Synthesizing Quotient Lenses

SOLOMON MAINA, University of Pennsylvnia, USA
ANDERS MILTNER, Princeton University, USA
KATHLEEN FISHER, Tufts University, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
DAVID WALKER, Princeton University, USA
STEVE ZDANCEWIC, University of Pennsylvania, USA

Quotient lenses are bidirectional transformations whose correctness laws are "loosened" by specified equivalence relations, allowing inessential details in concrete data formats to be suppressed. For example, a programmer could use a quotient lens to define a transformation that ignores the order of fields in XML data, so that two XML files with the same fields but in different orders would be considered the same, allowing a single, simple program to handle them both. Building on a recently published algorithm for synthesizing plain bijective lenses from high-level specifications, we show how to synthesize bijective quotient lenses in three steps. First, we introduce *quotient regular expressions* (QREs), annotated regular expressions that conveniently mark inessential aspects of string data formats; each QRE specifies, simulteneously, a regular language and an equivalence relation on it. Second, we introduce *QRE lenses*, i.e., lenses mapping between QREs. Our key technical result is a proof that every QRE lens can be transformed into a functionally equivalent lens that canonizes source and target data just at the "edges" and that uses a bijective lens to map between the respective canonical elements; no internal canonization occurs in a lens in this normal form. Third, we leverage this normalization theorem to *synthesize* QRE lenses from a pair of QREs and example input-output pairs, reusing earlier work on synthesizing plain bijective lenses. We have implemented QREs and QRE lens synthesis as an extension to the bidirectional programming language Boomerang. We evaluate the effectiveness of our approach by synthesizing QRE lenses between various real-world data formats in the Optician benchmark suite.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; *Application specific development environments*;

Additional Key Words and Phrases: Bidirectional Programming, Program Synthesis, Type-Directed Synthesis, Type Systems

ACM Reference Format:
Solomon Maina, Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing Quotient Lenses. *Proc. ACM Program. Lang.* 2, ICFP, Article 80 (September 2018), 29 pages. https://doi.org/10.1145/3236775

1 INTRODUCTION

Programmers often need to write programs that bidirectionally convert data between two different formats. For example, bibliographic data may be stored in BibTeX or EndNote formats, spreadsheet

Authors' addresses: Solomon Maina, University of Pennsylvnia, USA, smaina@seas.upenn.edu; Anders Miltner, Princeton University, USA, amiltner@cs.princeton.edu; Kathleen Fisher, Tufts University, USA, kfisher@eecs.tufts.edu; Benjamin C. Pierce, University of Pennsylvania, USA, bcpierce@cis.upenn.edu; David Walker, Princeton University, USA, dpw@cs.princeton.edu; Steve Zdancewic, University of Pennsylvania, USA, stevez@cis.upenn.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2018 Copyright held by the owner/author(s).

2475-1421/2018/9-ART80
https://doi.org/10.1145/3236775

data can be represented as either CSV and TSV files, and web APIs can return data as either JSON or XML objects. A time-tested and appealing way to implement such conversions is to use *lenses* [Foster et al. 2007], programs that simultaneously define pairs of functions for translating the data from the source format to the target format (the *get* direction) and back again (the *put* direction). Lens programming languages guarantee that these put and get functions satisfy certain *lens laws*, which imply that round trips, from source to target and back again, behave properly.

Recent work [Miltner et al. 2017] developed a tool called Optician to synthesize a subset of the *bijective lenses* that can be derived in the lens programming language *Boomerang* [Bohannon et al. 2008]. Each lens $\ell : S \Leftrightarrow T$ that Optician can synthesize specifies a bijection from the language of a source regular expression S to a target language of a target regular expression T:

$$\ell.\mathsf{get}\ (\ell.\mathsf{put}\ t) = t, \text{ and } \ell.\mathsf{put}\ (\ell.\mathsf{get}\ s) = s \tag{1}$$

Optician makes programming bijective lenses easier. However, many useful lenses are not strictly bijective in nature—the desired transformation might ignore whitespace or the exact ordering of data fields, for instance—and it is a shame that Optician's synthesis algorithm cannot be directly applied in such cases. One observation, however, is that non-bijective transformations can often be structured as a bijective "core" surrounded by some kind of data normalization at the edges. We can therefore hope to use the Optician algorithm as component in a system that synthesizes more complex lenses.

This paper applies this idea to the problem of synthesizing *quotient lenses* [Foster et al. 2008]. Quotient lenses are lenses in which the lens laws are loosened so that they hold modulo an equivalence relation on the source and target data respectively; in this paper we are concerned with *bijective quotient lenses* which are lenses for which Equation 1 holds modulo equivalence relations $\equiv_S$ and $\equiv_T$ defined on the source and target data respectively:

$$\ell.\mathsf{get}\ (\ell.\mathsf{put}\ t) \equiv_T t, \text{ and } \ell.\mathsf{put}\ (\ell.\mathsf{get}\ s) \equiv_S s \tag{2}$$

Quotient lenses are useful in situations where a programmer wishes for the transformation defined by a lens to have the same behavior on data that differ only in inessential details. For instance, a programmer may wish to "quotient away" the number of white space characters between data items, the capitalization of various strings, the sequence of fields in a record, or the order of items in a list.

Optician synthesizes bijective lenses from a pair (S, T) of regular expressions specifying the source and target types and a set of example input-output pairs that guide the synthesis algorithm. This presents a challenge for synthesizing quotient lenses since a specification of a lens's source and target formats needs to account for the equivalence relations defined on the respective formats. Our solution is to introduce *Quotient Regular Expressions* (QREs), which are regular expressions augmented with extra syntax that enables programmers to simultaneously specify a regular expression and an equivalence relation on the language of that regular expression. Further, given a QRE, we can automatically infer a *canonizer*—a function that converts strings in the language of the regular expression to a canonical form.

For example, consider the following QRE for writing author names:

```
let wsp_sp = collapse wsp → "␣"
let comma_name = last_name . "," . wsp_sp . first_name
```

In this example, wsp is an existing regular expression for a nonempty sequence of whitespace characters, and first_name and last_name are existing regular expressions for first and last names. The QRE wsp_sp is a QRE with the same underlying language as wsp, but with a single canonical representative: "␣". It is used as a component of comma_name, a QRE with an underlying language of two names, separated by a command and whitespace, where the names with a single space between them are canonical.

Our second main contribution in this paper is to introduce *QRE lenses*, with the end goal of synthesizing QRE lenses that map between QREs via a synthesized bijective lens between the respective canonical formats. This idea is simple and natural but begs the following question: do we give up expressiveness if we restrict ourselves to this form? Our main technical contribution to this question asserts that we do not—more specifically, we prove a normal form theorem (Theorem 5.2) that says that every QRE lens formed by freely composing QRE lenses, applying regular operators to them, and quotenting them with QREs can be rewritten to be the composition of a source canonizer, a bijective lens, and a target canonizer. This normalization property will enable us to (1) synthesize QRE lenses by extending the synthesis algorithm used by Optician, and (2) prove that if there is a QRE lens that satisfies the input specification, then this extended algorithm will return such a lens.

Given this framework, generating a QRE lens requires only a pair of QREs to describe the source and target formats and a (possibly empty) suite of examples demonstrating the mapping. For example, the following code

let ℓ = synth comma_name ⇔ space_name using {("Lovelace,␣Ada", "Ada␣Lovelace")}

binds ℓ to a synthesized QRE lens mapping between names in the comma-separated form described by the QRE comma_name and the space-separated form described by the QRE space_name.

In summary, our main contributions are:

(1) We introduce *Quotient Regular Expressions* (QREs), a compact, convenient notation for simultaneously defining a regular language modulo some equivalence relation and a canonizer for that relation (Section 4).
(2) We introduce *QRE lenses*, which translate between data formats specified using QREs (Section 5).
(3) We design a specific set of useful QRE lens combinators and prove that QRE lenses defined using these combinators can be put into a restricted normal form (Section 5). This is the main technical contribution of this paper.
(4) We leverage this result to reduce the problem of *synthesizing* QRE lenses to the problem of synthesizing bijective string lenses, which was previously studied in past work [Miltner et al. 2017] (Section 6).
(5) We extend Boomerang with QREs and QRE lens synthesis and demonstrate the utility and practicality of our approach by synthesizing QRE lenses between a variety of data formats drawn from the Optician benchmark suite (Section 7).

Sections 8 and 9 present related and future work.

2 BACKGROUND: BIJECTIVE STRING LANGUAGES

Before describing QREs and QRE lenses, we briefly review bijective string lenses. Consider a bidirectional transformation that converts between BibTeX citation records such as

```
@Book {Lovelace,
Author = "Ada Lovelace",
Title = {Notes}
}
```

and equivalent EndNote records like the following:

```
%0 Book
%F Lovelace
%A Ada Lovelace
```

```
let preamble : "@book{" ⇔ "%0␣Book\n%F␣" =
del "@book{" . ins "%0␣Book\n%F␣"

let author_lens : (",\n" . bib_author) ⇔ ("\n" . end_author) =
del ",\nauthor␣=␣\""
. ins "\n%A␣"
. name
. (del "␣and␣"
. ins "\n"
. (ins "%A␣" . name . del "␣and␣" . ins "\n")*
. ins "%A␣"
. name
|| "")

let title_lens : ("\",\n" . bib_title . "},\n}") ⇔ ("\n" . end_title) =
del "\",\ntitle␣=␣{" . ins "\n%T␣" . title . del "},\n}"

let bib_to_end : bibtex ⇔ end_note = preamble . label . author_lens . title_lens
```

Fig. 1. A plain bijective lens bib_to_end between data matching bibtex and end_note regular expressions (which are omitted for brevity).

```
%T Notes
```

Boomerang's bijective string lenses are designed to define bidirectional transformations such as this one, where the data formats can be matched in a one-to-one manner, in this case by matching the label, author and title fields. Boomerang encourages a compositional approach in which programmers define simple lenses and then compose them using a variety of combinators.

Primitive lenses include:

- del *s*: delete the constant string *s* in the get direction; insert it in the put direction.
- ins *s*: insert the constant string *s* in the get direction; delete it in the put direction.
- copy *R*: copy the text matching the regular expression *R* in both directions.

Such primitives may be combined using the regular operators Kleene star, alternation and concatenation, as well as lens composition. Figure 1 illustrates the use of these combinators to define a lens bib_to_end that transforms data in BibTeX to EndNote (and vice versa).

Recent work described the Optician algorithm/tool [Miltner et al. 2017], which can synthesize bijective lenses such as bib_to_end, obviating the sometimes tedious tasks involved in writing such lenses by hand. Given the directive

```
synth S ⇔ T using exs
```

Optician will synthesize a bijective lens between source and target formats described by regular expressions S and T, respectively, and constrained by a set of input–output example pairs exs specifying how the synthesized lens should behave on those examples.

3 QRE LENSES BY EXAMPLE

Optician greatly simplifies the task of programming bijective string lenses, but not all bidirectional transformations are bijective. For instance, BibTeX users are not typically interested in preserving whitespace between words. The order of author and title fields is also likely irrelevant, and there may be equivalent ways of writing the same name: “Lovelace, Ada” vs “Ada Lovelace.” Consequently, the following two BibTeX citations represent the same logical object even though they differ in nonessential details.

```
@Book {Lovelace,
Author = "Ada Lovelace",
Title = {Notes},
}
```

```
@Book{     Lovelace,
Title = {Notes},
Author = "Lovelace, Ada",    }
```

When mapping these records into another format, such as EndNote, we must decide what to do with the nonessential information. A bijective mapping must preserve all the information, including the extraneous details, which leads to complex and brittle lenses. A better approach is to identify records that differ only in the nonessential information, mapping them into a canonical representation. This canonical representation is then mapped into the target format. With this approach, both of the above BibTeX records would be mapped to the same EndNote record.

We use *Quotient Regular Expresions* (QREs) to specify the external format in full detail and to mark which pieces of it are inessential. From a QRE, we can infer a regular expression that describes only the essential information, which we call the internal format, and we can derive a canonizer that maps between the external and internal formats.

3.1 Specifying BibTeX Using QREs

In this subsection, we develop a QRE specification of BibTeX records, introducing various QRE combinators along the way. Our first step in this process is to define a whitespace format, which externally matches any non-zero number of whitespace characters. It converts any such whitespace into a single space character, its canonical form. We use the QRE collapse primitive to define this whitespace-normalizing QRE.

```
let wsp_sp = collapse wsp ⟶ "␣"
```

Sometimes there are multiple disjoint representations of the same data. In such situations, the QRE squash combinator creates a QRE that allows external data to be in either format, and converts any data in the first format to the second. For instance, assume that the comma_name format describes “Lovelace, Ada” and that the space_name format describes “Ada Lovelace” and c_to_s is a function from the first to the second. In this case, the following instance of squash creates the desired canonizer.

```
let name = squash comma_name ⟶ space_name using c_to_s
```

One way to define the c_to_s function is simply to write it from scratch in some ordinary programming language. However, we can synthesize such functions automatically—here, c_to_s is the get direction of a lens that can be synthesized using the synth combinator:

```
let ℓ = synth comma_name ⇔ space_name using {("Lovelace,␣Ada", "Ada␣Lovelace")}
let c_to_s = ℓ.get
```

The first line above synthesizes a lens between comma_name and space_name using the listed example transformation as a guide. The second line extracts the get direction transformation from the lens, which is what we need for squash.

The permutation QRE combinator, perm, allows data to be unordered. For example, the following instance of perm allows label, author, and title fields (which we assume have been defined earlier) to appear in any order.

```
let bib_fields = perm (label, bib_author, bib_title)
```

To normalize the field separators, one can specify in an optional with clause that the components of the permutation are conjoined by another QRE. For instance, below, we normalize whitespace between fields, leaving only a single newline.

```
let wsp = [␣\n\t\r]+
let wsp_sp = collapse wsp ⟶ "␣"
let last_name = [A–Z][a–z]
let first_name = [A–Z][a–z]

(* define name representations with a space and with a comma *)
let space_name = first_name . wsp_sp . last_name
let comma_name = last_name . "," . wsp_sp . first_name

(* synthesize a lens that maps comma representation to space representation *)
let ℓ = synth comma_name ⇔ space_name using {("Lovelace,␣Ada", "Ada␣Lovelace")}
let c_to_s = ℓ.get

(* squash QRE maps comma_name to space_name *)
let name = squash comma_name ⟶ space_name using c_to_s

(* define rest of bibtex fields *)
let bib_names = name . (wsp_sp . "and" . wsp_sp . name)*
let bib_author = "author␣=␣\"" . bib_names . "\""
let title = word . (wsp_sp . word)*
let bib_title = "title␣=␣{" . title . "}"

(* allow any permutation of fields interspersed with arbitrary whitespace *)
let bib_fields = perm (label, bib_author, bib_title) with (collapse ("," . wsp) ⟶ ",\n")
let bibtex = "@book{" . bib_fields . "}"
```

Fig. 2. QRE definition of BibTeX records.

```
let bib_fields = perm (label, bib_author, bib_title) with (collapse ("," . wsp) ⟶ ",\n")
```

Another QRE primitive is the functional composition combinator, written ";". For an example of its use, suppose we have already defined a QRE, canonized_whitespace, that accepts XML documents and chooses documents with no whitespace as canonical. Suppose that we also have defined a QRE, canonized_order, which accepts whitespace-normalized XML documents, and chooses a specific ordering of XML elements as canonical. We can use the functional composition combinator to combine these two QREs into canonized_whitespace ; canonized_order, a QRE that accepts all XML documents, and chooses ordered XML documents without whitespace as canonical.

