Synthesizing Lenses

Anders Francis Miltner

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: David Walker

September 2020

Abstract

Lenses are bidirectional functions that satisfy a set of "round-tripping laws." Lenses arise in a number of domains, and can implement serializer/deserializer pairs, parser/pretty printer pairs, and incremental data structure transformers.

Domain-specific languages like Boomerang, Augeas, GRoundTram, BiFluX, BiYacc, Brul, BiGUL, and HOBiT allow programmers to write both transformations with a single program, and guarantee the transformations satisfy the round-tripping laws. However, writing in domain-specific lens languages can be hard, requiring the user to reason about fiddly details of the transformations, often within the context of a complex type system.

In this work we introduce Optician, which provides an alternative method of developing lenses – synthesis. We develop a synthesis engine for Boomerang, a lens language for string transformations. Pairs of regular expressions serve as types for Boomerang lenses, which transform strings between the languages of those regular expressions. Instead of manually writing Boomerang lenses, programmers merely need to provide the type of the desired lens and a set of input/output examples describing the lens's behavior. Optician will then synthesize a lens between the provided types that exhibits the demonstrated behavior. We demonstrate how to synthesize three classes of lenses: bijective lenses, quotient lenses, and symmetric lenses. Bijective lenses encode bijections, quotient lenses encode bijections modulo an equivalence relation, and symmetric lenses encode bidirectional transformations that may discard information when going from one format to another.

We define a synthesizer that involves two cooperating procedures. One procedure proposes a candidate space of lenses, the second procedure searches through that space.

To synthesize quotient lenses, regular expressions are augmented with equivalence information, and the synthesizer generates lenses that translate between the repre-

sentative elements of the equivalence classes. To synthesize symmetric lenses, regular expressions are augmented with probability distributions, and the synthesis algorithm aims to avoid information loss.

We evaluate Optician from a variety of benchmarks taken from the lens and synthesis literature, and find that we are able to synthesize all the lenses in our benchmark suite. We have integrated Optician into the Boomerang codebase, enabling users to either synthesize their lenses or write them by hand.

Acknowledgements

I've been incredibly fortunate throughout my PhD to have been surrounded by wonderful collaborators, communities, and friends.

Of course, my PhD would not have been possible without my advisor, David Walker. Dave constantly demonstrates that his priorities lie with his students; focusing on our happiness and our education over his publication record, research agenda, and hands.

Much of my learning happened during the research process. My collaborators, Kathleen Fisher, Benjamin Pierce, Steve Zdancewic, Solomon Maina, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, Abhishek Udupa, Todd Millstein, and Saswat Padhi, have all shaped me as a researcher. In addition to teaching me how to research, they helped me work on some pretty cool projects!

The Princeton PL group was instrumental to keeping me happy and sane during the last 5 years. Instead of listing individual people, I will instead list a series of references and inside jokes, as that is more fun: beanbag chairs and lava lamps, sushi and escolar, 40 Wendy's chicken nuggets, PLSwole and Ryan's scale, 215!, when to leave for lunch talks, Charlie's baking, GC grilled chicken, Citron Saison, and The Always Fresh, Never Frozen PL Chat.

In addition to my friends in the PL group, I'd like to thank my internet gaming friend, Miles Paterson. It's been great being able take a break from the stresses of grad school, and replace them with the stresses of losing at video games.

Of course, a big thanks to Katie Cavanagh, my partner for basically the entirety of my PhD. Katie's helped me relax and escape from academics and PL. Katie has helped me through paper rejections, helped celebrate paper acceptances, and been a positive force throughgout my PhD.

I'd also like to thank my immediate and extended family. My dad Robert and my mom Susan have been there, bugging me to call them on Sundays, and generally just

being supportive of my experience as a grad student. My sister Lise has lived nearby, and has been able to give me weekends of fun when she visits, while also being there to empathize with the difficulties of grad school.

I've been gracefully accepted into a number of houses during grad school. My roommates, Whitney Mueller, Riley Simmons-Edler, and Weikun Yang have been a blast to live with (and play LOTS of board games with). Katie's parents, Norm Cavanagh and Patricia DeFusco let me stay with them for almost 3 months during Coronavirus. Katie's roommate, Shirley Zhu has also let me stay with her as I finish my thesis, as well as for a number of months during the winter and last summer. Good living situations help me stay happy, and I've been lucky to live with such great people.

I'd like to thank my examiners, Andrew Appel and Aarti Gupta, and my readers, Kathleen Fisher and Zak Kincaid. Their comments and revisions were indispensable in improving my thesis.

The work done in this dissertation was supported by DARPA award FA8750-17-2-0028 and DARPA SafeDocs (PRIME DARPA BAA #HR001118S0054).

Lastly, I'd like to thank Copper, my cute puppy! I got Copper during my last year, and she has been a blast to raise.

Contents

Chapter 1

Introduction

Programs that analyze consumer information, performance statistics, transaction logs, scientific records, and many other kinds of data are essential components in many software systems. Oftentimes, the data analyzed comes in *ad hoc* formats, making tools for reliably parsing, printing, cleaning, and transforming data increasingly important. Programmers often need to reliably transform back-and-forth between formats, not only transforming source data into a target format but also safely transforming target data back into the source format. *Lenses* [17] are back-and-forth transformations that provide strong guarantees about their round-trip behavior, guarding against data corruption while reading, editing, and writing data sources.

A lens comprises a number of functions, depending on the class of lenses. Generally, there are one or more transformations that translate from the *left-hand* or *source* format, into the *right-hand* or *target* format; and there are one or more transformations that translate from the target format back to the source format. A benefit of lens-based languages is that they use a single term to express both *left-to-right* and *right-to-left* transformations. Furthermore, well-typed lenses give rise to functions guaranteed to satisfy desirable invertibility properties.

```
let name = [A−Z][a−z]∗
let wsp  = [ \t\r\n\f]+
let lfm = name . ",," (wsp . name)∗
let fml = (name . wsp)∗ . name
let name_swap : (lens in fml <=> lmf) =
    swap
      (id(name) . del ",")
      (swap id(wsp) id(name))∗
```

Figure 1.1: Bijective lens that converts names formatted like "Miltner, Anders Francis" to names formatted like "Anders Francis Miltner" and back again.

Lens-based languages are present in a variety of tools and have found mainstream industrial use. Boomerang [4, 6] lenses provide guarantees on transformations between *ad hoc* string document formats. Augeas [35], a popular tool that reads Linux system configuration files, uses the left-to-right part of a lens to transform configuration files into a canonical tree representation that users can edit either manually or programmatically. It uses the lens's right-to-left transformations to merge the edited results back into the original string format. Other lens-based languages and tools include GRoundTram [24], BiFluX [49], BiYacc [63], Brul [62], BiGUL [29], bidirectional variants of relational algebra [7], spreadsheet formulas [37], graph query languages [23], and XML transformation languages [34].

Unfortunately, these languages are difficult to program in, as they force the programmer to juggle the multiple ways of interpreting their terms. In Boomerang, a single term encodes the allowable inputs from the source data format, the allowable inputs from the target data format, and how to transform the terms from each format to the other. When the two data formats contain Kleene stars in different places, programmers must mentally transform these formats into a common "aligned" form.

These languages impose fiddly constraints on their terms, making lens programming slow and tedious. For example, Boomerang programmers often must rearrange the order of data items by recursively using operators that swap adjacent fields. Furthermore, the Boomerang type checker is very strict, disallowing many programs

because they contain ambiguity about how certain data is transformed. Lens languages provide their strong bidirectional guarantees by putting additional burdens on the programmer. For example, consider the lens in Figure 1.1, written in the language of Bijective Lenses described in Chapter 3.

This transformation converts names like "Miltner, Anders Francis" to "Anders Francis Miltner" and back again. In the second format, the last name appears last. However, even this simple lens requires complex reasoning about how to reorder fields.

To make programming with lenses faster and easier, we have developed *Optician*, a tool for synthesizing lenses from simple, high-level specifications. This work continues a recent trend toward streamlining programming tasks by synthesizing programs in a variety of domain-specific languages [14, 20, 32, 50], many guided by types [14, 15, 19, 48, 51].

As inputs, Optician takes specifications for the source and target formats, plus a collection of concrete examples of the desired transformation. Format specifications are supplied as regular expressions, possibly with some light annotations. Because regular expressions are so widely understood, we anticipate such inputs will be substantially easier for everyday programmers to work with than the unfamiliar syntax of lenses. Moreover, these format descriptions communicate a great deal of information to the synthesis system. Thus, requiring user input of regular expressions makes synthesis robust, helps the system scale to large and complex data sources, and constrains the search space sufficiently that the user typically needs to give very few, if any, examples.

While we believe Optician will be helpful for lens programmers, like those who use Augeas, Optician primarily focuses on developers with little experience with lenses. For most users, learning a new, fiddly language is harder than just implementing the individual functions that comprise the lens. By requiring user input in the form of regular expressions, users needn't learn how to write Boomerang. While users do need to learn how to read Boomerang, to ensure that their lens is correct, we envision that

this is a much simpler task than writing the lenses themselves. In short, we would like to make the job of writing data synchronizers easier – we believe the process of synthesizing lenses is easier than either writing lenses by hand, or writing individual functions that comprise the lens by hand.

In this dissertation, we demonstrate how to synthesize three classes of lenses, Bijective Lenses (Chapter 3), Quotient Lenses (Chapter 4), and Symmetric Lenses (Chapter 5). Bijective lenses bidirectionally convert between data formats in bijective correspondence. Quotient lenses bidirectionally convert between data formats with equivalence relations, where the data formats are in bijective correspondence, modulo the equivalence relation. Symmetric lenses express sets of functions for bidirectionally converting between data formats, where each format may have information not present in the other format. The material for these chapters come from the papers: Synthesizing Bijective Lenses [42], Synthesizing Quotient Lenses [38], and Synthesizing Symmetric Lenses [45].

While regular expressions provide detailed information about the uses for the generated lenses, they also complicate synthesis procedures. Specifically, Boomerang's types are regular expression pairs, and each regular expression is equivalent to an infinite number of other regular expressions. To synthesize all Boomerang terms, a type-directed synthesizer must sometimes be able to find, amongst all possible equivalent regular expressions, the one with the right syntactic structure to guide the subsequent search for a well-typed, example-compatible Boomerang term.

To resolve these issues, we introduce a new language of *Disjunctive Normal Form (DNF) lenses*. Just as string lenses have pairs of regular expressions as types, DNF lenses have pairs of *DNF regular expressions* as types. The typing judgements for DNF lenses limit how equivalences can be used, greatly reducing the size of the search space. Despite the restrictive syntax and type system of DNF lenses, we prove our new

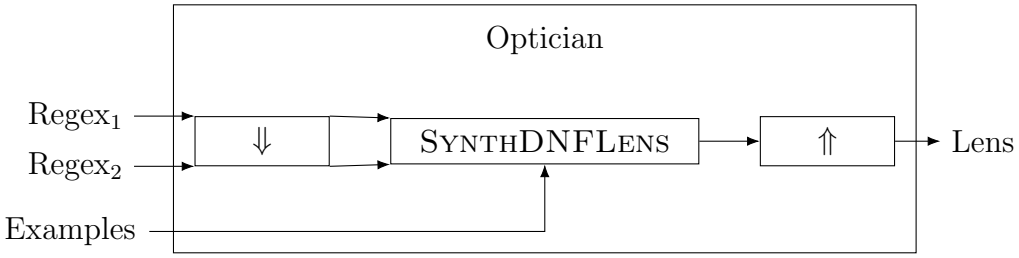


Figure 1.2: Schematic Diagram for Optician. Regular expressions, Regex_1 and Regex_2, and the examples, Examples, are given as input. First, the function $\Downarrow$ converts R and S into their respective DNF forms. Next, SynthDNFLens synthesizes a DNF lens from Examples and the DNF forms of Regex_1 and Regex_2. Finally, $\Uparrow$ converts the synthesized DNF lens into a boomerang lens, Lens.

language is equivalent to a natural, declarative specification of the bijective fragment of Boomerang.

Figure 1.2 shows a high-level, schematic diagram for Optician. First, Optician uses the function $\Downarrow$ to convert the input regular expressions into DNF regular expressions. Next, SynthDNFLens performs type-directed synthesis on these DNF regular expressions and the input examples to synthesize a DNF lens. Finally, this DNF lens is converted back into a regular lens with the function $\Uparrow$, and returned to the user. When synthesizing quotient lenses, the synthesizer performs an additional step. It extracts a canonizing function on the data formats and synthesizes a lens on the canonical representatives.

The SynthDNFLens procedure operates as a pair of synthesizers. The first synthesizer proposes candidate spaces of lenses, and the second procedure searches through that space. How these synthesizers operate, and terminate, depends on whether they are synthesizing bijective or symmetric lenses.

We have integrated Optician into the Boomerang codebase, enabling users to either synthesize their lenses or write them by hand. With this integration, users can use synthesized lenses as part of hand-written lenses, and the synthesis algorithm will utilize relevant hand-written lenses to simplify synthesis tasks.

We evaluate Optician on a variety of benchmarks taken from the lens and synthesis literature. Many of our benchmarks come from Augeas [35], a configuration editing system that uses lens combinators, and Flash Fill [20], a system that allows users to specify unidirectional string transformations by example. Our benchmarks are quite large and complex; the regular expressions for an average benchmark are defined in hundreds of AST nodes, and often have nested iterations and disjunctions. We go into extended detail in our benchmark suite construction in Chapter 3. On these benchmarks, we find Optician is relatively quick – all benchmarks are solved in under a minute.

In this thesis, we show that synthesis can be used to generate lenses between complex file formats. Optician is able to synthesize complex transformations between large data formats with nested iterations and disjunctions, largely because it leverages format descriptions. Prior work [20, 32] that synthesizes transformations purely through input-output examples eventually hits a complexity barrier, as such tools must learn both the transformation and the data formats being transformed. Realizing that format descriptions help in these synthesis tasks, we accept richer format descriptions in the forms of Quotient Regular Expressions (Chapter 4) and Stochastic Regular Expressions (Chapter 5) to help synthesize quotient and symmetric lenses. Generally, with richer user specifications, our synthesis algorithm is able to tackle increasingly complex classes of transformations.

Chapter 2

Preliminaries

Before going into the specifics of any of the synthesis problems, we provide a small section of preliminaries on regular expression syntax, unambiguity constraints, and equivalences.

2.0.1 Regular Expressions

We use Σ to denote the alphabet of individual characters c; strings s and t are elements of Σ^*. Regular expressions, abbreviated REs, are used to express *languages*, which are subsets of Σ^*. REs over Σ are:

$$R,S \quad ::= \quad s \quad | \quad \emptyset \quad | \quad R^* \quad | \quad R_1 \cdot R_2 \quad | \quad R_1 \mid R_2$$

$\mathcal{L}(R) \subseteq \Sigma^*$, the language of R, is defined as usual.

2.0.2 Regular Expression Unambiguity

The typing derivations of lenses require regular expressions to be written in a way that parses text unambiguously. R and S are *unambiguously concatenable*, written $R \cdot^! S$ if, for all strings $s_1, s_2 \in \mathcal{L}(R)$ and $t_1, t_2 \in \mathcal{L}(S)$, whenever $s_1 \cdot t_1 = s_2 \cdot t_2$ it is the case that $s_1 = s_2$ and $t_1 = t_2$. Similarly, R is *unambiguously iterable*, written

$$
\begin{array}{rcll}
R \mid \emptyset & \equiv & R & +\ Ident \\
R \cdot \emptyset & \equiv & \emptyset & 0\ Proj_R \\
\emptyset \cdot R & \equiv & \emptyset & 0\ Proj_L \\
(R \cdot R') \cdot R'' & \equiv & R \cdot (R' \cdot R'') & \cdot\ Assoc \\
(R \mid R') \mid R'' & \equiv & R \mid (R' \mid R'') & \mid\ Assoc \\
R \mid S & \equiv & S \mid R & \mid\ Comm \\
R \cdot (R' \mid R'') & \equiv & (R \cdot R') \mid (R \cdot R'') & Dist_R \\
(R' \mid R'') \cdot R & \equiv & (R' \cdot R) \mid (R'' \cdot R) & Dist_L \\
\epsilon \cdot R & \equiv & R & \cdot\ Ident_L \\
R \cdot \epsilon & \equiv & R & \cdot\ Ident_R \\
(R \mid S)^* & \equiv & (R^* \cdot S)^* \cdot R^* & Sumstar \\
(R \cdot S)^* & \equiv & \epsilon \mid (R \cdot (S \cdot R)^* \cdot S) & Prodstar \\
(R^*)^* & \equiv & R^* & Starstar \\
(R \mid S)^* & \equiv & ((R \mid S) \cdot S \mid (R \cdot S^*)^n \cdot R)^* & Dicyclicity \\
 & & \cdot(\epsilon \mid (R \mid S) \cdot ((R \cdot S^*)^0 \mid \ldots \mid (R \cdot S^*)^n)) &
\end{array}
$$

Figure 2.1: Conway's equational theory for regular expressions.

$R^{*!}$ if, for all $n, m \in \mathbb{N}$ and for all strings $s_1, \ldots, s_n, t_1, \ldots, t_m \in \mathcal{L}(R)$, whenever $s_1 \cdot \ldots \cdot s_n = t_1 \cdot \ldots \cdot t_m$ it is the case that $n = m$ and $s_i = t_i$ for all i.

A regular expression R is *strongly unambiguous* if one of the following holds: (a) $R = s$, or (b) $\mathcal{L}(R) = \{\}$, or (c) $R = R_1 \cdot R_2$ with $R_1 \cdot^! R_2$, or (d) $R = R_1 \mid R_2$ with $R_1 \cap R_2 = \emptyset$, or (e) $R = (R')^*$ with $(R')^{*!}$. Moreover, in the recursive cases, R_1, R_2, and R' must also be strongly unambiguous.

2.0.3 Regular Expression Equivalences

R and S are *equivalent*, written $R \equiv S$, if $\mathcal{L}(R) = \mathcal{L}(S)$. There exists an equational theory for determining whether two regular expressions are equivalent, presented by Conway [11], and proven complete by Krob [31] These axioms are shown in Figure 2.1.

While this equational theory is complete, naïvely using it in the context of lens synthesis presents several problems. In the context of lens synthesis, we instead use the equational theory corresponding to the axioms of a *star semiring* [13], shown in

$$
\begin{array}{rcll}
R \mid \emptyset & \equiv^s & R & +\ Ident \\
R \cdot \emptyset & \equiv^s & \emptyset & 0\ Proj_R \\
\emptyset \cdot R & \equiv^s & \emptyset & 0\ Proj_L \\
(R \cdot R') \cdot R'' & \equiv^s & R \cdot (R' \cdot R'') & \cdot\ Assoc \\
(R \mid R') \mid R'' & \equiv^s & R \mid (R' \mid R'') & \mid\ Assoc \\
R \mid S & \equiv^s & S \mid R & \mid\ Comm \\
R \cdot (R' \mid R'') & \equiv^s & (R \cdot R') \mid (R \cdot R'') & Dist_R \\
(R' \mid R'') \cdot R & \equiv^s & (R' \cdot R) \mid (R'' \cdot R) & Dist_L \\
\epsilon \cdot R & \equiv^s & R & \cdot\ Ident_L \\
R \cdot \epsilon & \equiv^s & R & \cdot\ Ident_R \\
R^* & \equiv^s & \epsilon \mid (R \cdot R^*) & Unrollstar_L \\
R^* & \equiv^s & \epsilon \mid (R^* \cdot R) & Unrollstar_R
\end{array}
$$

Figure 2.2: Star-semiring equivalences.

Figure 2.2. If two regular expressions are equivalent within this equational theory, they are *star semiring equivalent*, written $R \equiv^s S$. The star semiring axioms consist of the semiring axioms plus the following rules for equivalences involving the Kleene star: In Chapter 3, we provide intuition for why synthesis with full regular expression equivalence is problematic and justify our choice of using star semiring equivalence instead.

There exist other alternative axiomatizations of regular expression equivalences, such as Kozen's [30] and Salomaa's [55]. Kozen and Salomaa's axiomatizations are not equational theories: applying certain inference rules requires that side conditions must be satisfied. Consequently, using these axiomatizations does not permit a simple search strategy – our algorithm could no longer merely apply rewrite rules because it would need to confirm that the side conditions are satisfied. To avoid these complications (though in doing so, we accept the alternative complications of an infinitary theory), we focus on Conway's equational theory.

Chapter 3

Synthesizing Bijective Lenses

3.1 Introduction

Bijective lenses are the subset of lenses that are also bijections. This means that any change in one format reflects a change in the other format. We begin describing bijective lenses to develop the overall algorithm, and will show how to synthesize more complex lenses, like symmetric and quotient lenses, by building off the high-level approach used in bijective lenses. While bijective lenses are the most restrictive class of lenses, they still are quite useful. For example, name_swap introduced in §1 is a bijective lens.

```
let name = [A−Z][a−z]∗
let wsp  = [ \t\r\n\f]+
let lfm = name . "," (wsp . name)∗
let fml = (name . wsp)∗ . name
let name_swap : (lens in fml <=> lmf) =
  swap
    (id(name) . del "," )
    (swap id(wsp) id(name))∗
```

This lens is bijective, as each string in lfm corresponds to a unique string in fml. However, small changes to the formats can remove the bijective correspondence. Consider altering lfm to not permit arbitrary whitespace between names, but rather to enforce a single space:

```
let lfm = name . "," (" " . name)*
```

With this small change, the desired lens of type fml <=> lmf is no longer bijective.

This chapter will introduce the language of bijective lenses, and provide more detailed information about the general difficulties of lens synthesis. We will show how we overcome these difficulties by developing a synthesis for an alternative, equivalent language: DNF lenses. We evaluate our bijective synthesis algorithm on 39 benchmarks taken from the lens and synthesis literature, and find that we can synthesize all the bijections in under 5 seconds. Most of the content of this chapter comes from the paper "Synthesizing Bijective Lenses" [42].

