalence induced by q_2.

Of course, it is also useful to design primitive canonizers *de novo*. The canonizer law imposes fewer restrictions than the lens laws, giving us enormous latitude for writing specific canonizing transformations that would not be legal as lenses. Several useful canonizer primitives are discussed in Section 5.

4. Rational Operators

Having presented the semantic space of q-lenses and several generic combinators, we now turn our attention to q-lenses for the specific domain of strings. The next several q-lenses are direct generalizations of corresponding basic string lens operators (Bohannon et al. 2008).

First, a little notation. 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 by $L_1{\cdot}L_2 = \{u{\cdot}v \mid u \in L_1 \text{ and } v \in L_2\}$. The iteration of L is written L^*; i.e., $L^* = \bigcup_{n=0}^{\infty} L^n$, where L^n is the n-fold concatenation of L.

Some of the definitions below require that, for every string belonging to the concatenation of two given languages, there must be a unique way of splitting that string into two substrings belonging to the concatenated languages. We say that two languages L_1 and L_2 are unambiguously concatenable, written $L_1{\cdot}^{!}L_2$, when, 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 $L^{!*}$, when, for every $u_1, \ldots, u_m \in L$ and $v_1, \ldots, v_n \in L$, if $u_1{\cdots}u_m = v_1{\cdots}v_n$ then $m = n$ and $u_i = v_i$ for every i. It is decidable whether two regular languages L_1 and L_2 are unambiguously concatenable and whether a single language L is unambiguously iterable; see Bohannon et al. (2008).

Several of the primitives are parameterized on regular expressions

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

where u ranges over arbitrary strings (including ϵ). The notation $[\![E]\!]$ denotes the language described by $E \in \mathcal{R}$. The function $rep(E)$ picks an arbitrary representative of $[\![E]\!]$.

Our first string q-lens combinators are based on the rational operators union, concatenation, and iteration (Kleene star). The functional components of these combinators are identical to the basic lens versions defined in Bohannon et al. (2008), but the typing rules are different, since they define equivalence relations on the concrete and abstract domains.

We start by defining the concatenation of relations on sets of strings.

4.1 Definition [Relation Concatenation]: Let L_1 and L_2 be languages and let R_1 and R_2 be binary relations on L_1 and L_2. The relation $R_1 \cdot R_2$ is defined as $w \ (R_1 \cdot R_2) \ w'$ iff there exist $w_1, w_1' \in L_1$ and $w_2, w_2' \in L_2$ with $w = w_1 \cdot w_2$ and $w' = w_1' \cdot w_2'$ such that $w_1 \ R_1 \ w_1'$ and $w_2 \ R_2 \ w_2'$.

The concatenation of two equivalence relations $\sim_1$ and $\sim_2$ is not always an equivalence relation. (In particular, it may not be transitive.) However, it is guaranteed to be an equivalence in two important cases: when the concatenation of L_1 and L_2 is unambiguous, and when $\sim_1$ and $\sim_2$ is the identity relation.

The concatenation operator for q-lenses works in the obvious way. Note that the equivalence relations on both sides of the type are guaranteed to be an equivalence since the concatenations of both pairs of languages are unambiguous.

$$
\begin{array}{c}
C_1 \cdot^! C_2 \qquad A_1 \cdot^! A_2 \\
l_1 \ \in \ C_1/\sim_{C_1} \Longleftrightarrow A_1/\sim_{A_1} \\
l_2 \ \in \ C_2/\sim_{C_2} \Longleftrightarrow A_2/\sim_{A_2} \\
\underline{\sim_C = \sim_{C_1}\cdot\sim_{C_2} \qquad \sim_A = \sim_{A_1}\cdot\sim_{A_2}} \\
l_1 \cdot l_2 \ \in \ C_1 \cdot C_2/\sim_C \Longleftrightarrow A_1 \cdot A_2/\sim_A
\end{array}
$$

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

4.2 Lemma: $l_1 \cdot l_2 \in (C_1 \cdot C_2)/\sim_C \Longleftrightarrow (A_1 \cdot A_2)/\sim_A$.

Proof:

▶ **GetEquiv**: Let $c, c' \in C_1 \cdot C_2$ with $c \sim_C c'$. Then there exist unique $c_1, c_1' \in C_1$ and $c_2, c_2' \in C_2$ such that $c = c_1 \cdot c_2$ and $c' = c_1' \cdot c_2'$ and $c_1 \sim_{C_1} c_1'$ and $c_2 \sim_{C_2} c_2'$.

The equivalence

$$
\begin{aligned}
& (l_1 \cdot l_2).\textit{get } c \\
=\ & (l_1.\textit{get } c_1) \cdot (l_2.\textit{get } c_2) \\
\sim_A\ & (l_1.\textit{get } c_1') \cdot (l_2.\textit{get } c_2') \\
=\ & (l_1 \cdot l_2).\textit{get } c'
\end{aligned}
$$

follows from GETEQUIV for l_1 and l_2 and the definition of $\sim_A$.

▶ **PutEquiv**: Let $c, c' \in C_1 \cdot C_2$ and $a, a' \in A_1 \cdot A_2$. Then there exist unique $c_1, c_1' \in C_1$ and $c_2, c_2' \in C_2$ such that $c = c_1 \cdot c_2$ and $c' = c_1' \cdot c_2'$ and $c_1 \sim_{C_1} c_1'$ and $c_2 \sim_{C_2} c_2'$. There also exist unique $a_1, a_1' \in A_1$ and $a_2, a_2' \in A_2$ such that $a = a_1 \cdot a_2$ and $a' = a_1' \cdot a_2'$ and $a_1 \sim_{A_1} a_1'$ and $a_2 \sim_{A_2} a_2'$.

The equivalence

$$
\begin{aligned}
& (l_1 \cdot l_2).\textit{put } a \ c \\
=\ & (l_1.\textit{put } a_1 \ c_1) \cdot (l_2.\textit{put } a_2 \ c_2) \\
\sim_C\ & (l_1.\textit{put } a_1' \ c_1') \cdot (l_2.\textit{put } a_2' \ c_2') \\
=\ & (l_1 \cdot l_2).\textit{put } a' \ c'
\end{aligned}
$$

follows from PUTEQUIV for l_1 and l_2 and the definition of $\sim_C$.

▶ **CreatEquiv**: Analogous to the previous cases (using CREATEEQUIV for l_1 and l_2).

▶ **GetPut**: Let $c \in C_1 \cdot C_2$ and let $a \in A_1 \cdot A_2$ with $a = (l_1 \cdot l_2).\textit{get } c$. As C_1 and C_2 are unambiguously concatenable, there exist unique $c_1 \in C_1$ and $c_2 \in C_2$ such that $c = c_1 \cdot c_2$. Similarly, there exist unique $a_1 \in A_1$ and $a_2 \in A_2$ such that $a = a_1 \cdot a_2$. With the definition of $(l_1 \cdot l_2).\textit{get}$ we also have that $a_1 = \_1.\textit{get } c_1$ and $a_2 = \_2.\textit{get } c_2$.

Using these facts, we calculate as follows

$$
\begin{aligned}
&(l_1 \cdot l_2).put\ ((l_1 \cdot l_2).get\ c)\ c \\
={}&(l_1 \cdot l_2).put\ (a_1 \cdot a_2)\ (c_1 \cdot c_2) \\
={}&(l_1 \cdot l_2).put\ a\ c \\
={}&(l_1.put\ a_1\ c_1) \cdot (l_2.put\ a_2\ c_2) \\
={}&(l_1.put\ (l_1.get\ c_1)\ c_1) \cdot (l_2.put\ (l_2.get\ c_2)\ c_2) \\
\sim_C{}& c_1 \cdot c_2 \qquad\qquad\qquad\qquad\qquad\qquad\text{By GETPUT for } l_1 \text{ and } l_2 \text{ and definition of } \sim_C \\
={}& c
\end{aligned}
$$

and obtain the required equivalence.

▶ **PutGet**: Let $a \in A_1 \cdot A_2$ and $c \in C_1 \cdot C_2$. As A_1 and A_2 are unambiguously concatenable, there exist unique $a_1 \in A_1$ and $a_2 \in A_2$ such that $a = a_1 \cdot a_2$. Similarly, there exist unique $c_1 \in C_1$ and $c_2 \in C_2$ such that $c = c_1 \cdot c_2$. Using these facts, we calculate as follows

$$
\begin{aligned}
&(l_1 \cdot l_2).get\ ((l_1 \cdot l_2).put\ a\ c) \\
={}&(l_1 \cdot l_2).get\ ((l_1 \cdot l_2).put\ (a_1 \cdot a_2)\ (c_1 \cdot c_2)) \\
={}&(l_1 \cdot l_2).get\ ((l_1.put\ a_1\ c_1) \cdot (l_2.put\ a_2\ c_2)) \\
={}&(l_1.get\ (l_1.put\ a_1\ c_2)) \cdot (l_1.get\ (l_1.put\ a_2\ c_2)) \quad \text{as } ran((l_1 \cdot l_2).put) = C_1 \cdot C_2 \text{ and } C_1 \cdot^! C_2 \\
\sim_A{}& a_1 \cdot a_2 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \text{By PUTGET for } l_1 \text{ and } l_2 \text{ and definition of } \sim_A \\
={}& a
\end{aligned}
$$

and obtain the required equivalence.

▶ **CreateGet**: Analogous to the previous case (using CREATEGET for l_1 and l_2). □

Concatenation raises an interesting side point. Suppose that we have two canonizers, q_1 and q_2, and two q-lenses, l_1 and l_2, that we want to—in some order—concatenate and quotient on the left. There are two ways we could do this: we could quotient l_1 and l_2 first using q_1 and q_2, and combine the results by concatenating the q-lenses just defined, or we could concatenate the q-lenses l_1 and l_2 and the canonizers q_1 and q_2 and then quotient the results. Both are possible in our system and (when both are well-typed[5]) yield equivalent q-lenses. At the end of this section, we define the rational operators on canonizers, and prove this equivalence formally.

For Kleene star, we start by lifting iteration to relations.

4.3 Definition [Relation Iteration]: Let L_1 be a regular language, and let R_1 be a binary relation on R_1. The relation R_1^* is defined as $w\ R_1^*\ w'$ iff there exist strings $w_1, ..., w_n$ and $w'_1, ..., w'_n$ such for all i in $\{1...n\}$ we have $w_i\ R_1\ w'_i$.

Using this definition, the generalization of Kleene star to q-lenses is straightforward.

$$
\frac{l \ \in \ C/\sim_C \Longleftrightarrow A/\sim_A \qquad C^{!*} \qquad A^{!*}}{l^* \ \in \ C^*/\sim_C^* \Longleftrightarrow A^*/\sim_A^*}
$$

$$
\begin{aligned}
get\ (c_1 \cdots c_n) \quad &= (l.get\ c_1) \cdots (l.get\ c_n) \\
put\ (a_1 \cdots a_n)\ (c_1 \cdots c_m) &= c'_1 \cdots c'_n \\
\text{where } c'_i &= \begin{cases} l.put\ a_i\ c_i & i \in \{1, ..., \min(m, n)\} \\ l.create\ a_i & i \in \{m+1, ..., n\} \end{cases} \\
create\ (a_1 \cdots a_n) \quad &= (l.create\ a_1) \cdots (l.create\ a_n)
\end{aligned}
$$

4.4 Lemma: $l^* \in C^*/(\sim_C)^* \Longleftrightarrow A^*/(\sim_A)^*$.

---

[5] Quotienting the lenses first is a little more flexible, since the concatenation of the original q-lenses need not be typeable.