The final QRE combinator is the normalize combinator. This combinator allows a programmer to manually define a function f which sends each string that matches a regular expression R to some canonical representative in another regular expression R' where $\mathcal{L}(R') \subseteq \mathcal{L}(R)$. The equivalence relation defined by the normalize combinator is hence the equivalence relation defined by the *fibres* of f; that is, for all strings s and s' that match R, s is equivalent to s' if and only if $f(s) = f(s')$.

For instance, assume that $f(s)$ = "␣" (a space character) for all whitespace strings s. Then the collapse QRE wsp_sp defined above can be expressed using the normalize combinator as normalize (wsp,"␣", f).

Figure 2 gives a QRE definition for the simple BibTeX records we consider here.

3.2 QRE Lenses and QRE Lens Synthesis

At this point we have a tool for synthesizing bijective string lenses from a pair of regular expressions and a set of example input-output pairs (Optician), and we have a way of defining regular expressions

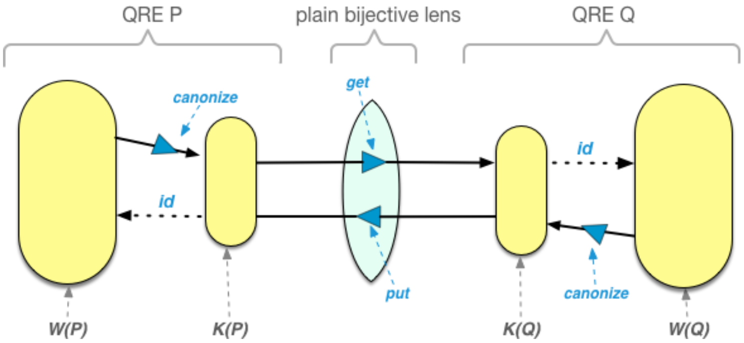


Fig. 3. QRE lens from source P to target Q. P and Q are QREs, each consisting of a "whole" set ($W(P)$ and $W(Q)$) and a "kernel" part ($K(P)$ and $K(Q)$) Between the two kernels is a plain bijective lens. The get function of the whole lens takes an argument from $W(P)$, applies the canonizer for P to obtain a canonical representative in $K(P)$, then applies the get of the plain lens, yielding an element of $K(Q)$, which is a subset of $W(Q)$. The put function does the reverse, mapping from $W(Q)$ to $K(P)$ (hence $W(P)$. This lens is in normal form, with canonizers (specified by QREs) at the outer edges and an ordinary bijective lens in the center.

with equivalence relations indicating the essential information (QREs). A tantalizing possibility would be to use the bijective string lens synthesis procedure as a subroutine for a quotient lens synthesis procedure. This new synthesizer would take as input source and target QREs and example input-output pairs, compute the canonical source/target formats from the QREs, map the example input-output pairs to their canonical representations, and then invoke the bijective string lens synthesis procedure on the canonical data formats and the canonical examples.

Indeed this idea is what motivates our definition of *QRE lenses*. Intuitively, our QRE lenses are bijective lenses with "canonizers at the edges". Figure 3 depicts the architecture of QRE lenses. Every QRE lens q has a type $P \Leftrightarrow Q$ where P and Q are QREs. In the *get* direction, a QRE lens $q : P \Leftrightarrow Q$ uses the source QRE P to compute a canonical representative for the data modulo the equivalence relation defined by P and then applies the get function of a bijective string lens ℓ to this representative. In the *put* direction, q operates similarly, but using the QRE Q and the put function of ℓ.

Because the QREs P and Q determine the internal formats for data after canonization, and because the algorithm for synthesizing bijective string lenses is directed by these formats, P and Q are all that is required to synthesize QRE lenses end-to-end.

However, our requirement that canonizers appear only at the edges raises a key technical question: Are we limiting the expressiveness of our transformations by demanding all programs fit into this normal form? It turns out that we are not—any lens that uses canonizers internally can be transformed into a lens that uses canonizers only at the edges. The main technical contribution of this paper (Theorem 5.2) is a proof of this fact. This technical result justifies using synthesis to produce QRE lenses instead of manually writing them, which can lead to substantial savings in program complexity. For instance, after defining the BibTeX and EndNote QREs and binding them to the variables bibtex and endnote respectively, then all of the code in Figure 1 may be replaced by a single call to the synthesis prodedure:

```
let bib_to_end : bibtex ⇔ endnote =
synth bibtex ⇔ endnote using {(bib_example, end_example)}
```

Here, the generated quotient lens synchronizes bibtex and endnote formats, using bib_example and end_example (the two concrete example strings given at the beginning of this section) to disambiguate. In addition, and as we saw earlier with the definition of c_to_s, the synthesis procedure itself can be used to create lenses that are in turn used to define other QREs. The ability to interleave QRE specification with QRE lens synthesis yields a powerful and flexible way of creating bidirectional transformations.

4 QUOTIENT REGULAR EXPRESSIONS

A Quotient Regular Expression (or QRE) is a regular expression R augmented with syntax that expresses an equivalence relation on the language of R. This section formalizes the set of QRE combinators that we introduced informally in Section 3.

4.1 Syntax and Semantics of QREs

Formally, the language of Quotient Regular Expressions (QREs) is given by the following grammar,

$$\begin{aligned} Q := {} & \mathsf{normalize}(R_1, R_2, f) \;\mid\; \mathsf{id}(R) \;\mid\; \mathsf{collapse}\ R \mapsto s \;\mid\; \mathsf{squash}\ Q_1 \rightarrow Q_2\ \mathsf{using}\ f \\ & \mid \mathsf{perm}(Q_1, \ldots, Q_n)\ \mathsf{with}\ Q \;\mid\; Q_1\ ;\ Q_2 \;\mid\; Q_1 \cdot Q_2 \;\mid\; (Q_1 \mid Q_2) \;\mid\; Q^* \end{aligned}$$

where Q ranges over QREs, R ranges over regular expressions, f ranges over functions between regular languages, and s ranges over character strings.

Using the conventional notation that $\mathcal{L}(R)$ is the language accepted by the regular expression R, each QRE Q yields four semantic objects:

$W(Q)$ A regular expression, denoting the "whole" of Q
$\equiv_Q$ An equivalence relation on $\mathcal{L}(W(Q))$
$K(Q)$ A regular expression, denoting the "kernel" of Q, such that $\mathcal{L}(K(Q))$ forms a complete set of representatives for $\equiv_Q$
$\mathsf{canonize}(Q)$ A "canonizing" function. Given any $w \in \mathcal{L}(W(Q))$, $\mathsf{canonize}(Q)(w)$ is the unique k in $\mathcal{L}(K(Q))$ such that $k \equiv_Q w$.

Intuitively, $W(Q)$ is the regular expression representing the external format, while $K(Q)$ is the regular expression representing the internal format. The equivalence relation $\equiv_Q$ groups together elements in the language of $W(Q)$ that contain the same essential information. The function $\mathsf{canonize}(Q)$ picks the representative element from each of the resulting equivalence classes.

The well-formedness constraints for QREs ensure that these four semantic objects fit together to form a coherent quotient $W(Q)/{\equiv_Q}$ whose equivalence classes are determined by $\mathsf{canonize}(Q)$.

4.2 The normalize Combinator

The relationship among the semantic objects of a QRE can be understood in terms of the combinator $\mathsf{normalize}(R_1, R_2, f)$, which expresses each of these pieces explicitly. Its whole language is just R_1, its kernel language is just R_2, and its canonizer is just f; its equivalence relation $\equiv$ is determined by the fibres of f, so we have $s_1 \equiv s_2 \Leftrightarrow f(s_1) = f(s_2)$ for s_1 and s_2 in $\mathcal{L}(R_1)$.

These components form a coherent quotient language when the canonization function f is surjective and idempotent (intuitively, f picks out a unique representative for each equivalence class). We also require that the kernel language be a subset of the whole language, which enables QRE composition. These considerations lead to the following well-formedness rule.

$$\frac{\mathcal{L}(R_2) \subseteq \mathcal{L}(R_1) \qquad f : \mathcal{L}(R_1) \longrightarrow \mathcal{L}(R_2) \qquad f \text{ is surjective} \qquad f = f^2}{\mathsf{normalize}(R_1, R_2, f)\ \mathsf{wf}} \;\text{(Normalize)}$$

Semantically, the normalize QRE is universal—each of the other combinators Q is equivalent to $\mathsf{normalize}(W(Q), K(Q), f)$ for some surjective, idempotent function $f : \mathcal{L}(W(Q)) \longrightarrow \mathcal{L}(K(Q))$.

Q	$W(Q)$	$K(Q)$
$\mathsf{id}(R)$	R	R
$\mathsf{collapse}\ R \mapsto s$	R	s
$\mathsf{squash}\ Q_1 \longrightarrow Q_2\ \mathsf{using}\ f$	$W(Q_1) \mid W(Q_2)$	$K(Q_2)$
$\mathsf{normalize}(R_1, R_2, f)$	R_1	R_2
$Q_1\ ;\ Q_2$	$W(Q_1)$	$K(Q_2)$
$Q_1 \cdot Q_2$	$W(Q_1) \cdot W(Q_2)$	$K(Q_1) \cdot K(Q_2)$
$Q_1 \mid Q_2$	$W(Q_1) \mid W(Q_2)$	$K(Q_1) \mid K(Q_2)$
Q^*	$W(Q)^*$	$K(Q)^*$

$$W(\mathsf{perm}(Q_1, \ldots, Q_n)\ \mathsf{with}\ Q) = \bigcup_{\sigma \in S_n} W(Q_{\sigma(1)}) \cdot W(Q) \cdot \ldots \cdot W(Q) \cdot W(Q_{\sigma(n)})$$

$$K(\mathsf{perm}(Q_1, \ldots, Q_n)\ \mathsf{with}\ Q) = K(Q_1) \cdot K(Q) \cdot \ldots \cdot K(Q) \cdot K(Q_n)$$

Fig. 4. Whole and Kernel regular expressions for QRE combinators. In describing regular expressions, we use the notations $\mid$ and $\bigcup$ for alternation, the notation $\cdot$ for concatenation, and the notation $*$ for Kleene closure. We use the notation S_n to denote the set of all permutations of the numbers 1 to n.

However, verifying that a canonization function f is surjective and idempotent is in general undecidable. Consequently, a programmer wishing to use normalize must discharge strong proof obligations, which is cumbersome in practice.[1]

The remaining QRE combinators, which we discuss next, provide simpler, more compositional ways of building canonizers that meet these requirements by construction. Nevertheless, the normalize combinator provides a useful guide in the design of these combinators because it gives a sufficient condition for the well-formedness of any potential QREs.

4.3 QRE Combinator Semantics

Figure 4 gives the inductive definitions of the whole and kernel languages $W(Q)$ and $K(Q)$ for all of the QRE combinators. The squash and permutation combinators have the two most interesting definitions. If $Q = \mathsf{squash}\ Q_1 \longrightarrow Q_2\ \mathsf{using}\ f$, then the whole language of Q is $W(Q_1) \mid W(Q_2)$ because the squash combinator merges the whole language $W(Q_1)$ of Q_1 with the whole language $W(Q_2)$ of Q_2. The function $f : \mathcal{L}(W(Q_1)) \longrightarrow \mathcal{L}(W(Q_2))$, maps $\mathcal{L}(W(Q_1))$ into $\mathcal{L}(W(Q_2))$ using f and then canonizes $W(Q_2)$ into $K(Q_2)$ using $\mathsf{canonize}(Q_2)$.

For the $\mathsf{perm}(Q_1, \ldots, Q_n)\ \mathsf{with}\ Q$ combinator, the whole language is the union of languages of the form $W(Q_{\sigma(1)}) \cdot W(Q) \cdot \ldots \cdot W(Q) \cdot W(Q_{\sigma(n)})$ for any permutation σ in S_n, where S_n is the set of all permutations of the numbers 1 to n. Intuitively, the permutation combinator allows for the string to match any permutation of the Q_i's while preserving the separator Q in between each of the Q_i's. The kernel of the permutation combinator is the language $K(Q_1) \cdot K(Q) \cdot \ldots \cdot K(Q) \cdot K(Q_n)$, because the canonical permutation is the identity permuation, with each of the parts of the input that match Q_i and Q canonized into $K(Q_i)$ and $K(Q)$ respectively.