3.2 Bijective Lens Language

Technically, all bijections between languages are considered lenses. We define bijective lenses to be bijections created from the following Boomerang lang combinators, ℓ.

$$
\begin{aligned}
\ell ::= \quad & \mathsf{const}(s_1 \in \Sigma^*, s_2 \in \Sigma^*) \\
\mid \quad & \ell^* \\
\mid \quad & \mathsf{concat}(\ell_1, \ell_2) \\
\mid \quad & \mathsf{swap}(\ell_1, \ell_2) \\
\mid \quad & \mathsf{or}(\ell_1, \ell_2) \\
\mid \quad & \ell_1 \,;\, \ell_2 \\
\mid \quad & \mathsf{id}(R)
\end{aligned}
$$

The denotation of a lens ℓ is $[\![\ell]\!] \subseteq \Sigma^* \times \Sigma^*$. If $(s_1, s_2) \in [\![\ell]\!]$, then ℓ *maps between* s_1 and s_2.

$$
\begin{aligned}
[\![\mathsf{const}(s_1, s_2)]\!] &= \{(s_1, s_2)\} \\
[\![\ell^*]\!] &= \{(s_1 \cdot \ldots \cdot s_n, t_1 \cdot \ldots \cdot t_n) \mid n \in \mathbb{N} \ \wedge \ \forall i \in [1, n], (s_i, t_i) \in [\![\ell]\!]\} \\
[\![\mathsf{concat}(\ell_1, \ell_2)]\!] &= \{(s_1 \cdot s_2, t_1 \cdot t_2) \mid (s_1, t_1) \in [\![\ell_1]\!] \ \wedge \ (s_2, t_2) \in [\![\ell_2]\!]\} \\
[\![\mathsf{swap}(\ell_1, \ell_2)]\!] &= \{(s_1 \cdot s_2, t_2 \cdot t_1) \mid (s_1, t_1) \in [\![\ell_1]\!] \ \wedge \ (s_2, t_2) \in [\![\ell_2]\!]\} \\
[\![\mathsf{or}(\ell_1, \ell_2)]\!] &= \{(s, t) \mid (s, t) \in [\![\ell_1]\!] \vee (s, t) \in [\![\ell_2]\!]\} \\
[\![\ell_1 \ ; \ell_2]\!] &= \{(s_1, s_3) \mid \exists s_2 (s_1, s_2) \in [\![\ell_1]\!] \ \wedge \ (s_2, s_3) \in [\![\ell_2]\!]\} \\
[\![\mathsf{id}(R)]\!] &= \{(s, s) \mid s \in \mathcal{L}(R)\}
\end{aligned}
$$

The simplest lens in the combinator language is the constant lens between strings s, and t, $\mathsf{const}(s, t)$. The lens $\mathsf{const}(s, t)$, when operated left-to-right, replaces the string s with t, and when operated right-to-left, replaces string t with s. It can oftentimes be useful to add strings to just one side, keeping the other as the empty string identity. For this common case, we use ins s as syntactic sugar for const("",s) and del s as syntactic sugar for const(s,""). In the name_swap lens, we can see a use of the constant lens: it deletes "," operating from left-to-right, and adds it back in when operating from right-to-left.

The identity lens on a regular expression, $\mathsf{id}(R)$, operates in both directions by applying the identity function to strings in $\mathcal{L}(R)$. In name_swap, the identity lens is used to retain name information when operating left-to-right and right-to-left.

The composition combinator, $\ell_1 \ ; \ell_2$, operates by applying ℓ_1 then ℓ_2 when operating left to right, and applying ℓ_2 then ℓ_1 when operating right to left.

Each of the other lenses manipulates structured data. For instance, $\mathsf{concat}(\ell_1, \ell_2)$ operates by applying ℓ_1 to the left portion of a string, and ℓ_2 to the right, and concatenating the results. The name_swap lens uses concatenations to build a lens that maintains names, but deletes ","s.

The combinator $\mathsf{swap}(\ell_1, \ell_2)$ does the same as $\mathsf{concat}(\ell_1, \ell_2)$ but it swaps the results before concatenating. In name_swap, the swap lens is the mechanism by which the first name appears at the end.

$$\textsc{Constant Lens}\quad \frac{s_1 \in \Sigma^* \qquad s_2 \in \Sigma^*}{\mathsf{const}(s_1, s_2) : s_1 \Leftrightarrow s_2}$$

$$\textsc{Iterate Lens}\quad \frac{\ell : R \Leftrightarrow S \qquad R^{*!} \qquad S^{*!}}{\ell^* : R^* \Leftrightarrow S^*}$$

$$\textsc{Concat Lens}\quad \frac{\ell_1 : R_1 \Leftrightarrow S_1 \quad \ell_2 : R_2 \Leftrightarrow S_2 \quad R_1 \mathbin{.^!} R_2 \qquad S_1 \mathbin{.^!} S_2}{\mathsf{concat}(\ell_1, \ell_2) : R_1 R_2 \Leftrightarrow S_1 S_2}$$

$$\textsc{Swap Lens}\quad \frac{\ell_1 : R_1 \Leftrightarrow S_1 \quad \ell_2 : R_2 \Leftrightarrow S_2 \quad R_1 \mathbin{.^!} R_2 \qquad S_2 \mathbin{.^!} S_1}{\mathsf{swap}(\ell_1, \ell_2) : R_1 R_2 \Leftrightarrow S_2 S_1}$$

$$\textsc{Or Lens}\quad \frac{\ell_1 : R_1 \Leftrightarrow S_1 \qquad \ell_2 : R_2 \Leftrightarrow S_2 \quad \mathcal{L}(R_1) \cap \mathcal{L}(R_2) = \emptyset \qquad \mathcal{L}(S_1) \cap \mathcal{L}(S_2) = \emptyset}{\mathsf{or}(\ell_1, \ell_2) : R_1 \mid R_2 \Leftrightarrow S_1 \mid S_2}$$

$$\textsc{Compose Lens}\quad \frac{\ell_1 : R_1 \Leftrightarrow R_2 \qquad \ell_2 : R_2 \Leftrightarrow R_3}{\ell_1 \,;\, \ell_2 : R_1 \Leftrightarrow R_3}$$

$$\textsc{Identity Lens}\quad \frac{R \text{ is strongly unambiguous}}{\mathsf{id}(R) : R \Leftrightarrow R}$$

$$\textsc{Rewrite Regex Lens}\quad \frac{\ell : R_1 \Leftrightarrow R_2 \qquad R_1 \equiv^s R_1' \qquad R_2 \equiv^s R_2'}{\ell : R_1' \Leftrightarrow R_2'}$$

Figure 3.1: Lens Typing Rules

The combinator $\mathsf{or}(\ell_1, \ell_2)$ operates by applying either ℓ_1 or ℓ_2 to the string, depending on which contains the string in the domain (or codomain).[1]

The combinator ℓ^* operates by repeatedly applying ℓ to subparts of a string.

Writing programs even as simple as swapping names around (as shown in Chapter 1) requires reasoning about which components must be swapped to make the last name appear at the end, and how to properly place whitespace between the names. These difficulties become even more apparent when writing the complex transformations that occur between large formats, and when ensuring lenses are well-typed.

3.2.1 Lens Typing

The typing judgement for lenses has the form $\ell : R \Leftrightarrow S$, meaning ℓ bijectively maps between $\mathcal{L}(R)$ and $\mathcal{L}(S)$. In particular, if $\ell : R \Leftrightarrow S$, then $(s, t) \in [\![\ell]\!]$ implies $s \in \mathcal{L}(R)$ and $t \in \mathcal{L}(S)$. Furthermore, if $(s, t) \in [\![\ell]\!]$ and $(s, t') \in [\![\ell]\!]$, then $t = t'$. In the same way, if $(s, t) \in [\![\ell]\!]$ and $(s', t) \in [\![\ell]\!]$, then $s = s'$. Because of this bijectivity, when $(s, t) \in [\![\ell]\!]$, we say $\ell.\mathsf{createR}\ s = t$ and $\ell.\mathsf{createL}\ t = s$. The functions, $\ell.\mathsf{createL}$ and $\ell.\mathsf{createR}$, are then inverses: $\ell.\mathsf{createL} \circ \ell.\mathsf{createR} = id$ and $\ell.\mathsf{createR} \circ \ell.\mathsf{createL} = id$.

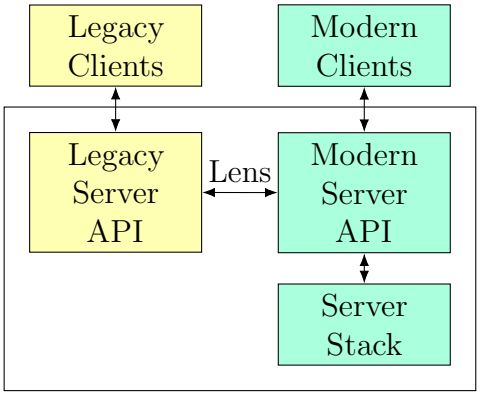
Figure 3.1 gives the typing relation. Many of the typing derivations require side conditions about unambiguity. These side conditions guarantee that the semantics of the language create a bijective function. For example, if $\ell_1 : R_1 \Leftrightarrow S_1$, and $\ell_2 : R_2 \Leftrightarrow S_2$, and R_1 is not unambiguously concatenable with R_2, then there would exist $s_1, s_1' \in \mathcal{L}(R_1)$, and $s_2, s_2' \in \mathcal{L}(R_2)$ where $s_1 \cdot s_2 = s_1' \cdot s_2'$, but $s_1 \neq s_1'$, and $s_2 \neq s_2'$. The lens $\mathsf{concat}(\ell_1, \ell_2)$ would no longer necessarily act as a function when applied from left to right, as ℓ_1 could be applied to both s_1 and to s_1'.

The typing rule for $\mathsf{id}(R)$ requires a strongly unambiguous regular expression. This unambiguity allows the identity lens to be derivable from other lenses.[2] This requirement does not, however, reduce expressiveness, as any regular expression is equivalent to a strongly unambiguous regular expression [8].

The last rule in Figure 3.1 is a type equivalence rule that lets the typing rules consider a lens $\ell : R_1 \Leftrightarrow R_2$ to have type $R_1' \Leftrightarrow R_2'$ so long as $R_1 \equiv^s R_1$ and $R_2 \equiv^s R_2'$. Notice that this rule uses star semiring equivalence as opposed to Conway equivalence. In theory, this reduces the expressiveness of the type system; in practice, we have not found it restrictive. We explain and justify the decision to use star semiring

[1] For nested concatenations and disjunctions, the terms $\mathsf{concat}(\ell_1, \ell_2)$ and $\mathsf{or}(\ell_1, \ell_2)$ are syntactically heavy. Instead, these can be written in the infix style of $\ell_1.\ell_2$ (for concatenations) and $\ell_1|\ell_2$ (for disjunctions).

[2] In practice, we allow regular expressions that aren't strongly unambiguous to appear in $\mathsf{id}(R)$, provided that they are expressed as a user defined regular expression. We elide such user-defined regular expression information from the theory for the sake of simplicity.



(a)

Legacy Work Item Representation	Modern Work Item Representation

<WorkItem>
<Field Id=1>Bug</Field>
<Field Id=2>Return 400 on bad PUT request</Field>
<Field Id=5>Smith, J</Fiel
</WorkItem>
	{
	Title: Return 400 on bad PUT request,
	Type: Bug,
	AssignedTo: J Smith,
	}

(b)

Figure 3.2: VSTS Architecture Using Lenses. In (a), we show the proposed architecture of VSTS using lenses. When a legacy client requests a work item, the server retrieves the data in a modern format through the new APIs, then the lens converts it into a legacy format to return to the client. When a legacy client updates a work item, it provides the data in the legacy format to the server. The lens then converts this data into the modern format for the new endpoints to process. Idealized Task representations from legacy and modern web service endpoints are given in (b).

equivalence in the next section. Furthermore, we find this system sufficient in practice: the name_swap lens, as well as all the lenses in our benchmark suite described in Section 3.7, are well-typed according to these typing rules.

It is worthwhile at this point to notice that the problem of finding a well-typed lens ℓ given a pair of regular expressions—the lens synthesis problem—would not be difficult if it were not for lens composition and the type equivalence rules. When read bottom up, these two rules require wild guesses at additional regular expressions to continue driving synthesis recursively in a type-directed fashion. In contrast, in the other rules, the shape of the lens is largely determined by the given types. The following sections elaborate on this problem and describe our solution.

3.3 Synthesis Overview

To highlight the difficulties in synthesizing lenses, we use an extended example inspired by the evolution of Microsoft's Visual Studio Team Services (VSTS). VSTS

is a collection of web services for team management – providing a unified location for source control (e.g. a Git server), task management (i.e. providing a means to keep track of TODOs and bugs), and more. In 2014, to increase third party developer interaction, VSTS released new web service endpoints [22]. However, despite VSTS introducing new, modernized web APIs, they must still maintain the old, legacy web APIs for continued support of legacy clients [40]. Instead of maintaining server code for each endpoint, we envision an architecture that uses a lens to convert resources of the old form into resources of the new form and vice versa, as shown in Figure 3.2a. Writing each of these converters by hand is slow and error prone. Optician expedites this process by only requiring users to input regular expressions and input-output examples. Furthermore, the lenses it generates are guaranteed to map between the provided regular expressions and to act correctly on the provided examples.

Consider a "Work Item," a resource that represents a task given to a team. Figure 3.2b shows an example work item in idealized legacy and modern formats. While the two representations contain the same information, they are presented differently – clients that expect one representation would fail if given the other. In our proposed architecture, if an old client performs an HTTP GET request to receive a work item, the server first retrieves that work item using the modern API, and then uses the lens's putL function to convert this task into the legacy format. Similarly, if an old client performs an HTTP PUT request to update a work item, the server first uses the lens's putR function to convert that data into the new format, and then inputs the work item in the new representation to the modern APIs.

For simplicity, let's consider finding only the mapping between the "Title" field of the legacy and modern work item formats. The legacy client accepts inputs of the form

```
let legacy_title = "<Field Id=2>" . text_char* . "</Field>"
```

while the modern client accepts inputs of the form

```
let modern_title = ("Title:" . text_char* . text_char . ",")
                 | ""
```

where text_char is a user-defined data type representing what characters can be present in a text field (like the title field). We would like to be able to synthesize ℓ, a lens that satisfies the typing judgement ℓ : legacy_title $\Leftrightarrow$ modern_title (i.e. ℓ maps between the legacy representation legacy_title, and the modern representation modern_title). Because the modern API omits the title field if it is blank, the lens must perform different actions depending on the number of text characters present, functionality provided by or lenses. An or lens applies one of two lenses, depending on which of the lenses' source types matches the input string.

However, the typing rule for or does not suffice to type check lenses that map between legacy_title and modern_title. While modern_title is a regular expression with an outermost disjunction, legacy_title is a regular expression with an outermost concatenation, so the rule cannot be immediately applied. We address this problem by allowing conversions between equivalent regular expression types with the type equivalence rule. Using this rule, a type-directed synthesis algorithm can convert legacy_title into

```
let legacy_title' = "<Field␣Id=2></Field>"
                  | ("<Field␣Id=2>" . text_char . text_char* . "</Field>")
```

There exist or lenses between legacy_title' and modern_title, and the two cases of an empty and a nonempty number of text characters can be handled separately. However, the need to find this equivalent type highlights a significant challenge in synthesizing bijective lenses.

Challenge 1: Multi-dimensional Search Space. Since regular expression equivalence is decidable, it is easy to *check* whether a given lens ℓ with type $R_1 \Leftrightarrow R_2$ also has type $R_1' \Leftrightarrow R_2'$. During synthesis, however, deciding when and how to use type

conversion is difficult because there are infinitely many regular expressions that are equivalent to the source and target regular expressions. Does the algorithm need to consider all of them? In what order? To convert from legacy_title to legacy_title', the algorithm must first unroll text_char* into "" | text_char text_char*, and then it must distribute this disjunction on the left and the right.

A related challenge arises from the composition operator, $\ell_1 \; ; \; \ell_2$. The typing rule for composition requires that the target type of l_1 be the source type of l_2. To synthesize a composition lens between R_1 and R_3, a sound synthesizer must find an intermediate type R_2 and lenses with types $R_1 \Leftrightarrow R_2$ and $R_2 \Leftrightarrow R_3$. Searching for the correct regular expression R_2 is again problematic because the search space is infinite.

Thus, naïvely applying type-directed synthesis techniques involves searching *in three infinite dimensions*. A complete naïve synthesizer must search for (1) a *type* consisting of two regular expressions equivalent to the given ones but with "similar shapes" and (2) a lens *expression* that has the given type and is consistent with the user's examples. Furthermore, whenever composition is part of the expression, naïve type-directed synthesis requires a further search for (3) an *intermediate regular expression*.

Our approach to this challenge is to define a new "DNF syntax" for regular languages and lenses that reduces the synthesis search space *in all dimensions*. In this new language, regular expressions are written in a disjunctive normal form, where disjunctions are fully distributed over concatenation and where binary operators are replaced by n-ary ones, eliminating associativity rules. Using DNF regular expressions, when presented with a synthesis problem with type $(A|B)C \Leftrightarrow A'C'|B'C'$, Optician will first convert this type into $\langle [A \cdot C] \mid [B \cdot C] \rangle \Leftrightarrow \langle [A' \cdot C'] \mid [B' \cdot C'] \rangle$, where $\langle \ldots \rangle$ represents n-ary disjunction and $[\ldots]$ represents n-ary concatenation. Like DNF regular expressions, DNF lenses are stratified, with disjunctions outside of concatenations, and they use n-ary operators instead of binary ones. Furthermore, DNF lenses do not

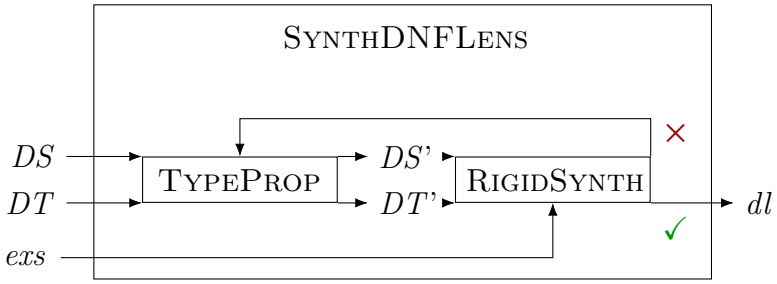


Figure 3.3: Schematic Diagram for DNF Lens Synthesis Algorithm. DNF regular expressions, DS and DT, and a set of examples *exs* are given as input. The synthesizer, TypeProp, uses these input DNF regular expressions to propose a pair of equivalent DNF regular expressions, DS' and DT'. The synthesizer RigidSynth then attempts to generate a DNF lens, dl, which goes between DS' and DT' and satisfies all the examples in *exs*. If RigidSynth is successful, dl is returned. If RigidSynth is unsuccessful, information of the failure is returned to TypeProp, which continues proposing candidate DNF regular expressions until RigidSynth finds a satisfying DNF lens.

need a composition operator, eliminating an entire dimension of search. This dual stratification and the lack of composition creates a very tight relationship between the structure of a DNF lens term and the DNF regular expression pair that forms its type.

Translating regular expressions into DNF form collapses many equivalent REs into the same syntactic form. However, this translation does not fully normalize regular expressions. Nor do we want it to: If a synthesizer normalized $\epsilon \mid BB* \text{ to } B*$, it would have trouble synthesizing lenses with types like $\epsilon \mid BB* \Leftrightarrow \epsilon \mid CD*$ where the first occurrence of B on the left needs to be transformed into C while the rest of the Bs need to be transformed into D. Normalization to DNF eliminates many, but not all, of the regular expression equivalences that may be needed before a simple, type-directed structural search can be applied—i.e., DNF regular expressions are only *pseudo-canonical*.

Consequently, a type-directed synthesis algorithm must still search through some equivalent regular expressions. To handle this search, SynthDNFLens is structured as two communicating synthesizers, shown in Figure 3.3. The first synthesizer, TypeProp, proposes DNF regular expressions equivalent to the input DNF regular

expressions. TypeProp uses the axioms of a star semiring to unfold Kleene star operators in one or both types, to obtain equivalent (but larger) DNF regular expression types. The second synthesizer, RigidSynth, performs a syntax-directed search based on the structure of the provided DNF regular expressions, as well as the input examples. If the second synthesizer finds a satisfying DNF lens, it returns that lens. If the second synthesizer fails to find such a lens, TypeProp learns of that failure, and proposes new candidate DNF regular expression pairs.

Star Semiring Equivalence and Rewriting One could try to search the space of DNF regular expressions equivalent to the input regular expressions by turning the Conway axioms into (undirected) rewrite rules operating on DNF regular expressions and then trying all possible combinations of rewrites. Doing so would be problematic because the Conway axiomatization itself is infinitely branching (due to the choice of n in the dyclicity axiom).