Proof:

▶ **GetEquiv**: Let $c, c' \in C^*$ with $c \sim_C^* c'$. By the definition of $\sim_C^*$ there exist $c_1, ..., c_n \in C$ with $c = c_1 \cdots c_n$ and $c'_1, ..., c'_n$ with $c' = c'_1 \cdots c'_n$ and $c_i \sim_C c'_i$ for $i \in \{1, ..., n\}$.

The equivalence

$$
\begin{aligned}
& l^*.get\ c \\
={}& (l.get\ c_1) \cdots (l.get\ c_n) \\
\sim_A^*{}& (l.get\ c'_1) \cdots (l.get\ c'_n) \\
={}& l^*.get\ c'
\end{aligned}
$$

follows using GETEQUIV for l and the definition of $\sim_A^*$.

▶ **PutEquiv**: Let $a, a' \in A^*$ with $a \sim_A^* a'$ and $c, c' \in C^*$ with $c \sim_C^* c'$. By the definition of $\sim_A^*$ there exist $a_1, ..., a_m \in A$ and $a'_1, ..., a'_m \in A$ with $a = a_1 \cdots a_m$ and $a' = a'_1 \cdots a'_m$ and $a_i \sim_A a'_i$ for $i \in \{1, ..., m\}$. Similarly, by the definition of $\sim_C$ there exist $c_1, ..., c_n \in C$ and $c'_1, ..., c'_n \in C$ with $c = c_1 \cdots c_n$ and $c' = c'_1 \cdots c'_n$ with $c_i \sim_C c'_i$ for $i \in \{1, ..., n\}$.

The equivalence

$$
\begin{aligned}
& l^*.put\ a\ c \\
={}& c''_1 \cdots c''_m \\
& \quad \text{where } c''_i = l.put\ a_i\ c_i \text{ for } i \in \{1, ..., \max(m, n)\} \\
& \quad \text{and } c''_i = l.create\ a_i \text{ for } i \in \{\max(m, n) + 1, ..., m\} \\
\sim_C^*{}& c'''_1 \cdots c'''_m \\
& \quad \text{where } c'''_i = l.put\ a'_i\ c'_i \text{ for } i \in \{1, ..., \max(m, n)\} \\
& \quad \text{and } c'''_i = l.create\ a'_i \text{ for } i \in \{\max(m, n) + 1, ..., m\} \\
={}& l^*.put\ a'\ c'
\end{aligned}
$$

follows using PUTEQUIV and CREATEEQUIV for l and the definition of $\sim_C$.

▶ **CreatEquiv**: Analogous to the previous cases (using CREATEEQUIV for l).

▶ **GetPut**: Let $c \in C^*$. By $C^{!*}$ there exist unique $c_1, ..., c_n \in C$ such that $c = c_1 \cdots c_n$. By the definition of $l^*.get$ we have $l^*.get\ c = (l.get\ c_1) \cdots (l.get\ c_n)$.

Using these facts, we calculate as follows

$$
\begin{aligned}
& l^*.put\ (l^*.get\ c)\ c \\
={}& (l.put\ (l.get\ c_1)\ c_1) \cdots (l.put\ (l.get\ c_n)\ c_n) && \text{as } A^{!*} \\
\sim_C^*{}& c_1 \cdots c_n && \text{By GETPUT for } l \text{ and definition of } \sim_C \\
={}& c
\end{aligned}
$$

and obtain the required equivalence.

▶ **PutGet**: Let $a \in A^*$ and $c \in C^*$. By $A^{!*}$, there exist unique $a_1, ..., a_m \in A$ such that $a = a_1 \cdot a_m$. Likewise, by $C^{!*}$ there exist unique $c_1, ..., c_n \in C$ such that $c = c_1 \cdots c_n$.

We calculate as follows

$$
\begin{aligned}
& l^*.get\ (l^*.put\ a\ c) \\
={}& l^*.get\ (c'_1 \cdots c'_m) && \text{where } c'_i = l.put\ a_i\ c_i \text{ for } i \in \{1, ..., \max(m, n)\} \\
& && \text{and } c'_i = l.create\ a_i \text{ for } i \in \{\max(m, n) + 1, ..., m\} \\
={}& (l_1.get\ (c'_1)) \cdots (l_1.get\ (c'_m)) && \text{as } ran(l.put) = ran(l.create) = C \\
& && \text{and } C^{!*} \text{ and definition } l^*.get \\
\sim_A^*{}& a_1 \cdots a_2 && \text{by PUTGET and PUTCREATE for } l \text{ and definition of } \sim_A \\
={}& a
\end{aligned}
$$

and obtain the required equivalence.

▶ **CreateGet**: Analogous to the previous case (using PUTCREATE for l.) $\qquad\square$

The q-lens version of union is more interesting.

$$
\begin{array}{c}
l_1 \in C_1/\sim_{C_1} \iff A_1/\sim_{A_1} \\
l_2 \in C_2/\sim_{C_2} \iff A_2/\sim_{A_2} \\
C_1 \cap C_2 = \emptyset \\
a \sim_A a' \wedge a \in A_1 \cap A_2 \text{ implies } a\sim_{A_1} a' \wedge a\sim_{A_2} a' \\
\sim_C = \sim_{C_1} \cup \sim_{C_2} \qquad \sim_A = \sim_{A_1} \cup \sim_{A_2} \\
\hline
l_1 \mid l_2 \in C_1 \cup C_2/\sim_C \iff A_1 \cup A_2/\sim_A
\end{array}
$$

$$
\textit{get } c \quad = \begin{cases} l_1.\textit{get } c & \text{if } c \in C_1 \\ l_2.\textit{get } c & \text{if } c \in C_2 \end{cases}
$$

$$
\textit{put } a\ c \quad = \begin{cases} l_1.\textit{put } a\ c & \text{if } c \in C_1 \wedge a \in A_1 \\ l_2.\textit{put } a\ c & \text{if } c \in C_2 \wedge a \in A_2 \\ l_1.\textit{create } a & \text{if } c \in C_2 \wedge a \in A_1 \setminus A_2 \\ l_2.\textit{create } a & \text{if } c \in C_1 \wedge a \in A_2 \setminus A_1 \end{cases}
$$

$$
\textit{create } a = \begin{cases} l_1.\textit{create } a & \text{if } a \in A_1 \\ l_2.\textit{create } a & \text{if } a \in A_2 \setminus A_1 \end{cases}
$$

4.5 Lemma: $l_1 \mid l_2 \in C_1 \cup C_2/\sim_C \iff A_1 \cup A_2/\sim_A$.

Proof:

▶ **GetEquiv**: Let $c, c' \in C_1 \cup C_2$ with $c \sim_C c'$. By the definition of $\sim_C$, and since $C_1 \cap C_2 = \emptyset$, we have that $c, c' \in C_1$ with $c \sim_{C_1} c'$ or $c, c' \in C_2$ with $c \sim_{C_2} c'$. We analyze each case separately.

Case $c, c' \in C_1$**:** We calculate as follows

$$
\begin{aligned}
& (l_1 \mid l_2).\textit{get } c \\
={}& l_1.\textit{get } c \\
\sim_A{}& l_1.\textit{get } c' && \text{by GETEQUIV for } l_1 \\
={}& (l_1 \mid l_2).\textit{get } c'
\end{aligned}
$$

Case $c, c' \in C_2$**:** Symmetric to the previous case, using l_1 instead of l_1.

▶ **PutEquiv**: Let $a, a' \in A_1 \cup A_2$ with $a \sim_A a'$ and $c, c' \in C_1 \cup C_2$ with $c \sim_C c'$.

By the definition of $\sim_A$ and the conditions on $\sim_{A_1}$ and $\sim_{A_2}$ on their intersection, $A_1 \cap A_2$, we have that $a, a' \in A_1$ with $a \sim_{A_1} a'$ or $a, a' \in A_2$ with $a \sim_{A_2} a'$. Also, by the definition of $\sim_C$ and since $C_1 \cap C_2 = \emptyset$, we have that $c, c' \in C_1$ with $c \sim_{C_1} c'$ or $c, c' \in C_2$ with $c \sim_{C_2} c'$. We analyze each case separately.

Case $a, a' \in A_1$ **and** $c, c' \in C_1$**:** We calculate as follows

$$
\begin{aligned}
& (l_1 \mid l_2).\textit{put } a\ c \\
={}& l_1.\textit{put } a\ c \\
\sim_C{}& l_1.\textit{put } a'\ c' && \text{by PUTEQUIV for } l_1 \\
={}& (l_1 \mid l_2).\textit{put } a'\ c'
\end{aligned}
$$

and obtain the required equivalence.

Case $a, a' \in A_2$ **and** $c, c' \in C_2$**:** Symmetric to the previous case, using l_2 instead of l_1.

Case $a, a' \in A_1 \setminus A_2$ **and** $c, c' \in C_2$**:** We calculate as follows

$$
\begin{aligned}
& (l_1 \mid l_2).put\ a\ c \\
&= l_1.create\ a \\
&\sim_C l_1.create\ a' \qquad\qquad \text{by CREATEEQUIV for } l_1 \\
&= (l_1 \mid l_2).put\ a'\ c'
\end{aligned}
$$

and obtain the required equivalence.

Case $a, a' \in A_2 \setminus A_1$ **and** $c, c' \in C_1$**:** Symmetric to the previous case, using l_2 instead of l_1.

▶ **CreatEquiv**: Analogous to the previous case.

▶ **GetPut**: Let $c \in C$. We analyze several cases.

Case $c \in C_1$**:** By the definition of $(l_1 \mid l_2).get$, we have that $(l_1 \mid l_2).get\ c = l_1.get\ c \in A_1$. Using this fact, we calculate as follows

$$
\begin{aligned}
& (l_1 \mid l_2).put\ ((l_1 \mid l_2).get\ c)\ c \\
&= l_1.put\ (l_1.get\ c)\ c \\
&\sim_C c \qquad\qquad\qquad\qquad\qquad\qquad \text{by GETPUT for } l_1
\end{aligned}
$$

and obtain the required equivalence.

Case $c \in C_2$ **and** $(l_1 \mid l_2).get\ c \in A_2$**:** Symmetric to the previous case, using l_2 instead of l_1.

▶ **PutGet**: Let $a, \in A_1 \cup A_2$ and $c \in C_1 \cup C_2$. We analyze several cases.

Case $a \in A_1$ **and** $c \in C_1$**:** We calculate as follows

$$
\begin{aligned}
& (l_1 \mid l_2).get\ ((l_1 \mid l_2).put\ a\ c) \\
&= l_1.get\ (l_1.put\ a\ c) \qquad\qquad \text{as } ran(l_1.put) = C_1 \text{ and } C_1 \cap C_2 = \emptyset \\
&\sim_A a \qquad\qquad\qquad\qquad\qquad\quad\ \text{by PUTGET for } l_1
\end{aligned}
$$

and obtain the required equivalence.

Case $a \in A_2$ **and** $c \in C_2$**:** Symmetric to the previous case, using l_2 instead of l_1.

Case $a \in A_1$ **and** $c \in C_2 \setminus C_1$**:** We calculate as follows

$$
\begin{aligned}
& (l_1 \mid l_2).get\ ((l_1 \mid l_2).put\ a\ c) \\
&= l_1.get\ (l_1.create\ a) \qquad\qquad \text{as } ran(l_1.create) = C_1 \text{ and } C_1 \cap C_2 = \emptyset \\
&\sim_A a \qquad\qquad\qquad\qquad\qquad\qquad \text{by CREATEGET for } l_1
\end{aligned}
$$

and obtain the required equivalence.