Figure 5 gives the inductive definitions of the canonize function for each QRE. The permutation combinator gives rise to the most interesting definition:

$$\begin{aligned}&\mathsf{canonize}(\mathsf{perm}(Q_1, \ldots, Q_n)\ \mathsf{with}\ Q)(w_{\sigma(1)} \cdot s_1 \cdot \ldots \cdot s_{n-1} \cdot w_{\sigma(n)})\\ &= \mathsf{canonize}(Q_1)(w_1) \cdot \mathsf{canonize}(Q)(s_1) \cdot \ldots \cdot \mathsf{canonize}(Q)(s_{n-1}) \cdot \mathsf{canonize}(Q_n)(w_n)\end{aligned}$$

[1] In our implementation we allow a programmer to use normalize at their own risk without checking these side conditions as an "escape hatch" for the case when other QRE combinators are insufficient.

$$
\begin{array}{lcl}
\mathsf{canonize}(\mathsf{id}(R))(w) & = & w \\
\mathsf{canonize}(\mathsf{collapse}\ R \mapsto s)(w) & = & s \\
\mathsf{canonize}(\mathsf{squash}\ Q_1 \rightarrow Q_2\ \mathsf{using}\ f)(w) & = & \begin{cases} \mathsf{canonize}(Q_1)(f(w)) & \text{if } w \in \mathcal{L}(W(Q_1)) \\ \mathsf{canonize}(Q_2)(w) & \text{otherwise} \end{cases} \\
\mathsf{canonize}(\mathsf{normalize}(R_1, R_2, f))(w) & = & f(w) \\
\mathsf{canonize}(Q_1\ ;\ Q_2)(w) & = & \mathsf{canonize}(Q_2)(\mathsf{canonize}(Q_1)(w)) \\
\mathsf{canonize}(Q_1 \cdot Q_2)(w_1 \cdot w_2) & = & \mathsf{canonize}(Q_1)(w_1) \cdot \mathsf{canonize}(Q_2)(w_2) \\
\mathsf{canonize}(Q_1 \mid Q_2)(w) & = & \begin{cases} \mathsf{canonize}(Q_1)(w) & \text{if } w \in \mathcal{L}(W(Q_1)) \\ \mathsf{canonize}(Q_2)(w) & \text{if } w \in \mathcal{L}(W(Q_2)) \end{cases} \\
\mathsf{canonize}(Q^*)(w_1 \cdot \ldots \cdot w_n) & = & \begin{cases} \epsilon & \text{if } n = 0 \\ \mathsf{canonize}(Q)(w_1) \cdot \ldots \cdot \mathsf{canonize}(Q)(w_n) & \text{if } n > 0 \end{cases} \\
\multicolumn{3}{l}{\mathsf{canonize}(\mathsf{perm}(Q_1, \ldots, Q_n)\ \mathsf{with}\ Q)(w_{\sigma(1)} \cdot s_1 \cdot \ldots \cdot s_{n-1} \cdot w_{\sigma(n)})} \\
\multicolumn{3}{l}{\ = \ \mathsf{canonize}(Q_1)(w_1) \cdot \mathsf{canonize}(Q)(s_1) \cdot \ldots \cdot \mathsf{canonize}(Q)(s_{n-1}) \cdot \mathsf{canonize}(Q_n)(w_n)}
\end{array}
$$

Fig. 5. QRE canonizers. Here we assume that the input w to the canonizers has been partitioned in the unique way guaranteed to exist by the lens unambiguity conditions.

$$
\begin{array}{lcl}
w \equiv_{\mathsf{normalize}(R_1, R_2, f)} w' & \Longleftrightarrow & f(w) = f(w') \\
w \equiv_{\mathsf{id}(R)} w' & \Longleftrightarrow & w = w' \\
w \equiv_{\mathsf{collapse}\ R \mapsto s} w' & \Longleftrightarrow & \mathit{True} \\
w \equiv_{\mathsf{squash}\ Q_1 \rightarrow Q_2\ \mathsf{using}\ f} w' & \Longleftrightarrow & f(w) \equiv_{Q_2} w' \text{ or } w \equiv_{Q_2} w' \\
w \equiv_{Q_1\ ;\ Q_2} w' & \Longleftrightarrow & \exists k, k' \in \mathcal{L}(K(Q_2)) \text{ such that } w \equiv_{Q_1} k,\ w' \equiv_{Q_1} k', \text{ and } k \equiv_{Q_2} k' \\
w \equiv_{Q_1 \cdot Q_2} w' & \Longleftrightarrow & w = r_1 \cdot r_2,\ w' = r'_1 \cdot r'_2 \text{ with } r_1 \equiv_{Q_1} r'_1,\ r_2 \equiv Q_n\, r'_2 \\
w \equiv_{Q_1 \mid Q_2} w' & \Longleftrightarrow & w \equiv_{Q_1} w' \text{ or } w \equiv_{Q_2} w' \\
w \equiv_{Q^*} w' & \Longleftrightarrow & w = r_1 \cdot \ldots \cdot r_n,\ w' = r'_1 \cdot \ldots \cdot r'_n \text{ and } r_i \equiv_Q r'_i \\
w \equiv_{\mathsf{perm}(Q_1, \ldots, Q_n)\ \mathsf{with}\ Q} w' & \Longleftrightarrow & w = r_{\sigma(1)} \cdot s_1 \cdot \ldots \cdot s_{n-1} \cdot r_{\sigma(n)},\ w' = r'_{\theta(1)} \cdot s'_1 \cdot \ldots \cdot s'_{n-1} \cdot r'_{\theta(n)} \\
 & & \text{for some } \sigma, \theta \in S_n,\ \text{with } r_i \equiv_{q_i} r'_i \text{ and } s_k \equiv_Q s'_k
\end{array}
$$

Fig. 6. QRE Equivalence Relations

which places the strings that match Q_i and Q according to the canonical permutation (i.e., the identity permutation) before applying the canonizers $\mathsf{canonize}(Q_i)$ and $\mathsf{canonize}(Q)$, respectively.

Finally, Figure 6 gives the inductive definition of the equivalence relation $\equiv_Q$, which is the set-theoretic semantics of a QRE Q as an equivalence relation on the regular language $W(Q)$.

4.4 Ambiguity and Well-Formed QREs

To ensure that regular combinations of QREs are well-formed, we need to enforce a variety of *unambiguity* constraints. Specifically, when applying regular combinators to QREs Q_1 and Q_2, we require that if a string s matches any of the regular expressions $W(Q_1) \cdot W(Q_2)$, $W(Q_1) \mid W(Q_2)$, $W(Q_1)^*$, $K(Q_1) \cdot K(Q_2)$, $K(Q_1) \mid K(Q_2)$, $K(Q_1)^*$, then s matches that regular expression in only one way. Formally, we say that regular expressions R and S are *unambiguosly concatenable*, written $R \cdot^! S$ if for all strings $r, r' \in \mathcal{L}(R)$ and $s, s' \in \mathcal{L}(S)$, if $r \cdot s = r' \cdot s'$, then $r = r'$ and $s = s'$. We say that a regular expression R is *unambiguosly iterable*, written $R^{*!}$ if for all strings $r_1, \ldots, r_m$

$$\frac{R \text{ is strongly unambiguous}}{\mathsf{id}(R)\ \mathsf{wf}}\ (\text{Id}) \qquad \frac{s \in \mathcal{L}(R)}{\mathsf{collapse}\ R \mapsto s\ \mathsf{wf}}\ (\text{Collapse})$$

$$\frac{Q_i, Q\ \mathsf{wf} \qquad \begin{matrix}\forall \sigma{\neq}\theta,\ W(Q_{\sigma(1)}){\cdot}W(Q){\cdot}\ldots{\cdot}W(Q_{\sigma(n)}) \\ \cap W(Q_{\theta(1)}){\cdot}W(Q){\cdot}\ldots{\cdot}W(Q_{\theta(n)}){=}\varnothing\end{matrix} \qquad K(Q_1) \cdot^! K(Q) \cdot^! \ldots \cdot^! K(Q) \cdot^! K(Q_n)}{\mathsf{perm}(Q_1, \ldots, Q_n)\ \mathsf{with}\ Q\ \mathsf{wf}}\ (\text{Perm})$$

$$\frac{Q_1, Q_2\ \mathsf{wf} \qquad \mathcal{L}(W(Q_1)) \cap \mathcal{L}(W(Q_2)) = \varnothing \qquad f : \mathcal{L}(W(Q_1)) \longrightarrow \mathcal{L}(W(Q_2))}{\mathsf{squash}\ Q_1 \rightarrow Q_2\ \mathsf{using}\ f\ \mathsf{wf}}\ (\text{Squash})$$

$$\frac{\mathcal{L}(R') \subseteq \mathcal{L}(R) \qquad f : \mathcal{L}(R) \longrightarrow \mathcal{L}(R') \qquad f \text{ is surjective} \qquad f = f^2}{\mathsf{normalize}(R, R', f)\ \mathsf{wf}}\ (\text{Normalize})$$

$$\frac{Q_1, Q_2\ \mathsf{wf} \qquad K(Q_1) = W(Q_2)}{Q_1\ ;\ Q_2\ \mathsf{wf}}\ (\text{Compose}) \qquad \frac{Q\ \mathsf{wf} \qquad W(Q)^{*!} \qquad K(Q)^{*!}}{Q^*\ \mathsf{wf}}\ (\text{Star})$$

$$\frac{Q_1, Q_2\ \mathsf{wf} \qquad W(Q_1) \cdot^! W(Q_2) \qquad K(Q_1) \cdot^! K(Q_2)}{Q_1 \cdot Q_2\ \mathsf{wf}}\ (\text{Concat})$$

$$\frac{Q_1, Q_2\ \mathsf{wf} \qquad W(Q_1) \cap W(Q_2) = \varnothing}{Q_1 \mid Q_2\ \mathsf{wf}}\ (\text{Union})$$

Fig. 7. Well-formed QREs

and $r'_1, \ldots, r'_n \in \mathcal{L}(R)$, if $r_1 \cdot \ldots \cdot r_m = r'_1 \cdot \ldots \cdot r'_n$, then $m = n$ and $r_i = r'_i$. We say that a regular expression R is *strongly unambiguous* [Sippu and Soisalon-Soininen 1988] if and only if (1) $R = \varnothing$, or (2) $R = S_1 \cdot S_2$ with S_1, S_2 strongly unambiguous and $S_1 \cdot^! S_n$, or (3) $R = S_1 \mid S_2$ with S_1, S_2 strongly unambiguous and $\mathcal{L}(S_1) \cap \mathcal{L}(S_2) = \varnothing$, or (4) $R = S^*$ with S strongly unambiguous and $S^{*!}$.

These unambiguity constraints can be a little fiddly in practice. As a simple example of this, consider writing a regular expression for comma-separated lists of strings. Our first impulse might be to write it as,

```
CSL = anychar+ . ("," . anychar+)*
```

(where . is concatenation), but this regular expression is ambiguous, as anychar is a big union of a of single characters including comma.

While the unambiguity constraints consequently appear to compromise compositionality of QREs, they can usually be circumvented by making a small changes to the offending regular expression: for instance, in the preceding example, then we need to write,

```
CSL = anycharexceptcomma+ . ("," . anycharexceptcomma+)*
```

(assuming anycharexceptcomma denotes a big union of every single character string except comma).

Unambiguity constraints are necessary because they ensure that the canonizing function of a QRE is well-defined. For example, consider the QRE "a"* . (collapse "a"* → "a"). The behavior of this QRE is not well-defined since the string "aaa" can be canonized to any of "a", "aa", or "aaa" depending on how "aaa" is parsed. The unambiguity constraints are also applied to the kernels of QREs since the underlying bijective string lens of a QRE lens operates on kernels, and bijective string lenses impose the same unambiguity restrictions so that they too are well defined as functions.

4.5 Well-formedness of QREs

Figure 7 gives the inference rules for deriving well-formed QREs. The unambiguity conditions are pertinent when defining QREs using the regular combinators. For example, the (Concat) inference rule says that the concatenation $Q_1 \cdot Q_2$ of QREs Q_1 and Q_2 is well formed only if the concatentions of $W(Q_1)$ and $W(Q_2)$, and $K(Q_1)$ and $K(Q_2)$ are unambiguous.

The most complicated inference rule is the (Perm) rule for the permutation combinator. The second hypothesis for the Perm rule says that for any two different permutations σ and θ, the languages $W(Q_{\sigma(1)}) \cdot W(Q) \cdot \ldots \cdot W(Q_{\sigma(n)})$ and $W(Q_{\theta(1)}) \cdot W(Q) \cdot \ldots \cdot W(Q_{\theta(n)})$ must be disjoint. This restriction is important because an input string could match any of the regular expressions $W(Q_{\theta(1)}) \cdot W(Q) \cdot \ldots \cdot W(Q_{\theta(n)})$ for some permutation θ, so we must require that all of them be disjoint. The third hypothesis says that the regular expression $K(Q_1) \cdot K(Q) \cdot \ldots \cdot K(Q) \cdot K(Q_n)$ must be unambiguous. This restriction arises because the underlying lens that maps to or from a perm QRE operates on the language $K(Q_1) \cdot K(Q) \cdot \ldots \cdot K(Q) \cdot K(Q_n)$, the kernel of the perm QRE. The underlying lens requires that the source and target regular expressions be unambiguous so that the lens can match strings uniquely.

5 QRE LENSES

As we have seen, QREs express a broad class of equivalence relations directly on regular languages. QREs are therefore a good *specification* language for quotient lenses. In this section we introduce *QRE Lenses*, a class of quotient lenses that map between data that is specified using QREs.

Recall that given regular expressions R, S and equivalence relations $\equiv_R$ and $\equiv_S$ defined on $\mathcal{L}(R)$ and $\mathcal{L}(S)$, a *bijective quotient lens* $q : R/\equiv_R \Leftrightarrow S/\equiv_S$ is a pair of functions $q.\text{get} : \mathcal{L}(R) \longrightarrow \mathcal{L}(S)$ and $q.\text{put} : \mathcal{L}(S) \longrightarrow \mathcal{L}(R)$ such that for all $r \in \mathcal{L}(R)$ and $s \in \mathcal{L}(S)$, we have $q.\text{put}(q.\text{get}(r)) \equiv_R r$ and $q.\text{get}(q.\text{put}(s)) \equiv_S s$. Moreover, if $r \equiv_R r'$, then $q.\text{get}(r) = q.\text{get}(r')$, and if $s \equiv_S s'$, then $q.\text{put}(s) = q.\text{put}(s')$. In words, a bijective quotient lens q from R to S modulo $\equiv_R$ and $\equiv_S$ is a pair of functions $q.\text{get}$ and $q.\text{put}$ such that $q.\text{get}$ respects $\equiv_R$ and $q.\text{put}$ respects $\equiv_S$, and such that the *get* and *put* functions lifted to $\mathcal{L}(R)/\equiv_R$ and $\mathcal{L}(S)/\equiv_S$ are mutual inverses. These laws are similar to the bijective lens laws Eq. (1), except that the equality restrictions in the bijective lens laws are loosened to allow for equivalence relations. Also, the condition that $r \equiv_R r'$ implies $q.\text{get}(r) = q.\text{get}(r')$ ensures that the get function induced on the equivalence classes of $\equiv_R$ is well-defined, and similarly for the put function.

Having identified QREs as a natural way of specifing equivalence relations on regular expressions, a natural next step in defining quotient lenses is to map between two QREs via a bijection between their kernels; indeed, this approach is the one we adopt in defining *QRE lenses*. More concretely, to define a language of QRE lenses, we (1) define a language of bijections (Section 5.1), (2) add quotients by allowing canonizers to be prepended or postpended to bijections, and (3) allow composition of such quotient lenses via the regular operators and functional composition (Sections 5.2 and 5.3).

However, we also have a secondary objective, which is to support lens synthesis. When provided with QREs describing the source and target languages, we would like to be able to generate quotient lenses automatically. One way to achieve that goal is to generate canonizers canonize(Q_1) and canonize(Q_2) from QREs Q_1 and Q_2 and then to synthesize bijective lenses between the kernel languages for Q_1 and Q_2. Unfortunately, the composition of two such quotient lenses does not have the form of a bijective lens with canonizers at the edges. Hence, an important technical question is whether we give up expressiveness if we restrict ourselves to this form. Fortunately, we can show that there is no loss of expressiveness if the bijections used to define QRE lenses are derived with a particular set of combinators. We describe these combinators next.