We also want DNF lenses to be closed under composition – if they are not then we need to be able to synthesize lenses containing composition operators. To be closed under composition, it is sufficient for the equivalence relation used in the type equivalence rule to be the equivalence closure of a rewrite system ($\rightarrow$) satisfying four conditions. First, if $R \rightarrow R'$, then $\mathcal{L}(R) = \mathcal{L}(R')$. Second, if $R \rightarrow R'$ and R is strongly unambiguous, then R' is also strongly unambiguous. The remaining two properties relate the rewrite rules to the typing derivations of DNF lenses, when those typing derivations do not use type equivalence. To express these properties, we use the notation $dl \,\tilde{:}\, DS \Leftrightarrow DT$ to mean that if dl is a DNF lens that goes between DNF regular expressions DS and DT, then the typing derivation contains no instances of the type equivalence rule. Using this notation, we can express the *confluence* property, as follows:

Definition 1 (Confluence)**.** A set of rewrite rules on regular expressions, $\rightarrow$, is confluent, if whenever $dl_1 \mathbin{:}\!\sim (\Downarrow R_1) \Leftrightarrow (\Downarrow S_1)$[3], if $R_1 \rightarrow R_2$ and $S_1 \rightarrow S_2$, then there exist regular expressions R_3, and S_3 and a DNF lens dl_3, such that:

1. $R_2 \rightarrow R_3$
2. $S_2 \rightarrow S_3$
3. $dl_3 \mathbin{:}\!\sim (\Downarrow R_3) \Leftrightarrow (\Downarrow S_3)$
4. $[\![dl_3]\!] = [\![dl_1]\!]$

We call the final property *bisimilarity*. Bisimilarity requires two symmetric conditions.

Definition 2 (Bisimilarity)**.** Whenever $dl_1 \mathbin{:}\!\sim (\Downarrow R_1) \Leftrightarrow (\Downarrow S_1)$ and $R_1 \rightarrow R_2$, there exist a regular expression S_2 and a DNF lens dl_2 such that

1. $S_1 \rightarrow S_2$
2. $dl_2 \mathbin{:}\!\sim (\Downarrow R_2) \Leftrightarrow (\Downarrow S_2)$
3. $[\![dl_2]\!] = [\![dl_1]\!]$

For the rewrites to be bisimilar, the symmetric property must also hold for $S_1 \rightarrow S_2$.

Our solution for handling type equivalence is to use $\equiv^s$, the equivalence relation generated by the axioms of a star semiring. This equivalence relation is compatible with our lens synthesis strategy, as orienting these unrolling rules from left to right presents us with a rewrite relation that is both confluent and bisimilar, and whose equivalence closure is $\equiv^s$. The star semiring axioms are the most coarse subset of regular expression equivalences we could find that is generated by a rewrite relation and is still confluent and bisimilar. We have not been able to prove that this relation

[3]Recall that $\Downarrow$ converts a regular expression into DNF form.

is the coarsest such relation possible, but it is sufficient to cover all the test cases in our benchmark suite (see §3.7). However, it is easy to show that Conway's axioms (*Prodstar* in particular) are not bisimilar, which is why we avoid this in our system.

Challenge 2: Large Types DNF lenses are equivalent in expressivity to lenses and the algorithm SynthDNFLens is quite fast. Unfortunately, the conversion to DNF incurs an exponential blowup. In practical examples, the regular expressions describing complex *ad hoc* data formats may be very large, causing the exponential blowup to have a significant impact on synthesis time. The key to addressing this issue is to observe that users naturally construct large types incrementally, introducing named abbreviations for major subcomponents. For example, in the specification of legacy_title and modern_title, the variable text_char describes which characters can be present in a title. To include a large disjunction representing all valid title characters instead of the concise variable text_char in the definitions of legacy_title and modern_title would be unmaintainable and difficult to read.

Unfortunately, leaving these variables opaque introduces a new dimension of search. In addition to searching through the rewrites on regular expressions, the algorithm must also search through possible *substitutions*, replacements of variables with their definitions. We designate these two types of equivalences *expansions*, using "rewrites" to denote expansions that arise from traversing rewrite rules on the regular expressions, and using "substitutions" to denote expansions that arise from replacing a variable with its definition.

Interestingly, Optician can exploit the structure inherent in these named abbreviations to speed up the search dramatically. For example, if text_char appears just once in both the source and the target types, the system hypothesizes that the identity lens can be used to convert between occurrences of text_char. On the other hand, if text_char appears in the source but not in the target, the system recognizes that, to

find a lens, text_char must be replaced by its definition. In this way, the positions of names can serve as a guide for applying substitutions and rewrites in the synthesis algorithm. By using these named abbreviations, TypeProp guides the transformation of regular expression types during search by deducing when certain expansions must be taken, or when one of a class of expansions must be taken.

3.4 DNF Regular Expressions

The first important step in Optician is to convert regular expression types into *disjunctive normal form* (DNF). A DNF regular expression, abbreviated DNF RE, is an n-ary disjunction of sequences, where a sequence alternates between concrete strings and atoms, and an atom is an iteration of DNF regular expressions. The grammar below describes the syntax of DNF regular expressions (DS,DT), sequences (SQ,TQ), and atoms (A, B) formally.

$$\begin{array}{lcl} A,B & ::= & DS^* \\ SQ,TQ & ::= & [s_0 \cdot A_1 \cdot \ldots \cdot A_n \cdot s_n] \\ DS,DT & ::= & \langle SQ_1 \mid \ldots \mid SQ_n \rangle \end{array}$$

Notice that it is straightforward to convert an arbitrary series of atoms and strings into a sequence: if there are multiple concrete strings between atoms, the strings may be concatenated into a single string. If there are multiple atoms between concrete strings, the atoms may be separated by empty strings, which will sometimes be omitted for readability. Notice also that a simple string with no atoms may be represented as a sequence containing just one concrete string. In our bijective lens synthesis implementation, names of user-defined regular expressions are also atoms. However, we elide such definitions from our theoretical analysis.

$\odot_{SQ} : \mathit{Sequence} \to \mathit{Sequence} \to \mathit{Sequence}$
$[s_0 \cdot A_1 \cdot \ldots \cdot A_n \cdot s_n] \odot_{SQ} [t_0 \cdot B_1 \cdot \ldots \cdot B_m \cdot t_m] = [s_0 \cdot A_1 \cdot \ldots \cdot A_n \cdot s_n \cdot t_0 \cdot B_1 \cdot \ldots \cdot B_m \cdot t_m]$

$\odot : \mathit{DNF} \to \mathit{DNF} \to \mathit{DNF}$
$\langle SQ_1 \mid \ldots \mid SQ_n \rangle \odot \langle TQ_1 \mid \ldots \mid TQ_m \rangle =$
$\langle SQ_1 \odot_{SQ} TQ_1 \mid \cdots \mid SQ_1 \odot_{SQ} TQ_m \mid \cdots \mid SQ_n \odot_{SQ} TQ_1 \mid \cdots \mid SQ_n \odot_{SQ} TQ_m \rangle$

$\oplus : \mathit{DNF} \to \mathit{DNF} \to \mathit{DNF}$
$\langle SQ_1 \mid \ldots \mid SQ_n \rangle \oplus \langle TQ_1 \mid \ldots \mid TQ_m \rangle = \langle SQ_1 \mid \ldots \mid SQ_n \mid TQ_1 \mid \ldots \mid TQ_m \rangle$

$\mathcal{D} : \mathit{Atom} \to \mathit{DNF}$
$\mathcal{D}(A) = \langle [\epsilon \cdot A \cdot \epsilon] \rangle$

Figure 3.4: DNF Regular Expression Functions

Intuitions about DNF regular expressions may be confirmed by their semantics, which we give by defining the language (set of strings) that each DNF regular expression denotes:

$$\begin{aligned} \mathcal{L}(DS^*) &= \{s_1 \cdot \ldots \cdot s_n \mid \forall i s_i \in \mathcal{L}(DS)\} \\ \mathcal{L}([s_0 \cdot A_1 \cdot \ldots \cdot A_n \cdot s_n]) &= \{s_0 \cdot t_1 \cdots t_n \cdot s_n \mid t_i \in \mathcal{L}(A_i)\} \\ \mathcal{L}(\langle SQ_1 \mid \ldots \mid SQ_n \rangle) &= \{s \mid s \in \mathcal{L}(SQ_i) \text{ and } i \in [1, n]\} \end{aligned}$$

A sequence of strings and atoms is *sequence unambiguously concatenable*, written $\cdot^!(s_0; A_1; \ldots; A_n; s_n)$, if, when $s_i', t_i' \in \mathcal{L}(A_i)$ for all i, then $s_0 s_1' \ldots s_n' s_n = s_0 t_1' \ldots t_n' s_n$ implies $s_i' = t_i'$ for all i. A DNF regular expression R is *unambiguously iterable*, written $R^{*!}$ if, for all $n, m \in \mathbb{N}$ and for all strings $s_1, \ldots, s_n, t_1, \ldots, t_m \in \mathcal{L}(R)$, if $s_1 \cdot \ldots \cdot s_n = t_1 \cdot \ldots \cdot t_m$ then $n = m$ and $s_i = t_i$ for all i.

Expressivity of DNF Regular Expressions Any regular expression may be converted into an equivalent DNF regular expression. To define the conversion function, we rely on several auxiliary functions defined in Figure 3.4. Intuitively, $DS \odot DS$ concatenates two DNF regular expressions, producing a well-formed DNF regular expression as a result. Similarly, $DS \oplus DS$ generates a new DNF regular expression representing the union of two DNF regular expressions. Finally, $\mathcal{D}(A)$ converts a naked atom into a well-formed DNF regular expression. The conversion

algorithm itself, written $\Downarrow R$, is defined below.

$$\begin{aligned}
\Downarrow s &= \langle [s] \rangle \\
\Downarrow \emptyset &= \langle \rangle \\
\Downarrow (R^*) &= \mathcal{D}((\Downarrow R)^*) \\
\Downarrow (R_1 \cdot R_2) &= \Downarrow R_1 \odot \Downarrow R_2 \\
\Downarrow (R_1 \mid R_2) &= \Downarrow R_1 \oplus \Downarrow R_2
\end{aligned}$$

Using $\Downarrow$, the definition of legacy_title' gets converted into the DNF regular expression:

```
dnf_legacy_title =
  ⟨[ "<Field ␣ Id=2></Field>" ]
  | [ "<Field ␣ Id=2>" · text_char · "" · ⟨ [ text_char ] ⟩∗ · "<Field ␣ Id=2>" ]⟩
```

and the definition of modern_title gets converted into the DNF regular expression:

```
dnf_modern_title =
  ⟨[ "Title:" · text_char · "" · ⟨[ text_char ]⟩∗ · "," ]
  | [""]⟩
```

We formalize the correspondence between regular expressions and DNF regular expressions via the following theorem.

Theorem 1 ($\Downarrow$ Soundness)**.** For all regular expressions R, $\mathcal{L}(\Downarrow R) = \mathcal{L}(R)$.

Note that the proofs for this theorem, and all subsequent theorems in this section, are contained in the full version of the "Synthesizing Bijective Lenses" paper [41].

DNF Regular Expression Rewrites There are many fewer equivalences on DNF regular expressions than there are on regular expressions, but there still remain pairs of DNF regular expressions that, while syntactically different, are semantically identical. Figure 3.5 defines a collection of rewrite rules on DNF regular expressions designed to search the space of equivalent DNF REs. This directed rewrite system helps limit

$$\textsc{Atom Unrollstar}_{\text{L}} \quad \frac{}{DS^* \rightarrow_A \langle [\epsilon] \rangle \oplus (DS \odot \mathcal{D}(DS^*))} \qquad \textsc{Atom Unrollstar}_{\text{R}} \quad \frac{}{DS^* \rightarrow_A \langle [\epsilon] \rangle \oplus (\mathcal{D}(DS^*) \odot DS)}$$

$$\textsc{Atom Structural Rewrite} \quad \frac{DS \rightarrow DS'}{DS^* \rightarrow_A \langle [DS'^*] \rangle}$$

$$\textsc{DNF Structural Rewrite} \quad \frac{A_j \rightarrow_A DS}{\begin{array}{c} \langle SQ_1 \mid \ldots \mid SQ_{i-1} \rangle \\ \oplus \langle [s_0 \cdot A_1 \cdot \ldots \cdot s_{j-1}] \rangle \odot \mathcal{D}(A_j) \odot \langle [s_j \cdot \ldots \cdot A_m \cdot s_m] \rangle \oplus \\ \langle SQ_{i+1} \mid \ldots \mid SQ_n \rangle \\ \rightarrow \\ \langle SQ_1 \mid \ldots \mid SQ_{i-1} \rangle \\ \oplus \langle [s_0 \cdot A_1 \cdot \ldots \cdot s_{j-1}] \rangle \odot DS \odot [s_j \cdot \ldots \cdot A_m \cdot s_m] \oplus \\ \langle SQ_{i+1} \mid \ldots \mid SQ_n \rangle \end{array}}$$

Figure 3.5: DNF Regular Expression Rewrite Rules

the search space more than the non-directional equivalence $\equiv^s$ relation. Because the rewrite rules are confluent, it is just as powerful as the $\equiv^s$ relation.

Because disjunctive normal form flattens a series of unions or concatenations into an n-ary sum-of-products, there is no need for rewriting rules that manage associativity or distributivity. Moreover, the lens term language and synthesis algorithm itself manages out-of-order summands, so we also have no need of rewriting rules to handle commutativity of unions. Hence, the rewriting system need only focus on rewrites that involve Kleene star. The rule Atom Unrollstar$_{\text{L}}$ is a directed rewrite rule designed to mirror *Unrollstar*$_L$. Intuitively, it unfolds any atom DS^* into $\epsilon \mid (DS \cdot DS^*)$. However, $\epsilon \mid (DS \cdot DS^*)$ is not a *DNF* regular expression. Hence, the rule uses DNF concatenation ($\odot$) and union ($\oplus$) in place of regular expression concatenation and union to ensure a DNF regular expression is constructed. The rule Atom Unrollstar$_{\text{R}}$ mirrors the rule *Unrollstar*$_R$ in a similar way.

The rules Atom Structural Rewrite and DNF Structural Rewrite make it possible to rewrite terms involving Kleene star that are nested deep within a DNF regular expression, while ensuring that the resulting term remains in DNF form.

3.5 DNF Lenses

The syntax of DNF lenses (dl), sequence lenses (sql) and atom lenses (al) is defined below. DNF lenses and sequence lenses both contain permutations (σ) that describe how these lenses transform their subcomponents.

$$\begin{aligned} al &::= dl^* \\ sql &::= ([(s_0, t_0) \cdot al_1 \cdot \ldots \cdot al_n \cdot (s_n, t_n)], \sigma) \\ dl &::= (\langle sql_1 \mid \ldots \mid sql_n \rangle, \sigma) \end{aligned}$$

A DNF lens consists of a list of sequence lenses and a permutation. Much like a DNF regular expression is a list of disjuncted sequences, a DNF lens contains a list of ored sequence lenses. DNF lenses also contain a permutation σ that provides information about which sequences are mapped to which by the internal sequence lenses. As an example, consider a DNF lens that maps between data with type dnf_legacy_title and data with type dnf_modern_title. In such a lens, the permutation σ indicates whether data matching ["<Field␣Id=2></Field>"] will be translated to data matching ["Title:" · text_char · "" · ⟨[text_char]⟩* · ","] or [""], and likewise for the other sequence in dnf_legacy_title. In this case, we would use the permutation that swaps the order, as the first sequence in dnf_legacy_title gets mapped to the second in dnf_modern_title, and vice-versa. As we will see in a moment, these permutations make it possible to construct a well-typed lens between two DNF regular expressions regardless of the order in which clauses in a DNF regular expression appear, thereby eliminating the need to consider equivalence modulo commutativity of these clauses.

A sequence lens consists of a list of atom lenses separated by pairs of strings, and a permutation. Intuitively, much like a sequence is a list of concatenated atoms

and strings, a sequence lens is a list of concatenated atom lenses and string pairs. Sequence lenses also contain a permutation that makes createL and createR reorder data, allowing sequence lenses to take the job of both concat and swap. If there are n elements in the series then the DNF sequence lens divides an input string up into n substrings. The i^{th} such substring is transformed by the i^{th} element of the series. More precisely, if that i^{th} element is an atom lens, then the i^{th} substring is transformed according to that atom lens. If the i^{th} element is a pair of strings (s_1, s_2) then that pair of strings acts like a constant lens: when used from left-to-right, such a lens translates string s_1 into s_2; when used from right-to-left, such a lens translates string s_2 into s_1. After all of the substrings have been transformed, the permutation describes how to rearrange the substrings transformed by the atom lenses to obtain the final output. As an example, consider a sequence lens that maps between data with type ["Title:" · text_char · "" · ⟨[text_char]⟩* · ","] and data with type ["<Field␣Id=2>" · text_char · "" · ⟨[text_char]⟩* · "<Field␣Id=2>"]. We desire no reorderings between the atoms text_charand ⟨[text_char]⟩*, so the permutation associated with this lens would be the identity permutation.

An atom lens is an iteration of a DNF lens; its semantics is similar to the semantics of ordinary iteration lenses. In our implementation, identity transformations between user-defined regular expressions are also atom lenses. However, we elide such definitions from our theoretical analysis.

The semantics of DNF Lenses, sequence lenses and atom lenses are defined formally below.

$$\begin{aligned}
[\![dl^*]\!] ::= \ & \{(s_1 \cdot \ldots \cdot s_n, t_1 \cdot \ldots \cdot t_n) \\
& \quad \mid n \in \mathbb{N} \ \wedge \ (s_i, t_i) \in [\![dl]\!]\} \\
[\![([(s_0, t_0) \cdot al_1 \cdot \ldots \cdot al_n \cdot (s_n, t_n)], \sigma)]\!] ::= \ & \{(s_0 s'_1 \ldots s'_n s_n, t_0 t'_{\sigma(1)} \ldots t'_{\sigma(n)} t_n) \\
& \quad \mid (s'_i, t'_i) \in [\![al_i]\!]\} \\
[\![(\langle sql_1 \mid \ldots \mid sql_n \rangle, \sigma)]\!] ::= \ & \{(s, t) \mid (s, t) \in sql_i \text{ for some } i\}
\end{aligned}$$

$$\frac{dl \,\tilde{:}\, DS \Leftrightarrow DT \qquad DS^{*!} \qquad DT^{*!}}{dl^* \,\tilde{:}\, DS^* \Leftrightarrow DT^*}$$

$$\frac{\begin{array}{c} al_1 \,\tilde{:}\, A_1 \Leftrightarrow B_1 \qquad \ldots \qquad al_n \,\tilde{:}\, A_n \Leftrightarrow B_n \\ \sigma \in S_n \qquad \cdot^!(s_0; A_1; \ldots; A_n; s_n) \qquad \cdot^!(t_0; B_{\sigma(1)}; \ldots; B_{\sigma(n)}; t_n) \end{array}}{([(s_0, t_0) \cdot al_1 \cdot \ldots \cdot al_n \cdot (s_n, t_n)], \sigma) \,\tilde{:}\, [s_0 \cdot A_1 \cdot \ldots \cdot A_n \cdot s_n] \Leftrightarrow [t_0 \cdot B_{\sigma(1)} \cdot \ldots \cdot B_{\sigma(n)} \cdot t_n]}$$

$$\frac{\begin{array}{c} sql_1 \,\tilde{:}\, SQ_1 \Leftrightarrow TQ_1 \qquad \ldots \qquad sql_n \,\tilde{:}\, SQ_n \Leftrightarrow TQ_n \\ \sigma \in S_n \qquad i \neq j \Rightarrow \mathcal{L}(SQ_i) \cap \mathcal{L}(SQ_j) = \emptyset \qquad i \neq j \Rightarrow \mathcal{L}(TQ_i) \cap \mathcal{L}(TQ_j) = \emptyset \end{array}}{(\langle sql_1 \mid \ldots \mid sql_n \rangle, \sigma) \,\tilde{:}\, \langle SQ_1 \mid \ldots \mid SQ_n \rangle \Leftrightarrow \langle TQ_{\sigma(1)} \mid \ldots \mid TQ_{\sigma(n)} \rangle}$$

$$\frac{DS' \rightarrow^* DS \qquad DT' \rightarrow^* DT \qquad dl \,\tilde{:}\, DS \Leftrightarrow DT}{dl : DS' \Leftrightarrow DT'}$$

Figure 3.6: DNF Lens Typing

Type Checking Figure 3.6 presents the type checking rules for DNF lenses. In order to control where regular expression rewriting may be used (and thereby reduce search complexity), the figure defines two separate typing judgements. The first judgement has the form $dl \,\tilde{:}\, DS \Leftrightarrow DT$. If $dl \,\tilde{:}\, DS \Leftrightarrow DT$, then not only is $[\![dl]\!]$ a bijective mapping between $\mathcal{L}(DS)$ and $\mathcal{L}(DT)$, but the terms and types are all well aligned – if $dl \,\tilde{:}\, DS \Leftrightarrow DT$ then there must be the same number of sequence lenses in dl as there are sequences in each of DS and DT (and similarly for their subcomponents). The second judgement has the form $dl : DS \Leftrightarrow DT$. If $dl : DS \Leftrightarrow DT$, then $[\![dl]\!]$ is a bijective mapping between DS and DT. However, because this judgement allows for rewriting in its derivation, the terms and types may not be aligned.

One of the key differences between these typing judgements and the judgements for ordinary lenses are the permutations. For example, in the rule for typing DNF lenses, the permutation σ indicates how to match sequence types in the left-hand format $(SQ_1 \ldots SQ_n)$ and the right-hand format $(TQ_{\sigma(1)} \ldots TQ_{\sigma(n)})$. Permutations are used in a similar way in the typing rule for sequence lenses.

These DNF lenses only express bijections that are already expressible in the language of lenses, and they can express everything expressible in the language of lenses.

Properties While DNF lenses have a restrictive syntax, they remain as powerful as ordinary bijective lenses. The following theorems characterize the relationship between the two languages.

Theorem 2 (DNF Lens Soundness)**.** If there exists a derivation of $dl : DS \Leftrightarrow DT$, then there exist a lens, $\Uparrow dl$, and regular expressions, R and S, such that $\Uparrow dl : R \Leftrightarrow S$ and $\Downarrow R = DS$ and $\Downarrow S = DT$ and $[\![\Uparrow dl]\!] = [\![dl]\!]$.