Case $a \in A_2$ **and** $c \in C_1 \setminus C_2$**:** Symmetric to the previous case, using l_2 instead of l_1.

▶ **CreateGet**: Analogous to the previous case. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

The relations $\sim_C$ and $\sim_A$ are formed by taking the the union of the corresponding relations from l_1 and l_2; the side conditions in the typing rule ensure that these are equivalence relations. The side condition on $\sim_A$ is also essential for ensuring the q-lens laws. It stipulates that $\sim_{A_1}$ and $\sim_{A_2}$ may only relate elements of the intersection $A_1 \cap A_2$ to other elements in $A_1 \cap A_2$ and that $\sim_{A_1}$ and $\sim_{A_2}$ must agree in the intersection. To see why this is needed, suppose we have a in $A_1 \cap A_2$ and $a' \in A_2 \setminus A_1$ with $a \sim_A a'$, and let $c \in C_1$ with $get\ c = a$. Then $put\ a'\ c = l_2.create\ a'\ c$. Since $dom(l_1.put) \cap dom(l_2.create) = \emptyset$, the result cannot be related to c by $\sim_C$—i.e., GETPUT fails.

The final q-lens combinator in this section, *permute*, is like concatenation, but reorders of the abstract string it constructs according to a fixed permutation σ. As an example, let σ be the permutation that maps 1 to 2, 2 to 3, and 3 to 1. The *get* component of *permute* σ (*copy a*) (*copy b*) (*copy c*) maps *abc* to *cab*.

$$
\begin{array}{c}
\sigma \in \mathsf{Perms}(\{1,...,n\}) \qquad i_1 \triangleq \sigma^{-1}(1) \ ... \ i_n \triangleq \sigma^{-1}(n) \\
C_1 \cdot^! C_2 \ ... \ C_{n-1} \cdot^! C_n \qquad A_{i_1} \cdot^! A_{i_2} \ ... \ A_{i_{n-1}} \cdot^! A_{i_n} \\
\forall i \in \{1,...,n\}. \, l_i \ \in \ C_i/\sim_{C_i} \iff A_i/\sim_{A_i} \\
\sim_C \triangleq \sim_{C_1} \cdot ... \cdot \sim_{C_n} \qquad C \triangleq C_1 \cdot ... \cdot C_n \\
\sim_A \triangleq \sim_{A_{i_1}} \cdot ... \cdot \sim_{A_{i_n}} \qquad A \triangleq A_{i_1} \cdot ... \cdot A_{i_n} \\
\hline
permute \ \sigma \ l_1 \ ... \ l_n \ \in \ C/\sim_C \iff A/\sim_A
\end{array}
$$

$$
\begin{aligned}
get \ (c_1 \cdot ... \cdot c_n) \qquad\qquad &= (l_{i_1}.get \ c_{i_n}) \cdot ... \cdot (l_{i_1}.get \ c_{i_n}) \\
put \ (a_{i_1} \cdot ... \cdot a_{i_n}) \ (c_1 \cdot ... \cdot c_n) &= (l_1.put \ a_1 \ c_1) \cdot ... \cdot (l_n.put \ a_n \ c_n) \\
create \ (a_{i_1} \cdot ... \cdot a_{i_n}) \qquad\quad\ &= (l_1.create \ a_1) \cdot ... \cdot (l_n.create \ a_n)
\end{aligned}
$$

The *permute* lens is used in many of our examples, including the lenses for genomic data described in Section 8 and in the translation from XSugar programs to q-lenses in Section 9.

Now we lift each of the rational operators to canonizers. Since canonizers only have to satisfy the weaker RECANONIZE law, we have some additional flexibility compared to basic lenses. For example, the concatenation operator on q-lenses requires that the concatenations of the languages on the left and on the right each be unambiguous; with canonizers, we only need the concatenation on the left be unambiguous:

$$
\begin{array}{c}
q_1 \in C_1 \leftrightarrow B_1/\sim_{B_1} \qquad q_2 \in C_2 \leftrightarrow B_2/\sim_{B_2} \\
C_1 \cdot^! C_2 \qquad \sim_B = \mathsf{TransClosure}(\sim_{B_1} \cdot \sim_{B_2}) \\
split \in (B_1 \cdot B_2) \to (B_1 \times B_2) \\
\hline
q_1 \cdot q_2 \in C_1 \cdot C_2 \leftrightarrow B_1 \cdot B_2/\sim_B
\end{array}
$$

$$
\begin{aligned}
canonize \ c_1 \cdot c_2 &= (q_1.canonize \ c_1) \cdot (q_2.canonize \ c_2) \\
choose \ b \qquad &= (q_1.choose \ b_1) \cdot (q_2.choose \ b_2) \\
&\text{where } spilt \ b = (b_1, b_2)
\end{aligned}
$$

The *split* function determines how strings in $(B_1 \cdot B_2)$, which may be ambiguous, should be split. (In Boomerang, we have two different operators for concatenating canonizers: one instantiates *split* with a function that uses a longest-match policy, and another that uses a shortest-match policy.) Note that we take the transitive closure of $(\sim_{B_1} \cdot \sim_{B_2})$ (to ensure that it is an equivalence).

4.6 Lemma: $q_1 \cdot q_2 \in C_1 \cdot C_2 \leftrightarrow B_1 \cdot B_2/\sim_B$.

Proof: Let $b \in B_1 \cdot B_2$ with *split* $b = (b_1, b_2)$. We calculate as follows

$$
\begin{aligned}
&(q_1 \cdot q_2).canonize \ ((q_1 \cdot q_2).choose \ b) \\
={}& (q_1 \cdot q_2).canonize \ ((q_1.choose \ b_1) \cdot (q_2.choose \ b_2)) \\
={}& (q_1.canonize \ (q_1.choose \ b_1)) \cdot (q_2.canonize \ (q_2.choose \ b_2)) \quad \text{as } C_1 \cdot^! C_2 \\
\sim_B{}& (b_1 \cdot b_2) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{by RECANONIZE for } q_1 \text{ and } q_2 \\
={}& b
\end{aligned}
$$

and obtain the required equivalence. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

Having defined the concatenation of canonizers, we now prove a result described earlier: that canonizing and quotienting on the left in either order yields equivalent lenses

4.7 Lemma: Let

$$
\begin{aligned}
l_1 &\in B_1/\sim_{B_1} \iff A_1/\sim_{A_1} & q_1 &\in C_1 \leftrightarrow B_1/\sim_{B_1} \\
l_2 &\in B_2/\sim_{B_2} \iff A_2/\sim_{A_2} & q_2 &\in C_2 \leftrightarrow B_2/\sim_{B_2}
\end{aligned}
$$

be q-lenses and canonizers. Define q-lenses

$$l \triangleq (lquot\ q_1\ l_1) \cdot (lquot\ q_2\ l_2) \qquad \text{and} \qquad l' \triangleq lquot\ (q_1 \cdot q_2)\ (l_1 \cdot l_2)$$

If l and l' are both well-typed, then l and l' are equivalent q-lenses.

Proof: Since l is well typed, we have that $C_1 \cdot^! C_2$ and $A_1 \cdot^! A_2$. Also, since l' is well typed, we have that $B_1 \cdot^! B_2$. Using these facts, we prove that the component functions of l and l' are equivalent.

▶ **Get**: Let $c = c_1 \cdot c_2 \in (C_1 \cdot C_2)$. We calculate as follows

$$
\begin{aligned}
&l.get\ c \\
={}& ((lquot\ q_1\ l_1).get\ c_1) \cdot ((lquot\ q_2\ l_2).get\ c_2) \\
={}& (l_1.get\ (q_1.canonize\ c_1)) \cdot (l_2.get\ (q_2.canonize\ c_2)) \\
={}& (l_1 \cdot l_2).get\ ((q_1 \cdot q_2).canonize\ (c_1 \cdot c_2)) \qquad\qquad \text{as } B_1 \cdot^! B_2 \text{ and } C_1 \cdot^! C_2 \\
={}& (lquot\ (q_1 \cdot q_2)\ (l_1 \cdot l_2)).get\ c \\
={}& l'.get\ c
\end{aligned}
$$

and obtain the required equality.

▶ **Put**: Let $a = a_1 \cdot a_2 \in A_1 \cdot A_2$ and $c = c_1 \cdot c_2 \in C_1 \cdot C_2$. We calculate as follows:

$$
\begin{aligned}
&l.put\ a\ c \\
={}& ((lquot\ q_1\ l_1).put\ a_1\ c_1) \cdot ((lquot\ q_1\ l_1).put\ a_2\ c_2) \\
={}& (q_1.choose\ (l_1.put\ a_1\ (q_1.canonize\ c_1))) \cdot (q_2.choose\ (l_2.put\ a_2\ (q_2.canonize\ c_2))) \\
={}& (q_1 \cdot q_2).choose\ ((l_1 \cdot l_2).put\ a\ ((q_1 \cdot q_2).canonize\ c)) \\
& \qquad \text{as } A_1 \cdot^! A_2 \text{ and } B_1 \cdot^! B_2 \text{ and } C_1 \cdot^! C_2 \\
={}& (lquot\ (q_1 \cdot q_2)\ (l_1 \cdot l_2)).put\ a\ c \\
={}& l'.put\ a\ c
\end{aligned}
$$

and obtain the required equality.

▶ **Create**: Analogous to the case for *put*. □

The iteration operator on canonizers is similar:

$$
\frac{
\begin{array}{c}
q_1 \in C_1 \leftrightarrowtriangle B_1/\sim_{B_1} \\
C_1^{!*} \qquad \sim_B = \mathsf{TransClosure}(\sim^*_{B_1}) \\
split \in (B_1{}^*) \to \mathrm{List}(\mathrm{B_1})
\end{array}
}{
q_1{}^* \in C_1{}^* \leftrightarrowtriangle B_1{}^*/\sim^*_{B_1}
}
$$

$$
\begin{aligned}
canonize\ c_1 \cdots c_n ={}& (q_1.canonize\ c_1) \cdots (q_1.canonize\ c_n) \\
choose\ b ={}& (q_1.choose\ b_1) \cdots (q_n.choose\ b_n) \\
& \text{where } split\ b = [b_1, ..., b_n]
\end{aligned}
$$

4.8 Lemma: $q_1{}^* \in C_1{}^* \leftrightarrowtriangle B_1{}^*/\sim^*_{B_1}$.

Proof: Let $b \in B_1{}^*$ and let $split\ b = [b_1, ..., b_n]$. We calculate as follows

$$
\begin{aligned}
&(q_1{}^*).canonize\ ((q_1{}^*).choose\ b) \\
={}& (q_1{}^*).canonize\ ((q_1.choose\ b_1) \cdots (q_1.choose\ b_n)) \\
={}& (q_1.canonize\ (q_1.choose\ b_1)) \cdots (q_1.canonize\ (q_1.choose\ b_n)) \quad \text{as } C_1^{!*} \\
\sim_B{}& b_1 \cdots b_n \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{by } \textsc{ReCanonize} \text{ for } q_1
\end{aligned}
$$

and obtain the required equivalence. □

Note that as in the concatenation operator for canonizers, the iteration of B_1 is allowed to be ambiguous. The *split* function handles the splitting of strings in $B_1{}^*$ into substrings belonging to B_1. Boomerang has two variants of canonizer iteration: one with a longest-match *split* and another that uses shortest-match.