5.1 Bijective String Lenses

We define the set of *bijective string lenses* to be the set of bijections between regular languages constructed using the following combinators,

$$\ell := \mathsf{id}(R) \mid \mathsf{const}(s,t) \mid \mathsf{swap}(\ell_1,\ell_2) \mid \ell_1 \cdot \ell_2 \mid (\ell_1 \mid \ell_2) \mid \ell^* \mid \ell_1 \ ; \ \ell_2$$

where R ranges over regular expressions and s ranges over character strings.

The $\mathsf{const}(s,t)$ lens replaces the string s with t in the source data in the forward direction, and t with s in the target data in the backward direction, while $\mathsf{id}(R)$ applies the identity function to the source and target in $\mathcal{L}(R)$ in both directions. The composition lens $\ell_1 \ ; \ \ell_2$ applies ℓ_1 followed by then ℓ_2 to the source in the forward direction, and applies ℓ_2 followed by ℓ_1 to the target in the backward direction. The lens $\ell_1 \cdot \ell_2$ first splits the string s into s_1 and s_2, applies ℓ_1 and ℓ_2 to s_1 and s_2 to get t_1 and t_2 respectively, then concatenates t_1 and t_2 and returns $t_1 \cdot t_2$ as the final result in the forward direction. $\ell_1 \cdot \ell_2$ operates similarly in the backward direction, but with s, s_1 and s_2 substituted for t, t_1 and t_2. The $\mathsf{swap}(\ell_1,\ell_2)$ lens operates like $\ell_1 \cdot \ell_2$, except that it swaps t_1 and t_2 before concatenating the two for a final result of $t_2 \cdot t_1$ in the forward direction. In the backward direction, $\mathsf{swap}(\ell_1,\ell_2)$ first undoes the swap, then behaves like the concatenation lens. The $\ell_1 \mid \ell_2$ lens chooses to apply ℓ_1 or ℓ_2 depending on whether the source (resp. target) data is matched by ℓ_1 or ℓ_2 in the forward (resp. backward direction). The ℓ^* lens splits the string s into strings $s_1, \ldots, s_n$, applies ℓ_1 to each s_i to obtain t_i, and then concatenates each of the t_i's for a final result of $t_1 \cdot \ldots \cdot t_n$ in the forward direction. The iteration lens ℓ^* operates similarly in the backward direction, but with s and s_i substituted for t and t_i.

The denotation of a lens ℓ_1 is $[\![\ell_1]\!] \subseteq \mathit{String} \times \mathit{String}$. If $(s_1,s_2) \in [\![\ell_1]\!]$, then ℓ_1 maps between s_1 and s_2.

$$\begin{aligned}
[\![\mathsf{id}(R)]\!] &= \{(r,r) \mid r \in \mathcal{L}(R)\} \\
[\![\mathsf{const}(r,s)]\!] &= \{(r,s)\} \\
[\![\mathsf{swap}(\ell_1,\ell_2)]\!] &= \{(s \cdot t, t' \cdot s') \mid (s,s') \in [\![\ell_1]\!] \text{ and } (t,t') \in [\![\ell_2]\!]\} \\
[\![\ell_1 \cdot \ell_2]\!] &= \{(s \cdot t, s' \cdot t') \mid (s,s') \in [\![\ell_1]\!] \text{ and } (t,t') \in [\![\ell_2]\!]\} \\
[\![\ell_1 \mid \ell_2]\!] &= \{(s \cdot t) \mid (s,t) \in [\![\ell_1]\!] \text{ or } (s,t) \in [\![\ell_2]\!]\} \\
[\![\ell^*]\!] &= \{(s_1 \cdot \ldots \cdot s_n, t_1 \cdot \ldots \cdot t_n) \mid (s_i,t_i) \in [\![\ell]\!] \text{ for } 1 \leq i \leq n\} \\
[\![\ell_1 \ ; \ \ell_2]\!] &= \{(r,t) \mid \text{there exists } s \text{ with } (r,s) \in [\![\ell_1]\!] \text{ and } (s,t) \in [\![\ell_2]\!]\}
\end{aligned}$$

Each bijective string lens ℓ has a type $\ell : R \Leftrightarrow S$ where R and S are regular expressions. If $\ell : R \Leftrightarrow S$, then the source language of ℓ is $\mathcal{L}(R)$ and the target language of ℓ is $\mathcal{L}(S)$. The final rule in the figure deserves special attention. It states that a lens $\ell : R \Leftrightarrow S$ can also be considered to be of type $\ell : R' \Leftrightarrow S'$ provided that R (resp. S) can be proven to be equivalent to R' (resp. S) from the star-semiring axioms (associativity and commutativity of $\mid$ and $\cdot$, identity of the empty string ϵ, distibutivity of $\cdot$ over $\mid$, annihilative property of $\varnothing$ with respect to $\cdot$, and that $R^* = \epsilon \mid R{\cdot}R^* = \epsilon \mid R^*{\cdot}R$ for all R):

$$\frac{\ell_1 : R \Leftrightarrow S \qquad R \equiv^s R' \qquad S \equiv^s S'}{\ell_1 : R' \Leftrightarrow S'}$$

This rule makes it significantly more difficult to synthesize a bijective lens from its type.

5.2 Syntax of QRE Lenses

Having given a brief overview of the class of bijective string lenses, we now introduce the class of QRE lenses. The language of QRE lenses is given by following grammar,

$$q := \mathsf{lift}(\ell) \mid q_1 \cdot q_2 \mid \mathsf{swap}(q_1,q_2) \mid (q_1 \mid q_2) \mid q^* \mid q_1 \ ; \ q_2 \mid \mathsf{lquot}(q,Q) \mid \mathsf{rquot}(Q,q)$$

$$\frac{s_1 \in \Sigma^* \qquad s_2 \in \Sigma^*}{\text{const}\ (s_1, s_t) : s_1 \Leftrightarrow s_2} \qquad \frac{R \text{ is strongly unambiguous}}{\text{id}(R) : R \Leftrightarrow R}$$

$$\frac{\begin{matrix}\ell_1{:}R_1{\Leftrightarrow}S_1 \\ \ell_2{:}R_2{\Leftrightarrow}S_2\end{matrix} \qquad \begin{matrix}R_1 \cdot^! R_2 \\ S_1 \cdot^! S_2\end{matrix}}{\ell_1 \cdot \ell_2 : R_1 \cdot R_2 \Leftrightarrow S_1 \cdot S_2} \qquad \frac{\begin{matrix}\ell_1{:}R_1{\Leftrightarrow}S_1 \\ \ell_2{:}R_2{\Leftrightarrow}S_2\end{matrix} \qquad \begin{matrix}R_1 \cdot^! R_2 \\ S_2 \cdot^! S_1\end{matrix}}{\text{swap}\ (\ell_1, \ell_2) : R_1 \cdot R_2 \Leftrightarrow S_2 \cdot S_1}$$

$$\frac{\ell_1 : R_1 \Leftrightarrow R_2 \qquad \ell_2 : R_2 \Leftrightarrow R_3}{\ell_1 \ ; \ \ell_2 : R_1 \Leftrightarrow R_3} \qquad \frac{\begin{matrix}\ell_1{:}R_1{\Leftrightarrow}S_1 \\ \ell_2{:}R_2{\Leftrightarrow}S_2\end{matrix} \qquad \begin{matrix}\mathcal{L}(R_1)\cap\mathcal{L}(R_2)=\varnothing \\ \mathcal{L}(S_1)\cap\mathcal{L}(S_2)=\varnothing\end{matrix}}{\ell_1 \mid \ell_2 : R_1 \mid R_2 \Leftrightarrow S_1 \mid S_2}$$

$$\frac{\ell : R \Leftrightarrow S \qquad R^{*!} \qquad S^{*!}}{\ell^* : R \Leftrightarrow S} \qquad \frac{\ell : R \Leftrightarrow S \qquad R \equiv^s R' \qquad S \equiv^s S'}{\ell : R' \Leftrightarrow S'}$$

Fig. 8. Bijective String Lens Typing Rules

where Q ranges over QREs.

Other than the lift combinator which allows a bijective lens to be treated as a quotient lens, the QRE lens combinators are the same as the bijective string lens combinators but with two extra combinators where quotienting actually occurs: the lquot and rquot combinators. The lquot(Q, q) combinator takes a quotient lens q and a QRE Q and quotients the source data using Q, assuming that the source data forms a complete set of representatives for the equivalence relation $\equiv_Q$. The rquot(q, Q) combinator does the same but on the target data.

For example, recall that in the BibTeX to EndNote transformation, we had the QREs bibtex and endnote and the bijective lens bib_to_end that maps between bibtex and endnote. The quotient lens that maps between bibtex and endnote is then given by the following QRE lens:

let bib_to_end_q : bibtex ⇔ endnote = rquot (lquot(bibtex, bib_to_end), endnote)

5.3 Semantics of QRE Lenses

Each QRE lens q has a type $q : Q_1 \Leftrightarrow Q_2$ where Q_1, Q_2 are QREs. If $q : Q_1 \Leftrightarrow Q_2$, then the source format is described by $W(Q_1)$ and the canonical set of representative for the source data is described by $K(Q_1)$. Similarly, the target format is described by $W(Q_2)$ and the canonical set of representatives for the target data is described by $K(Q_2)$. The underlying lens of q is a bijective lens $\ell : K(Q_1) \Leftrightarrow K(Q_2)$.

The denotation $[\![q]\!]$ of a QRE lens $q : Q_1 \Leftrightarrow Q_2$ is a quotient lens $[\![q]\!] : W(Q_1)/{\equiv_{Q_1}} \Longleftrightarrow W(Q_2)/{\equiv_{Q_2}}$. The typing rules and denotation of QRE lenses are given in Figure 9. The trickiest typing rule is the typing rule for composition:

$$\frac{\begin{matrix}q_1{:}Q_1{\Leftrightarrow}Q_2 \\ q_2{:}Q_3{\Leftrightarrow}Q_4\end{matrix} \qquad \begin{matrix}\mathcal{L}(W(Q_2))=\mathcal{L}(W(Q_3)) \\ K(Q_2)\equiv^s K(Q_3)\end{matrix} \qquad \text{canonize}(Q_2) = \text{canonize}(Q_3)}{q_1 \ ; \ q_2 : Q_1 \Leftrightarrow Q_4}$$

This rule essentially says that the composition $q_1 \ ; \ q_2$ is well defined if and only if the intermediary QREs Q_3 and Q_4 define the same equivalence relation on regular expressions that are equivalent modulo the star-semiring axioms. (See [Foster et al. 2008, §4] for an example of what goes wrong if this premise is dropped). The condition $\mathcal{L}(W(Q_2)) = \mathcal{L}(W(Q_3))$ says that the intermediary language is the same on both sides, while the condition $K(Q_2) \equiv^s K(Q_3)$ says that the kernel regular expressions are equivalent modulo the star-semiring axioms. Finally, the condition canonize(Q_2) = canonize(Q_3) says that Q_2 and Q_3 define the same equivalence relation on $\mathcal{L}(W(Q_2)) = \mathcal{L}(W(Q_3))$.

Of course checking the condition canonize(Q_2) = canonize(Q_3) is undecidable in general; indeed the approach used to type quotient lenses in Boomerang is to classify equivalences according to

$$\frac{\ell : R \Leftrightarrow S}{\mathsf{lift}(\ell) : id(R) \Leftrightarrow id(S)}\;(Lift) \qquad \begin{array}{l}\llbracket \mathsf{lift}(\ell) \rrbracket.\mathsf{get} = \llbracket \ell \rrbracket \\ \llbracket \mathsf{lift}(\ell) \rrbracket.\mathsf{put} = \llbracket \ell \rrbracket^{-1}\end{array}$$

$$\frac{q : Q_2 \Leftrightarrow Q_3 \quad Q_1 \text{ wf} \quad K(Q_1) = W(Q_2)}{\mathsf{lquot}(Q_1, q) : Q_1 \,;\, Q_2 \Leftrightarrow Q_3}\;(Lquot) \qquad \begin{array}{l}\llbracket \mathsf{lquot}(Q_1, q) \rrbracket.\mathsf{get} = \llbracket q \rrbracket.\mathsf{get} \circ \mathsf{canonize}(Q_1) \\ \llbracket \mathsf{lquot}(Q_1, q) \rrbracket.\mathsf{put} = \llbracket q \rrbracket.\mathsf{put}\end{array}$$

$$\frac{q : Q_1 \Leftrightarrow Q_3 \quad Q_3 \text{ wf} \quad W(Q_3) = K(Q_2)}{\mathsf{rquot}(q, Q_1) : Q_1 \Leftrightarrow Q_2 \,;\, Q_3}\;(Rquot) \qquad \begin{array}{l}\llbracket \mathsf{rquot}(q, Q_1) \rrbracket.\mathsf{get} = \llbracket q \rrbracket.\mathsf{get} \\ \llbracket \mathsf{rquot}(q, Q_1) \rrbracket.\mathsf{put} = \llbracket q \rrbracket.\mathsf{put} \circ \mathsf{canonize}(Q_3)\end{array}$$

$$\frac{q : Q_1 \Leftrightarrow Q_2 \quad W(Q_1)^{*!}, W(Q_2)^{*!} \quad K(Q_1)^{*!}, K(Q_2)^{*!}}{q^* : {Q_1}^* \Leftrightarrow {Q_2}^*}\;(Star) \qquad \begin{array}{l}\llbracket q^* \rrbracket.\mathsf{get} = (\llbracket q \rrbracket.\mathsf{get})^* \\ \llbracket q^* \rrbracket.\mathsf{put} = (\llbracket q \rrbracket.\mathsf{get})^*\end{array}$$

$$\frac{\begin{array}{ccc} q_1{:}Q_1{\Leftrightarrow}Q_3 & W(Q_1)\cdot^{!}W(Q_2) & W(Q_3)\cdot^{!}W(Q_4) \\ q_2{:}Q_2{\Leftrightarrow}Q_4 & K(Q_1)\cdot^{!}K(Q_2) & K(Q_3)\cdot^{!}K(Q_4)\end{array}}{q_1 \cdot q_2 : Q_1 \cdot Q_2 \Leftrightarrow Q_3 \cdot Q_4}\;(Concat) \qquad \begin{array}{l}\llbracket q_1 \cdot q_2 \rrbracket.\mathsf{get} = \llbracket q_1 \rrbracket.\mathsf{get} \cdot \llbracket q_2 \rrbracket.\mathsf{get} \\ \llbracket q_1 \cdot q_2 \rrbracket.\mathsf{put} = \llbracket q_1 \rrbracket.\mathsf{put} \cdot \llbracket q_2 \rrbracket.\mathsf{put}\end{array}$$