Theorem 3 (DNF Lens Completeness)**.** If there exists a derivation for $\ell : R \Leftrightarrow S$, then there exists a DNF lens dl such that $dl : (\Downarrow R) \Leftrightarrow (\Downarrow S)$ and $[\![\ell]\!] = [\![dl]\!]$.

Discussion DNF lenses are significantly better suited to synthesis than regular bijective lenses. First, they contain no composition operator. Second, the use of equivalence (rewriting) is highly constrained: Rewriting may only be used once at the top-most level as opposed to interleaved between uses of the other rules. Consequently, a type-directed synthesis algorithm may be factored into two discrete steps: one step that searches for an effective pair of regular expressions and a second step that is directed by the syntax of the regular expression types that were discovered in the first step.

3.6 Algorithm

Synthesis Overview Algorithm 1 presents the synthesis procedure. SynthLens takes the source and target regular expressions R and S, and a list of examples *exs*, as input. First, SynthLens validates the unambiguity of the input regular expressions,

R and S, and confirms that they parse the input/output examples, *exs*. Next, the algorithm converts R and S into DNF regular expressions DS and DT using the $\Downarrow$ operator. It then calls SynthDNFLens on DS, DT, and the examples to create a DNF lens dl. Finally, it uses $\Uparrow$ to convert dl to a Boomerang lens. The details of $\Uparrow$ are relatively uninteresting, and so we elide them here. The $\Uparrow$ function merely converts the disjunctions to a series of ors, and converts permutations to a series of swaps and concatenations.

Algorithm 1 SynthLens

```
1:  function SynthDNFLens(DS, DT, exs)
2:      Q ← CreatePQueue((DS, DT), 0)
3:      while true do
4:          (qe, Q) ← Pop(Q)
5:          (DS', DT', e) ← qe
6:          dlo ← RigidSynth(DS', DT', exs)
7:          match dlo with
8:              | Some dl → return dl
9:              | None →
10:                 qes ← Expand(DS, DT, e)
11:                 Q ← EnqueueMany(qes, Q)

12: function SynthLens(R, S, exs)
13:     Validate(R, S, exs)
14:     (DS, DT) ← (⇓R, ⇓S)
15:     dl ← SynthDNFLens(DS, DT, exs)
16:     return ⇑dl
```

SynthDNFLens starts by creating a priority queue Q to manage the search for a DNF lens. Each element qe in the queue is a tuple of the source DNF regular expression DS', the target DNF regular expression DT', and a count e of the number of expansions needed to produce this pair of DNF regular expressions from the originals DS and DT. (Recall that an expansion is a use of a rewrite rule or the substitution of a user-defined definition for its name.) The priority of each queue element is e; DNF regular expressions that have undergone fewer expansions will get priority. The algorithm initializes the queue with DS and DT, which have an expansion count

of zero. The algorithm then proceeds by iteratively examining the highest priority element from the queue (this examination corresponds to TypeProp in Figure 3.3), and using the function RigidSynth to try to find a rewriteless DNF lens between the popped source and target DNF regular expressions that satisfy the examples *exs*. If successful, the algorithm returns the DNF lens *dl*. Otherwise, the function Expand produces a new set of candidate DNF regular expression pairs that can be obtained from *DS* and *DT* by applying various expansions to the source and target DNF regular expressions.

Expand Intelligent expansion inference is key to the efficiency of Optician. Expand, shown in Algorithm 2, codifies this inference. It makes critical use of the locations of user-defined data types, measured by their *star depth*, which is the number of nested *'s the data type occurs beneath. Star depth locations are useful because we can quickly compute the current star depths of user-defined data types (with GetCurrentSet) and the star depths of user-defined data types reachable via expansions (with GetTransitiveSet). Furthermore, the star depths of user-defined data types have the following useful property:

Property 1. If U is present at star depth i in DS and there exists a rewriteless DNF lens dl such that $dl : DS \Leftrightarrow DT$, then U is also present at star depth i in DT. The symmetric property is true if U is present at star depth i in DT.

Property 1 means that if there is a rewriteless DNF lens between two DNF regular expressions, then the same user-defined data types must be present at the same locations in both of the DNF regular expressions. We use this property to determine when certain rules must be applied and to direct the search to rules that make progress towards this required alignment.

Expand has three major components: ExpandRequired, FixProblemElts, and ExpandOnce, which we discuss in turn. ExpandRequired performs expansions

Algorithm 2 EXPAND

```
1: function ExpandRequired(DS, DT, e)
2:     CS_DS ← GetCurrentSet(DS)
3:     CS_DT ← GetCurrentSet(DT)
4:     TS_DS ← GetTransitiveSet(DS)
5:     TS_DT ← GetTransitiveSet(DT)
6:     r ← false
7:     foreach (U, i) ∈ CS_DS \ TS_DT
8:         r ← true
9:         (DS, e) ← ForceExpand(DS, U, i, e)
10:    foreach (U, i) ∈ CS_DT \ TS_DS
11:        r ← true
12:        (DT, e) ← ForceExpand(DT, U, i, e)
13:    if r then
14:        return ExpandRequired(DS, DT, e)
15:    return (DS, DT, e)

16: function FixProblemElts(DS, DT, e)
17:    CS_DS ← GetCurrentSet(DS)
18:    CS_DT ← GetCurrentSet(DT)
19:    qes ← []
20:    foreach (U, i) ∈ CS_DT \ CS_DS
21:        qes ← qes ++ Reveal(DS, U, i, e, DT)
22:    foreach (U, i) ∈ CS_DS \ CS_DT
23:        qes ← qes ++ Reveal(DT, U, i, e, DS)
24:    return qes

25: function Expand(DS, DT, e)
26:    (DS, DT, e) ← ExpandRequired(DS, DT, e)
27:    qes ← FixProblemElts(DS, DT, e)
28:    match qes with
29:        | [] → return ExpandOnce(DS, DT, e)
30:        | _ → return qes
```

that *must* be taken. In particular, if a user-defined data type U at star depth i is impossible to reach through any number of expansions on the opposite side, then that user-defined data type *must* be replaced by its definition at that depth. For example, consider trying to find a lens between $\langle[\mathsf{legacy\_title}]\rangle$ and $\langle[\mathsf{modern\_title}]\rangle$. No matter how many expansions are performed on modern_title, the user-defined type legacy_title will not be exposed because the set of possible reachable pairs of data types and star depths in modern_title is $\{(\mathsf{modern\_title}, 0), (\mathsf{text\_char}, 0), (\mathsf{text\_char}, 1)\}$. Because no number of expansions will reveal legacy_title on the right, the algorithm must replace legacy_title with its definition on the left in order to find a lens. ExpandRequired continues until it finds all forced expansions.

ExpandRequired finds all the expansions that must be performed, but it does not perform any other expansions. However, there are many situations where it is possible to infer that one of a set of expansions must be performed without forcing any individual expansion. In particular, for any pair of types that have a rewriteless lens between them, for each (user-defined type, star depth) pair (U, i) on one side, that same pair must be present on the other side. FixProblemElts identifies when there is a (U, i) pair present on only one side. After identifying these problem elements, it calls Reveal to find the expansions that will reveal this element. For example, after $\langle[\mathsf{legacy\_title}]\rangle$ has been expanded to

$$\langle[\text{"<Field Id=2>"} \cdot \langle[\mathsf{text\_char}]\rangle^* \cdot \text{"</Field>"}\,]\rangle$$

and $\langle[\mathsf{modern\_title}]\rangle$ has been expanded to

$$\langle[\text{"Title:"} \cdot \mathsf{text\_char} \cdot \text{""} \cdot \langle[\mathsf{text\_char}]\rangle^* \cdot \text{","}] \mid [\text{""}]\rangle$$

we can see that the modern expansion has an instance of text_char at depth 0, where the legacy one does not. For a lens to exist between the two types, text_char must be

revealed at star depth 0 in the legacy expansion. Revealing text_char at depth zero will give back two candidate DNF regular expressions, one from an application of Atom UnrollstarL, and one from an application of Atom UnrollstarR.

Together ExpandRequired and FixProblemElts apply many expansions, but by themselves they are not sufficient. Typically, when FixProblemElts and ExpandRequired do not find all the necessary expansions, the input data formats expect large amounts of similar information. For example, in trying to synthesize the identity transformation between "" | U | UU(U∗) and "" | U(U∗), ExpandRequired and FixProblemElts find no forced expansions. An expansion is necessary, but the set of pairs $(\{(U,0),(U,1)\})$ is the same for both sides. When this situation arises, the algorithm uses the ExpandOnce function to conduct a purely enumerative search, implemented by performing all single-step expansions.

RigidSynth The function RigidSynth, shown in Algorithm 3, implements the portion of SynthLens that generates a lens from the types and examples without using any equivalences. Intuitively, it aligns the structures of the source and target regular expressions by finding appropriate permutations of nested sequences and nested atoms, taking into account the information contained in the examples. Once it finds an alignment, it generates the corresponding lens.

For integrating the example information, RigidSynth starts by merging the parse trees of the examples into the DNF regular expressions, creating *exampled DNF regular expressions*, where *ils* is a set of int lists.

Definition 3. An exampled atom, exampled sequence, and exampled DNF regular expression are:

$$\begin{array}{lll} EA,EB & ::= & (EDS^{*},ils) \\ ESQ,ETQ & ::= & ([s_0 \cdot EA_1 \cdot \ldots \cdot EA_n \cdot s_n], ils) \\ EDS,EDT & ::= & (\langle ESQ_1 \mid \ldots \mid ESQ_n \rangle, ils) \end{array}$$

If the int list $[0, 2]$ is part of the int list set of an exampled DNF regular expression, that means that the DNF regular expression is under one stars, and it parses the second iteration of the first example. If the int list $[0, 2, 3]$ is part of the int list set of an exampled DNF regular expression, then that means it parses the third iteration of the nested star, and the second iteration of the outermost star, of the first example.

We use a NFA matching algorithm to embed the parse trees of the examples into the DNF regular expressions. The function $\{s_1, \ldots, s_n\} \in DS \rightsquigarrow EDS$ represents embedding the strings $s_1, \ldots, s_n$ into DS, creating EDS.

Searching for aligning permutations requires care, as naïvely considering all permutations between two exampled DNF regular expressions $\langle SQ_1 \mid \ldots \mid SQ_n \rangle$ and $\langle TQ_1 \mid \ldots \mid TQ_n \rangle$ would require time proportional to $n!$. A better approach is to identify elements of the source and target DNF regular expressions that match and to leverage that information to create candidate permutations.

RigidSynth performs this identification via orderings on sequences ($\leq_{Seq}$), and atoms ($\leq_{Atom}$). To determine if one expression is less than the other, the algorithm converts each expression into a sorted list of its subterms and returns whether the lexicographic ordering determines the first list less than the second. These orderings are carefully constructed so that equivalent terms have lenses between them. For example, between two sequences, SQ and TQ, there is a lens $sql \mathbin{⁝} SQ \Leftrightarrow TQ$ if, and only if, $ESQ \leq_{Seq} ETQ$ and $ETQ \leq_{Seq} ESQ$, where $\{\} \in SQ \rightsquigarrow ESQ$ and $\{\} \in TQ \rightsquigarrow ETQ$. Through these orderings, aligning the components reduces to merely sorting and zipping lists.

Furthermore, through composing the permutations required to sort the lists (extracted through the function, *sorting*), the algorithm discovers the permutation used in the lens. Finally, if there are lenses between all the subcomponents (checked with the AllSome function), the sublenses are combined with the permutation to return the full DNF lens.

Algorithm 3 RigidSynth

function RSAtom$((EDS^*, ils_1), (EDT^*, ils_2))$
  if $ils_1 \neq ils_2$ **then**
    return *None*
  match RSInt(EDS, EDT) **with**
    | *Some* $dl \rightarrow$ **return** dl^*
    | *None* $\rightarrow$ **return** *None*

function RSSeq$(([s_0 \cdot EA_1 \cdot \ldots \cdot EA_n \cdot s_n], ils_1), ([t_0 \cdot EB_1 \cdot \ldots \cdot EB_m \cdot t_m], ils_2))$
  if $ils_1 \neq ils_2$ **then**
    return *None*
  if $n \neq m$ **then**
    return *None*
  $\sigma_1 \leftarrow sorting(\leq_{Atom}, [EA_1 \cdot \ldots \cdot EA_n])$
  $\sigma_2 \leftarrow sorting(\leq_{Atom}, [EB_1 \cdot \ldots \cdot EB_n])$
  $\sigma \leftarrow \sigma_1^{-1} \circ \sigma_2$
  $EABs \leftarrow$ Zip$([EA_1 \cdot \ldots \cdot EA_n], [EB_{\sigma(1)} \cdot \ldots \cdot EB_{\sigma(n)}])$
  $alos \leftarrow$ Map(RSAtom, $EABs$)
  match AllSome$(alos)$ **with**
    | *Some* $[al_1 \cdot \ldots \cdot al_n] \rightarrow$ **return** *Some* $([(s_0, t_0) \cdot al_1 \cdot \ldots \cdot al_n \cdot (s_n, t_n)], \sigma^{-1})$
    | *None* $\rightarrow$ **return** *None*

function RSInt$((\langle ESQ_1 \mid \ldots \mid ESQ_n \rangle, ils_1), (\langle ETQ_1 \mid \ldots \mid ETQ_m \rangle, ils_2))$
  if $ils_1 \neq ils_2$ **then**
    return *None*
  if $n \neq m$ **then**
    return *None*
  $\sigma_1 \leftarrow sorting(\leq_{Seq}, [ESQ_1 \mid \ldots \mid ESQ_n])$
  $\sigma_2 \leftarrow sorting(\leq_{Seq}, [ETQ_1 \mid \ldots \mid ETQ_n])$
  $\sigma \leftarrow \sigma_1^{-1} \circ \sigma_2$
  $ESTQs \leftarrow$ Zip$([ESQ_1 \mid \ldots \mid ESQ_n], [ETQ_{\sigma(1)} \mid \ldots \mid ETQ_{\sigma(n)}])$
  $sqlos \leftarrow$ Map(RSSeq, $ESTQs$)
  match AllSome$(sqlos)$ **with**
    | *Some* $[sql_1 \mid \ldots \mid sql_n] \rightarrow$ **return** *Some* $(\langle sql_1 \mid \ldots \mid sql_n \rangle, \sigma^{-1})$
    | *None* $\rightarrow$ **return** *None*

function RigidSynth$(DS, DT, [(s_1, t_1); \ldots; (s_n, t_n)])$
  $\{s_1, \ldots, s_n\} \in DS \rightsquigarrow EDS$
  $\{t_1, \ldots, t_n\} \in DS \rightsquigarrow EDT$
  return RSInt(EDS, EDT)

As an example, consider trying to find a DNF lens between

$$\langle\ [\text{"<Field␣Id=2></Field>"}]$$
$$\mid\ [\ \text{"<Field␣Id=2>"}\cdot \text{text\_char}\cdot \text{""}\cdot \langle[\text{text\_char}]\rangle^{*}\cdot \text{"</Field>"}]\ \rangle$$

and

$$\langle\ [\ \text{"Title:"}\cdot \text{text\_char}\cdot \text{""}\cdot \langle[\ \text{text\_char}\ ]\rangle\cdot \text{","}\ ]$$
$$\mid\ [\text{""}]\ \rangle$$

As RigidSynth considers the legacy DNF regular expression, it orders its two sequences by maintaining the existing order: first [" <Field ␣ Id=2></Field>"], then [" <Field ␣ Id=2>" · text_char · "" · ⟨[text_char]⟩* · " <Field ␣ Id=2>"]. In contrast, RigidSynth reorders the two sequences of the modern DNF regular expression, making [""] first, and ["Title:" · text_char · "" · ⟨[text_char]⟩* · ","] second; the overall permutation is a swap. As a result, the two string sequences become aligned, as do the two complex sequences.

Then, the algorithm calls RSSeq on the two aligned sequence pairs. There are no atoms in both [" <Field ␣ Id=2></Field>"] and [""], trivially creating the sequence lens:

([(" <Field ␣ Id=2></Field>","")],id)

Next, the sequences [" <Field ␣ Id=2>" · text_char · "" · ⟨[text_char]⟩* · " <Field ␣ Id=2>"] and ["Title=" · text_char · "" · ⟨[text_char]⟩* · ","] would be sent to RSSeq. In RSSeq, the atoms would not be reordered, aligning text_char with text_char, and ⟨[text_char]⟩* with ⟨[text_char]⟩*. Immediately, RSAtom finds the identity transformation on text_char, and will recurse to find the identity transformation for ⟨[text_char]⟩*: These generated atom lenses are then combined into a sequence lens. Lastly, the two sequence lenses are combined with the swapping permutation to create the final DNF lens.

By incorporating information about how examples are parsed in the orderings, SynthLens guarantees not only that there will be a lens between the regular expres-

sions, but also that the lens will satisfy the examples. For example if $ESQ \leq_{Seq} TQ$ and $ETQ \leq_{Seq} SQ$ then there is not only a sequence lens between SQ and TQ, but there is one that also satisfies the examples. Incorporating parse tree information lets the synthesis algorithm differentiate between previously indistinguishable subcomponents; a text_char that parsed only "a" would become less than a text_char that parsed only "b".

Because RigidSynth reduces to merely ordering subterms, the runtime of individual RigidSynth calls is $n * log(n) + m * log(m)$, where n and m are the sizes of DS and DT.

Correctness We have proven two theorems demonstrating the correctness of our algorithm.

Theorem 4 (Algorithm Soundness)**.** For all lenses ℓ, regular expressions R and S, and examples exs, if $\ell = \textsc{SynthLens}(R, S, exs)$, then $\ell : R \Leftrightarrow S$ and for all (s, t) in exs, $(s, t) \in [\![\ell]\!]$.

Theorem 5 (Algorithm Completeness)**.** Given regular expressions R and S, and a set of examples exs, if there exists a lens ℓ such that $\ell : R \Leftrightarrow S$ and for all (s, t) in exs, $(s, t) \in [\![\ell]\!]$, then $\textsc{SynthLens}(R, S, exs)$ will return a lens.

Theorem 4 states that when we return a lens, that lens will match the specifications. Theorem 5 states that if there is a DNF lens that satisfies the specification, then we will return a lens, but not necessarily the same one. However, from Theorem 4, we know that this lens will match the specifications.

Termination If there are no bijective lenses between R and S that satisfies the examples exs, then $\textsc{SynthLens}(R, S, exs)$ will not terminate, as long as $\mathcal{L}(R)$ is infinite or $\mathcal{L}(S)$ is infinite. However, we theorize that there likely is a way of identifying, by looking at the regular expressions and examples, whether a well-typed bijective

lens exists that satisfies the examples. However, we did not investigate building such an identification system; in part because if our symmetric synthesizer (§5) is complete, it is guaranteed to terminate.

Simplification of Generated Lenses While our system takes in only partial specifications, there can be multiple lenses that satisfy the specifications. To help users determine if the synthesized lens is correct, Optician transforms the generated code to make it easily readable. Optician (1) maximally factors the concats and ors, (2) turns lenses that perform identity transformations into identity lenses, and (3) simplifies the regular expressions the identity lenses take as an argument. Performing these transformations and pretty printing the generated lenses make the synthesized lenses much easier to understand. For example, without minimization, the title field transformation is:

```
const("<Field␣Id=2></Field>","")
  | (const("<Field␣Id=2>","Title:␣")
      . id(text_char)
      . const("","")
      . (const("","") . id(text_char) . const("",""))*
      . const("","")
      . const("</Field>",",␣"))
```

where with minimization, the title field transformation is:

```
const("<Field␣Id=2>","")
  . (id("")
      | (id(text_char) . id(text_char)* . const("", ",")))
  . const("</Field>",""))
```

Compositional Synthesis Most synthesis problems can be divided into subproblems. For example, if the format $R_1 \cdot R_2$ must be converted into $S_1 \cdot S_2$, one might first work on the $R_1 \Leftrightarrow S_1$ and $R_2 \Leftrightarrow S_2$ subproblems. After those subproblems have been solved, the lenses they generate can be combined into a solution for $R_1 \cdot R_2 \Leftrightarrow S_1 \cdot S_2$.

Our tool allows users to specify multiple synthesis problems in a single file, and allows the later, more complex problems to use the results generated by earlier problems. This tactic allows Optician to scale to problems of just about any size and complexity with just a bit more user input. This compositional interface also provides users greater control over the synthesized lenses and allows reuse of intermediate synthesized abstractions. The compositional synthesis engine allows lenses previously defined manually by the user, and lenses in the Boomerang standard library to be included in synthesis.

3.7 Evaluation

We have implemented Bijective Optician in 3713 lines of OCaml code. We have integrated our synthesis algorithm into Boomerang, so users can input synthesis tasks in place of lenses. We have published our implementation in a public GitHub repo [43].

We evaluate our synthesis algorithm by applying it to 39 benchmark programs. All evaluations were performed on a 2.5 GHz Intel Core i7 processor with 16 GB of 1600 MHz DDR3 running macOS Sierra.

Benchmark Suite Construction We constructed our benchmarks by adapting examples from Augeas [35] and Flash Fill [20] and by handcrafting specific examples to test various features of the algorithm.

Both Augeas and Flash Fill permit non-bijective transformations. In adopting these benchmarks, we had to modify the formats to address two forms of non-bijectivity: (1) whitespace was present in one format but not the other, and (2) useful information

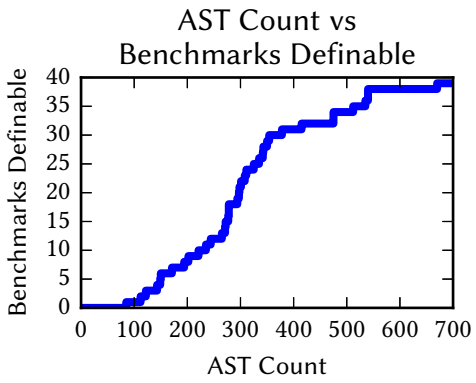
was projected away when going from one format to the other. The first form of non-bijectivity was by far the most common, applying to many Augeas examples. To address this form of non-bijectivity, we added whitespace in the format where typically whitespace is unnecessary. The second form of non-bijectivity was less prevalent, but occurred, typically in the FlashFill examples. To address this form of non-bijectivity, we added projected information to the end of the format missing that information.