The final combinator in this section forms the union of two canonizers.

$$\frac{q_1 \ \in \ C_1 \leftrightarrow\!\!\!\rightarrow B_1/\!\!\sim_{B_1} \qquad C_1 \cap C_2 = \emptyset \qquad}{q_1 \mid q_2 \ \in \ C_1 \cup C_2 \leftrightarrow\!\!\!\rightarrow (B_1 \cup B_2)/(\sim_{B_1} \cup \sim_{B_2})}$$
$$q_2 \ \in \ C_2 \leftrightarrow\!\!\!\rightarrow B_2/\!\!\sim_{B_2} \qquad \sim_B = \mathsf{TransClosure}(\sim_{B_1} \cup \sim_{B_2})$$

$$canonize\ c = \begin{cases} q_1.canonize\ c & \text{if } c \in C_1 \\ q_2.canonize\ c & \text{if } c \in C_2 \end{cases}$$

$$choose\ b \ \ = \begin{cases} q_1.choose\ b & \text{if } b \in B_1 \\ q_2.choose\ b & \text{if } b \in B_2 \end{cases}$$

4.9 Lemma: $q_1 \mid q_2 \in C_1 \cup C_2 \leftrightarrow\!\!\!\rightarrow (B_1 \cup B_2)/(\sim_{B_1} \cup \sim_{B_2})$.

Proof: Let $b \in B$. We analyze two subcases. If $b \in B_1$ then we calculate as follows

$$
\begin{aligned}
& (q_1 \mid q_2).canonize\ ((q_1 \mid q_2).choose\ b) \\
&= q_1.canonize\ (q_1.choose\ b) && \text{as } b \in B_1 \text{ and } ran(q_1.choose) = C_1 \text{ and } C_1 \cap C_2 = \emptyset \\
& \sim_B b && \text{by } \textsc{ReCanonize} \text{ for } q_1
\end{aligned}
$$

and obtain the required equivalence. The case where $b \in B_2 \setminus B_1$ is symmetric. $\qquad\square$

5. Primitives

Now we define some primitive q-lenses and canonizers. As background, we recall two basic string lenses defined in Bohannon et al. (2008):

$$\frac{E \in \mathcal{R}}{copy\ E \in \llbracket E \rrbracket \Longleftrightarrow \llbracket E \rrbracket}$$

$$
\begin{aligned}
get\ c \ \ &= c \\
put\ a\ c \ &= a \\
create\ a &= a
\end{aligned}
$$

$$\frac{E \in \mathcal{R} \qquad u \in \Sigma^* \qquad v \in \llbracket E \rrbracket}{const\ E\ u\ v \in \llbracket E \rrbracket \Longleftrightarrow \{u\}}$$

$$
\begin{aligned}
get\ c \ \ &= u \\
put\ a\ c \ &= c \\
create\ a &= v
\end{aligned}
$$

The first, *copy E*, behaves like the identity on $\llbracket E \rrbracket$ in both directions. The second, *const E u v*, maps every string belonging to $\llbracket E \rrbracket$ to a constant u in the *get* direction and restores the concrete string in the *put* direction. The argument v is a default for *create*. Some other useful lenses are now expressible as derived forms:

$$
\begin{aligned}
E \leftrightarrow u &\triangleq const\ E\ u\ (rep(E)) \\
E \leftrightarrow u &\in \llbracket E \rrbracket \Longleftrightarrow \{u\} \\
del\ E &\triangleq E \leftrightarrow \epsilon \\
del\ E &\in \llbracket E \rrbracket \Longleftrightarrow \{\epsilon\} \\
ins\ u &\triangleq \epsilon \leftrightarrow u \\
ins\ u &\in \{\epsilon\} \Longleftrightarrow \{u\}
\end{aligned}
$$

$E \leftrightarrow u$ is like *const* but automatically chooses an element of E for *create*; *del E* deletes a concrete string belonging to E in the *get* direction and restores it in the *put* direction; *ins u* inserts a fixed string u in the *get* direction and removes it in the other direction. Each of these basic lenses can be converted to a q-lens using *lift*. To avoid clutter, we assume from now on that *copy*, *const*, etc. denote the (lifted) q-lens versions.

Now let us explore some q-lens variants of these basic lenses. The *get* component of *del* deletes a string and the *put* component restores it. In most situations, this is the behavior we want. However, if the deleted data is "ignorable"—e.g., whitespace—then we may prefer to have a *put* component that produces a canonical default $e \in \llbracket E \rrbracket$ instead of restoring the original. This transformation cannot be a basic lens because it violates GETPUT, but it is easy to define as a q-lens using left quotient:

$$qdel\ E\ e \quad \triangleq \quad lquot\ (canonizer\ (const\ E\ e\ e))\ (del\ e)$$

Operationally, the *get* function works by canonizing the input string to e, and then deleting it. The *put* restores e and passes it to the canonizer's *choose* component (i.e., the *create* component of *const*), which also produces e. The type of *qdel* E e, which is $\llbracket E \rrbracket / \mathsf{Tot}(\llbracket E \rrbracket) \Longleftrightarrow \{\epsilon\}/{=}$, records the fact that every string in $\llbracket E \rrbracket$ is treated equivalently. ($\mathsf{Tot}(\llbracket E \rrbracket)$ is the total relation on $\llbracket E \rrbracket$.)

Next we define a q-lens for inserting information into the abstract string. The q-lens *qins* e E behaves like *ins* e in the *get* direction, but its *put* component accepts the set $\llbracket E \rrbracket$ (where $e \in \llbracket E \rrbracket$). We often use *qins* in examples to insert canonical whitespace in the forward direction while accepting arbitrary whitespace in the other direction. Its definition is straightforward using right quotient:

$$qins\ e\ E \quad \triangleq \quad rquot\ (ins\ e)\ (canonizer\ (const\ E\ e\ e))$$

Again, the type of *qins* e E, namely $\{\epsilon\}/{=} \Longleftrightarrow \llbracket E \rrbracket / \mathsf{Tot}(\llbracket E \rrbracket)$, records the fact that the equivalence relation on the abstract domain is the total relation.

Q-lens versions of *const* and $E \leftrightarrow u$ are similar:

$$
\begin{aligned}
qconst\ u\ E\ D\ v \quad \triangleq \quad & lquot\ (canonizer\ (const\ E\ u\ u)) \\
& \quad (rquot\ (u \leftrightarrow v) \\
& \quad\quad (canonizer\ (const\ D\ v\ v))) \\
E \leftrightarrow D \quad \triangleq \quad & qconst\ (rep(E))\ E\ D\ (rep(D))
\end{aligned}
$$

The q-lens *qconst* accepts any E in the *get* direction and maps it to v; in the *put* direction, it accepts any D and maps it to u. Its type is $\llbracket E \rrbracket / \mathsf{Tot}(\llbracket E \rrbracket) \Longleftrightarrow \llbracket D \rrbracket / \mathsf{Tot}(\llbracket D \rrbracket)$. The q-lens $E \leftrightarrow D$ has the same type; it maps between E and D, producing arbitrary representatives in each direction.

The next primitive duplicates data. Duplication operators in the context of bidirectional languages have been extensively studied by Hu et al. (2004) and Liu et al. (2007) in settings with different laws. Our own previous lens languages have not supported duplication operators because their semantics is incompatible with the strict versions of the lens laws (and the types used in those languages, as we discuss in Section 6). In the more relaxed semantic space of q-lenses, however, duplication causes no problems.

$$
\frac{l \in C/{\sim_C} \Longleftrightarrow A_1/{\sim_{A_1}} \qquad f \in C \to A_2 \qquad A_1 \cdot^! A_2 \qquad {\sim_A} = {\sim_{A_1}} \cdot \mathsf{Tot}(A_2)}{dup_1\ l\ f \in C/{\sim_C} \Longleftrightarrow A_1 \cdot A_2/{\sim_A}}
$$

$$
\begin{aligned}
get\ c &= (l.get\ c) \cdot (f\ c) \\
put\ (a_1 \cdot a_2)\ c &= (l.put\ a_1\ c) \\
create\ (a_1 \cdot a_2) &= (l.create\ a_1)
\end{aligned}
$$

The q-lens dup_1 is parameterized on a q-lens l and a function f having the same domain as l (f is typically the *get* component of a q-lens; in Boomerang, the built-in function get extracts a q-lens's *get* component). The *get* function of dup_1 supplies one copy of the concrete string c to l's *get* component, sends a second copy to f, and concatenates the results. The *put* and *create* components discard the portion of the abstract string generated by f and invoke the corresponding component of l on the remainder of the abstract string. For example, if e and e' belong to $\llbracket E \rrbracket$, then

the *get* component of dup_1 (*copy E*) ((*copy E*).*get*) maps e to $e \cdot e$, and the *create* component maps $e \cdot e'$ to e. The typing rule records the fact that dup_1 ignores the part of the abstract string generated by f. The symmetric operator $dup_2 \ f \ l$ discards the first copy instead of the second in the *put/create* direction.

5.1 Lemma: $dup_1 \ l \ f \in C/{\sim_C} \iff A_1 \cdot A_2/{\sim_A}$.

Proof:

▶ **GetEquiv**: Let $c, c' \in C$ with $c \sim_C c'$. We calculate as follows

$$
\begin{aligned}
&(dup_1 \ l \ f).get \ c \\
={} &(l.get \ c) \cdot (f \ c) \\
\sim_A {} &(l.get \ c') \cdot (f \ c') \qquad \text{by } \textsc{GetEquiv} \text{ for } l \\
={} &(dup_1 \ l \ f).get \ c'
\end{aligned}
$$

and obtain the required equivalence.

▶ **PutEquiv**: Let $a, a' \in A$ with $a \sim_A a'$ and let $c, c' \in C$ with $c \sim_C c'$. Then there exist $a_1, a_1' \in A_1$ and $a_2, a_2' \in A_2$ with $a = a_1 \cdot a_2$ and $a = a_1' \cdot a_2'$ and $a_1 \sim_{A_1} a_1'$.

Using these facts, we calculate as follows

$$
\begin{aligned}
&(dup_1 \ l \ f).put \ a \ c \\
={} &(dup_1 \ l \ f).put \ (a_1 \cdot a_2) \ c \\
={} &l.put \ a_1 \ c \\
\sim_C {} &l.put \ a_1' \ c \qquad \text{by } \textsc{PutEquiv} \text{ for } l \\
={} &(dup_1 \ l \ f).put \ (a_1' \cdot a_2') \ c' \\
={} &(dup_1 \ l \ f).put \ a' \ c'
\end{aligned}
$$

and obtain the required equivalence.

▶ **CreatEquiv**: Analogous to the previous case, using $\textsc{CreateEquiv}$ for l.

▶ **GetPut**: Let $c \in C$. We calculate as follows

$$
\begin{aligned}
&(dup_1 \ l \ f).put \ ((dup_1 \ l \ f).get \ c) \ c \\
={} &(dup_1 \ l \ f).put \ ((l.get \ c) \cdot (f \ c)) \ c \\
={} &l.put \ (l.get \ c) \ c \qquad \text{as } ran(l.get) = A_1 \text{ and } A_1 \cdot^! A_2 \\
\sim_C {} &c \qquad\qquad\qquad\qquad\quad \text{by } \textsc{GetPut} \text{ for } l
\end{aligned}
$$

and obtain the required equivalence.