$$\frac{\begin{array}{ccc} q_1{:}Q_1{\Leftrightarrow}Q_3 & W(Q_1)\cdot^{!}W(Q_2) & W(Q_4)\cdot^{!}W(Q_3) \\ q_2{:}Q_2{\Leftrightarrow}Q_4 & K(Q_1)\cdot^{!}K(Q_2) & K(Q_4)\cdot^{!}K(Q_3)\end{array}}{q = q_1 \cdot q_2 : Q_1 \cdot Q_2 \Leftrightarrow Q_4 \cdot Q_3}\;(Swap) \qquad \begin{array}{l}\llbracket q \rrbracket.\mathsf{get}(s_1 \cdot s_2) = \llbracket q_2 \rrbracket.\mathsf{get}(s_2) \cdot \llbracket q_1 \rrbracket.\mathsf{get}(s_1) \\ \llbracket q \rrbracket.\mathsf{put}(t_2 \cdot t_1) = \llbracket q_1 \rrbracket.\mathsf{put}(t_1) \cdot \llbracket q_2 \rrbracket.\mathsf{put}(t_2)\end{array}$$

$$\frac{\begin{array}{cc} q_1{:}Q_1{\Leftrightarrow}Q_3 & \mathcal{L}(W(Q_1)){\cap}\mathcal{L}(W(Q_2)){=}\varnothing \\ q_2{:}Q_2{\Leftrightarrow}Q_4 & \mathcal{L}(W(Q_3)){\cap}\mathcal{L}(W(Q_4)){=}\varnothing\end{array}}{q_1 \mid q_2 : (Q_1 \mid Q_2) \Leftrightarrow (Q_3 \mid Q_4)}\;(Or) \qquad \begin{array}{l}\llbracket q_1 \mid q_2 \rrbracket.\mathsf{get}(s) = \begin{cases}\llbracket q_1 \rrbracket.\mathsf{get}(s) & \text{if } s \in \mathcal{L}(W(Q_1)) \\ \llbracket q_2 \rrbracket.\mathsf{get}(s) & \text{if } s \in \mathcal{L}(W(Q_2))\end{cases} \\ \llbracket q_1 \mid q_2 \rrbracket.\mathsf{put}(s) = \begin{cases}\llbracket q_1 \rrbracket.\mathsf{put}(s) & \text{if } s \in \mathcal{L}(W(Q_3)) \\ \llbracket q_2 \rrbracket.\mathsf{put}(s) & \text{if } s \in \mathcal{L}(W(Q_4))\end{cases}\end{array}$$

$$\frac{\begin{array}{ccc} q_1{:}Q_1{\Leftrightarrow}Q_2 & \mathcal{L}(W(Q_2)){=}\mathcal{L}(W(Q_3)) & \\ q_2{:}Q_3{\Leftrightarrow}Q_4 & K(Q_2){\equiv^s}K(Q_3) & \end{array} \quad \mathsf{canonize}(Q_2) = \mathsf{canonize}(Q_3)}{q_1 \,;\, q_2 : Q_1 \Leftrightarrow Q_4}\;(\text{Compose})$$

$$\llbracket q_1 \,;\, q_2 \rrbracket.\mathsf{get} = \llbracket q_2 \rrbracket.\mathsf{get} \circ \llbracket q_1 \rrbracket.\mathsf{get}$$

$$\llbracket q_1 \,;\, q_2 \rrbracket.\mathsf{put} = \llbracket q_1 \rrbracket.\mathsf{put} \circ \llbracket q_2 \rrbracket.\mathsf{put}$$

Fig. 9. Denotation and Typing Rules for QRE Lenses

whether they are or are not the equality relation. While this decision seemingly restricts the power of composition in Boomerang significantly, the practice of writing quotient lenses shows that this restriction is not overly restrictive. This is because most quotient lenses originate as lifted basic lenses, and therefore have types whose equivalence relations are both equality, and further, equality is preserved by many of the quotient lens combinators [Foster et al. 2008]. Foster et al also discuss a second possible approach to typing quotient lenses, where equivalence relations are represented by rational functions that induce them. While this second approach is more refined than the first, Boomerang favours the first approach since the second appears to be too expensive to be useful in practice.

Thankfully, we do not face the issue of checking equivalence relation equality in our work since our end goal is to synthesize lenses and not write them my hand, so the programmer will never actually need to typecheck a functional composition expression.

The semantics defined on QRE lenses imply the following theorem:

Theorem 5.1. *If there is a derivation $q : Q_1 \Leftrightarrow Q_2$, then $\llbracket q \rrbracket : W(Q_1)/{\equiv_{Q_1}} \Leftrightarrow W(Q_2)/{\equiv_{Q_2}}$ is a well-defined quotient lens.*

5.4 Normal Forms of QRE Lenses

Recall that our approach in defining QRE lenses is to have each QRE lens $q : Q_1 \Leftrightarrow Q_2$ be such that

$$\llbracket q \rrbracket.\mathsf{get} = \ell \circ \mathsf{canonize}(Q_1)$$
$$\llbracket q \rrbracket.\mathsf{put} = \ell^{-1} \circ \mathsf{canonize}(Q_2)$$

for some bijective lens ℓ. In other words, each QRE lens is the same as a bijective lens with canonizers at the edges. The following theorem, which is the main technical contribution of this paper, confirms that this indeed is the case:

Theorem 5.2. *If there is a derivation $q : Q_1 \Leftrightarrow Q_2$, then there exists a bijective lens $\ell : K(Q_1) \Leftrightarrow K(Q_2)$ such that:*

$$\llbracket q \rrbracket.\mathsf{get} = \llbracket \ell \rrbracket \circ \mathsf{canonize}(Q_1)$$
$$\llbracket q \rrbracket.\mathsf{put} = \llbracket \ell \rrbracket^{-1} \circ \mathsf{canonize}(Q_2)$$

Proof. The proof follows by induction on the derivation of $q : c \Leftrightarrow c$. The most interesting part of the proof is the case for functional composition as we must demonstrate that it is possible to eliminate the canonizers in the middle of the term.

The derivation rule and denotation for composition are as follows:

$$\frac{\begin{array}{c} q_1{:}Q_1{\Leftrightarrow}Q_2 \\ q_2{:}Q_3{\Leftrightarrow}Q_4 \end{array} \qquad \begin{array}{c} \mathcal{L}(W(Q_2)){=}\mathcal{L}(W(Q_3)) \\ K(Q_2){\equiv}K(Q_3) \end{array} \qquad \mathsf{canonize}(Q_2) = \mathsf{canonize}(Q_3)}{q_1 \ ; \ q_2 : Q_1 \Leftrightarrow Q_4}$$

$$\llbracket q_1 \ ; \ q_2 \rrbracket.\mathsf{get} = \llbracket q_2 \rrbracket.\mathsf{get} \circ \llbracket q_1 \rrbracket.\mathsf{get}$$
$$\llbracket q_1 \ ; \ q_2 \rrbracket.\mathsf{put} = \llbracket q_1 \rrbracket.\mathsf{put} \circ \llbracket q_2 \rrbracket.\mathsf{put}$$

By the induction hypothesis, there exist bijective lenses, $\ell_1 : K(Q_1) \Leftrightarrow K(Q_2)$ and $\ell_2 : K(Q_3) \Leftrightarrow K(Q_4)$ such that:

$$\begin{array}{ll} \llbracket q_1 \rrbracket.\mathsf{get} = \llbracket \ell_1 \rrbracket \circ \mathsf{canonize}(Q_1) & \llbracket q_2 \rrbracket.\mathsf{get} = \llbracket \ell_2 \rrbracket \circ \mathsf{canonize}(Q_3) \\ \llbracket q_1 \rrbracket.\mathsf{put} = \llbracket \ell_1 \rrbracket^{-1} \circ \mathsf{canonize}(Q_2) & \llbracket q_2 \rrbracket.\mathsf{put} = \llbracket \ell_2 \rrbracket^{-1} \circ \mathsf{canonize}(Q_4) \end{array}$$

Consequently:

$$\begin{aligned} \llbracket q_2 \rrbracket.\mathsf{get} \circ \llbracket q_1 \rrbracket.\mathsf{get} &= (\llbracket \ell_2 \rrbracket \circ \mathsf{canonize}(Q_3)) \circ (\llbracket \ell_1 \rrbracket \circ \mathsf{canonize}(Q_1)) \\ &= \llbracket \ell_2 \rrbracket \circ (\mathsf{canonize}(Q_3) \circ \llbracket \ell_1 \rrbracket) \circ \mathsf{canonize}(Q_1) \\ &= (\llbracket \ell_2 \rrbracket \circ \llbracket \ell_1 \rrbracket) \circ \mathsf{canonize}(Q_1) \\ &= \llbracket \ell_1 \ ; \ \ell_2 \rrbracket \circ \mathsf{canonize}(Q_1) \end{aligned}$$

We are permitted to claim the third step from the second since $\mathsf{canonize}(Q_3)$ is the identity function on $K(Q_2)$ which is syntactically equal to $K(Q_3)$ by assumption. A similar argument shows that:

$$\llbracket q_1 \rrbracket.\mathsf{put} \circ \llbracket q_2 \rrbracket.\mathsf{put} = \llbracket \ell_1 \ ; \ \ell_2 \rrbracket^{-1} \circ \mathsf{canonize}(Q_4)$$

The other cases of the proof are similar, proceeding by a straightforward application of the induction hypothesis followed by unrolling the equations that give the denotation for QRE lenses. Full details can be found in the appendix. □

6 SYNTHESIZING QRE LENSES

QRE lenses address some of the limitations of bijective lenses because a single lens program expresses both the canonizers and the transformation between kernel languages simultaneously, which reduces programmer effort. But we can go even further by recognizing that the type structure of QRE lenses contains information that can be exploited to automatically synthesize lenses from their types. Rather than writing the QRE lens manually, the programmer can instead specify the desired behavior of a lens by giving its interface types and providing examples, if necessary, to disambiguate among possible implementations. This way of constructing lenses can often be much simpler than building them by hand, as we saw in Section 3.2.

The Optician framework [Miltner et al. 2017] showed how to do such lens synthesis in the case for bijective lenses. Here we show how to reduce QRE lens synthesis to that case, so that we can re-use the Optician algorithm but in the more expressive context of QRE lenses. The basic idea is straightforward: we run the Optician algorithm to synthesize a lens between the kernels of two QREs and then apply the canonizers at the edges to recover a lens between the whole languages. This simple strategy turns out to be remarkably effective, and the idea of using Optician in this way inspired the design of our QRE lenses.

The Optician bijective lens synthesis algorithm works as follows. Given regular expressions R and S and a set of example input–output pairs $\{(r_1, s_1), \ldots, (r_n, s_n)\}$, Optician will try to find a bijective lens $\ell : R \Leftrightarrow S$ that agrees on the examples, *i.e.*, it maps r_i to s_i in the forward direction and s_i to r_i in the backward direction. The bijective lens synthesis algorithm is guaranteed to succeed (eventually!) if such a bijective lens exists. There are typically a large number of such lenses, so this algorithm chooses the one that corresponds to a minimal *alignment* of the regular expressions. Additional details on this algorithm can be found in the original paper on synthesizing bijective lenses [Miltner et al. 2017].

In our setting, we want to synthesize a quotient lens $q : Q_1 \Leftrightarrow Q_2$ from the QREs Q_1 and Q_2 and a set of example input-output pairs $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ where the x_i's are in $W(Q_1)$ and the y_i's are in $W(Q_2)$. We furthermore wish q to map the equivalence class of x_i to the equivalence class of y_i and vice versa:

$$q.\mathsf{get}(x_i) \equiv_{Q_2} y_i, \text{ and}$$
$$q.\mathsf{put}(y_i) \equiv_{Q_1} x_i$$

Our approach to synthesizing QRE lenses is guided by Theorem 5.2, which says that, if there is a derivation $q : Q_1 \Leftrightarrow Q_2$ of a QRE lens, then there exists a bijective lens $\ell : K(Q_1) \Leftrightarrow K(Q_2)$ such that:

$$[\![q]\!].\mathsf{get} = [\![\ell]\!] \circ \mathsf{canonize}(Q_1)$$
$$[\![q]\!].\mathsf{put} = [\![\ell]\!]^{-1} \circ \mathsf{canonize}(Q_2)$$

For the examples, the x_i's are in $W(Q_1)$ and the y_i's are in $W(Q_2)$, so we can construct $x'_i = \mathsf{canonize}(Q_1)(x_i)$ in $K(Q_1)$ and $y'_i = \mathsf{canonize}(Q_2)(y_i)$ is in $K(Q_2)$. To synthesize the desired quotient lens $q : Q_1 \Leftrightarrow Q_2$ that is consistent with the input-output examples $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ it suffices to synthesize a bijective lens $\ell : K(Q_1) \Leftrightarrow K(Q_2)$ that is consistent with the canonized examples $\{(x'_1, y'_1), \ldots, (x'_n, y'_n)\}$ and then apply the canonizers at the outside. Our procedure for QRE lens synthesis is given formally in Algorithm 1.

Algorithm 1 SynthQRELens

```
function SynthQRELens($Q_1, Q_2, exs$)
    $R_1 \leftarrow K(Q_1)$
    $R_2 \leftarrow K(Q_2)$
    $c_1 \leftarrow \text{canonize}(Q_1)$
    $c_2 \leftarrow \text{canonize}(Q_2)$
    $exs' \leftarrow \textsc{Map}(exs, fun\,(ex_l, ex_r) \rightarrow (c_1(ex_l), c_2(ex_r)))$
    $l \leftarrow \textsc{SynthBijectiveLens}(R_1, R_2, exs')$
    return $\text{rquot}(\text{lquot}(Q_1, \ell), Q_2)$
```

Theorem 6.1. *Given QREs Q_1 and Q_2, and a set of examples $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, if there is a QRE lens $q : Q_1 \Leftrightarrow Q_2$ such that $q.\text{get}(x_i) \equiv_{Q_2} y_i$ and $q.\text{put}(y_i) \equiv_{Q_1} x_i$, then SynthQRELens$(Q_1, Q_2, exs)$ will return such a lens.*

(This follows from the correctness of the Optician algorithm and Theorem 5.2.)

Returning to the BibTeX to EndNote transformation of Section 3.2, the QREs bibtex and endnote describe a BibTeX record and an Endnote record respectively. We also had the example pair (bib_example , end_example), where:

```
bib_example =
"@Book␣{Lovelace,
␣Author␣=␣\"Ada␣Lovelace\",
␣Title␣=␣{Generic␣Title},
}"
```

```
end_example =
"%0␣Book
␣%T␣Generic␣Title
␣%A␣Ada␣Lovelace
␣%F␣Lovelace"
```

There exists a bijective lens between the kernel of bibtex and the kernel of endnote that maps the normalized form of bib_example to the normalized form of end_example, so calling SynthQRELens on the QREs bibtex and endnote, with the example set of {(bib_example, end_example)} will return a satisfying lens. In this instance, the example set guides the algorithm to also find the desired lens.