Augeas is a configuration editing system for Linux that uses lens combinators similar to those in Boomerang. However, it transforms strings on the left to structured trees on the right rather than transforming strings to strings. We adapted these Augeas lenses to our setting by converting the right-hand sides to strings that correspond to serialized versions of the tree formats. We derived 29 of the benchmark tests by adapting the first 27 lenses in alphabetical order, as well as the lenses aug/xml-firstlevel and aug/xml that were referenced by the 'A' lenses. Furthermore, the 12 last synthesis problems derived from Augeas were tested after Optician was finalized, demonstrating that the optimizations were not overtuned to perform well on the testing data.

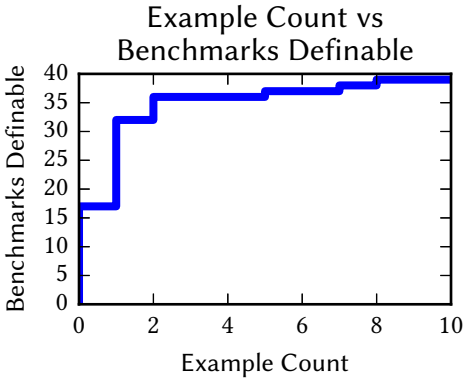
Flash Fill is a system that allows users to specify common string transformations by example [20]. We derived three benchmarks from the first few examples in the paper and one from the running example on extracting phone numbers.

Finally, we added custom examples to highlight weaknesses of our algorithm (cap-prob and 2-cap-prob) and to test situations for which we thought the tool would be particularly useful (workitem-probs, date-probs, bib-prob, and addr-probs). These examples convert between work item formats, date formats, bibliography formats, and address formats, respectively.

Figure 3.7 shows the complexity of our regular specifications as well as our example counts. An average benchmark has a regular specification written in 310 AST nodes, and uses 1.1 input/output examples. Our benchmarks vary from simple problems, like changing how dates are represented (with a specification size of 85, and a generated lens



(a)



(b)

Figure 3.7: Sizes of Specifications. In (a), we show how many benchmarks are defined in our suite using a given number of AST nodes or fewer. In (b), we show how many benchmarks are defined in our suite using a given number of examples or fewer.

size of 79), to complex tasks, like transforming configuration files for server monitoring software into dictionary form (with a specification size of 670 and a generated lens size of 651). On average, the size of the generated lens is 89% the size of its type specifications.

Impact of Optimizations We developed a series of optimizations that improve the performance of the synthesis algorithm dramatically. To determine the relative importance of these optimizations, we developed the 5 different modes that run the synthesis algorithm with various optimizations enabled. These modes are:

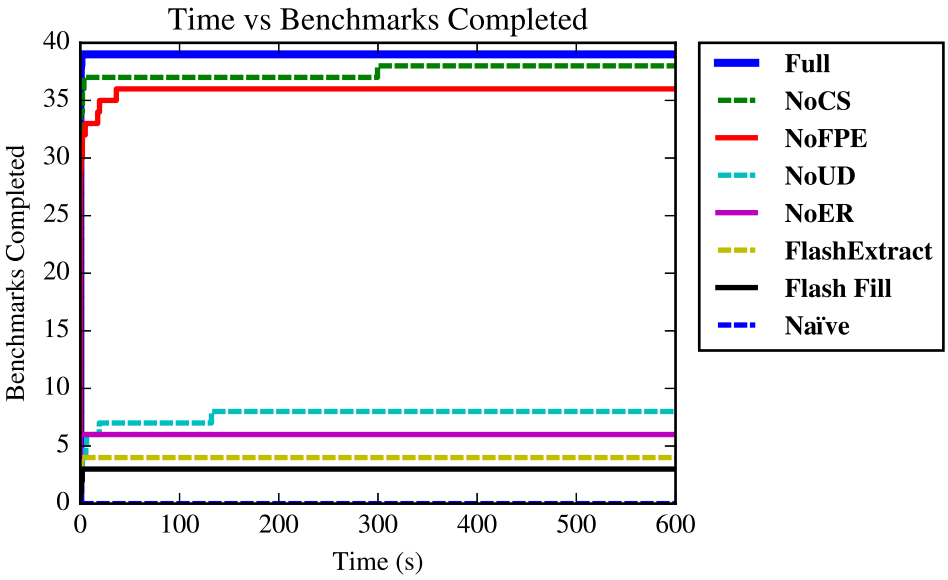


Figure 3.8: Number of benchmarks that can be solved by a given algorithm in a given amount of time. **Full** is the full synthesis algorithm. **NoCS** is the synthesis algorithm using all optimizations but without using a library of existing lenses. **NoFPE** is the core DNF synthesis algorithm augmented with user-defined data types with forced expansions performed. **NoER** is the core synthesis augmented with user-defined data types. **NoUD** is the core synthesis algorithm. **FlashExtract** is the existing FlashExtract system. **Flash Fill** is the existing Flash Fill system. **Naïve** is naïve type-directed synthesis on the bijective lens combinators. Our synthesis algorithm performs better than the naïve approach and other string transformation systems, and our optimizations speed up the algorithm enough that all tasks become solvable.

Full: All optimizations are enabled, and compositional synthesis is used.

NoCS: Like **Full**, but compositional synthesis is not used.

NoFPE: Like **NoCS**, but FixProblemElts is never called, expansions are only forced through ExpandRequired or processed enumeratively through ExpandOnce.

NoER: Like **NoFPE**, but all the expansions taken are generated through enumerative search from ExpandOnce.

NoUD: User-defined data types are no longer kept abstract until needed. All user-defined regular expressions get replaced by their definition at the start of synthesis.

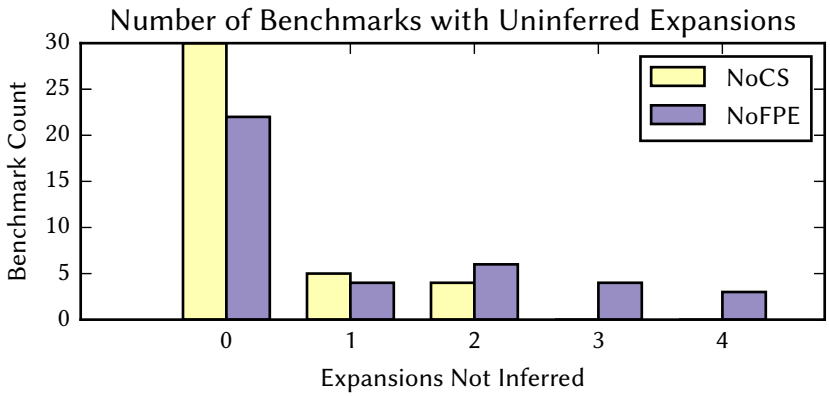


Figure 3.9: Number of expansions found using enumerative search for tasks requiring a given number of expansions. **NoCS** is using the full inference algorithm. **NoFPE** only counts forced inferences as found by the ExpandRequired function. Both systems are able to infer the vast majority of expansions. Full inference only rarely requires enumerative search.

We ran Optician in each mode over our benchmark suite. Figure 3.8 summarizes the results of these tests. **Full** synthesized all 39 benchmarks, **NoCS** synthesized 38 benchmarks, **NoFPE** synthesized 36 benchmarks, **NoER** synthesized 6 benchmarks, **NoUD** synthesized 8 benchmarks, and **Naïve** synthesized 0 benchmarks. Optician's optimizations make synthesis effective against a wide range of complex data formats.

Interestingly, **NoER** performs *worse* than **NoUD**. Adding in user defined data types introduces the additional search through substitutions. The cost of this additional search outweighs the savings that these data type abstractions provide. In particular, because of the large fan-out of possible expansions, **NoER** can only synthesize lenses which require 5 or fewer expansions. However, some lenses require over 50 expansions. Without a way to intelligently traverse expansions, the need to search through substitutions makes synthesis unbearably slow.

In **NoFPE**, we can determine that many expansions are forced, so an enumerative search is often unnecessary. Figure 3.9 shows that in a majority of examples, all the expansions can be identified as required, minimizing the impact of the large fan-out. While unable to infer every expansion for all the benchmarks, the full algorithm is

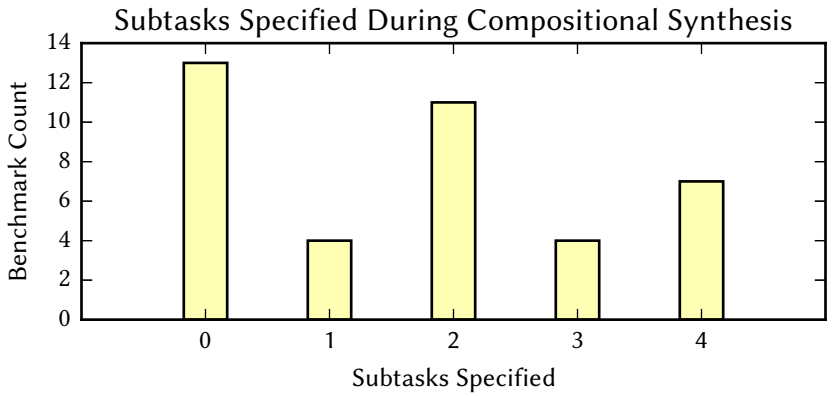


Figure 3.10: Number of subtasks specified during compositional synthesis. Splitting the task into just a few subtasks provides huge performance benefits at the cost of a small amount of additional user work.

able to infer quite a bit. In our benchmark suite, ExpandRequired infers a median of 13 and a maximum of 75 expansions.

Merely inferring the forced expansions makes almost all the synthesis tasks solvable. In many cases, **NoFPE** infers *all* the expansions. In 22 of the 38 examples solvable by **NoCS**, all expansions were forced. However, the remaining 16 still require some enumerative search. This enumerative search causes the **NoFPE** version of the algorithm to struggle with some of the more complex benchmarks. Incorporating FixProblemElts speeds up these slow benchmarks. When using full inference (FixProblemElts and ExpandRequired), the synthesis algorithm can recognize that one of a few expansions must be performed. Adding in these types of inferred expansions directs the remaining search even more, both speeding up existing problems and solving previously unmanageable benchmarks.

When combined, these optimizations implement an efficient synthesis algorithm, which can synthesize lenses between a wide range of data formats. However, some of the tasks are still slow, and one remains unsolved. Using compositional synthesis lets the system scale to the most complex synthesis tasks, synthesizing all lenses in under 5 seconds. Additional user interaction is required for compositional synthesis,

but the amount of interaction is minimal, as shown in Figure 3.10. The number of subtasks used was in no way the minimal number of subtasks needed for synthesis under 5 seconds, but rather subtasks were introduced where they naturally arose.

The benchmark that only completes with compositional synthesis is also the slowest benchmark in **Full**, aug/xml[4]. Optician can only synthesize a lens for this example when compositional synthesis is used because it is a complex data format, it requires a large number of expansions, and relatively few expansions are forced. When not using compositional synthesis, the algorithm must perform a total of 398 expansions, of which only 105 are forced. The synthesis algorithm is able to force so few expansions because of the highly repetitive nature of the aug/xml specification. XML tags occur at many different levels, and they all use the same user-defined data types. This repetitive nature causes our expansion inference to find only a few of the large number of required expansions. The large fan-out of expansions, combined with the large number of expansions that must be performed, creates a search space too large for our algorithm to effectively search. However, the synthesis algorithm is able to succeed on the easier task of finding the desired transformation when provided with two additional subtasks: synthesis on XML of depth one, and synthesis of XML of depth up to two.

Importance of Examples To evaluate how many user-supplied examples the algorithm requires in practice, we *randomly* generated appropriate source/target pairs, mimicking what a naïve user might do. We did not write the examples by hand out of concern that our knowledge of the synthesis algorithm might bias the selection. Figure 3.11 shows the number of randomly generated examples it takes to synthesize the correct lens averaged over ten runs. The synthesis algorithm almost never needs any examples: only 5 benchmarks need a nonzero number of examples to synthesize the correct lens and only one, cust/workitem-probs required over 10 randomly generated

[4]Since xml syntax is context-free, the source and target regular expressions describe only xml expressions up to depth 3.

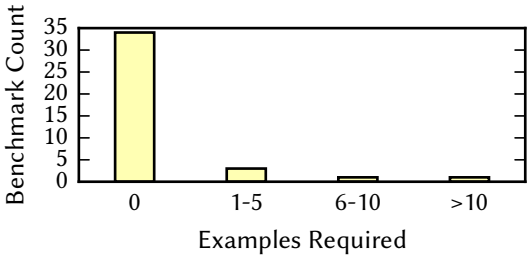


Figure 3.11: Average number of random examples required to synthesize benchmark programs. **Experimental Average** is the average number of randomly generated examples needed to correctly synthesize the lens. **Determinize Permutations** is the theoretical number of examples required to determinize the choice all the permutations in RigidSynth. In practice, far fewer examples are needed to synthesize the correct lens than would be predicted by the number required to determinize permutations.

examples. A clever user may be able to reduce the number of examples further by selecting examples carefully; we synthesized cust/workitem-probs with only 8 examples.

These numbers are low because there are relatively few well-typed bijective lenses between any two source and target regular expressions. As one would expect, the benchmarks where there are multiple ways to map source data to the target (and vice versa) require the most examples. For example, the benchmark cust/workitem-probs requires a large number of examples because it must differentiate between data in different text fields in both the source and target and map between them appropriately. As these text fields are heavily permuted (the legacy format ordered fields by a numeric ID, where the modern format ordered fields alphabetically) and fields can be omitted, a number of examples are needed to correctly identify the mapping between fields.

The average number of examples to infer the correct lens does not tell the whole story. The system will stop as soon as it finds a well typed lens that satisfies the supplied examples. This inferred lens may or may not correctly handle unseen examples that correspond to unexercised portions of the source and target regular expressions. Figure 3.11 lists the number of examples that are required to determinize the generation of permutations in RigidSynth. Intuitively, this number represents the maximum number of examples that a user must supply to guide the synthesis engine if it always

guesses the wrong permutation when multiple permutations can be used to satisfy the specification.

The average number of examples is so much lower than the maximum number of required examples because of correspondences in how we wrote the regular expressions for the source and target data formats. Specifically, when we had corresponding disjunctions in both the source and the target, we ordered them the same way. The algorithm uses the supplied ordering to guide its search, and so the system requires fewer examples. We did not write the examples in this style to facilitate synthesis, but rather because maintaining similar subparts in similar orderings makes the types much easier to read. We expect that most users would do the same.

Comparison Against Other Tools We are the first tool to synthesize bidirectional transformations between data formats, so there is no tool to which we can make an apple-to-apples comparison. Instead, we compare against tools for generating unidirectional transformations instead. Figure 3.8 includes a comparison against two other well-known tools that synthesize text transformation and extraction functions from examples – Flash Fill and FlashExtract. For this evaluation, we used the version of these tools distributed through the PROSE project [52].

To generate specifications for Flash Fill, we generated input/output specifications by generating random elements of the source language, and running the lens on those elements to generate elements of the target language. These were then fed to Flash Fill.

To generate specifications for FlashExtract, we extracted portions of strings mapped in the generated lens either through an identity transformation or through a previously synthesized lens, whereas strings that were mapped through use of const were considered boilerplate and so not extracted.

As these tools were designed for a broader audience, they put less of a burden on the user. These tools only use input/output examples (for Flash Fill), or marked text regions (for FlashExtract), as opposed to Optician's use of regular expressions to constrain the format of the input and output. By using regular expressions, Optician is able to synthesize significantly more programs than either existing tool.

Flash Fill and FlashExtract have two tasks: to determine how the data is transformed, they must also infer the structure of the data, a difficult job for complex formats. In particular, neither Flash Fill nor FlashExtract was able to synthesize transformations or extractions present under two iterations, a type of format that is notoriously hard to infer. These types of dual iterations are pervasive in Linux configuration files, making Flash Fill and FlashExtract ill suited for many of the synthesis tasks present in our test suite.

Furthermore, as unidirectional transformations, Flash Fill and FlashExtract have a more expressive calculus. To guarantee bidirectionality, our syntax must be highly restrictive, providing a smaller search space to traverse.

Chapter 4

Synthesizing Quotient Lenses

4.1 Introduction

Chapter 3 describes how to synthesize bijections from format descriptions. 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. 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 use the bijective lens synthesis as component in a system that synthesizes lenses that involve normalization.

This paper applies this idea to the problem of synthesizing *quotient lenses* [18]. 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 that express bijections modulo equivalence relations $\equiv_S$ and $\equiv_T$ defined on the source and target data respectively.

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.

Additionally, we 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? We prove that we do not—more specifically, we prove a normal form theorem (Theorem 7) that says that

every QRE lens formed by freely composing QRE lenses, applying regular operators to them, and quotienting 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 the bijective synthesizer, 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.

Most of the content in this chapter comes from the paper "Synthesizing Quotient Lenses" [38].

4.2 Quotient Lens Definition

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 $q.\mathsf{createR} : \mathcal{L}(R) \longrightarrow \mathcal{L}(S)$ and $q.\mathsf{createL} : \mathcal{L}(S) \longrightarrow \mathcal{L}(R)$ such that for all $r \in \mathcal{L}(R)$ and $s \in \mathcal{L}(S)$, we have $q.\mathsf{createL}(q.\mathsf{createR}(r)) \equiv_R r$ and $q.\mathsf{createR}(q.\mathsf{createL}(s)) \equiv_S s$. Moreover, if $r \equiv_R r'$, then $q.\mathsf{createR}(r) = q.\mathsf{createR}(r')$, and if $s \equiv_S s'$, then $q.\mathsf{createL}(s) = q.\mathsf{createL}(s')$. In words, a bijective quotient lens q from R to S modulo $\equiv_R$ and $\equiv_S$ is a pair of functions $q.\mathsf{createR}$ and $q.\mathsf{createL}$ such that $q.\mathsf{createR}$ respects $\equiv_R$ and $q.\mathsf{createL}$ respects $\equiv_S$, and such that the createR and createL functions lifted

to $\mathcal{L}(R)/\equiv_R$ and $\mathcal{L}(S)/\equiv_S$ are mutual inverses. These laws are similar to stating that the functions are inverses (as was shown for bijective lenses in Chapter 3), except that the strict syntactic equality relations in the definition of inverses are loosened to allow for equivalence relations. Also, the condition that $r \equiv_R r'$ implies $q.\mathsf{createR}(r) = q.\mathsf{createR}(r')$ ensures that the createR function induced on the equivalence classes of $\equiv_R$ is well-defined, and similarly for the createL function.

4.3 QRE Lenses by Example

Chapter 3 shows how to simplify 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.

4.3.1 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 left-to-right 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.createR
```

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 createR 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 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 $f(s) = f(s')$.