▶ **PutGet**: Let $a \in A$ and $c \in C$. Then there exist $a_1 \in A_1$ and $a_2 \in A_2$ with $a = a_1 \cdot a_2$. We calculate as follows

$$
\begin{aligned}
&(dup_1 \ l \ f).get \ ((dup_1 \ l \ f).put \ a \ c) \\
={} &(dup_1 \ l \ f).get \ ((dup_1 \ l \ f).put \ (a_1 \cdot a_2) \ c) \\
={} &(dup_1 \ l \ f).get \ (l.put \ a_1 \ c) \\
={} &(l.get \ (l.put \ a_1 \ c)) \cdot (f \ (l.put \ a_1 \ c)) \\
\sim_A {} &a \qquad\qquad\qquad\qquad\quad \text{by } \textsc{PutGet} \text{ for } l \text{ and definition of } \sim_A
\end{aligned}
$$

and obtain the required equivalence.

▶ **CreateGet**: Analogous to the previous case, using $\textsc{CreateGet}$ for l.

□

In both of these q-lenses, the handling of duplicated data is quite simple. (In particular, unlike the duplication operators proposed and extensively studied by Hu et al. (2004), *put* and *create* do not make any attempt to merge

changes to the duplicated data in the abstract string.) Nevertheless, they suffice for many practical examples. For example, when f is an aggregation operator such as count *count*,[6] discarding the aggregate valued while retaining the other copy often makes sense (see Section 8).

So much for primitive q-lenses; for the rest of this section, let us consider some primitive canonizers. The first is a generic combinator that builds a canonizer from a function mapping a set of structures onto a "normalized" subset of itself.

$$\frac{f \ \in \ C \to C_0 \qquad C_0 \subseteq C \qquad \forall c \in C_0. \ f \ c = c}{\textit{normalize } f \ \in \ C \leftrightarrow\!\!\!\to C_0/=}$$

$$\textit{canonize } c = f \ c$$
$$\textit{choose } c \quad = c$$

The *canonize* component is the given function f, and the *choose* component is the identity function.

5.2 Lemma: *normalize* $f \in C \leftrightarrow\!\!\!\to C_0/=$.

Proof: Let $c \in C_0$. We calculate as follows

$$\begin{aligned}
&(\textit{normalize } f).\textit{canonize } ((\textit{normalize } f).\textit{choose } c) \\
&= (\textit{normalize } f).\textit{canonize } c \\
&= f \ c \\
&= c \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{as } c \in C_0
\end{aligned}$$

and obtain the required equality. □

Using *normalize*, we can build a canonizer to put substrings of a larger string in sorted order. To lighten the presentation, we describe the binary version; the generalization to an n-ary sort is straightforward. Let C_1 and C_2 be regular languages that can be unambiguously concatenated in either order and such that $C_1{\cdot}C_2 \cap C_2{\cdot}C_1 = \emptyset$. Let $f \ C_1 \ C_2$ be the function that takes a string $c_1{\cdot}c_2 \in C_1{\cdot}C_2$ or $c_2{\cdot}c_1 \in C_2{\cdot}C_1$ and produces $c_1{\cdot}c_2$ in either case. It is easy to check that this function satisfies the side condition in the typing rule for *normalize* with $C_0 = C_1{\cdot}C_2$, since already sorted strings map to themselves. The canonizer *sort* $C_1 \ C_2$ is defined as *normalize* $(f \ C_1 \ C_2)$. It has the canonizer type

$$(C_1{\cdot}C_2 \cup C_2{\cdot}C_1) \leftrightarrow\!\!\!\to C_1{\cdot}C_2/=.$$

(We consider a variant of the sorting primitive in Section 6.)

It is occasionally necessary to introduce special primitives to meet the requirements of particular applications. The next primitive canonizer, *columnize*, is one such. It was designed specifically for processing the UniProt format described in Section 8, wrapping long lines of text by replacing spaces with newlines so that they do not spill over into the margin. However, it is easy to imagine *columnize* being handy in other situations—for processing textual documents or other kinds of string data presented in fixed-width formats.

$$\frac{(\Sigma^*{\cdot}nl{\cdot}\Sigma^*) \cap C_0 = \emptyset}{C = [(s \cup nl)/s]C_0}{\textit{columnize } C_0 \ s \ nl \ \in \ C \leftrightarrow\!\!\!\to C_0/=}$$

canonize c : *replace nl with s in c*
choose c : *replace s with nl as needed in c to wrap long lines*

---

[6] In Boomerang, the function *count* R takes a string u belonging to R^* and returns the number of substrings belonging to R that u can be split into.

It takes as arguments a set of strings C_0, a "space" string s, and a "newline" string nl. Its *canonize* component replaces every occurrence of the newline string with space; the *choose* component wraps long lines by replacing some of their spaces with newlines. The typing rule for *columnize* requires that nl not appear in strings in C_0 and assigns to the entire canonizer the type $C \leftrightarrow\!\!\!\triangleright\!\!\!\rightarrow C_0/\!\!=$, where C is obtained by widening C_0 so that nl may appear anywhere that s may.

5.3 Lemma: *columnize* C_0 s $nl \in C \leftrightarrow\!\!\!\triangleright\!\!\!\rightarrow C_0/\!\!=$.

Proof: Let $c \in C_0$. Then (*columnize* C_0 s nl).*canonize* (*columnize* C_0 s nl).*choose* $c = c$ since nl does not appear in c, (*columnize* C_0 s nl).*choose* only replaces some s characters with nl, and (*columnize* C_0 s nl).*canonize* replaces all nl characters with s. □

6. Loosening Lens Types with Canonizers

We were originally motivated to study q-lenses by the need to work "modulo insignificant details" when writing lenses to transform real-world data formats. However, as we began using our language to build larger examples, we discovered a significant—and completely unexpected—additional benefit: q-lenses allow us to assign many bidirectional transformations *coarser* types than the strict lens laws permit, easing some serious tractability concerns that arise in languages with extremely precise type systems.

The need for precise types stems from a fundamental choice in our design: the *put* component of every lens is a total function. Totality is attractive to users of lenses, because it guarantees that any valid abstract structure can be put back with any valid concrete structure—i.e., the lens can handle an arbitrary edit to the abstract view, as long as it stays within the specified type.[7] However, for exactly the same reason, totality makes it difficult to design lens primitives—the *put* function must do something reasonable with every pair of valid abstract and concrete structures, and the only way that a lens can avoid having to handle certain structures is by excluding them from its type. Thus, in practice, a lens language with a sufficiently rich set of primitives has to be equipped with a correspondingly rich set of types.

Working in a language with very precise types has many advantages. For example, Boomerang's type checker, which is based on regular languages, uncovered a subtle source of ambiguity in the UniProt ASCII format. But it also imposes burdens—both on programmers, who must write programs to satisfy a very picky type checker, and on implementations, where mechanizing these precise analyses often requires expensive algorithms. Fortunately, the increased flexibility of q-lenses and canonizers can be exploited to loosen types and alleviate these burdens. We give three examples.

The first example involves the *columnize* transformation, which was defined as a primitive canonizer in Section 5. The mappings between long lines of text and blocks of well-wrapped lines form a bijection and so trivially satisfy the lens laws. Thus, we could also define *columnize* as a basic lens. However, the type of this lens, which describes the set of minimally-split, well-wrapped blocks (i.e., sequences of lines that must be broken exactly at the margin column, or ones that must be broken at the column just before the margin because the next two characters are not spaces, or lines that must be broken at the second-to-last column..., and so on) is horribly complicated and cumbersome—both for programmers and in implementations. We could loosen the type to match the one we gave to the *columnize* canonizer—i.e., to arbitrary blocks of text, including blocks containing "extra" newlines—but changing the type in this way also requires changing the *put* function in order to avoid violating the GETPUT law. In particular, if we take a concrete block of text containing some extra newlines, map it to an abstract line by *get*, and immediately map it back to a concrete block by *put*, then the strict version of GETPUT stipulates that all of the extra newlines must be restored exactly. Thus, the *put* function cannot ignore its concrete argument and insert the minimal number of newlines needed to avoid spilling over into the margin; it must also examine the concrete string and restore any extra newlines from it. Formulating *columnize* as a canonizer rather than a lens, avoids both of these complications. By exploiting the additional flexibility permitted by the canonizer law, we obtain a primitive whose type and behavior are both simple.

---

[7] In the terminology of Hegner (1990), the abstract structures are "closed views."

The second example of a transformation whose type can be simplified using canonizers is *sort*. As with *columnize*, it is possible to define a basic lens version of *sort*. To sort $C_1...C_k$, we form the union of lenses that recognize the concatenations of permutations of the C_is, and apply the appropriate permutation to put them in sorted order. This lens has the behavior we want, but its type on the concrete side is the set of all concatenations of permutations of C_is—a type whose size grows as the factorial of k! As the number of languages being sorted increases, the size of this type rapidly becomes impractical. Fortunately, this combinatorial blowup can be avoided by widening the concrete type to $(C_1 \mid ... \mid C_n)^*$. This type overapproximates the set of strings that we actually want to sort, but has an *enormously* more compact representation—one that grows linearly with k. Of course, having widened the type in this way, we also need to extend the canonizer's functional components to handle this larger set of strings. In particular, we must extend *canonize* to handle the case where several or no substrings belong to a given R_i. A reasonable choice, which works well for many examples including sorting XML attributes and BibTeX fields, is to simply discard any extras and fill in any missing ones with defaults.

The final example involves the duplication operator. Consider the simple instance $dup_1\ (copy\ E)\ ((copy\ E).get)$ whose *get* function maps strings $e \in [\![E]\!]$ to $e \cdot e$ (assume the concatenation of E with itself is unambiguous). There *is* a pure basic lens with this behavior, but but in order to satisfy PUTGET, the domain of its *put* component must be restricted to abstract strings where the two copies of e are equal. (If it were defined on $e \cdot e'$ with e different from e', then no matter what concrete string e'' it produced, the *get* function would produce a string $e'' \cdot e''$, violating PUTGET.) Thus, as a basic lens, the type of dup_1 must include an equality constraint in its abstract component: $[\![E]\!] \iff \{e \cdot e' \in [\![E]\!] \cdot [\![E]\!] \mid e = e'\}$. Unfortunately, this type is not regular and cannot be expressed in Boomerang's type system. Since we were not prepared to deal with these equality constraints in our type system, we were forced to exclude dup_1 as a primitive in earlier versions of Boomerang. However, if we take dup_1 as a q-lens, we can assign it a more flexible type with no equality constraint: $[\![E]\!]/= \iff [\![E]\!] \cdot [\![E]\!]/= \cdot \mathsf{Tot}([\![E]\!])$. Executing a round-trip via *put* and *get* on an abstract string $e \cdot e'$ with e different from e' is no problem. Although the result, $e \cdot e'$, and original string $e \cdot e$ are different, they are related by $= \cdot \mathsf{Tot}([\![E]\!])$. Hence, the PUTGET law is satisfied.

7. Typechecking

The typing rules for some of the q-lens combinators—including left and right quotienting, sequential composition, and union—place constraints on the equivalence relation components in the types of the q-lenses they combine. For example, to check that an instance of sequential composition $l\ ;\ k$ is well formed, we need to verify that l's abstract equivalence relation and k's concrete one are identical. In this section, we describe two different approaches to implementing these rules. The first uses a coarse analysis, simply classifying equivalences according to whether they are or are not the equality relation. Surprisingly, this very simple analysis captures our most common programming idioms and turns out to be sufficient for all of the applications we have built. The second approach is much more refined: it represents equivalence relations by rational functions that induce them. This works, in principle, for a large class of equivalence relations including most of our canonizers (except for those that do reordering). However, it appears too expensive to be useful in practice.