7 IMPLEMENTATION AND EVALUATION

We have implemented QREs and the quotient lens synthesis algorithm described above as an extension to the Boomerang interpreter [Bohannon et al. 2008; Foster et al. 2008]. We have extended the existing Optician tool to synthesize QRE lenses from QREs. We will use "QRE-enhanced Optician" to denote our extended version of Optician, and just plain "Optician" to denote the pre-QRE version of Optician. The synthesis algorithm produces Boomerang lens values, so Boomerang gives synthesized lenses the same first-class status as hand-written ones. To evaluate the effectiveness of QREs, QRE lenses, and QRE lens synthesis, we conducted experiments to answer the following questions:

- **Ease of use.** Does synthesizing QRE lenses from QREs permit an easier development process than writing lenses by hand? Does synthesizing QRE lenses from QREs permit an easier development process than manually writing canonizers and then synthesizing lenses between their canonized forms?
- **Performance.** Is the synthesis algorithm/implementation fast enough to be used as part of a standard development process?

All evaluations were performed on a 2.5 GHz Intel Core i7 processor with 16 GB of 1600 MHz DDR3 running macOS High Sierra.

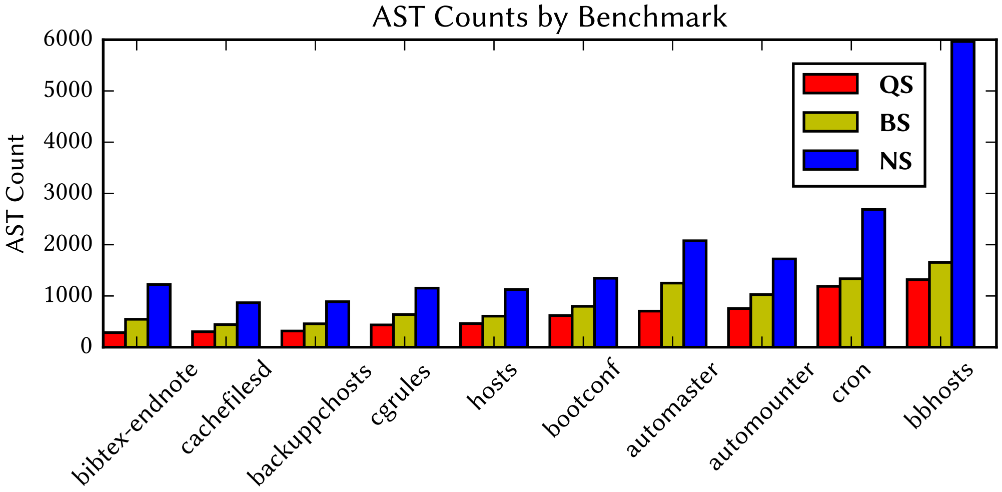


Fig. 10. AST node measurements for each of the three approaches on each of the 10 non-bijective benchmark problems. Benchmarks are sorted in order of increasing complexity as measured by the number of AST nodes in the source and target format descriptions. QRE Synthesis requires far fewer AST nodes than the other two approaches.

7.1 Benchmark Suite Construction

To evaluate our QRE implementation, we adapted 39 lens synthesis tasks from the benchmark suite of the original Optician system. These benchmarks are a combination of custom benchmarks, benchmarks derived from FlashFill [Gulwani 2011], and benchmarks derived from Augeas [Lutterkort 2008] (we also experimented using our QRE implementation to synthesize quotient lenses between XML, RDF and and JSON formats using data from the data.gov database; the data consisted of census statistics, demographic statistics, wage comparosion data, and crime index data). In these 39 benchmarks, 10 of the data formats had to be modified to work with the bijectivity constraints that Optician required. For instance, when one representation permits whitespace where the other does not, we modified the original version of the benchmark to allow more whitespace, thereby restoring bijectivity (but altering the data format). With the new QRE support, we were able to remove these alterations. This experience alone suggests that QREs make the lens development process more flexible.

7.2 Ease of Use

To evaluate the impact of QRE lens synthsis on programmer effort, we focus our attention on the 10 problems in the benchmark suite that are not bijective and hence require non-trivial canonizers. (Optician already handles the other problems with minimal programmer effort.)

We are interested in comparing three different approaches, which vary in the amount of synthesis used. In the first approach, which we call **QS** for QRE Synthesis, the programmer uses QRE lens synthesis. She must write QRE specifications of the source and target formats and she may give examples. In the second approach, which we call **BS** for Bijective Synthesis, the programmer uses bijective lens synthesis à la Optician. She must write canonizers by hand, along with regular expressions to describe the external representations of the source and target formats. (The internal formats can be inferred from the canonizers.) She may also provide examples to help in the synthesis of the bijective lens. In the third approach, which we call **NS** for No Synthesis, the programmer writes the lens between the source and target formats entirely by hand, including the descriptions of the source and target formats.

For each problem in the benchmark suite, we calculate the following measures as proxies for the level of programmer effort when using each the three approaches:

QS: The number of AST nodes in the QRE specifications for the source and target formats, including examples.

BS: The sum of (1) the number of AST nodes in $W(q)$ for each QRE q in the source and target formats, (2) the number of AST nodes in canonize(q) for each QRE q with a non-trivial canonizer, and (3) the number of AST nodes in the examples. We use (1) to estimate the burden of describing the external source and target formats and (2) to estimate the burden of writing the requisite canonizers by hand. We count the nodes in the examples because they would be fed to the bijective synthesizer. These counts are an approximation, as both $W(q)$ and canonize(q) are automatically generated from the corresponding QRE q, and it is possible that a human-written version might be smaller.

NS: The sum of (1) the number of AST nodes in $W(q)$ for each QRE q in the source and target formats and (2) the number of AST nodes in the synthesized QRE lens. We use (1) to estimate the burdern of describing the source and target formats and (2) to estimate the burdern of writing the appropriate lens by hand. These counts are also approximations, as $W(q)$ and the synthesized lens may be larger than one written by hand.

Figure 10 shows each of these measures for the 10 non-bijective problems in the benchmark suite. On average (using a geometric mean), **BS** used 38.5% more AST nodes than **QS**, requiring an average of 214 more AST nodes. On average, **NS** used 180% more AST nodes than **QS**, requiring an average of 998 more AST nodes. These figures suggest that introducing QREs saves programmers significant effort compared to both Optician and basic Boomerang.

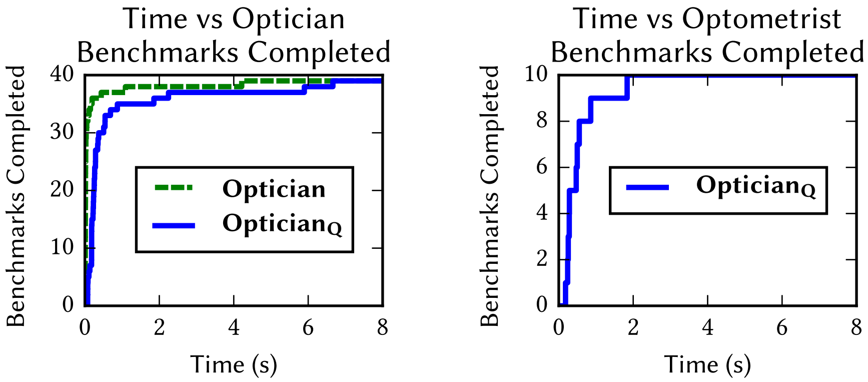
7.3 Maintaining Competitive Performance

To assess the performance of QRE synthesis, we are interested in two different questions. First, how does the performance of QRE-enhanced Optician compare to the performance of Optician on benchmarks that do not require QREs? The answer to this question tells us how much overhead we have introduced by adopting the more general mechanism. Figure 11(a) shows that QRE-enhanced Optician was able to synthesize all of the Optician benchmarks at a speed competitive with the old version. There is a small amount of additional overhead introduced by QREs in calculating the W and K functions, resulting in a slight decrease in performance.

Second, how much time does it take for QRE-enhanced Optician to synthesize a QRE lens when running on a non-bijective benchmark problem? Figure 11(b) shows the amount of time required to infer a lens for each of the 10 benchmark programs with nontrivial quotients. We find that QRE-enhanced Optician is able to synthesize all quotient lenses in under 10 seconds, and typically finishes in under 5 seconds.

8 RELATED WORK

This paper builds on the work of Foster et al [Foster et al. 2008] who introduced the theory of quotient lenses and implemented quotient lenses as a refinement of the bidirectional string processing language Boomerang [Bohannon et al. 2008]. As we mentioned in Section 4.4, all our QRE combinators can be expressed using just the *normalize* combinator, which is one of the canonizer primitives that Boomerang already supports. Also, all our QRE lens combinators



(a) (b)

Fig. 11. Runtimes measurements. In (a), we run Optician and QRE-enhanced Optician on the Optician benchmarks. We find that there is only a negligible performance overhead incurred by using QREs. In (b), we run QRE-enhanced Optician on the 10 Optician benchmarks previously edited to make them bijective, after removing those edits and then extending the synthesis specification to include QREs. (In other words, we restored them to their original state, added QREs, and then ran QRE-enhanced Optician). We find that QRE-enhanced Optician is able to synthesize all quotient lenses in under 10 seconds, and typically finishes in under 5 seconds.

are already supported in Boomerang. Consequently Boomerang quotient lenses are at least as expressive as our language of QRE lenses.

Boomerang's canonizers allow one to canonize a regular language R to by mapping it to another regular language S which may not be contained in S. Formally, given sets C and B and an equivalence relation on B, Foster et al defined a *canonizer* q from C to $B/{\equiv_B}$ to be a pair of functions $q.\mathsf{canonize} : C \longrightarrow B$ and $q.\mathsf{choose} : B \longrightarrow C$ such that for every $b \in B$:

$$q.\mathsf{canonize}\ (q.\mathsf{choose}\ b) \equiv_B\ b$$

This definition gives allows much more latitude for defining canonizers than QREs. For example, if $\equiv_B$ is equal to $\mathsf{Tot}(B)$, the equivalence relation that relates every element in B to every other element in B, then every function from C to B is a canonizer.

Because of this extra elbow-room, Boomerang is able to offer two primitive *duplication* quotient lenses, the first of which can be derived using the following inference rule,

$$\frac{\ell : C/{\equiv_C} \Leftrightarrow A_1/{\equiv_{A_1}} \qquad f : C \longrightarrow A_2 \qquad A_1 \cdot^! A_2 \qquad {\equiv_A} = {\equiv_{A_1}} \cdot \mathsf{Tot}(A_2)}{\mathsf{dup}_1 \ell\ f\ \ :\ C/{\equiv_C} \Longrightarrow A_1 \cdot A_2/{\equiv_A}}$$

$$(\mathsf{dup}_1\ \ell\ f).\mathsf{get}\ c = (\ell.\mathsf{get}\ c) \cdot (f\ c)$$
$$(\mathsf{dup}_1\ \ell\ f).\mathsf{put}\ (a_1 \cdot a_2)\ c = \ell.\mathsf{put}\ a_1\ c$$
$$(\mathsf{dup}_1\ \ell\ f).\mathsf{create}\ (a_1 \cdot a_2) = \ell.\mathsf{create}\ a_1$$

with the symmetric dup_2 combinator discarding the first copy instead of the second in the put/create direction.

Boomerang's more general definition for canonizers also allows quotient lenses to be used as canonizers by using the taking the canonize function to be the get component of a lens and the choose function to be its create component. Naturally, QREs also take advantage of this ability to use lenses as canonizers by allowing for user-defined functions to be used by the squash and normalize combinators. Moreover, the added functionality of synthesizing lenses further makes it easier to define canonizers, as well as lenses.

Foster et al also discuss other bidirectional programming languages that support quotienting of data including XSugar [Brabrand et al. 2008], biXid [Kawanaka and Hosoya 2006] and X/Inv [Hu et al. 2004; Mu et al. 2004; MU et al. 2006]; we refer the reader to the related work section in [Foster et al. 2008] for this discussion.

A newer language which supports quotenting is FliPpr [Matsuda and Wang 2013]. FliPpr is a program transformation system that uses program inversion to produce a CFG parser from a pretty-printer. FliPpr includes biased choice combinators that choose canonical representatives for sets of strings. For example, one can define variants of (white)spaces with the choice operator as,

```
nil = text "" <+ space
space = (text "␣" <+ text "\n") <> nil
```

where nil and space pretty-print "" and "␣" respectively, but represent zero-or-more and one-or-more whitespaces in parsing. FliPpr also allows annotations for optional parentheses in programs. These features enable a user to easily define a pretty-printer that also recognizes ugly strings that get mapped to a canonical representative modulo whitespace and parentheses quotienting.

Another programming language that supports quotienting is BiFluX [Pacheco et al. 2014], a bidirectional functional update language for XML. BiFluX is inspired by the FLUX XML update language [Cheney 2008], and adopts a bidirectional programming by update paradigm, where a program succintly and precisely describes *how* to update a source document with a target document, in an intuitive way, such that there is a unique “inverse” source query for each update program. The source and view types of BiFluX programs are given by regular expressions.

In BiFluX, a regular expression R is said to be unambiguous if there is only one way to parse a *flat* value of type R to a structured value of type R; here a flat value of R is a value of R with all left/right tags, parentheses and list brackets removed. The bidirectional part of BiFluX involves transforming source regular expressions to view regular expressions so that data sequences described by the two types can be matched. This process can result in intermediate types that are ambiguous, and this ambiguity can cause unintended updates to be made the source, especially when information is discarded in this process.

The typechecking phase of a BiFluX program therefore includes a *type normalization* phase that tries to normalize a source types using automata reduction techniques, and derive a lens between ambiguous and unambiguous types. On the view side, this phase only tries to normalize view types into isomorphic types. Pacheco et. al. do not claim that their normalization procedures are complete in the sense that they can disambiguate any ambiguous type, but they do show that when these procedures succeed the normalized types are unambiguous.