```
let wsp = [#\,\textvisiblespace\,#\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.createR

(* 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 . "}"
```

Figure 4.1: QRE definition of BibTeX records.

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

4.3.2 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 with equivalence relations indicating the essential

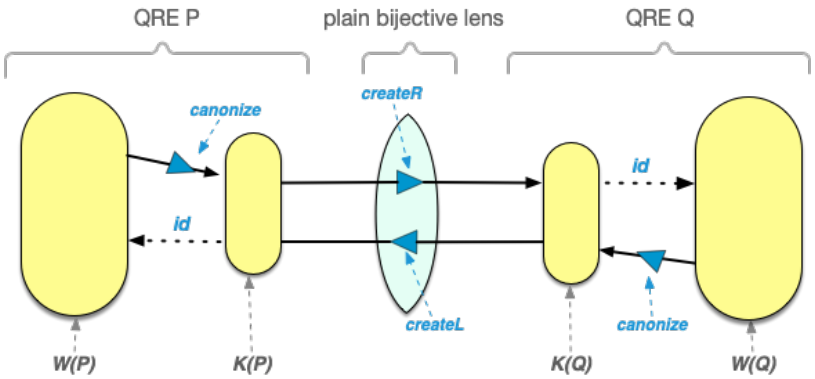


Figure 4.2: QRE lens from source P to target Q. P and Q are QREs, each consisting of a "whole" set ($W(P)$ and $W(Q)$) and a "kernel" part ($K(P)$ and $K(Q)$) Between the two kernels is a plain bijective lens. The createR function of the whole lens takes an argument from $W(P)$, applies the canonizer for P to obtain a canonical representative in $K(P)$, then applies the createL of the plain lens, yielding an element of $K(Q)$, which is a subset of $W(Q)$. The createL 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.

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 4.2 depicts the architecture of QRE lenses. Every QRE lens q has a type $P \Leftrightarrow Q$ where P and Q are QREs. In the *left-to-right* 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 createR function of a bijective string lens ℓ to this representative. In the *right-to-left* direction, q operates similarly, but using the QRE Q and the createL 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 chapter (Theorem 7) 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 writing a synchronizer from BibTeX to EndNote is as simple as 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.

4.4 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.

4.4.1 Syntax and Semantics of QREs

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

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

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.

In the next section, I will provide a judgement, $Q\ \mathsf{wf}$, that defines the well-formed QREs. 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 $\mathsf{canonize}(Q)$.

4.4.2 The normalize Combinator

The relationship among the semantic objects of a QRE can be understood in terms of the combinator 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 quotient 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.

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

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

[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.

Q	$W(Q)$	$K(Q)$
id(R)	R	R
collapse $R \mapsto s$	R	s
squash $Q_1 \to Q_2$ using f	$W(Q_1) \mid W(Q_2)$	$K(Q_2)$
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^*	$W(Q)^*$	$K(Q)^*$

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

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

Figure 4.3: 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.

combinators because it gives a sufficient condition for the well-formedness of any potential QREs.

4.4.3 QRE Combinator Semantics

Figure 4.3 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 $Q =$ squash $Q_1 \to Q_2$ 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 canonize(Q_2).

For the perm$(Q_1, \ldots, Q_n)$ 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

$$
\begin{aligned}
&\mathsf{canonize}(\mathsf{id}(R))(w) &&= w \\
&\mathsf{canonize}(\mathsf{collapse}\ R \mapsto s)(w) &&= s \\
&\mathsf{canonize}(\mathsf{squash}\ Q_1 \to Q_2\ \mathsf{using}\ f)(w) &&= \begin{cases} \mathsf{canonize}(Q_2)(f(w)) & \text{if } w \in \mathcal{L}(W(Q_1)) \\ \mathsf{canonize}(Q_2)(w) & \text{otherwise} \end{cases} \\
&\mathsf{canonize}(\mathsf{normalize}(R_1, R_2, f))(w) &&= f(w) \\
&\mathsf{canonize}(Q_1\ ;\ Q_2)(w) &&= \mathsf{canonize}(Q_2)(\mathsf{canonize}(Q_1)(w)) \\
&\mathsf{canonize}(Q_1 \cdot Q_2)(w_1 \cdot w_2) &&= \mathsf{canonize}(Q_1)(w_1) \cdot \mathsf{canonize}(Q_2)(w_2) \\
&\mathsf{canonize}(Q_1 \mid Q_2)(w) &&= \begin{cases} \mathsf{canonize}(Q_1)(w) & \text{if } w \in \mathcal{L}(W(Q_1)) \\ \mathsf{canonize}(Q_2)(w) & \text{if } w \in \mathcal{L}(W(Q_2)) \end{cases}
\end{aligned}
$$

$$
\mathsf{canonize}(Q^*)(w_1 \cdot \ldots \cdot w_n) = \begin{cases} \epsilon & \text{if } n = 0 \\ \mathsf{canonize}(Q)(w_1) \cdot \ldots \cdot \mathsf{canonize}(Q)(w_n) & \text{if } n > 0 \end{cases}
$$

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

Figure 4.4: 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.

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

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

which places the strings that match Q_i and Q according to the canonical permutation (i.e., the provided ordering) before applying the canonizers $\mathsf{canonize}(Q_i)$ and $\mathsf{canonize}(Q)$, respectively. Essentially, the permutation combinator permits any ordering of the fields in $W(Q)$, and has a specific ordering in $K(Q)$.

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

Figure 4.5: QRE Equivalence Relations

Finally, Figure 4.5 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 $W(Q)$.

4.4.4 Ambiguity and Well-Formed QREs

To ensure that regular combinations of QREs are well-formed, we need to enforce a variety of *unambiguity* constraints. Like in lens programming, 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,

$$\mathsf{CSL} = \mathsf{anychar}+ \ .\ (\text{","}\ .\ \mathsf{anychar}+)*$$

(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,

$$\mathsf{CSL} = \mathsf{anycharexceptcomma}+ \;.\; (\text{","} \;.\; \mathsf{anycharexceptcomma}+)*$$

(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 $\text{"a"}^* \;.\; (\mathsf{collapse}\ \text{"a"}^* \to \text{"a"})$. 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.

4.4.5 Well-formedness of QREs

Figure 4.6 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

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

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

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

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

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

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

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

Figure 4.6: Well-formed QREs

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.

4.5 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 $q.\mathsf{createR} : \mathcal{L}(R) \longrightarrow \mathcal{L}(S)$ and $q.\mathsf{createL} : \mathcal{L}(S) \longrightarrow \mathcal{L}(R)$ such that for all $r \in \mathcal{L}(R)$ and $s \in \mathcal{L}(S)$, we have $q.\mathsf{createL}(q.\mathsf{createR}(r)) \equiv_R r$ and $q.\mathsf{createR}(q.\mathsf{createL}(s)) \equiv_S s$. Moreover, if $r \equiv_R r'$, then $q.\mathsf{createR}(r) = q.\mathsf{createR}(r')$, and if $s \equiv_S s'$, then $q.\mathsf{createL}(s) = q.\mathsf{createL}(s')$. In words, a bijective quotient lens q from R to S modulo $\equiv_R$ and $\equiv_S$ is a pair of functions $q.\mathsf{createR}$ and $q.\mathsf{createL}$ such that $q.\mathsf{createR}$ respects $\equiv_R$ and $q.\mathsf{createL}$ respects $\equiv_S$, and such that the createR and createL functions lifted to $\mathcal{L}(R)/\equiv_R$ and $\mathcal{L}(S)/\equiv_S$ are mutual inverses. Also, the condition that $r \equiv_R r'$ implies $q.\mathsf{createR}(r) = q.\mathsf{createR}(r')$ ensures that the createR function induced on the equivalence classes of $\equiv_R$ is well-defined, and similarly for the createL 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 add quotients by allowing canonizers to be prepended or postpended to bijections and allow composition of such quotient lenses via the regular operators and functional composition (Sections 4.5.1 and 4.5.2).

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 $\mathsf{canonize}(Q_1)$ and $\mathsf{canonize}(Q_2)$ 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.

4.5.1 Syntax of QRE Lenses

The language of QRE lenses is given by following grammar,

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

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 $\mathsf{lquot}(Q, q)$ combinator takes a quotient lens q and a QRE Q and quotients the source data using Q. The $\mathsf{rquot}(q, Q)$ 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)
```

4.5.2 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 $[\![q]\!] : W(Q_1)/\equiv_{Q_1} \Longleftrightarrow W(Q_2)/\equiv_{Q_2}$. Because the semantics are more complex, they cannot be expressed as merely a subset of $\Sigma^* \times \Sigma^*$. Instead, the semantics are provided directly as the two functions createR and createL. Note that, for many lenses, the semantics are only well-defined when the lens is well-typed. The typing rules of QRE lenses are given in Figure 4.7 and the semantics are given in Figure 4.8. The trickiest typing rule is the typing rule for composition:

$$\textsc{Compose}\quad \frac{\mathcal{L}(W(Q_2)) = \mathcal{L}(W(Q_2')) \qquad K(Q_2) \equiv^s K(Q_2') \qquad q_1 : Q_1 \Leftrightarrow Q_2 \qquad q_2 : Q_2' \Leftrightarrow Q_3 \qquad \mathsf{canonize}(Q_2) = \mathsf{canonize}(Q_2')}{q_1 \ ; \ q_2 : Q_1 \Leftrightarrow Q_3}$$

This rule essentially says that the composition $q_1 \ ; \ q_2$ is well defined if and only if the intermediary QREs Q_2 and Q_2' define the same equivalence relation on regular

$$\textsc{Lift}\quad \frac{\ell : R \Leftrightarrow S}{\mathsf{lift}(\ell) : \mathsf{id}(R) \Leftrightarrow \mathsf{id}(S)} \qquad \textsc{LQuot}\quad \frac{q : Q_2 \Leftrightarrow Q_3 \qquad Q_1\ \mathsf{wf} \qquad K(Q_1) = W(Q_2)}{\mathsf{lquot}(Q_1, q) : Q_1\ ;\ Q_2 \Leftrightarrow Q_3}$$

$$\textsc{RQuot}\quad \frac{q : Q_1 \Leftrightarrow Q_3 \qquad Q_3\ \mathsf{wf} \qquad W(Q_3) = K(Q_2)}{\mathsf{rquot}(q, Q_1) : Q_1 \Leftrightarrow Q_2\ ;\ Q_3}$$

$$\textsc{Star}\quad \frac{q : Q_1 \Leftrightarrow Q_2 \qquad W(Q_1)^{*!}, W(Q_2)^{*!} \qquad K(Q_1)^{*!}, K(Q_2)^{*!}}{q^* : {Q_1}^* \Leftrightarrow {Q_2}^*}$$

$$\textsc{Concat}\quad \frac{\begin{array}{c} q_1 : Q_1 \Leftrightarrow Q_3 \qquad q_2 : Q_2 \Leftrightarrow Q_4 \\ W(Q_1) \cdot^! W(Q_2) \qquad K(Q_1) \cdot^! K(Q_2) \qquad W(Q_3) \cdot^! W(Q_4) \qquad K(Q_3) \cdot^! K(Q_4) \end{array}}{q_1 \cdot q_2 : Q_1 \cdot Q_2 \Leftrightarrow Q_3 \cdot Q_4}$$

$$\textsc{Swap}\quad \frac{\begin{array}{c} q_1 : Q_1 \Leftrightarrow Q_3 \qquad q_2 : Q_2 \Leftrightarrow Q_4 \\ W(Q_1) \cdot^! W(Q_2) \qquad K(Q_1) \cdot^! K(Q_2) \qquad W(Q_4) \cdot^! W(Q_3) \qquad K(Q_4) \cdot^! K(Q_3) \end{array}}{\mathsf{swap}(q_1, q_2) : Q_1 \cdot Q_2 \Leftrightarrow Q_4 \cdot Q_3}$$

$$\textsc{Or}\quad \frac{\begin{array}{c} q_1 : Q_1 \Leftrightarrow Q_3 \\ q_2 : Q_2 \Leftrightarrow Q_4 \qquad \mathcal{L}(W(Q_1)) \cap \mathcal{L}(W(Q_2)) = \varnothing \qquad \mathcal{L}(W(Q_3)) \cap \mathcal{L}(W(Q_4)) = \varnothing \end{array}}{q_1 \mid q_2 : (Q_1 \mid Q_2) \Leftrightarrow (Q_3 \mid Q_4)}$$

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

Figure 4.7: Typing Rules for QRE Lenses

$$
\begin{aligned}
[\![\mathsf{lift}(\ell)]\!].\mathsf{createL} &= [\![\ell]\!] \\
[\![\mathsf{lift}(\ell)]\!].\mathsf{createR} &= [\![\ell]\!]^{-1}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{lquot}(Q_1, q)]\!].\mathsf{createL} &= [\![q]\!].\mathsf{createL} \circ \mathsf{canonize}(Q_1) \\
[\![\mathsf{lquot}(Q_1, q)]\!].\mathsf{createR} &= [\![q]\!].\mathsf{createR}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{rquot}(q, Q_1)]\!].\mathsf{createL} &= [\![q]\!].\mathsf{createL} \\
[\![\mathsf{rquot}(q, Q_1)]\!].\mathsf{createR} &= [\![q]\!].\mathsf{createR} \circ \mathsf{canonize}(Q_3)
\end{aligned}
$$

$$
\begin{aligned}
[\![q^*]\!].\mathsf{createL} &= ([\![q]\!].\mathsf{createL})^* \\
[\![q^*]\!].\mathsf{createR} &= ([\![q]\!].\mathsf{createL})^*
\end{aligned}
$$

$$
\begin{aligned}
[\![q_1 \cdot q_2]\!].\mathsf{createL} &= [\![q_1]\!].\mathsf{createL} \cdot [\![q_2]\!].\mathsf{createL} \\
[\![q_1 \cdot q_2]\!].\mathsf{createR} &= [\![q_1]\!].\mathsf{createR} \cdot [\![q_2]\!].\mathsf{createR}
\end{aligned}
$$

$$
\begin{aligned}
[\![q]\!].\mathsf{createL}(s_1 \cdot s_2) &= [\![q_2]\!].\mathsf{createL}(s_2) \cdot [\![q_1]\!].\mathsf{createL}(s_1) \\
[\![q]\!].\mathsf{createR}(t_2 \cdot t_1) &= [\![q_1]\!].\mathsf{createR}(t_1) \cdot [\![q_2]\!].\mathsf{createR}(t_2)
\end{aligned}
$$

$$
\begin{aligned}
[\![q_1 \mid q_2]\!].\mathsf{createL}(s) &= \begin{cases} [\![q_1]\!].\mathsf{createL}(s) & \text{if } s \in \mathcal{L}(W(Q_1)) \\ [\![q_2]\!].\mathsf{createL}(s) & \text{if } s \in \mathcal{L}(W(Q_2)) \end{cases} \\
[\![q_1 \mid q_2]\!].\mathsf{createR}(s) &= \begin{cases} [\![q_1]\!].\mathsf{createR}(s) & \text{if } s \in \mathcal{L}(W(Q_3)) \\ [\![q_2]\!].\mathsf{createR}(s) & \text{if } s \in \mathcal{L}(W(Q_4)) \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
[\![q_1 \ ; \ q_2]\!].\mathsf{createL} &= [\![q_2]\!].\mathsf{createL} \circ [\![q_1]\!].\mathsf{createL} \\
[\![q_1 \ ; \ q_2]\!].\mathsf{createR} &= [\![q_1]\!].\mathsf{createR} \circ [\![q_2]\!].\mathsf{createR}
\end{aligned}
$$

Figure 4.8: Semantics for QRE Lenses

expressions that are equivalent modulo the star-semiring axioms. (See [18, §4] for an example of what goes wrong if this premise is dropped). The condition $\mathcal{L}(W(Q_2)) = \mathcal{L}(W(Q_2'))$ says that the intermediary language is the same on both sides, while the condition $K(Q_2) \equiv^s K(Q_2')$ says that the kernel regular expressions are equivalent modulo the star-semiring axioms. Finally, the condition $\mathsf{canonize}(Q_2) = \mathsf{canonize}(Q_2')$ says that Q_2 and Q_2' define the same equivalence relation on $\mathcal{L}(W(Q_2)) = \mathcal{L}(W(Q_2'))$.

Of course checking the condition $\mathsf{canonize}(Q_2) = \mathsf{canonize}(Q_2')$ is undecidable in general; indeed Boomerang's typechecker only permits compositions between quotients when the associated canonizers are the identity. In other words, two Boomerang lenses

$q_1 : Q_1 \Leftrightarrow Q_2$ and $q_2 : Q_2' \Leftrightarrow Q_3$ can be composed if $W(Q_2) = K(Q_2) = K(Q_2') = W(Q_2')$.

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 [18]. 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 6. If there is a derivation $q : Q_1 \Leftrightarrow Q_2$, then $[\![q]\!] : W(Q_1)/\equiv_{Q_1} \Leftrightarrow W(Q_2)/\equiv_{Q_2}$ is a well-defined quotient lens.

4.5.3 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

$$[\![q]\!].\mathsf{createL} = \ell \circ \mathsf{canonize}(Q_1)$$

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

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 chapter, confirms that this indeed is the case:

Theorem 7. 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:

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

Proof. The proof follows by induction on the derivation of $q : Q_1 \Leftrightarrow Q_2$. 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:

$$\frac{\mathcal{L}(W(Q_2)) = \mathcal{L}(W(Q_2')) \qquad K(Q_2) \equiv^s K(Q_2') \qquad q_1 : Q_1 \Leftrightarrow Q_2 \qquad q_2 : Q_2' \Leftrightarrow Q_3 \qquad \mathsf{canonize}(Q_2) = \mathsf{canonize}(Q_2')}{q_1 \ ; \ q_2 : Q_1 \Leftrightarrow Q_3}$$

$$[\![q_1 \ ; \ q_2]\!].\mathsf{createL} = [\![q_2]\!].\mathsf{createL} \circ [\![q_1]\!].\mathsf{createL}$$
$$[\![q_1 \ ; \ q_2]\!].\mathsf{createR} = [\![q_1]\!].\mathsf{createR} \circ [\![q_2]\!].\mathsf{createR}$$

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

$$[\![q_1]\!].\mathsf{createL} = [\![\ell_1]\!] \circ \mathsf{canonize}(Q_1) \qquad [\![q_2]\!].\mathsf{createL} = [\![\ell_2]\!] \circ \mathsf{canonize}(Q_2')$$
$$[\![q_1]\!].\mathsf{createR} = [\![\ell_1]\!]^{-1} \circ \mathsf{canonize}(Q_2) \qquad [\![q_2]\!].\mathsf{createR} = [\![\ell_2]\!]^{-1} \circ \mathsf{canonize}(Q_3)$$

Consequently:

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

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

$$[\![q_1]\!].\mathsf{createR} \circ [\![q_2]\!].\mathsf{createR} = [\![\ell_1 \ ; \ \ell_2]\!]^{-1} \circ \mathsf{canonize}(Q_3)$$

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 of the full version of the original paper [38]. □

4.6 Synthesis Algorithm

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.

Chapter 3 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 core synthesis 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.

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:

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

Our approach to synthesizing QRE lenses is guided by Theorem 7, 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:

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

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 4.

Algorithm 4 SynthQRELens

```
1: function SynthQRELens(Q1, Q2, exs)
2:     R1 ← K(Q1)
3:     R2 ← K(Q2)
4:     c1 ← canonize(Q1)
5:     c2 ← canonize(Q2)
6:     exs' ← Map(exs, λ(exl, exr) → (c1(exl), c2(exr)))
7:     l ← SynthBijectiveLens(R1, R2, exs')
8:     return rquot(lquot(Q1, ℓ), Q2)
```

Theorem 8. 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 $q.\mathsf{createL}(x_i) \equiv_{Q_2} y_i$ and $q.\mathsf{createR}(y_i) \equiv_{Q_1} x_i$, then SynthQRELens$(Q_1, Q_2, exs)$ will return such a lens.

(This follows from Theorems 4, 5, and 7.)

Returning to the BibTeX to EndNote transformation of Section 4.3.2, 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 =                    end_example =
"@Book␣{Lovelace,                "%0␣Book
␣Author␣=␣\"Ada␣Lovelace\",      ␣%T␣Generic␣Title
␣Title␣=␣{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.

4.7 Evaluation

We have implemented QREs and the quotient lens synthesis algorithm described above as an extension to the Boomerang interpreter [6, 18]. 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 bijective version of Optician. 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.

4.7.1 Benchmark Suite Construction

To evaluate our QRE implementation, we adapted 39 lens synthesis tasks from the benchmark suite of Chapter 3. These benchmarks are a combination of custom benchmarks, benchmarks derived from FlashFill [20], and benchmarks derived from

Augeas [36] (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 due to a lack of quotients. 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.

4.7.2 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 Boomerang 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 4.9 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, while finding semantically equivalent lenses.

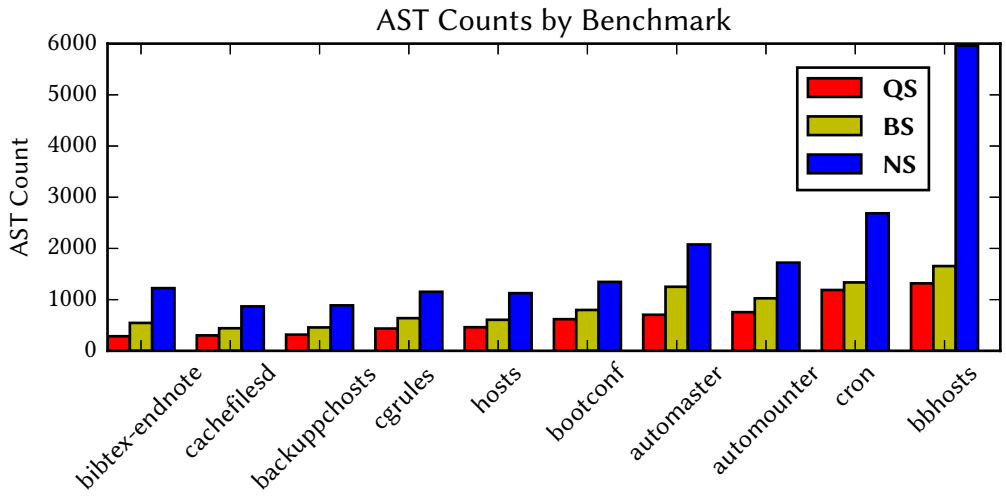
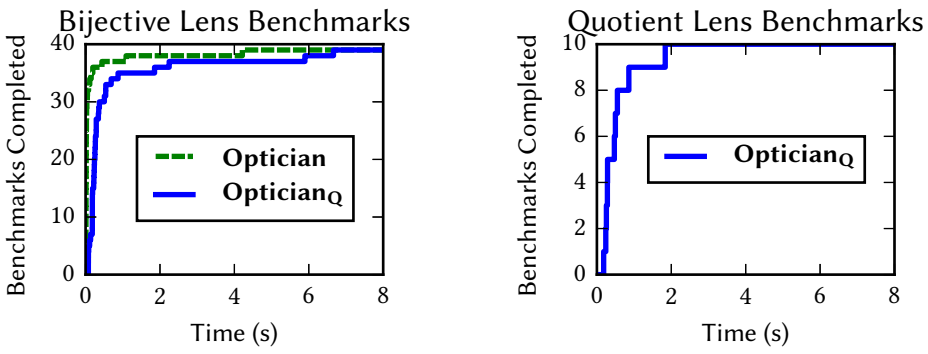


Figure 4.9: AST node measurements for each of the three approaches on each of the 10 non-bijective benchmark problems. Benchmarks are sorted in order of increasing complexity as measured by the number of AST nodes in the source and target format descriptions. QRE Synthesis requires far fewer AST nodes than the other two approaches. **QS** represents using the full quotient Optician system. **BS** represents using bijective Optician to write bijective lenses, and manually writing canonizers for the edges. **NS** represents manually writing the full lens.

4.7.3 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 4.10(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 (two additional passes over the QREs are required, where inputs could be used as-is), resulting in a very 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 4.10(b) shows the amount of time required to infer a lens for each of the 10 benchmark programs



(a) (b)

Figure 4.10: 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.

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.

Chapter 5

Synthesizing Symmetric Lenses

5.1 Introduction

One reason that bijective and quotient lens synthesis can be so effective, even relative to successful synthesis tools in other domains, is that there are typically not very many bijections (or bijections, up to equivalence) between a given pair of data formats, particularly in the context of user-defined regular expressions, where only the identity map can be used on them. If the synthesis algorithm finds any bijection, it is fairly likely to be the intended one.