The first type system is based on two simple observations: first, that most q-lenses originate as lifted basic lenses, and therefore have types whose equivalence relations are both equality; and second, that equality is preserved by many of our combinators including all of the rational operators, *permute*, sequential composition, and even (on the non-quotiented side) the left and right quotient operators. These observations suggest a coarse classification of equivalence relations into two sorts:

$$\tau\ ::=\ Identity\ \mid\ Any$$

We can now restrict the typing rules for our combinators to only allow sequential composition, quotienting, and union of types whose equivalence relation type is $Identity$. Although this restriction seems draconian (it disallows many q-lenses that are valid according to the typing rules presented in earlier sections), it turns out to be surprisingly successful in practice—we have not needed anything more to write many thousands of lines of demo applications. The reasons for this are twofold. First, is that it allows two q-lenses to be composed, whenever the uses of *lquot* are all in the lens on the left and the uses of *rquot* on the right, a very common case. And second, it allows arbitrary

q-lenses (with any equivalences) to be concatenated as long as the result is not further composed, quotiented, or unioned—another very natural idiom. This is the typechecking algorithm currently implemented in Boomerang.

We can (at least in theory) go further by replacing the *Identity* sort with a tag carrying an arbitrary rational function f (i.e., a function computable by a finite state transducer):

$$\tau ::= \; Rational \text{ of } f \; | \; Any$$

Equivalence relations induced by rational functions are a large class that includes nearly all of the equivalence relations that can be formed using our combinators—everything except q-lenses constructed from canonizers based on *sort* and *permute*. Moreover, we can decide equivalence for these relations.

7.1 Definition: Let $f \in A \longrightarrow B$ be a rational function. Denote by $\sim_f$ the relation $\{(x, y) \in A \times A \; | \; f(x) = f(y)\}$.

7.2 Lemma: Let $f \in A \longrightarrow B$ and $g \in A \longrightarrow C$ be rational and surjective functions. Define a rational relation $h \subseteq C \times B$ as $(f \circ g^{-1})$. Then $(\sim_g \subseteq \sim_f)$ iff h is functional.

Proof: Let's expand the the definition of h

$$h(c) = \{f(a) \; | \; a \in A \text{ and } g(a) = c\}$$

Observe that, by the surjectivity of g we have $h(c) \neq \emptyset$.

($\Rightarrow$) Suppose that $\sim_g \subseteq \sim_f$.

Let $b, b' \in h(c)$. Then by the definition of h, there exist $a, a' \in A$ with $b = f(a)$ and $b' = f(a')$ and $g(a) = c = g(a')$. We have that $a \sim_g a'$, which implies that $a \sim_f a'$, and so $b = b'$. Since b and b' were arbitrary elements of $h(c)$, we conclude that h is functional.

($\Leftarrow$) Suppose that h is functional.

Let $a, a' \in A$ with $a \sim_g a'$. Then there exists $c \in C$ such that $g(a) = g(a') = c$. By the definition of h, and our assumption that h is functional, we have that $f(a) = h(c) = f(a')$ and so $a \sim_f a'$. Since a and a' were arbitrary, we conclude that $\sim_g \subseteq \sim_f$. $\qquad\square$

7.3 Corollary: Let f and g be rational functions. It is decidable whether $\sim_f = \sim_g$.

Proof: Recall that rational relations are closed under composition and inverse. Observe that $\sim_f = \sim_g$ iff both $f \circ g^{-1}$ and $g \circ f^{-1}$ are functional. Since these are both rational relations, the result follows using the decidability of functionality for rational relations (Blattner 1977). $\qquad\square$

The condition mentioned in union can also be decided using an elementary construction on rational functions. Thus, this finer system gives decidable type checking for a much larger set of q-lenses. Unfortunately, the constructions involved seem quite expensive to implement.

We are currently investigating the decidability of extensions capable of handling our full set of canonizers, including those that permute and sort data.

8. Experience: Q-lenses for Genomic Data

In this section we describe our experiences using Boomerang to implement a q-lens that maps between XML and ASCII versions of the UniProtKB/Swiss-Prot protein sequence database. We described a preliminary version of this lens in previous work (Bohannon et al. 2008), but while that lens handled the essential data in each format, it did not handle the full complexity of either. On the XML side, it only handled databases in a certain canonical form—e.g., with attributes in a particular order. On the ASCII side, it did not conform to the UniProt conventions for wrapping long lines, and it did not handle duplicated or aggregated data. We initially considered implementing custom canonizers (in OCaml) for the ASCII format, but this turned out to be quite complicated due to the slightly

different formatting details used to represent lines for various kinds of data. Re-engineering this program as a q-lens was a big improvement. Our new version, about 4200 lines of Boomerang code, handles both formats fully, using just the canonizer and q-lens primitives described above. In this section we sketch some highlights from this development, focusing on interesting uses of canonizers. Along the way, we describe our generic XML library, which encapsulates many details related to processing and transforming XML trees bidirectionally.

Let's start with a very simple lens that gives a taste of programming with q-lenses in Boomerang, focusing on the canonization of XML trees. In the XML presentation of UniProt databases, patent citations are represented as XML elements with three attributes:

```
<citation type="patent" date="1990-09-20"
    number="WO9010703"/>
```

In ASCII, they appear as RL lines:

```
RL  Patent number WO9010703, 20-SEP-1990.
```

The bidirectional between these formats is essentially bijective—the patent number can be copied verbatim from the attribute to the line, and the date just needs to be transformed from YYYY-MM-DD to DD-MMM-YYYY—but, because the formatting of the element may include extra whitespace and the attributes may appear in any order, building a lens that maps between all valid representations of patent citations in XML and ASCII formats is more complicated than it might first seem.

A bad choice (the only choice available with just basic lenses) would be to treat the whitespace and the order of attributes as data that should be explicitly discarded by the *get* function and restored by the *put*. This complicates the lens, since it then has to explicitly manage all this irrelevant data. Slightly better would be to write a canonizer that standardizes the representation of the XML tree and compose this with a lens that operates on the canonized data to produce the ASCII form. But we can do even better by mixing together the functions of this canonizer and lens in a single q-lens. (The code uses some library and auxiliary functions that are described later.)

```
let patent_xml : lens =
  ins "RL  " .
  Xml.attr3_elt_no_kids NL2 "citation"
    "type" ("patent" <-> "Patent number" . space)
    "number" (escaped_pcdata . comma . space)
    "date" date .
  dot
```

This lens transforms concrete XML to abstract ASCII in a single pass. The first line inserts the RL tag and spaces into the ASCII format. The second line is a library function from the Xml module that encapsulates details related to the processing of XML elements. The first argument, a string NL2, is a constant representing the second level of indentation. It is passed as an argument to an *qdel* instance that constructs the leading whitespace for the XML element in the reverse direction. The second argument, citation, is the name of the element. The remaining arguments are the names of the attributes and the lenses used for processing their corresponding values. These are given in canonical order. Internally, the attr3_elt_no_kids function sorts the attributes to put them into this order. The space, comma, and dot lenses insert the indicated characters; escaped_pcdata handles unescaping of PCDATA; date performs the bijective transformation on dates illustrated above.

The next fragment demonstrates quotienting on the abstract ASCII side. In XML, taxonomic lineages of source organisms are represented like this:

```
<lineage>
  <taxon>Eukaryota</taxon>
  <taxon>Lobosea</taxon>
  <taxon>Euamoebida</taxon>
  <taxon>Amoebidae</taxon>
  <taxon>Amoeba</taxon>
</lineage>
```

In ASCII, these are flattened onto lines tagged with `OC`:

```
OC   Eukaryota; Lobosea; Euamoebida; Amoebidae; Amoeba.
```

The code that converts between these formats is:

```
let oc_taxon : lens =
  Xml.pcdata_elt NL3 "taxon" esc_pcdata in

let oc_xml : lens =
  ins "OC   " .
  Xml.elt NL2 "lineage"
    (iter_with_sep oc_taxon (semi . space)) .
  dot
```

The first lens, `oc_taxon`, processes a single `taxon` element using a library function `pcdata_elt` that extracts encapsulated PCDATA from an element. As in the previous example, the NL3 argument is a constant representing canonical whitespace. The second lens, `oc_xml`, processes a `lineage` element. It inserts the `OC` tag into the ASCII line and then processes the children of the `lineage` element using a generic library function `iter_with_sep` that iterates its first argument using Kleene-star, and inserts its second argument between iterations. The `dot` lens terminates the line.

The lineage for amoeba is compact enough to fit onto a single `OC` line, but most lineages are not:[8]

```
OC   Eukaryota; Metazoa; Chordata; Craniata; Vertebrata;
OC   Euteleostomi; Mammalia; Eutheria; Euarchontoglires;
OC   Primates; Haplorrhini; Catarrhini; Hominidae; Homo.
```

The q-lens that maps between single-line `OC` strings produced by `oc_xml` and the final line-wrapped format:

```
let oc_q : canonizer =
  columnize (atype oc_xml) " " "\nOC   "
let oc_line : lens = rquot oc_xml oc_q
```

(The `atype` primitive extracts the abstract part of the type of a q-lens; `ctype`, used below, extracts the concrete part.)

Lastly, let us look at two instances where data is duplicated. In a few places in the UniProt database, there is data that is represented just once on the XML side but several times on the ASCII side. For example, the count of the number of amino acids in the actual protein sequence for an entry is listed as an attribute in XML

```
<sequence length="262" ...>
```

but appears twice in ASCII, in the `ID` line...

```
ID   GRAA_HUMAN Reviewed; 262 AA.
```

...and again in the `SQ` line:

```
SQ   SEQUENCE   262 AA;  28969 MW;
```

Using dup_2, we can write a lens that copies the data from the XML attribute and onto both lines in the ASCII format. The backwards direction of dup_2 discards the copy on the `ID` line, a reasonable policy for this application.

Another place where duplication is needed is when data is aggregated. The ASCII format of the information about alternative splicings of the gene is

```
CC   -!- ALTERNATIVE PRODUCTS:
CC       Event=Alternative initiation; Named isoforms=2;
CC       Name=Long; Synonyms=Cell surface;
CC         IsoId=P08037-1; Sequence=Displayed;
```

---

[8] To fit the human lineage into a single column, we have split lines longer than 45th column; in a real UniProt instance, the lines would be split at the 75th column.

```
CC       Name=Short; Synonyms=Golgi complex;
CC          IsoId=P08037-2; Sequence=VSP_018801;
```

where the `Named isoforms` field in the second line is the count of the number of `Name` blocks that follow below. The Boomerang code that generates these lines uses dup_2 and *count* to generate the appropriate integer in the *get* direction; in the reverse direction, it simply discards the integer generated by *count*.

9. Related Work

The idea of bidirectional transformations that work up to an equivalence relation is quite general. In this section, we exploit the framework of quotient lenses to illuminate some previously proposed systems—XSugar (Brabrand et al. 2007), biXid (Kawanaka and Hosoya 2006), and X/Inv (Hu et al. 2004). The comparison with XSugar is the most interesting, and we carry it out in detail, showing how a core fragment of XSugar can be translated into our notation. The others are discussed more briefly.