Formlenses [Rajkumar et al. 2014] are a variant of lenses that perform transformations up to equivalence. Formlenses are a bidirectional generalization of *formlets* [Cooper et al. 2008], which are a high-level abstraction for building Web forms. Formlets encapsulate several low-level details including selecting field names for elements and parsing data from client responses. In their work, Rajkumar et. al. represent a formlens as a function that takes an optional initial value of a form and a list of integers as a name source and produces a triple consisting of an HTML document, a *collect* function, and a modified namesource:

```
type Formlens a = Maybe a → [Int] → (Html,Env → Maybe a,[Int])
```

The names of any generated form fields are drawn from the name source, and the *collect* function looks up precisely these names

Rajkumar et. al. consider only formlenses that are well-formed in the sense that a well-formed formlens should only draw names from the namesource provided as an argument, and the *collect* function should only look up corresponding names. Under this assumption, Rajkumar et. al. define what it means for a formlens to be well behaved by defining a version of the classical GetPut and PutGet laws for formlenses. These laws hold modulo an equivalence relation on environments, since a formlens may be invoked with two separate namespaces, but this should not affect the value stored in the information encapsulated in the type a. Consequently, the GetPut and PutGet laws defined for formlenses are in spirit analogous to the GetPut and PutGet laws for quotient lenses, since they hold up to renaming of environment variables.

Another recent line of work that exploits equivalence relations on data is by Hilken et. al. in [Hilken et al. 2016]. This work is concerned mainly with testing, and uses *Equivalence Partitioning*, a software testing technique that divides the input data of a software unit into partitions of equivalent data from which test cases can be derived. This approach has the advantage that it can greatly reduce the number of test cases and hence the amount of time spent testing since only one representative from each class is tested.

More concretely, their approach gives the developer an explicit option to formulate her understanding of two object models being different. The technical realization is as follows: The developer specifies "classifying term", a closed Object Constraint Language (OCL) query term that can be evaluated in an object model and returns a characteristic value. Two object models with the same characteristic value belong to the same equivalence class. For example under a configuration requiring at least 2 and at most 4 Person objects, the classifying term Person.allInstances()−>size() would yield three object models with 2,3 and 4 Person objects respectively.

Classifying terms are therefore analogous to our QREs or to Boomerang canonizers in the sense that they define an equivalence relation on data. However, while QREs and canonizers aim at identifying or characterizing the elements that should be treated equally by a lens, the focus of the work by Hilken et al. is on classes that represent specific patterns of particular relevance to the modeler who is interested in analyzing the behavior of a transformation.

Finally, Cunha [Cunha 2010] uses relational algebra to encompass many of the existing approaches to bidirectional programming. For example, using $\circ$ to express relational composition, the GetPut law states that $\mathsf{get} \circ \mathsf{put} \subseteq \equiv_V \circ \pi_1$, where π_1 is the projection onto the first factor. In addition to its generality, this "point-free" approach to defining bidirectional transformations has the advantage of creating equational proofs of lens laws that proceed by folding and unfolding combinator laws.

Next we turn to related work in program synthesis. Much of the research in synthesis assumes that the synthesizer is provided with a collection of examples. Optician and QRE-Enhanced Optician differ in that they require the programmer to supply both examples *and* format descriptions in the form of regular expressions or QREs, though these two systems are far from the only ones to consider type-based synthesis. There is a trade-off here. On the one hand, a user must have some programming expertise to write regular expression (or QRE) specifications, and it requires some work. On the other hand, such specifications provide a great deal of information to the synthesis system, which decreases the number of examples needed (often to zero), makes the system scale well, and allows it to handle large, complex formats. By providing these format specifications, the synthesis engine does not have to both infer the format of the data as well as the transformations on it, obviating the need to infer tricky formats like those involving nested iterations.

There are many other recent results showing how to synthesize functions from type-based specifications [Augustsson 2004; Feser et al. 2015; Frankle et al. 2015; Osera and Zdancewic 2015; Polikarpova et al. 2016; Scherer and Rèmy 2015]. These systems enumerate programs of their target language, orienting their search procedures to process only terms that are well-typed. Optician is distinctive in that it synthesizes terms in a language with many type equivalences. Perhaps the system most similar to Optician is InSynth [Gvero et al. 2013], a system for synthesizing terms in the simply-typed lambda calculus that addresses equivalences on types. Instead of trying to directly synthesize terms of the simply-typed lambda calculus, InSynth synthesizes a well-typed term in the succinct calculus, a language with types that are equivalent "modulo isomorphisms of products and currying" [Gvero et al. 2013]. The type structure used in Optician is significantly more complex. In particular, because Optician types do not have full canonical forms, Miltner et al. [Miltner et al. 2017] used a pseudo-canonical form that captures part of the equivalence relation over types. To preserve completeness, they pushed some of the remaining parts of the type equivalence relation into a set of rewriting rules and other parts into the synthesis algorithm itself.

Morpheus [Feng et al. 2017] is another synthesis system that uses two communicating synthesizers to generate programs. In both Morpheus and Optician, one synthesizer provides an outline for the program, and the other fills in that outline with program details that satisfy the user's specifications. This approach works well in large search spaces that require some enumerative search. One important way that Optician differs from Morpheus is that in Morpheus, an outline is a sketch—an *expression* containing holes—whereas an outline in Optician is a pair of regular expressions, i.e., a *type*. Moreover, in order to implement an efficient search procedure, Anders et. al had to create both a new type language and a new term language for lenses. Once they did so, they proved their new, more constrained language designed for synthesis was just as expressive as the original, more flexible and compositional language designed for human programmers.

9 CONCLUSION AND FUTURE WORK

In this paper, we showed how to synthesize a class of bijective quotient lenses using the bijective lens synthesis system Optician as a plug-in for our synthesis algorithm. In order to achieve this, we first introduced *Quotient regular languages* to specify regular languages with an equivalence relation defined on them. Then, we introduced *QRE lenses*, which are bijective quotient lenses that map between QREs via bijective lenses. We proved a normal form theorem for QREs that enabled us to (1) extend the synthesis algorithm used by Optician to synthesize QRE lenses, and (2) prove that if there is a QRE lens that satisfies the input specification, then the extended algorithm returns such a lens. Finally we tested QRE-enhanced optician on the Optician benchmark suite and demonstrated that QREs can save programmers significant effort compared to both Optician and basic Boomerang, and that QRE-enhanced Optician maintains competitive performance over both of these systems.

Looking ahead, we are experimenting with larger examples of quotiented bijective lenses. So far, the three "specialized" QRE combinators (squash, perm, and collapse) have proved sufficient, but we expect eventually to encounter examples that will lead us to consider adding further combinators to supplement these.

Along a different dimension, we would like to investigate whether the techniques proposed here can be applied to synthesizing quotiented variants of richer classes of lenses, such as "classic" asymmetric lenses [Foster et al. 2007] or symmetric lenses [Hofmann et al. 2011]. The main technical issues are developing synthesis procedures for the unquotientied variants of a given class—we are currently attacking the case of symmetric lenses—and showing that quotiented lenses in this class can be normalized along the lines of Theorem 5.2.

APPENDICES

A PROOF OF NORMAL FORM THEOREM FOR QRE LENSES

Theorem 5.2 claims that if there is a derivation $q : Q_1 \Leftrightarrow Q_2$, then there exists a bijective lens $\ell : K(Q_1) \Leftrightarrow K(Q_2)$ such that:

$$[\![q]\!].\mathsf{get} = [\![\ell]\!] \circ \mathsf{canonize}(Q_1) \text{ and } [\![q]\!].\mathsf{put} = [\![\ell]\!]^{-1} \circ \mathsf{canonize}(Q_2)$$

We now prove this theorem.

Proof. We proceed by induction over the derivation $q : Q_1 \Leftrightarrow Q_2$.

(1) $\mathsf{lift}(\ell) : R/id(R) \Leftrightarrow S/id(S)$ where $\ell : R \Leftrightarrow S$. Then:

$$[\![\mathsf{lift}(\ell)]\!].\mathsf{get} = [\![\ell]\!] = [\![\ell]\!] \circ id_{\mathcal{L}(R)} = [\![\ell]\!] \circ \mathsf{canonize}(id(R)), \text{ and}$$
$$[\![\mathsf{lift}(\ell)]\!].\mathsf{put} = [\![\ell]\!]^{-1} = [\![\ell]\!]^{-1} \circ id_{\mathcal{L}(S)} = [\![\ell]\!]^{-1} \circ \mathsf{canonize}(id(S))$$

(2) $\mathsf{lquot}(Q_1, q) : Q_1 \ ; \ Q_2 \Leftrightarrow Q_3$ where $q : Q_2 \Leftrightarrow Q_3$, Q_1 is well formed and $K(Q_1) = W(Q_2)$. Then:

$$[\![\mathsf{lquot}(Q_1, q)]\!].\mathsf{get} = [\![q]\!].\mathsf{get} \circ \mathsf{canonize}(Q_1) \text{ and } [\![\mathsf{lquot}(Q_1, q)]\!].\mathsf{put} = [\![q]\!].\mathsf{put}$$

By the induction hypothesis, there exists a bijective lens $\ell : K(Q_2) \Leftrightarrow K(Q_3)$ such that:

$$[\![q]\!].\mathsf{get} = [\![\ell]\!] \circ \mathsf{canonize}(Q_2) \text{ and } [\![q]\!].\mathsf{put} = [\![\ell]\!]^{-1} \circ \mathsf{canonize}(Q_3)$$

Consequently

$$[\![\mathsf{lquot}(Q_1, q)]\!].\mathsf{get} = ([\![\ell]\!] \circ \mathsf{canonize}(Q_2)) \circ \mathsf{canonize}(Q_1) = [\![\ell]\!] \circ (\mathsf{canonize}(Q_1 \ ; \ Q_2))$$
$$[\![\mathsf{lquot}(Q_1, q)]\!].\mathsf{put} = [\![\ell]\!]^{-1} \circ \mathsf{canonize}(Q_3)$$

(3) $\mathsf{rquot}(q, Q_3) : Q_1 \Leftrightarrow Q_2 \ ; \ Q_3$ where $q : Q_1 \Leftrightarrow Q_2$, Q_3 is well formed and $K(Q_3) = W(Q_2)$. Proceed as in the previous case.

(4) $q_1 \ ; \ q_2 : c \Leftrightarrow Q_2$ where $q_1 : c \Leftrightarrow Q_1$ and $q_2 : Q_1 \Leftrightarrow Q_2$. Then:

$$[\![q_1 \ ; \ q_2]\!].\mathsf{get} = [\![q_2]\!].\mathsf{get} \circ [\![q_1]\!].\mathsf{get} \text{ and } [\![q_1 \ ; \ q_2]\!].\mathsf{put} = [\![q_1]\!].\mathsf{put} \circ [\![q_2]\!].\mathsf{put}$$

By the induction hypothesis, there exist bijective lenses, $\ell_1 : K(c) \Leftrightarrow K(Q_1)$ and $\ell_2 : K(Q_1) \Leftrightarrow K(Q_2)$ such that,

$$\begin{array}{ll} [\![q_1]\!].\mathsf{get} = [\![\ell_1]\!] \circ \mathsf{canonize}(c) & [\![q_2]\!].\mathsf{get} = [\![\ell_2]\!] \circ \mathsf{canonize}(Q_1) \\ [\![q_1]\!].\mathsf{put} = [\![\ell_1]\!]^{-1} \circ \mathsf{canonize}(Q_1) & [\![q_2]\!].\mathsf{put} = [\![\ell_2]\!]^{-1} \circ \mathsf{canonize}(Q_2) \end{array}$$

Consequently:

$$\begin{aligned} [\![q_1 \ ; \ q_2]\!].\mathsf{get} &= [\![q_2]\!].\mathsf{get} \circ [\![q_1]\!].\mathsf{get} \\ &= ([\![\ell_2]\!] \circ \mathsf{canonize}(Q_1)) \circ ([\![\ell_1]\!] \circ \mathsf{canonize}(c)) \\ &= [\![\ell_2]\!] \circ (\mathsf{canonize}(Q_1) \circ [\![\ell_1]\!]) \circ \mathsf{canonize}(c) \\ &= ([\![\ell_2]\!] \circ [\![\ell_1]\!]) \circ \mathsf{canonize}(c) \\ &= [\![\ell_1 \ ; \ \ell_2]\!] \circ \mathsf{canonize}(c) \end{aligned}$$

A similar argument shows that $[\![q_1 \ ; \ q_2]\!].\mathsf{put} = [\![\ell_1 \ ; \ \ell_2]\!]^{-1} \circ \mathsf{canonize}(c)$

(5) $q^* : Q_1{}^* \Leftrightarrow Q_2{}^*$ where $q : Q_1 \Leftrightarrow Q_2$, $W(Q_1)^{*!}$ and $W(Q_2)^{*!}$ and $K(Q_1)^{*!}$ and $K(Q_2)^{*!}$. Then:

$$[\![q^*]\!].\mathsf{get} = ([\![q]\!].\mathsf{get})^* \text{ and } [\![q^*]\!].\mathsf{put} = ([\![q]\!].\mathsf{put})^*$$

By the induction hypothesis there exists a bijective lens $\ell : K(Q_1) \Leftrightarrow K(Q_2)$ such that:

$$[\![q]\!].\mathsf{get} = [\![\ell]\!] \circ \mathsf{canonize}(Q_1) \text{ and } [\![q]\!].\mathsf{put} = [\![\ell]\!]^{-1} \circ \mathsf{canonize}(Q_2)$$

Consequentlty:

$$[\![q^*]\!].\mathsf{get} = ([\![\ell]\!] \circ \mathsf{canonize}(Q_1))^* = [\![\ell]\!]^* \circ \mathsf{canonize}(Q_1)^* = [\![\ell^*]\!] \circ \mathsf{canonize}(Q_1^*)$$
$$[\![q^*]\!].\mathsf{put} = ([\![\ell]\!]^{-1} \circ \mathsf{canonize}(Q_2))^* = ([\![\ell]\!]^{-1})^* \circ \mathsf{canonize}(Q_2)^* = [\![\ell^*]\!]^{-1} \circ \mathsf{canonize}(Q_2^*)$$

(6) $q_1 \cdot q_2 : Q_1 \cdot Q_2 \Leftrightarrow Q_3 \cdot Q_4$, where $q_1 : Q_1 \Leftrightarrow Q_3$, $q_2 : Q_2 \Leftrightarrow Q_4$, $W(Q_1) \cdot^{!} W(Q_2)$, $K(Q_1) \cdot^{!} K(Q_2)$, $W(Q_3) \cdot^{!} W(Q_4)$ and $K(Q_3) \cdot^{!} K(Q_4)$. Then:

$$[\![q_1 \cdot q_2]\!].\mathsf{get} = [\![q_1]\!].\mathsf{get} \cdot [\![q_2]\!].\mathsf{get} \text{ and } [\![q_1 \cdot q_2]\!].\mathsf{put} = [\![q_1]\!].\mathsf{put} \cdot [\![q_2]\!].\mathsf{put}$$