However, the set of real-world use cases for bijective lenses and quotient bijective lenses, where two data formats contain different arrangements of precisely the same information, is limited. Oftentimes, two data formats share just *some* of their information content. For instance, one ad-hoc system-configuration file might include some format-specific metadata, such as a date or a reference number, while the same configuration file on another operating system does not. Indeed among our benchmark suite, all of the benchmarks taken from Flash Fill [20] and many of the benchmarks that synchronize between two ad hoc file formats have this characteristic.

Can Optician's basic synthesis procedure be extended to a richer class of lenses? Could we even imagine synthesizing all *symmetric lenses* [25]—a much larger class that includes bijective lenses, "asymmetric" lenses (where the transformation from a source structure to a target can throw away information that must then be restored when transferring from target to source), and even more flexible transformations that allow each side to throw away information?

One might first hope that extending the bijective lens synthesis algorithms to synthesize symmetric rather than bijective lenses would be relatively straightforward: Simply replace the bijective combinators with symmetric ones and search using similar heuristics. However, this naïve approach encounters two difficulties.

The first of these is pragmatic. Symmetric lenses as presented by Hofmann et al. [25] operate over three structures: a "left" structure X, a "right" structure Y and a "complement" C that contains the information not present in either X or Y. These complements must be stored and managed somehow. More fundamentally, complements complicate synthesis *specifications*—instead of just giving single examples of source and target pairings, users would have to give longer "interaction sequences" to show how a lens should behave. To avoid these complexities, we define a restricted variant of symmetric lenses, called *simple symmetric lenses*. Intuitively, simple symmetric lenses are symmetric lenses that do not require external "memory" to recover data from past instances of X or Y when making a round trip. They only need the most recent instance.

Formally, we characterize the expressiveness of simple symmetric lenses by proving that they are exactly the symmetric lenses that satisfy an intuitive property called *forgetfulness*. We also show they are expressive enough for many practical uses by adding simple symmetric lenses to the Boomerang language [6] and applying them to a range of real-world applications. This exercise also demonstrates that simple

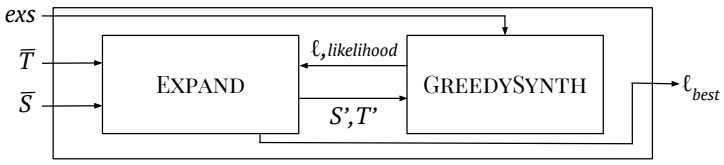


Figure 5.1: Schematic diagram for the simple symmetric lens synthesis algorithm. The user provides regular expressions $\overline{R}$and $\overline{S}$and a set of examples *exs* as input. EXPANDfirst converts $\overline{R}$and $\overline{S}$to stochastic regular expressions $R$and $S$with default probabilities. It then finds pairs of stochastic regular expressions equivalent to $R$and $S$and iteratively proposes them to GREEDYSYNTH. GREEDYSYNTH finds a lens typed between the supplied SREs. When the algorithm finds a likely lens, it returns it.

symmetric lenses can coexist with, and be extended by, other advanced lens features provided by Boomerang, including *quotient lenses* [18, 38] and *matching lenses* [5].

This leaves us with the second difficulty in synthesizing symmetric lenses: Whereas the number of bijective lenses between two given formats is typically tiny, the number of simple symmetric lenses is typically enormous. If a naïve search algorithm just selects the first simple symmetric lens it finds, the returned lens will generally *not* be the one the user wanted. We need a new principle for identifying "more likely" lenses and a more sophisticated synthesis algorithm that uses this principle to search the space more intelligently.

For these, we turn to information theory. We consider "likely" lenses to be ones that propagate "a lot" of information from the left data format to the right and vice versa. Conversely, "unlikely" lenses are ones that require a large amount of additional information to recover one of the formats given the other. By default, our synthesis algorithm prefers the lenses that propagate more information. This preference is formalized using *stochastic regular expressions* (SREs) [10, 54], which simultaneously define a set of strings and a probability distribution over those strings. Using this probability distribution, we can calculate the likelihood of a given lens.

With simple symmetric lenses and this SRE-based likelihood measure in hand, we propose a new algorithm for synthesizing likely lenses. To tame this complexity, we divide the synthesis algorithm into two communicating search procedures (Figure 5.1),

following the approach introduced in Chapter 3. The first, Expand, uses rewriting rules to propose new pairs of stochastic regular expressions equivalent to the original pair. The second, GreedySynth (which replaces RigidSynth) uses a greedy, type-directed algorithm to find a simple symmetric lens between input SRE pairs, returning the lens and its likelihood score to Expand. The whole synthesis algorithm heuristically terminates when a sufficiently likely lens is found.

We added this synthesis procedure to the Boomerang system and explored its effectiveness on a range of applications. Users set up a Boomerang synthesis task by providing two regular expressions and optionally supplying input-output examples. Users can also override the default mechanism for calculating the information content of a SRE by asserting that certain strings are *essential* or *irrelevant*, forcing certain data to either be retained or discarded during the transformations.

We evaluate our implementation, the effects of optimizations, and our inclusion of relevance annotations on a set of 48 lens synthesis benchmarks drawn from data cleaning, view maintenance, and file synchronization tasks, including the 39 benchmarks described in Chapter 3. We find the system can synthesize simple symmetric lenses for all of the benchmarks in under 30 seconds (§5.8). Most of the content in this chapter comes from the paper "Synthesizing Symmetric Lenses" [45].

5.2 Simple Symmetric Lenses

Symmetric lens are used to synchronize files, where each file has relevant information that is not present in the other file. Consider the example of synchronizing management and HR employee files, shown in Figure 5.2. In this company, management and human resources (HR) store information about employees in separate text files: management stores the names of employees and their salaries while HR stores the names of employees and their health insurance providers.

Management's data	HR's data
Jane Doe: 38000 John Public: 37500	FirstLast,Company Jane Doe,Healthcare Inc. John Public,Insurance Co.
Management's type	**HR's type**
let salary = number | "unk" let emp_salary = name . " ␣" . name . ": ␣" salary let emp_salaries = "" | emp_salary . ("\n" emp_salary)*	let company = (co_name . ("Co." | "Inc." | "Ltd.")) | "UNK" let emp_ins = name . " ␣" . name "," company let header = "FirstLast,Company" let emp_insurance = header . ("\n" . emp_ins)*

Figure 5.2: Hypothetical example data files and corresponding regular expressions used by management and HR at a company to represent employee salaries and health insurance providers, respectively.

While symmetric lenses are already a well-studied class of lenses, capable of synchronizing these formats, they are not very amenable to synthesis. Instead, we focus on simple symmetric lenses, a restriction on symmetric lenses. Semantically, a simple symmetric lens between sets X and Y comprises four functions subject to four round-tripping laws.

$$\mathsf{createR} : X \to Y \qquad \mathsf{putL}\,(\mathsf{createR}\,x)\,x = x \quad (\textsc{CreatePutRL})$$

$$\mathsf{createL} : Y \to X \qquad \mathsf{putR}\,(\mathsf{createL}\,y)\,y = y \quad (\textsc{CreatePutLR})$$

$$\mathsf{putR} : X \to Y \to Y \qquad \mathsf{putL}\,(\mathsf{putR}\,x\,y)\,x = x \quad (\textsc{PutRL})$$

$$\mathsf{putL} : Y \to X \to X \qquad \mathsf{putR}\,(\mathsf{putL}\,y\,x)\,y = y \quad (\textsc{PutLR})$$

The two create functions are used to fill in default values when introducing new data (*e.g.*, on create the "unk" salary entry is added alongside the name to the management file when HR inserts a new employee). The two put functions propagate edits from one format to the other by combining a new value from one with an old value from the other. The record projection notation $\ell.\mathsf{putR}$ extracts the putR function from the lens ℓ. These four functions can be used to keep the common information between two file types in sync. For example, if a new file of the left-hand format is created, the createR function will build a new file in the right-hand format. If the right-hand file is then edited, the putL function can update the left-hand file with the changed information.

The round-tripping laws guarantee that pushing an "unedited" value from one side (the result of a put or create) back through a lens in the other direction will

get back to exactly where we began. For example, consider a lens synchronizing the employee data formats in Figure 5.2. Applying the createR function to a salary file creates an insurance file with the same employee names in the file, with "UNK" for each insurance company. The round-tripping laws guarantees that applying the putL function to the generated insurance file and the original salary file will return the original salary file, unmodified.

Simple symmetric lenses differ from "classical" symmetric lenses in that they do not involve a *complement*. We give a detailed comparison in Chapter 6.

5.3 Simple Symmetric Lens Language

Simple symmetric lenses can be written using the following lens combinators, ℓ.

$$
\begin{aligned}
\ell ::= \quad & \mathsf{id}(R) \\
\mid \; & \ell^* \\
\mid \; & \mathsf{concat}(\ell_1, \ell_2) \\
\mid \; & \mathsf{swap}(\ell_1, \ell_2) \\
\mid \; & \mathsf{or}(\ell_1, \ell_2) \\
\mid \; & \ell_1 \; ; \; \ell_2 \\
\mid \; & \mathsf{invert}(\ell_1)\ell_2 \\
\mid \; & \mathsf{disconnect}(R, S, s \in \Sigma^*, t \in \Sigma^*)
\end{aligned}
$$

If R and S are regular expressions, then $\ell : \overline{R} \Leftrightarrow \overline{S}$ indicates that ℓ is a simple symmetric lens between $\mathcal{L}(R)$ and $\mathcal{L}(S)$. (We will use undecorated variables later for stochastic regular expressions, so for the remainder of this chapter, we mark plain REs with overbars.) We illustrate some of these combinators by defining lenses on subcomponents of the employee data formats.

The semantics for simple symmetric lenses are only well-defined when the lens is well-typed, so we shall introduce the lens typing rules at the same time as the semantics.

$$\frac{}{\mathsf{id}(\overline{R}) : \overline{R} \Leftrightarrow \overline{R}}$$

$$\begin{aligned} \mathsf{createR}\, s &= s \\ \mathsf{createL}\, s &= s \\ \mathsf{putR}\, s\, t &= s \\ \mathsf{putL}\, s\, t &= s \end{aligned}$$

The simplest combinator is the identity lens id, which takes as an argument a regular expression $\overline{R}$ and propagates data unchanged in both directions. For example, as names should be synchronized across the data formats, an id lens could perform that synchronization.

$\mathsf{id}(\mathsf{name})$: $\mathsf{name} \Leftrightarrow \mathsf{name}$

The identity lens moves data back and forth from source to target without changing it. Both the createR and createL functions are the identity function (so $\mathsf{createR}\, s = s$), and the put functions merely return the first argument (so $\mathsf{putR}\, s\, t = s$). Because the two formats are fully synchronized, no knowledge of the prior data is needed.

$$\frac{s \in \mathcal{L}(\overline{R}) \qquad t \in \mathcal{L}(\overline{S})}{\mathsf{disconnect}(\overline{R}, \overline{S}, s, t) : \overline{R} \Leftrightarrow \overline{S}}$$

$$\begin{aligned} \mathsf{createR}\, s' &= t \\ \mathsf{createL}\, t' &= s \\ \mathsf{putR}\, s'\, t' &= t' \\ \mathsf{putL}\, t'\, s' &= s' \end{aligned}$$

In contrast to the identity lens, disconnect(R, S, s, t) does not propagate any data at all from one format to the other. The disconnect lens takes four arguments: two regular expressions (R, S) and two strings (s, t). The regular expressions specify the formats on the two sides, while the strings provide default values. Disconnect lenses are used to remove information – for example, removing salary information when transforming from the management format to the HR format.

disconnect(salary, "", "unk", "") : salary $\Leftrightarrow$ ""

On creates, the input values are thrown away, and default values are returned (createL t ="unk"), and on puts, the second argument is used and the first is thrown away (putR s t = t). For example, the salary field is only present in management files, so the disconnect lens can ensure salary edits do not cause updates to the HR file. With the above lens, putL"" 60000 will return 60000, and putR will always return "".

The insert lens ins and the delete lens del are syntactic sugar for uses of the disconnect lens in which a string constant is omitted entirely from the source or target format.

ins(t) = disconnect("", t, "", t)

del(s) = disconnect(s, "", s, "")

The ins lens inserts a constant string when going from left to right, while del inserts a string when going from right to left.

$$\frac{\ell_1 : \overline{R}_1 \Leftrightarrow \overline{S}_1 \qquad \ell_2 : \overline{R}_2 \Leftrightarrow \overline{S}_2 \qquad \overline{R}_1 \cdot^! \overline{R}_2 \qquad \overline{S}_1 \cdot^! \overline{S}_2}{\mathsf{concat}(\ell_1, \ell_2) : \overline{R}_1 \cdot \overline{R}_2 \Leftrightarrow \overline{S}_1 \cdot \overline{S}_2}$$

$$\begin{aligned}
\mathsf{createR}\ s_1 s_2 &= (\ell_1.\mathsf{createR}\ s_1)(\ell_2.\mathsf{createR}\ s_2) \\
\mathsf{createL}\ t_1 t_2 &= (\ell_1.\mathsf{createL}\ t_1)(\ell_2.\mathsf{createL}\ t_2) \\
\mathsf{putR}\ (s_1 s_2)\ (t_1 t_2) &= (\ell_1.\mathsf{putR}\ s_1\ t_1)(\ell_2.\mathsf{putR}\ s_2\ t_2) \\
\mathsf{putL}\ (t_1 t_2)\ (s_1 s_2) &= (\ell_1.\mathsf{putL}\ t_1\ s_1)(\ell_2.\mathsf{putL}\ t_2\ s_2)
\end{aligned}$$

Concat is similar to concatenation in our bijective lens language. The concat lens structurally combines its sublenses; we could use the lens

$\mathsf{id}(\mathsf{name})\ .\ \mathsf{id}(\text{"}\,␣\text{"})\ .\ \mathsf{id}(\mathsf{name})\ .\ \mathsf{del}(\text{":}\,␣\text{"})\ .\ \mathsf{ins}(\text{","})$

to transform "Jane Doe: " to "Jane Doe," in the left-to-right direction.

$$\frac{\ell_1 : \overline{R}_1 \Leftrightarrow \overline{S}_1 \qquad \ell_2 : \overline{R}_2 \Leftrightarrow \overline{S}_2 \qquad \overline{R}_1 \mathbin{.^!} \overline{R}_2 \qquad \overline{S}_2 \mathbin{.^!} \overline{S}_1}{\mathsf{swap}(\ell_1, \ell_2) : \overline{R}_1 \cdot \overline{R}_2 \Leftrightarrow \overline{S}_2 \cdot \overline{S}_1}$$

$$\begin{aligned}
\mathsf{createR}\ s_1 s_2 &= (\ell_2.\mathsf{createR}\ s_2)(\ell_1.\mathsf{createR}\ s_1) \\
\mathsf{createL}\ t_2 t_1 &= (\ell_1.\mathsf{createL}\ t_1)(\ell_2.\mathsf{createL}\ t_2) \\
\mathsf{putR}\ (s_1 s_2)\ (t_2 t_1) &= (\ell_2.\mathsf{putR}\ s_2\ t_2)(\ell_1.\mathsf{putR}\ s_1\ t_1) \\
\mathsf{putL}\ (t_2 t_1)\ (s_1 s_2) &= (\ell_1.\mathsf{putL}\ t_1\ s_1)(\ell_2.\mathsf{putL}\ t_2\ s_2)
\end{aligned}$$

The swap combinator is similar to concat, though the second regular expression is swapped.

$$\frac{\ell_1 : \overline{R}_1 \Leftrightarrow \overline{S}_1 \qquad \ell_2 : \overline{R}_2 \Leftrightarrow \overline{S}_2 \qquad \mathcal{L}(\overline{R}_1) \cap \mathcal{L}(\overline{R}_2) = \emptyset \qquad \mathcal{L}(\overline{S}_1) \cap \mathcal{L}(\overline{S}_2) = \emptyset}{\mathsf{or}(\ell_1, \ell_2) : \overline{R}_1 \mid \overline{R}_2 \Leftrightarrow \overline{S}_1 \mid \overline{S}_2}$$

$$\mathsf{createR}\ s = \begin{cases} \ell_1.\mathsf{createR}\ s & \text{if } s \in \mathcal{L}(\overline{R}_1) \\ \ell_2.\mathsf{createR}\ s & \text{if } s \in \mathcal{L}(\overline{R}_2) \end{cases}$$

$$\mathsf{createL}\ t = \begin{cases} \ell_1.\mathsf{createL}\ t & \text{if } t \in \mathcal{L}(\overline{S}_1) \\ \ell_2.\mathsf{createL}\ t & \text{if } t \in \mathcal{L}(\overline{S}_2) \end{cases}$$

$$\mathsf{putR}\ s\ t = \begin{cases} \ell_1.\mathsf{putR}\ s\ t & \text{if } s \in \mathcal{L}(\overline{R}_1) \ \wedge\ t \in \mathcal{L}(\overline{S}_1) \\ \ell_2.\mathsf{putR}\ s\ t & \text{if } s \in \mathcal{L}(\overline{R}_2) \ \wedge\ t \in \mathcal{L}(\overline{S}_2) \\ \ell_1.\mathsf{createR}\ s & \text{if } s \in \mathcal{L}(\overline{R}_1) \ \wedge\ t \in \mathcal{L}(\overline{S}_2) \\ \ell_2.\mathsf{createR}\ s & \text{if } s \in \mathcal{L}(\overline{R}_2) \ \wedge\ t \in \mathcal{L}(\overline{S}_1) \end{cases}$$

$$\mathsf{putL}\ t\ s = \begin{cases} \ell_1.\mathsf{putL}\ t\ s & \text{if } t \in \mathcal{L}(\overline{S}_1) \ \wedge\ s \in \mathcal{L}(\overline{R}_1) \\ \ell_2.\mathsf{putL}\ t\ s & \text{if } t \in \mathcal{L}(\overline{S}_2) \ \wedge\ s \in \mathcal{L}(\overline{R}_2) \\ \ell_1.\mathsf{createL}\ t & \text{if } t \in \mathcal{L}(\overline{S}_1) \ \wedge\ s \in \mathcal{L}(\overline{R}_2) \\ \ell_2.\mathsf{createL}\ s & \text{if } t \in \mathcal{L}(\overline{S}_2) \ \wedge\ s \in \mathcal{L}(\overline{R}_1) \end{cases}$$

The or lens deals with data that can come in one form or another. If the data gets changed from one format to the other, information in the old format is lost.

$$\frac{\ell : \overline{R} \Leftrightarrow \overline{S} \qquad \overline{R}^{*!} \qquad \overline{S}^{*!}}{\ell^* : \overline{R}^* \Leftrightarrow \overline{S}^*}$$

$$\mathsf{createR}\ s_1 \ldots s_n = (\ell.\mathsf{createR}\ s_1) \ldots (\ell.\mathsf{createR}\ s_n)$$

$$\mathsf{createL}\ t_1 \ldots t_n = (\ell.\mathsf{createL}\ t_1) \ldots (\ell.\mathsf{createL}\ t_n)$$

$$\mathsf{putR}\ (s_1 \ldots s_n)\ (t_1 \ldots t_m) = t'_1 \ldots t'_n \text{ where } t'_i = \begin{cases} \ell.\mathsf{putR}\ s_i\ t_i & \text{if } i \leq m \\ \ell.\mathsf{createR}\ s_i & \text{otherwise} \end{cases}$$

$$\mathsf{putL}\ (t_1 \ldots t_m)\ (s_1 \ldots s_n) = s'_1 \ldots s'_n \text{ where } s'_i = \begin{cases} \ell.\mathsf{putR}\ t_i\ s_i & \text{if } i \leq n \\ \ell.\mathsf{createR}\ t_i & \text{otherwise} \end{cases}$$

The iterate lens is useful for synchronizing a series of items or rows in a table. For example given a lens employee_lens that synchronizes data for a single employee, the lens

(id("\n") . employee_lens)* : ("\n" . emp_salary)* ⇔ ("\n" . emp_ins)*

transforms a list of employees in employees in the Management format to a list of employees in the HR format and vice versa.

$$\frac{\ell_1 : \overline{R}_1 \Leftrightarrow \overline{S} \qquad \ell_2 : \overline{R}_2 \Leftrightarrow \overline{S} \qquad \mathcal{L}(\overline{R}_1) \cap \mathcal{L}(\overline{R}_2) = \emptyset}{\mathsf{merge\_right}(\ell_1, \ell_2) : \overline{R}_1 \mid \overline{R}_2 \Leftrightarrow \overline{S}}$$

$$\mathsf{createR}\ s = \begin{cases} \ell_1.\mathsf{createR}\ s & \text{if } s \in \mathcal{L}(\overline{R}_1) \\ \ell_2.\mathsf{createR}\ s & \text{if } s \in \mathcal{L}(\overline{R}_2) \end{cases}$$

$$\mathsf{createL}\ t = \ell_1.\mathsf{createL}\ t$$

$$\mathsf{putR}\ s\ t = \begin{cases} \ell_1.\mathsf{putR}\ s\ t & \text{if } s \in \mathcal{L}(\overline{R}_1) \\ \ell_2.\mathsf{putR}\ s\ t & \text{if } s \in \mathcal{L}(\overline{R}_2) \end{cases}$$

$$\mathsf{putL}\ t\ s = \begin{cases} \ell_1.\mathsf{putL}\ t\ s & \text{if } s \in \mathcal{L}(\overline{R}_1) \\ \ell_2.\mathsf{putL}\ t\ s & \text{if } s \in \mathcal{L}(\overline{R}_2) \end{cases}$$

The merge_right lens is interesting because when executed from left to right, it converts data in either format $\overline{R}_1$ or $\overline{R}_2$ into a common format $\overline{S}$. In previous work [6], this was combined into or, where or could have ambiguous types.