9.1 XSugar

XSugar is a language for writing conversions between XML and ASCII formats. Conversions are specified using pairs of intertwined grammars, in which the nonterminal names are shared and the right-hand side of each production specifies both a string and an XML representation (separated by =). For instance, our composers example from the introduction would be written as follows:

```
db    : [comps cs] = <composers> [comps cs] </>
comps : [comp c] [comps cs] = [comp c] [comps cs]
      : =
comp  : [Name n] "," [Birth b] "-" [Death d] =
          <composer>
            <name> [Name n] </>
            <years birth=[Birth b] death=[Death d]/>
            <nationality> [Nationality] </> </>
```

(The pattern ": = " used in the third line indicates that the list of composers can be empty in both formats.) XSugar programs transform strings by parsing them according to one grammar and pretty printing the resulting parse tree using the other grammar as a template. Additionally, on the XML side, the representation of trees is standardized using a generic canonizer.

Well-formed XSugar programs are guaranteed to be bijective modulo an equivalence relation that captures XML normalization, replacement of items mentioned on just one side of the grammar with defaults, and reordering of order-insensitive data. Since every bijection is a lens (semantically), every XSugar program is trivially a q-lens. However, it is not immediately clear that there should be a connection between XSugar and q-lenses at the level of *syntax*, since XSugar programs are specified using variables and recursion, while q-lenses in our notation are written using "point-free" combinators and no recursion.

To make the connection, we describe how to compile a core subset of XSugar into Boomerang. First, we identify a syntactic restriction on XSugar grammars that ensures regularity. Second, we compile the individual patterns used in XSugar grammars to lenses. Third, we apply a standard rewriting technique on grammars to eliminate recursion. And finally, we quotient the resulting lens by a canonizer that standardizes the representation of XML trees.

XSugar productions are given by the following grammar

$$
\begin{aligned}
p &::= r : q_1 : ... : q_k \\
q &::= \alpha^* = \beta^* \\
\alpha &::= "s" \mid [R] \mid [r\ z] \\
\beta &::= \alpha \mid <t(n{=}\alpha)^*>\beta^*</>
\end{aligned}
$$

where productions p contain a non-terminal r and a set of patterns q_1 to q_k, each of the form $\alpha^* = \beta^*$. The αs describe the ASCII format and the βs describe the XML format. The symbols used in patterns include literals $"s"$, unnamed

regular expressions $[R]$, non-terminals binding variables $[r\ z]$, and XML elements $\texttt{<}t\ n_1\texttt{=}\alpha_1...n_k\texttt{=}\alpha_k\texttt{>}\beta^*\texttt{</>}$. (The full XSugar language also includes several extensions, including XML namespaces, precedence declarations, and unordered productions. We discuss unordered productions below and ignore the others.) Well-formed XSugar programs satisfy two syntactic properties: every variable (i.e., z in $[r\ z]$) occurring in a pattern is used exactly once on each side of the pattern, and productions are unambiguous. (Ambiguity of context-free languages is undecidable, but the XSugar system employs a conservative algorithm that is said to perform well in practice.)

The first step in our compilation imposes an additional syntactic restriction on grammars to ensure regularity.[9] Recall that a language is regular iff it can be defined by a right-linear grammar. However, requiring that *every* non-terminal appear in the right-most position is clunky and needlessly restrictive; we can use a slightly less draconian restriction in which productions are sorted into mutually recursive groups of rules. Within each group, recursion must be right-linear, but references to non-terminals defined in earlier (in the order of the topological sort) recursion groups may be used freely. We construct the recursion groups by building the graph of references to non-terminals between rules, collapsing cycles by coalescing mutually-dependent groups, and performing a topological sort. The recursion groups for the composers example are as follows:

$$\{\texttt{comp}\} < \{\texttt{comps}\} < \{\texttt{db}\}$$

It turns out that, since XSugar patterns are linear in their variables, imposing right-linearity separately on the ASCII and XML portions of patterns ensures a kind of joint right-recursion: every pattern has one of two forms

$$\alpha_1...\alpha_k = \beta_1...\beta_l \qquad \text{or}$$
$$\alpha_1...\alpha_k\ [r_i\ z] = \beta_1...\beta_l\ [r_i\ z],$$

where each non-terminal except for the final, optional $[r_i\ z]$ refers to a rule defined in a preceding recursion group. This restriction rules out many XSugar programs—in particular, it obviously cannot handle XML with recursive schemas—but still captures a large class of useful transformations, including most of the demos in the XSugar distribution.

Next, we compile patterns to lenses. There are two cases. For non-recursive patterns, we construct a lens that maps between the XML and ASCII patterns using *permute*. For example, the $\texttt{comp}$ rule in the example compiles to the lens

```
permute [1;2;3;4;5;6;7]
  [("<composer><name>" <-> ""); (copy Name);
   ("</><years birth=" <-> ","); (copy Birth);
   (" death="          <-> "-"); (copy Death);
   (("/><nationality> . Nationality . "</></>") <-> "")]
```

where $\texttt{Name}$, $\texttt{Birth}$, and $\texttt{Death}$ are bound to the appropriate regular expressions and $\texttt{[1;2;3;4;5;6;7]}$ represents the identity permutation. Note that since both sides of a pattern may contain regular expressions, it is essential that the $\texttt{<->}$ be a q-lens (the basic lens variant of $\texttt{<->}$ only allows a string on the abstract side.) For recursive patterns, we compile the prefix of the pattern in the same way, but associate it with the final non-terminal in the right-most position. The result, after compiling each pattern, is a grammar in which the terminal symbols are lenses and any recursive non-terminals are right-recursive. For example, the $\texttt{comps}$ rule compiles to the following:

$$comps : (\texttt{permute [1] [comp]})\ comps$$
$$: \texttt{permute []}$$

(The second lens, a 0-ary permutation, is equivalent to *copy* ϵ.)

---

[9] This is essential—while full XSugar can be used to describe context-free languages, the types of string lenses are always regular, so without this restriction we would be trying to compile a context-free formalism into a regular one, which is clearly not possible! Note that we are not claiming that our q-lens syntax subsumes all of the functionality of XSugar, but rather illustrating the generality of our account of bidirectional programming modulo equivalences by drawing a connection with a completely different style of syntax for bidirectional programs.

The final step in the construction is to replace right-recursion by iteration, using a generalization of a standard construction on grammars. We calculate the lens for each non-terminal in a recursion group separately. There are again two cases. If r_i : $p_1...p_k$ is the only non-terminal in the group, then we partition the patterns into two sets: recursive patterns go into S_1, and non-recursive patterns into S_2. We then construct the following lens for r_i: $\left(\cup_{(k\ r_i)\in S_1}k\right)^*\cdot(\cup_{l\in S_2}l)$. It is easy to see that this lens describes the same transformation. For the case where the recursion group contains more than one non-terminal, we eliminate one non-terminal by replacing references to it with a similarly constructed lens, and then repeat the compilation. For the composers example, this compilation produces the following lenses (after replacing trivial permutations with concatenations):

```
let comp : lens =
  ("<composer><name>" <-> """) . (copy Name) .
  ("</><years birth=" <-> ",") . (copy Birth) .
  (" death=" <-> "-") . (copy Death) .
  ("/><nationality>" . Nationality . "</></>" <-> """)

let comps : lens = comp* . copy ""

let db : lens =
  (del "<composers>" <-> "") . comps . ("</>" <-> "")
```

One additional restriction of the translation should be mentioned. The typing rules for q-lenses check unambiguity *locally*—every concatenation and iteration—and demand that unions be disjoint. Our compilation only produces well-typed lenses for grammars that are "locally unambiguous" and "locally disjoint" in this sense.

The lenses produced by this compilation expect XML trees in a canonical form. To finish the job, we need to build a canonizer that standardizes the representation of input trees. We can do this with an analogous compilation that only uses the XML side of the grammar. For example, a canonizer for comp is

```
del WS . copy "<composer>" . del WS .
copy "<name>" . copy Name . "</name>" <-> "</>" .
del WS . copy "<years" . del WS .
sort2
 (ins " " . copy ("birth=\"" . Birth . "\"") .
    del WS)
 (ins " " . copy ("death=\"" . Death . "\"") .
    del WS) .
copy "/>" . del WS .
copy "<nationality>" . copy Nationality .
"</nationality>" <-> "</>"
```

where we have elided the coercions from q-lenses to canonizers, and where WS indicates whitespace. The final q-lens is built by quotienting the compiled lens by this canonizer.

XSugar also supports productions where patterns are tagged as unordered. These result in transformations that canonize order. We believe that an extension to unordered patterns is feasible using additional *sort* primitives in the compiled canonizer, but we leave this extension as future work.

9.2 biXid

The biXid language (Kawanaka and Hosoya 2006) specifies bidirectional conversions between pairs of XML documents. As in XSugar, biXid transformations are specified using pairs of grammars, but biXid grammars may be ambiguous and may contain non-linear variable bindings. These features are central to biXid's design. They are used critically, for example, in transformations such as the following, which converts between different representations of browser bookmark files.

```
relation contents_reorder =
    (var nb | var nf)* <-> (var xb)*, (var xf)*
```

```
where bookmark(nb, xb), folder(nf,xf)
```

One transformation, read from right to left, parses a sequences of bookmarks (`nb`) or folders (`nf`), interleaved in any order, converts them to the other format using using `bookmark` and `folder` transformations, and constructs a sequence in which the bookmarks (`xb`) appear before the folders (`xf`). The other transformation, read from left to right, parses a sequence consisting of bookmarks followed by folders, converts each of these to the first format, and then produces a sequence where bookmarks and folders may be freely interleaved (in fact, the biXid implementation does preserve the order of parsed items—i.e., the constructed sequence has bookmarks followed by folders—but this behavior is not forced by the semantics.) Ambiguity is also used in biXid to generate data that only appears on one side (analogous to XSugar's "unnamed" items [R]) and to handle data that may be represented in multiple ways— e.g., string values that can be placed either in an attribute or as PCDATA in a nested element.

In principle, we could identify a syntactically restricted subset of biXid, and compile it to Boomerang like we did for XSugar. However, since the computation models are so different—in particular, ambiguity is fundamental to biXid—the restrictions needed to make this work would be heavy and would likely render the comparison uninteresting. Instead, we discuss informally how each of the idioms requiring ambiguity, as identified by the designers of biXid, can be implemented instead using q-lenses and canonizers. Ambiguity arising from "freedom of ordering", as in the bookmarks transformation, can be handled using a q-lens that canonizes the order of the interleaved pattern using *sort*. Ambiguity due to unused data can be handled using combinators like *qins*, *qdel* and $E \leftrightarrow D$. Finally, ambiguity due to multiple representations of data can be handled by canonizing the various representations; for each of the examples discussed in the biXid paper, these canonizers are simple and local transformations.

9.3 Languages with Duplication

Bidirectional languages capable of duplicating data in the *get* direction, either by explicit combinators or implicitly by non-linear uses of variables, have been the focus of recent work by the Programmable Structured Documents group at Tokyo.

In early work, Mu et al. (2004) designed an injective language called Inv with a primitive duplication combinator and demonstrated that it satisfies variants of the basic lens laws—our GETPUT and the more relaxed PUTGETPUT law that we showed in the introduction. A key aspect of their approach is that transformations manipulate tagged values that carry edit annotations. The idea is that, using these annotations, the *put* direction of the duplication operator can check if a copied value has been modified (by looking for edit tags in the data), and, if so, incorporate these changes in its result. The semantics of other primitives in Inv are generalized to propagate tagged values. In particular, using a synchronization primitive for list values, they demonstrate that it is possible to achieve a sophisticated backwards semantics for several intricate operations on lists, even in the presence of duplication. Inv was later used as the foundation for a high-level bidirectional language for tree transformations, called X, and a structured document editor (Hu et al. 2004; Mu et al. 2006).