By the induction hypothesis, there exist bijective lenses $\ell_1 : K(Q_1) \Leftrightarrow K(Q_3)$ and $\ell_2 : K(Q_2) \Leftrightarrow K(Q_4)$ such that,

$$\begin{array}{ll} [\![q]\!].\mathsf{get} = [\![\ell_1]\!] \circ \mathsf{canonize}(Q_1) & [\![q_2]\!].\mathsf{get} = [\![\ell_2]\!] \circ \mathsf{canonize}(Q_2) \\ [\![q]\!].\mathsf{put} = [\![\ell_1]\!]^{-1} \circ \mathsf{canonize}(Q_3) & [\![q_2]\!].\mathsf{put} = [\![\ell_2]\!]^{-1} \circ \mathsf{canonize}(Q_4) \end{array}$$

Consequently:

$$\begin{aligned} [\![q_1 \cdot q_2]\!].\mathsf{get} &= ([\![\ell_1]\!] \circ \mathsf{canonize}(Q_1)) \cdot ([\![\ell_2]\!] \circ \mathsf{canonize}(Q_2)) \\ &= ([\![\ell_1]\!] \cdot [\![\ell_2]\!]) \circ (\mathsf{canonize}(Q_1) \cdot \mathsf{canonize}(Q_2)) \\ &= [\![\ell_1 \cdot \ell_2]\!] \circ \mathsf{canonize}(Q_1 \cdot Q_2) \end{aligned}$$

Similarly, $[\![q_1 \cdot q_2]\!].\mathsf{put} = [\![\ell_1 \cdot \ell_2]\!]^{-1} \circ \mathsf{canonize}(Q_3 \cdot Q_4)$

(7) $\mathsf{swap}(q_1, q_2) : Q_1 \cdot Q_2 \Leftrightarrow Q_4 \cdot Q_3$, where $q_1 : Q_1 \Leftrightarrow Q_3$, $q_2 : Q_2 \Leftrightarrow Q_4$, $W(Q_1) \cdot^{!} W(Q_2)$, $K(Q_1) \cdot^{!} K(Q_2)$, $W(Q_4) \cdot^{!} W(Q_3)$ and $K(Q_4) \cdot^{!} K(Q_3)$. Then:

$$\begin{aligned} [\![\mathsf{swap}(q_1, q_2)]\!].\mathsf{get}(s_1 \cdot s_2) &= [\![q_2]\!].\mathsf{get}(s_2) \cdot [\![q_1]\!].\mathsf{get}(s_1), \text{ and} \\ [\![\mathsf{swap}(q_1, q_2)]\!].\mathsf{put}(t_1, t_2) &= [\![q_1]\!].\mathsf{put}(t_1) \cdot [\![q_2]\!].\mathsf{put}(t_2) \end{aligned}$$

By the induction hypothesis, there exist bijective lenses $\ell_1 : K(Q_1) \Leftrightarrow K(Q_3)$ and $\ell_2 : K(Q_2) \Leftrightarrow K(Q_4)$ such that,

$$\begin{array}{ll} [\![q_1]\!].\mathsf{get} = [\![\ell_1]\!] \circ \mathsf{canonize}(Q_1) & [\![q_2]\!].\mathsf{get} = [\![\ell_2]\!] \circ \mathsf{canonize}(Q_2) \\ [\![q_1]\!].\mathsf{put} = [\![\ell_1]\!]^{-1} \circ \mathsf{canonize}(Q_3) & [\![q_2]\!].\mathsf{put} = [\![\ell_2]\!]^{-1} \circ \mathsf{canonize}(Q_4) \end{array}$$

Consequently:

$$\begin{aligned} [\![\mathsf{swap}(q_1, q_2)]\!].\mathsf{get}(s_1 \cdot s_2) &= [\![q_2]\!].\mathsf{get}(s_2) \cdot [\![q_1]\!].\mathsf{get}(s_1) \\ &= ([\![\ell_2]\!] \circ \mathsf{canonize}(Q_2))(s_2) \cdot ([\![\ell_1]\!] \circ \mathsf{canonize}(Q_1))(s_1) \\ &= ([\![\mathsf{swap}(\ell_1, \ell_2)]\!]) \circ (\mathsf{canonize}(Q_1) \cdot \mathsf{canonize}(Q_2))(s_1, s_2) \end{aligned}$$

Similarly, $[\![\mathsf{swap}(q_1, q_2)]\!].\mathsf{put} = ([\![\mathsf{swap}(\ell_1, \ell_2)]\!])^{-1} \circ (\mathsf{canonize}(Q_4) \cdot \mathsf{canonize}(Q_3))$

(8) $q_1 = q_1 \mid q_2$ where $q_1 : Q_1 \Leftrightarrow Q_3$, $q_2 : Q_2 \Leftrightarrow Q_4$, $\mathcal{L}(W(Q_1)) \cap \mathcal{L}(W(Q_2)) = \varnothing$ and $\mathcal{L}(W(Q_3)) \cap \mathcal{L}(W(Q_4)) = \varnothing$. Then:

$$[\![q_1 \mid q_2]\!].\mathsf{get}(s) = \begin{cases} [\![q_1]\!].\mathsf{get}(s) & \text{if } s \in \mathcal{L}(W(Q_1)) \\ [\![q_2]\!].\mathsf{get}(s) & \text{if } s \in \mathcal{L}(W(Q_2)) \end{cases}$$

$$[\![q_1 \mid q_2]\!].\mathsf{put}(s) = \begin{cases} [\![q_1]\!].\mathsf{put}(s) & \text{if } s \in \mathcal{L}(W(Q_3)) \\ [\![q_2]\!].\mathsf{put}(s) & \text{if } s \in \mathcal{L}(W(Q_4)) \end{cases}$$

By the induction hypothesis, there exist bijective lenses $\ell_1 : K(Q_1) \Leftrightarrow K(Q_3)$ and $\ell_2 : K(Q_2) \Leftrightarrow K(Q_4)$ such that,

$$\begin{array}{ll} [\![q_1]\!].\mathsf{get} = [\![\ell_1]\!] \circ \mathsf{canonize}(Q_1) & [\![q_2]\!].\mathsf{get} = [\![\ell_2]\!] \circ \mathsf{canonize}(Q_2) \\ [\![q_1]\!].\mathsf{put} = [\![\ell_1]\!]^{-1} \circ \mathsf{canonize}(Q_3) & [\![q_2]\!].\mathsf{put} = [\![\ell_2]\!]^{-1} \circ \mathsf{canonize}(Q_4) \end{array}$$

Consequently:

$$[\![q_1 \mid q_2]\!].\mathsf{get}(s) = \begin{cases} [\![\ell_1]\!] \circ \mathsf{canonize}(Q_1)(s) & \text{if } s \in \mathcal{L}(W(Q_1)) \\ [\![\ell_2]\!] \circ \mathsf{canonize}(Q_2)(s) & \text{if } s \in \mathcal{L}(W(Q_2)), \end{cases}$$

so $[\![q_1 \mid q_2]\!].\mathsf{get} = [\![\ell_1 \mid \ell_2]\!] \circ \mathsf{canonize}(Q_1 \mid Q_2)$. Similarly, $[\![q_1 \mid q_2]\!].\mathsf{put} = [\![\ell_1 \mid \ell_2]\!]^{-1} \circ \mathsf{canonize}(Q_3 \mid Q_4)$.
This completes the proof.

□

B PROOF OF CORRECTNESS OF SYNTHQRELENS

Theorem 6.1 states that Given QREs Q_1 and Q_2, and a set of examples $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, if there is a QRE lens $q : Q_1 \Leftrightarrow Q_2$ such that $q.\mathsf{get}(x_i) \equiv_{Q_2} y_i$ and $q.\mathsf{put}(y_i) \equiv_{Q_1} x_i$, then $\textsc{SynthQRELens}(Q_1, Q_2, exs)$ will return such a lens. We prove that here.

Lemma B.1 (Algorithm Soundness). *If $q = \textsc{SynthQRELens}(Q_1, Q_2, exs)$, then $q : Q_1 \Leftrightarrow Q_2$ and $q.\mathsf{get}(x_i) \equiv_{Q_2} y_i$ and $q.\mathsf{put}(y_i) \equiv_{Q_1} x_i$ for all $(x_i, y_i) \in exs$.*

Proof. As $q = \textsc{SynthQRELens}(Q_1, Q_2, exs)$, we know that $q = \mathsf{rquot}(\mathsf{lquot}(Q_1, \ell), Q_2)$. Furthermore, we know that $\ell = \textsc{SynthBijectiveLens}(R_1, R_2, exs')$, where $exs' = \textsc{Map}(exs, fun\,(ex_l, ex_r) \rightarrow (\mathsf{canonize}(Q_1)(ex_l), \mathsf{canonize}(Q_2)(ex_r)))$. By the correctness of SynthBijectiveLens, we know that $\ell.\mathsf{get}(x'_i) = y'_i$ and $\ell.\mathsf{put}(y'_i) = x'_i$ for all $(x'_i, y'_i) \in exs'$. By definitions of rquot and lquot, this means that:
$\mathsf{rquot}(\mathsf{lquot}(Q_1, \ell), Q_2).\mathsf{get}(x_i)$
$= \ell.get(\mathsf{canonize}(Q_1)(x_i))$
$= \ell.get(x'_i) = y'_i = \mathsf{canonize}(Q_2)(y_i) \equiv_{Q_2} y_i$ and

$\mathsf{rquot}(\mathsf{lquot}(Q_1, \ell), Q_2).\mathsf{put}(y_i)$
$= \ell.\mathsf{put}(\mathsf{canonize}(Q_2)(y_i))$
$= \ell.\mathsf{put}(y'_i) = x'_i = \mathsf{canonize}(Q_1)(x_i) \equiv_{Q_1} x_i$ as desired. □

Lemma B.2 (Algorithm Completeness). *If there exists a QRE lens $q : Q_1 \Leftrightarrow Q_2$ such that $q.\mathsf{get}(x_i) \equiv_{Q_2} y_i$ and $q.\mathsf{put}(y_i) \equiv_{Q_1} x_i$ for all $(x_i, y_i) \in exs$, then $\textsc{SynthQRELens}(Q_1, Q_2, exs)$ terminates.*

Proof. By Theorem 5.2, there exists a bijective lens $\ell : K(Q_1) \Leftrightarrow K(Q_2)$, such that $q' = \mathsf{rquot}(\mathsf{lquot}(Q_1, \ell), Q_2)$ is semantically equal to q.

This means that $q'.\mathsf{get}(x_i) \equiv_{Q_2} y_i$ and $q'.\mathsf{put}(y_i) \equiv_{Q_1} x_i$ for all $(x_i, y_i) \in exs$. Unfolding the definitions of rquot and lquot, we get:
$q'.\mathsf{get}(x_i) = \ell.\mathsf{get}(\mathsf{canonize}(Q_1)(x_i)) = \mathsf{canonize}(Q_2)(y_i)$ and
$q'.\mathsf{put}(y_i) = \ell.\mathsf{put}(\mathsf{canonize}(Q_2)(y_i)) = \mathsf{canonize}(Q_1)(x_i)$ for all $(x_i, y_i) \in exs$.

By correctness of SynthBijectiveLens, we know that $\textsc{SynthBijectiveLens}(R_1, R_2, exs')$ terminates, so the entire algorithm terminates. □

With soundness and completeness, we trivially get the correctness theorem.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their useful feedback and discussions and Nate Foster for his extensive assistance in integrating QRE-enhanced QRE-enhanced Optician into Boomerang. This research has been supported in part by DARPA award FA8750-17-2-0028 and ONR 568751 (SynCrypt).

REFERENCES

Lennart Augustsson. 2004. [Haskell] Announcing Djinn, version 2004-12-11, a coding wizard. Mailing List. http://www.haskell.org/pipermail/haskell/2005-December/017055.html.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: Resourceful Lenses for String Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM.

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. 2008. Dual Syntax for XML Languages. *Inf. Syst.* 33, 4-5 (June 2008).

James Cheney. 2008. FLUX: functional updates for XML. In *ACM Sigplan Notices*, Vol. 43. ACM, 3–14.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2008. The essence of form abstraction. In *Asian Symposium on Programming Languages and Systems*. Springer, 205–220.

Alcino Cunha. 2010. A relational approach to bidirectional transformations. (2010).

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM. http://doi.acm.org/10.1145/3062341.3062351

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Trans. Program. Lang. Syst.* 29, 3, Article 17 (May 2007). https://doi.org/10.1145/1232420.1232424

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. 2008. Quotient Lenses. *SIGPLAN Not.* 43, 9 (Sept. 2008), 383–396. https://doi.org/10.1145/1411203.1411257

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2015. *Example-Directed Synthesis: A Type-Theoretic Interpretation (extended version)*. Technical Report MS-CIS-15-12. University of Pennsylvania.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM.

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Frank Hilken, Martin Gogolla, Loli Burgueño, and Antonio Vallecillo. 2016. Testing models and model transformations using classifying terms. *Software & Systems Modeling* (2016), 1–28.

Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2011. Symmetric lenses. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 371–384.

Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. 2004. A Programmable Editor for Developing Structured Documents Based on Bidirectional Transformations. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '04)*. ACM, New York, NY, USA, 178–189. https://doi.org/10.1145/1014007.1014025

Shinya Kawanaka and Haruo Hosoya. 2006. biXid: A Bidirectional Transformation Language for XML. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM.

David Lutterkort. 2008. Augeas–a configuration API. In *Linux Symposium, Ottawa, ON*. 47–56.

Kazutaka Matsuda and Meng Wang. 2013. FliPpr: A prettier invertible printing system. In *European Symposium on Programming*. Springer, 101–120.

Anders Miltner, Kathleen Fisher, Benjamin C Pierce, David Walker, and Steve Zdancewic. 2017. Synthesizing bijective lenses. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1.

Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. 2004. *An Algebraic Approach to Bi-directional Updating*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2–20. https://doi.org/10.1007/978-3-540-30477-7_2

Shin-Cheng MU, Zhenjiang HU, and Masato TAKEICHI. 2006. Bidirectionalizing Tree Transformation Languages: A Case Study. *Computer Software* 23, 2 (2006), 129–141. https://doi.org/10.11309/jssst.23.2_129

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM.

Hugo Pacheco, Tao Zan, and Zhenjiang Hu. 2014. BiFluX: A bidirectional functional update language for XML. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. ACM, 147–158.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM. http://doi.acm.org/10.1145/2908080.2908093

Raghu Rajkumar, Nate Foster, Sam Lindley, and James Cheney. 2014. Lenses for web data. *Electronic Communications of the EASST* 57 (2014).

Gabriel Scherer and Didier Rèmy. 2015. Which simple types have a unique inhabitant?. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Seppo Sippu and Eljas Soisalon-Soininen. 1988. *Regular Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 65–114. https://doi.org/10.1007/978-3-642-61345-6_3