A second line of work from the same group investigates bidirectional languages with variable binding. Languages that allow unrestricted occurrences of variables implicitly support duplication, since data can be copied by programs that use a variable several times. The goal of this work is to develop a bidirectional semantics for XQuery (Liu et al. 2007). As in the earlier work, they propose relaxed variants of the lens laws and develop a semantics based on sophisticated propagation of annotated values.

One possible connection between their work and q-lenses is an informal condition proposed in the journal version of Hu et al. (2004). This is formulated in terms of an ordering on edited values that captures when one value is "more edited" than another. They propose strengthening the laws to require that composing *put* and *get* produce an abstract structure that is more edited in this sense, calling this property *update preservation*. We hope to investigate the relationship between our q-lens PUTGET law and their PUTGETPUT plus update preservation. (The comparison may prove difficult to make, however, because our framework is "state based"—the *put* function only sees the state of the data structure resulting from some set of edits, not the edits themselves—while theirs assumes an "operation-based" world in which the locations and effects of edit operations are explicitly indicated in the data.)

10. Integration with Dictionary Lenses

So far, we have focused on the refinement of basic lenses to q-lenses. The lenses used in Boomerang, however, are actually more complicated structures called *dictionary lenses*. In this section we argue that it makes sense to mix the two flavors of lenses in the same language by demonstrating how dictionary lenses can also be enhanced with quotienting.

Dictionary lenses are designed to manipulate data that may be reordered. Basic lenses have *put* functions that aligns data in the concrete and abstract structures by position—e.g., when processing a list of structures using Kleene-star, a dictionary lens matches up "chunks" of the concrete and abstract by "key" . Dictionary lenses are specified using special combinators that delineate certain parts of the data as reorderable "chunks", and a portion of each chunk as a "key". The *put* function works by parsing up its concrete argument into a dictionary structure, where chunks are organized by key, and a "skeleton" structure. The *put* function rebuilds a new concrete string, using both the information in the skeleton and the chunks in the dictionary. Critically, when the updated abstract structure is obtained by reordering chunks, it finds the corresponding chunks in the dictionary.

The formal definition of a dictionary lens from C to A with skeleton type S and dictionary type D, also written $C \overset{S,D}{\Longleftrightarrow} A$, is a structure with components:

$$
\begin{aligned}
l.get &\in C \longrightarrow A \\
l.put &\in A \longrightarrow S \times D \longrightarrow C \times D \\
l.create &\in A \longrightarrow D \longrightarrow C \times D \\
l.parse &\in C \longrightarrow S \times D \\
l.key &\in A \longrightarrow K
\end{aligned}
$$

The important thing to note is the *parse* function, which takes a concrete string and produces a dictionary and a "skeleton" structure that remains after the chunks are removed, and the *put* function, which operates on a skeleton and dictionary and produces the new concrete string and the dictionary that remains after removing any chunks needed to produce the concrete string. We refer the reader to the original Boomerang paper (Bohannon et al. 2008) for more details.

The way to see how these functions work, is to see how one can package up a dictionary lens l as a basic lens $\bar{l}$:

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

The symbol $\{\}$ denotes the empty dictionary. Note that the *parse* and *put* function are evaluated in strict sequence; this separation of phases facilitates the global reordering of concrete chunks that is performed by the *put* function.

Just as we refined basic lenses to q-lenses, we can revise dictionary lenses to qd-lenses. The GETPUT law for quotient-dictionary lenses (the notation $+\!\!+$ denotes the operation that concatenates two dictionary structures):

$$
\frac{s, d' = l.parse\ c \qquad a \sim_A l.get\ c \qquad l.put\ a\ (s, (d' +\!\!+ d)) = c', d''}{c \sim_C c' \text{ and } d'' = d} \tag{GETPUT}
$$

states that the *put* function restores information to c up to $\sim_C$, and also that it consumes the dictionary constructed from c exactly.

Similarly, we can revise the quotienting operators to wrap the arguments for the functional components, including *parse* and *key*.

$$l \in C/\!\sim_C \stackrel{S,D}{\Longleftrightarrow} B/\!\sim_B \qquad q \in A \leftrightarrow\!\!\!\mapsto B/\!\sim_B$$
$$a \sim_A a' \text{ iff } q.canonize\ a \sim_B q.canonize\ a'$$
$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$
$$rquot\ l\ q\ \in\ C/\!\sim_C \stackrel{S,D}{\Longleftrightarrow} A/\!\sim_A$$

$$
\begin{aligned}
get\ c &= q.choose\ (l.get\ c) \\
put\ a\ (s,d) &= l.put\ (q.canonize\ a)\ (s,d) \\
create\ a\ d &= l.create\ (q.canonize\ a)\ d \\
parse\ c &= l.parse\ c \\
key\ a &= l.key\ (q.canonize\ a)
\end{aligned}
$$

$$l \in B/\!\sim_B \stackrel{S,D}{\Longleftrightarrow} A/\!\sim_A \qquad q \in C \leftrightarrow\!\!\!\mapsto B/\!\sim_B$$
$$c \sim_C c' \text{ iff } q.canonize\ c \sim_B q.canonize\ c'$$
$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$
$$lquot\ q\ l\ \in\ C/\!\sim_C \stackrel{S,D}{\Longleftrightarrow} A/\!\sim_A$$

$$
\begin{aligned}
get\ c &= l.get\ (q.canonize\ c) \\
put\ a\ (s,d) &= q.choose\ (l.put\ a\ (s,d)) \\
create\ a\ d &= q.choose\ (l.create\ a\ d) \\
parse\ c &= l.parse\ (q.canonize\ c) \\
key\ a &= l.key\ a
\end{aligned}
$$

We can also use a dictionary lenses l as a canonizer by composing the conversion $\bar{l}$, which packages a dictionary lens up with the interface of a basic lens, with *canonizer*, which maps a basic lens to a canonizer. Note that this conversion produces a canonizer that does not use dictionaries at all (the *create* function is supplied with $\{\}$); hence, dictionary lenses do not behave in surprising ways when they are used as canonizers.

Besides the lens laws, dictionary lenses also obey an additional law, which captures, abstractly, the fact that they should be oblivious to the order of chunks. It is phrased in terms of an equivalence relation $\sim_R$ that identifies concrete strings that differ only in key-respecting reorderings of chunks. With quotients, this new law is as follows:

$$\frac{c \sim_R c'}{\bar{l}.put\ a\ c = \bar{l}.put\ a\ c'} \qquad\qquad (\textsc{EquivPut})$$

(Note that EQUIVPUT is stated using the basic lens version of l.) There are two equivalence relations on C in play—-$\sim_C$ and $\sim_R$. These relations capture orthogonal aspects of bidirectional transformations on ordered data: the first captures irrelevant differences between concrete strings, while the second describes which data can be reordered. Every quotient-dictionary lens program constructed using our combinators obeys EQUIVPUT for a $\sim_R$ that can be read off from its description.

Finally, we note that the interaction of dictionaries across sequential composition is not yet well understood. However, the behavior of composition in some simple common cases—e.g., when only one of l or k in $l\,;k$ contains "chunks"—is clear.

11. Conclusion

Q-lenses generalize basic lenses by allowing their forward and backward transformations to treat certain data as "ignorable." This extension, while modest at the semantic level, turns out have an elegant syntactic story based on canonizers and quotienting operators—a story that is both parsimonious (the same core primitives are used as lenses and as canonizers) and compositional (unlike previous approaches, where canonization is kept at the edges of transformations, our canonizers can be arbitrarily interleaved with the processing of data). Moreover, the additional flexibility offered by q-lenses make it possible to define many useful primitives such as duplication and sorting.

Our experience suggests that canonizers and q-lenses are essential for handling the details of real-world ad hoc data formats. Although many of these details appear minor at first sight, attempting to sidestep them makes the transformations we write essentially useless. Quotient lenses are the critical piece of technology that makes it possible to build precisely the bidirectional transformations we want. Thus, q-lenses and canonizers fill a much-needed gap between theory and practice of bidirectional languages.

Naturally, there are still many interesting issues left to be investigated. On the theoretical side, we would like to understand better how to characterize the set of programs for which the simple, coarse type analysis described in Section 7 is sufficient, and whether this simple analysis can be refined to admit more programs without going as far as the very expensive analysis in terms of rational functions. We would also like to investigate q-lenses for other structures besides strings, such as trees. On the engineering side, we are working on scaling up the Boomerang implementation to handle large datasets such as full-size (1Gb) UniProt databases. In particular, we believe it would be useful to have an algebraic theory of program equivalence for q-lenses as a basis for an optimizing compiler.

Acknowledgments

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.

Amos Bairoch, Rolf Apweiler, Cathy H. Wu, Winona C. Barker, Brigitte Boeckmann, Serenella Ferro, Elisabeth Gasteiger, Hongzhan Huang, Rodrigo Lopez, Michele Magrane, Maria J. Martin, Darren A. Natale, Claire O'Donovan, Nicole Redaschi, and Lai. The Universal Protein Resource (UniProt). *Nucleic Acids Research*, 33(Database issue):D154–9, January 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.

Pablo Berdaguer, Alcino Cunha, Hugo Pacheco, and Joost Visser. Coupled schema transformation and data conversion for XML and SQL. In *International Symposium on Practical Aspects of Declarative Languages (PADL), Nice France*, volume 4354, pages 290–304, 2007.

Meera Blattner. Single-valued a-transducers. *Journal of Computer and System Sciences*, 15(3):310–327, 1977.

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. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*, January 2008.

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 2007. To appear. Extended abstract in *Database Programming Languages (DBPL)* 2005.

A. Cunha, J.N. Oliveira, and J. Visser. Type-safe two-level data transformation. *European Symposium on Formal Methods*, 4085:284–299.

Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *TODS*, 7(3): 381–416, September 1982.

David T. Eger. Bit level types, 2005. Unpublished manuscript. Available from `http://www.yak.net/random/blt/blt-drafts/03/blt.pdf`.

Sander Evers, Peter Achten, and Rinus Plasmeijer. Disjoint forms in graphical user interfaces. In *Trends in Functional Programming*, volume 5, pages 113–128. Intellect, 2006. ISBN 1-84150-144-1.

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.

Michael Greenberg and Shriram Krishnamurthi. Declarative Composable Views, 2007. Undergraduate Honors Thesis. Department of Computer Science, Brown University.

Stephen J. Hegner. Foundations of canonical update support for closed database views. In *International Conference on Database Theory (ICDT), Paris, France*, pages 422–436, New York, NY, USA, 1990. Springer-Verlag New York, Inc. ISBN 0-387-53507-1. URL `http://www.cs.umu.se/ hegner/Publications/PDF/icdt90.pdf`.

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

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.

Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. Bidirectional interpretation of xquery. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM), Nice, France*, pages 21–30, New York, NY, USA, 2007.

David Lutterkort. Augeas: A Linux configuration API, February 2007. Available from `http://augeas.net/`.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, volume 42, pages 47–58. ACM Press New York, NY, USA, 2007.

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

Renée J. Miller, Mauricio A. Hernandez, Laura M. Haas, Lingling Yan, C. T. Howard Ho, Ronald Fagin, and Lucian Popa. The clio project: Managing heterogeneity. 30(1):78–83, March 2001.

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

Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. Bidirectionalizing tree transformation languages: A case study. *JSSST Computer Software*, 23(2):129–141, 2006.

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.

Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS), Nashville, TN*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2007. ISBN 978-3-540-75208-0.

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