$$\frac{\ell_1 : \overline{R} \Leftrightarrow \overline{S}_1 \qquad \ell_2 : \overline{R} \Leftrightarrow \overline{S}_2 \qquad \mathcal{L}(\overline{S}_1) \cap \mathcal{L}(\overline{S}_2) = \emptyset}{\mathsf{merge\_left}(\ell_1, \ell_2) : \overline{R} \Leftrightarrow \overline{S}_1 \mid \overline{S}_2}$$

$$\mathsf{createR}\ s = \ell_1.\mathsf{createR}\ s$$

$$\mathsf{createL}\ t = \begin{cases} \ell_1.\mathsf{createL}\ t & \text{if } t \in \mathcal{L}(\overline{S}_1) \\ \ell_2.\mathsf{createL}\ t & \text{if } t \in \mathcal{L}(\overline{S}_2) \end{cases}$$

$$\mathsf{putR}\ s\ t = \begin{cases} \ell_1.\mathsf{putR}\ s\ t & \text{if } t \in \mathcal{L}(\overline{S}_1) \\ \ell_2.\mathsf{putR}\ s\ t & \text{if } t \in \mathcal{L}(\overline{S}_2) \end{cases}$$

$$\mathsf{putL}\ t\ s = \begin{cases} \ell_1.\mathsf{putL}\ t\ s & \text{if } t \in \mathcal{L}(\overline{S}_1) \\ \ell_2.\mathsf{putL}\ t\ s & \text{if } t \in \mathcal{L}(\overline{S}_2) \end{cases}$$

The merge_left lens is symmetric to merge_right.

$$\frac{\ell_1 : \overline{R} \Leftrightarrow \overline{S} \qquad \ell_2 : \overline{S} \Leftrightarrow \overline{T}}{\ell_1 \; ; \; \ell_2 : \overline{R} \Leftrightarrow \overline{T}}$$

$$\begin{aligned}
\mathsf{createR}\; s &= \ell_2.\mathsf{createR}\; (\ell_1.\mathsf{createR}\; s) \\
\mathsf{createL}\; t &= \ell_1.\mathsf{createL}\; (\ell_2.\mathsf{createL}\; t) \\
\mathsf{putR}\; s\; t &= \ell_2.\mathsf{putR}\; (\ell_1.\mathsf{putR}\; s\; (\ell_2.\mathsf{createL}\; t))\; t \\
\mathsf{putL}\; t\; s &= \ell_1.\mathsf{putL}\; (\ell_2.\mathsf{putL}\; t\; (\ell_2.\mathsf{createR}\; s))\; s
\end{aligned}$$

The putR and putL function of composed lens $\ell_1 \; ; \; \ell_2$ is interesting. Because puts require intermediary data, we recreate that intermediary data with creates.

$$\frac{\ell : \overline{R} \Leftrightarrow \overline{S}}{\mathsf{invert}(\ell) : \overline{S} \Leftrightarrow \overline{R}}$$

$$\begin{aligned}
\mathsf{createR}\; t &= \ell.\mathsf{createL}\; t \\
\mathsf{createL}\; s &= \ell.\mathsf{createR}\; s \\
\mathsf{putR}\; t\; s &= \ell.\mathsf{putL}\; t\; s \\
\mathsf{putL}\; s\; t &= \ell.\mathsf{putR}\; s\; t
\end{aligned}$$

The invert combinator is particularly useful when chaining many compositions together, as it can be used to align the central types. For example, with the lenses $\ell_1 : R_1 \Leftrightarrow R_2$ and $\ell_2 : R_3 \Leftrightarrow R_2$, we can construct the composition $\ell_1 \; ; \; \mathsf{invert}(\ell_2) : R_1 \Leftrightarrow R_3$.

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

Type equivalence enables a lens of type $S \Leftrightarrow T$ to be used as a lens of type $S' \Leftrightarrow T'$ if S equivalent to S' and T is equivalent to T'. Type equivalence is useful both for addressing type annotations, and for making well-typed compositions.

```
let name_lens = id(name) . id(" ␣") . id(name) . del(": ␣") . ins(",")
let employee_lens = name_lens . disconnect(salary,"","unk","")
                                . disconnect("",company,"","UNK")
let employees_lens = ins("\n") . employee_lens . iterate(id("\n") . employee_lens)
let full_lens  :  emp_salaries ⇔ emp_insurance = ins(header) . employees_lens
```

Figure 5.3: A lens that synchronizes management and HR employee files

These combinators are combined in figure 5.3 to construct a complete lens between the employee formats.

5.4 Synthesis Overview

Given a regular expression type $\overline{R} \Leftrightarrow \overline{S}$ and a set of input-output examples, we want to find a simple symmetric lens $\ell : \overline{R} \Leftrightarrow \overline{S}$ that satisfies all the input/output examples. Note that, unlike previous synthesis specifications, these input/output examples can take the form of *put examples*, (e.g. putR $s\, t = t'$), instead of just create examples (e.g. createR $s = t$). This can help demonstrate that specific data from one side should be propogated. For our running example, suppose we wish to synthesize a lens between emp_salaries and emp_insurance, using as the input-output examples: createR mgmt_ex = hr_ex and createL hr_ex = mgmt_ex.

mgmt_ex	hr_ex
Jane Doe: 38000 John Public: 37500	FirstLast,Company Jane Doe,Healthcare Inc. John Public,Insurance Co.
emp_salaries	**emp_insurance**
let salary = number | "unk" let emp_salary = name . " ␣" . name . ": ␣" salary let emp_salaries = "" | emp_salary . ("\n" emp_salary)*	let company = (co_name . ("Co." | "Inc." | "Ltd.")) | "UNK" let emp_ins = name . " ␣" . name "," company let header = "FirstLast,Company" let emp_insurance = header . ("\n" . emp_ins)*

One challenge is that the simple symmetric lens combinators permit many well-typed lenses between a given pair of regular expressions. For example, Figure 5.3 gives one possible lens with type emp_salaries ⇔ emp_insurance, but

disconnect(emp_salaries, emp_insurance, mgmt_ex, hr_ex)

is another well-typed lens that satisfies the example in Figure 5.2. In general, *many* examples may be required to rule out all possible occurrences of disconnect lenses, particularly in complex formats. Instead of merely finding *any* satisfying lens, we wish to synthesize a satisfying lens that is likely to please the user.

How can we identify such a "likely" lens? We propose the following heuristic: A satisfying lens is "more likely" if it uses more data from one format to construct the other. For example, the identity lens (which uses all the data) is more likely than the disconnect lens (which uses none). Formally, we define the *likelihood* of a satisfying lens as the expected number of bits required to recover data in one format from data in the other; higher likelihoods correspond to fewer bits. Two strings s and t are *synchronized according to lens* ℓ if ℓ.putR $s\ t = t$ and ℓ.putL $t\ s = s$. We can *recover* s from t using bits b and lens ℓ if we can reconstruct s from t, b, and ℓ. For example, given the id lens, we can recover s from t using no bits because, in this case, s is just t. In contrast, given the disconnect lens, we need enough bits to fully encode s in order to recover it from t because all the information in t gets thrown away. Thus, if both id and disconnect are satisfying lenses for a particular pair of formats, id will be more likely.

The expected number of bits required to recover a piece of data corresponds to the well-known information-theoretic concept of *entropy* [57]. Calculating entropy requires a probability distribution over the space of possible values for the data. Specifically, given a set S and a probability distribution $P : S \to \mathbb{R}$ over S, the entropy $\mathbb{H}(S, P)$ is $-\Sigma_{s \in S} P(s) log_2 P(s)$. In information theory, the *information content* of each element of $s \in S$ is the number of bits required to specify s in a perfect encoding scheme $(-log_2 P(s))$. The entropy of S is the expected number of bits required to specify an element drawn from S—the probability of each element times its information content. Entropy captures the intuition that, if a data source contains many possible elements

and none have significantly higher probability than others, it will have high entropy. Data sources with just a few high probability elements have low entropy. When P is clear from context, we will often use the shorthand of $\mathbb{H}(S)$ for $\mathbb{H}(S, P)$.

In the present setting, we already have a way of expressing sets of data: regular expressions. To calculate entropy, what we need is a way to express probability distributions over those sets. To that end, we adopt *stochastic regular expressions* [54, 10] (SREs), which are regular expressions in which each operator is annotated with a probability (see §5.5). A stochastic regular expression thus specifies both a set of strings at the same time as a probability distribution over that set.

We use entropy to gauge the relative likelihood of lenses in our synthesis algorithm (see §5.6). For any lens ℓ, we can calculate the expected number of bits required to recover a string t in $\mathcal{L}(S)$ from a synchronized string s in $\mathcal{L}(R)$. This expectation is the *conditional entropy of R given S and ℓ*, formally $\sum_{s \in R} P_R(s) \cdot \mathbb{H}(\{t \mid \ell.\mathsf{putR}\, s\, t = t\})$. The likelihood we assign to ℓ is the sum of the conditional entropy of R given S and ℓ and the conditional entropy of S given R and ℓ. This metric assigns higher entropy (or lower likelihood) to lenses where knowing the string on one side provides little information about the string on the other side. It assigns zero entropy to bijections because given a string $s \in R$, the bijection exactly determines the corresponding string in S.

To obtain SREs from the plain regular expressions that users write, we use a default heuristic that attempts to assign probability annotations giving each string in the language equal probability (not prioritizing one piece of information over another).

Sometimes users already know that certain data should or should not be used to construct the other format. We introduce relevance annotations to SREs that enable users to specify whether a piece of data should be used to construct the other format (with require) or not (with skip). In our example, salary and company could safely be skipped (as they are disconnected), where name in emp_salary and emp_ins could be

annotated as required (as they are converted with the identity lens). Doing so both constrains the problem (as names must be synchronized) and makes it easier (as the algorithm does not waste time looking for salary information in the insurance format). In this way, users can tweak the lens likelihoods with their external knowledge.

5.4.1 Searching for Likely Lenses

Given a stochastic regular expression type $R \Leftrightarrow S$ and a set of input-output examples, our algorithm will search for a likely lens with that type. Similarly to when synthesizing bijective lenses, we split the search into two communicating procedures. The first, Expand, navigates the space of semantically equivalent regular expressions by applying rewrite rules that preserve both semantics and probability distributions. This algorithm ranks pairs of stochastic regular expressions by the number of rewrite rule applications required to obtain each pair from the one given as input. It passes the pairs off to the second search procedure, GreedySynth, in rank order with the smallest first.

GreedySynth looks for highest-likelihood lenses between a given pair of stochastic regular expressions R and S by performing a type-directed search. It first converts the stochastic regular expressions provided by Expand into *stochastic DNF regular expressions*—a constrained representation of stochastic regular expressions with disjunctions distributed over concatenations and with concatenations and disjunctions normalized to operating over lists. Then it uses the syntax of these n-ary DNF regular expressions to find normalized lenses in a form we call *simple symmetric n-ary DNF lenses*. These involve neither a composition operator nor a type equivalence rule. These restrictions mean that there are comparatively few simple symmetric n-ary DNF lenses that are well typed between a given pair of stochastic n-ary DNF regular expressions, so GreedySynth's search space for a given pair of regular expressions is finite. Finally, GreedySynth yields a simple symmetric lens by converting the n-ary syntax back to the binary forms provided in the surface language.

This architecture of communicating synthesizers gives us a way to enumerate pairs of stochastic regular expressions of increasing rank and to efficiently search through them, but it poses a problem: when should Expand stop proposing new SRE pairs? We might have found a promising lens between a pair of stochastic regular expressions, but a different pair we haven't yet discovered may give rise to an even better lens. The search algorithm must resolve a tension between the quality of the inferred lens and the amount of time it takes to return a result. For example, if the algorithm has already found the lens in Figure 5.3, we don't want to spend a lot of time searching for an even better lens. To resolve this tension, the algorithm uses heuristics to judge whether to return the current best satisfying lens to the user or to pass the next set of equivalent SREs to GreedySynth. The heuristics favor stopping if the current best satisfying lens is very likely, indicating the lens is very promising (for example, if the satisfying lens loses no information, the algorithm should terminate for no other lens will be more likely). The heuristics also favor stopping if Expand has delivered to GreedySynth all pairs of stochastic regular expressions at a given rank and there is a large number of pairs at the next rank, because searching through all such pairs will take a long time (*i.e.*, if a satisfying lens loses only a little information and searching for a better one will take a long time, the discovered lens is returned). With this approach, the algorithm can quickly return a satisfying lens with relatively high likelihood. If the user is unhappy with the result, they can either refine the search by supplying additional examples, which serve both to rule out previously proposed lenses and to reduce the size of the search space by cutting down on the number of satisfying lenses, or they can supply annotations on the source and target SREs indicating that certain information either must be retained or must be discarded by the lens (see §5.7.3).

5.5 Stochastic Regular Expressions

To characterize likely lenses, we must compute the expected number of bits needed to recover a string in one data source from a synchronized string in the other data source. To do this, we first develop a probabilistic model for our language using *stochastic regular expressions* (SREs)—regular expressions annotated with probability information [10, 54]—that jointly express a language and a probability distribution over that language.

$$R, S \quad ::= \quad s \quad | \quad \emptyset \quad | \quad R^{*_p} \quad | \quad R_1 \cdot R_2 \quad | \quad R_1 \mid_p R_2$$

(Lowercase s ranges over constant strings and p ranges over real numbers between 0 and 1, exclusive.) The semantics of an SRE S is a probability distribution P_S defined as follows.

$$\begin{aligned}
P_s(s'') &= \begin{cases} 1 & \text{if } s = s'' \\ 0 & \text{otherwise} \end{cases} \\
P_\emptyset(s) &= 0 \\
P_{R_1 \cdot R_2}(s) &= \Sigma_{s=s_1 s_2} P_{R_1}(s_1) P_{R_2}(s_2) \\
P_{R_1 \mid_p R_2}(s) &= p P_{R_1}(s) + (1-p) P_{R_2}(s) \\
P_{R^{*_p}}(s) &= \Sigma_n \Sigma_{s=s_1 \ldots s_n} p^n (1-p) \Pi_{i=1}^n P_R(s_i)
\end{aligned}$$

One may think of SREs as string generators. Under this interpretation, the constant SRE s always generates the string s and never any other string. The SRE $R_1 \mid_p R_2$ generates a string from R_1 with probability p and generates a string from R_2 with probability $1-p$. The SRE R^{*_p} generates strings in R^n with probability $p^n(1-p)$. For example, R^{*_p} will generate a string in R with probability $p(1-p)$ and a string in $R \cdot R$ with probability $p^2(1-p)$.

$$
\begin{array}{rcll}
\emptyset \cdot R & \equiv^s & \emptyset & 0\ Proj_L \\
R \cdot \emptyset & \equiv^s & \emptyset & 0\ Proj_R \\
\epsilon \cdot R & \equiv^s & R & \cdot\ Ident_L \\
R \cdot \epsilon & \equiv^s & R & \cdot\ Ident_R \\
R \mid_1 \emptyset & \equiv^s & R & +\ Ident \\
R \mid_p S & \equiv^s & S \mid_{1-p} R & \mid\ Comm \\
R \cdot (R' \mid_p R'') & \equiv^s & (R \cdot R') \mid_p (R \cdot R'') & Dist_R \\
(R' \mid_p R'') \cdot R & \equiv^s & (R' \cdot R) \mid_p (R'' \cdot R) & Dist_L \\
R^{*_p} & \equiv^s & \epsilon \mid_{1-p} (R \cdot R^{*_p}) & Unrollstar_L \\
R^{*_p} & \equiv^s & \epsilon \mid_{1-p} (R^{*_p} \cdot R) & Unrollstar_R \\
(R \cdot R') \cdot R'' & \equiv^s & R \cdot (R' \cdot R'') & \cdot\ Assoc \\
(R \mid_{p_1} R') \mid_{p_2} R'' & \equiv^s & R \mid_{p_1 p_2} (R' \mid_{\frac{(1-p_1)p_2}{1-p_1 p_2}} R'') & \mid\ Assoc
\end{array}
$$

Figure 5.4: Stochastic Regular Expression Star-Semiring Equivalence

5.5.1 Stochastic Regular Expression Equivalences

The EXPAND algorithm enumerates SREs that are "equivalent" to a given one. However, existing work does not define any notion of stochastic regular expression equivalence. Figure 5.4 shows how we extend the *star-semiring* [33] equivalences (a *finer* notion of equivalence than semantic equivalence) to SREs.

Theorem 9. If $R \equiv^s S$, then $P_R = P_S$.

This theorem will come in handy as we traverse (with EXPAND) or normalize across (as part of GREEDYSYNTH) star-semiring equivalences.

5.5.2 Stochastic Regular Expression Entropy

The entropy of a data source S is the expected number of bits required to describe an element drawn from S (formally $-\Sigma_{s \in S}\, P(s) \cdot log_2 P(s)$). The entropy of a SRE can be computed directly from its syntax, when each string is uniquely parsable (*i.e.* the SRE is unambiguous) and contains no empty subcomponents.

$$
\begin{aligned}
\mathbb{H}(s) &= 0 \\
\mathbb{H}(R^{*_p}) &= \tfrac{p}{1-p}(\mathbb{H}(R) - log_2 p) - log_2(1-p) \\
\mathbb{H}(R \cdot S) &= \mathbb{H}(R) + \mathbb{H}(S) \\
\mathbb{H}(R \mid_p S) &= p(\mathbb{H}(R) - log_2(p)) + (1-p)(\mathbb{H}(S) - log_2(1-p))
\end{aligned}
$$

For example, the entropy of $R = \texttt{"a"} \mid_{.5} \texttt{"b"}$ is 1. The best encoding of a stream of elements from R will use, on average, 1 bit per element to determine whether that element is $\texttt{"a"}$ or $\texttt{"b"}$. As an additional example, a fixed string has no information content, and so has no entropy.

Theorem 10. If R is unambiguous and does not contain $\emptyset$ as a subterm, $\mathbb{H}(R)$ is the entropy of R.

To understand the difficulties caused by ambiguity, consider the SRE $\texttt{"a"} \mid_{.5} \texttt{"a"}$. The formula above defines the entropy to be 1, but the true entropy is 0 (*i.e.*, no bits are needed to know the generated string will be $\texttt{"a"}$). Similar issues occur when trying to find the entropy when $\emptyset$ is a subterm (what should the entropy of $\emptyset^{*_{.5}}$ be?), so we do not define entropy on $\emptyset$.

Fortunately, we already require unambiguous regular expressions as input to our synthesis procedure to guarantee the synthesized lenses are well-typed, and we can easily preprocess empty subexpressions out of SREs that are themselves nonempty using the star-semiring equivalences (*e.g.*, $\emptyset \mid_{.5} \texttt{"s"} \equiv^s \texttt{"s"}$).

5.6 Lens Likelihoods

Our likelihood metric is based on the expected amount of information required to recover a string in one data format from the other. We use the function $\mathbb{H}^{\rightarrow}(S \mid \ell, R)$ to calculate bounds on the expected amount of information required to recover a string in S from a string in R, synchronized by ℓ. Similarly, we use the function $\mathbb{H}^{\leftarrow}(R \mid \ell, S)$ to calculate bounds on the expected amount of information required to

recover a string in R from a string in S, synchronized by ℓ. We write $a[b,c]$ to mean $[ab, ac]$, and $[a,b] + [c,d]$ to mean $[a+c, b+d]$.

$$
\begin{aligned}
\mathbb{H}^{\rightarrow}(S \mid \mathsf{id}(S), S) &= [0,0] \\
\mathbb{H}^{\rightarrow}(S \mid \mathsf{disconnect}(R,S,s,t), R) &= [\mathbb{H}(S),\mathbb{H}(S)] \\
\mathbb{H}^{\rightarrow}(S^{*_q} \mid \ell^*, R^{*_p}) &= \tfrac{p}{1-p}\mathbb{H}^{\rightarrow}(S \mid \ell, R) \\
\mathbb{H}^{\rightarrow}(S_1 \cdot S_2 \mid \mathsf{concat}(\ell_1,\ell_2), R_1 \cdot R_2) &= \mathbb{H}^{\rightarrow}(S_1 \mid \ell_1, R_1) + \mathbb{H}^{\rightarrow}(S_2 \mid \ell_2, R_2) \\
\mathbb{H}^{\rightarrow}(S_2 \cdot S_1 \mid \mathsf{swap}(\ell_1,\ell_2), R_1 \cdot R_2) &= \mathbb{H}^{\rightarrow}(S_2 \mid \ell_1, R_2) + \mathbb{H}^{\rightarrow}(S_1 \mid \ell_1, R_1) \\
\mathbb{H}^{\rightarrow}(S_1 \mid_q S_2 \mid \mathsf{or}(\ell_1,\ell_2), R_1 \mid_p R_2) &= p\mathbb{H}^{\rightarrow}(R_1 \mid \ell_1, S_1) + (1-p)\mathbb{H}^{\rightarrow}(R_2 \mid \ell_2, S_2) \\
\mathbb{H}^{\rightarrow}(S \mid \mathsf{merge\_right}(\ell_1,\ell_2), R_1 \mid_p R_2) &= p\mathbb{H}^{\rightarrow}(S \mid \ell_1, R_1) + (1-p)\mathbb{H}^{\rightarrow}(S \mid \ell_2, R_2) \\
\mathbb{H}^{\rightarrow}(S_1 \mid_q S_2 \mid \mathsf{merge\_left}(\ell_1,\ell_2), R) &= [0, \mathbb{H}^{\rightarrow}(S_1 \mid \ell_1, R) + \mathbb{H}^{\rightarrow}(S_2 \mid \ell_2, R) + 1] \\
\mathbb{H}^{\rightarrow}(R \mid \mathsf{invert}(\ell), S) &= \mathbb{H}^{\leftarrow}(R \mid \ell, S)
\end{aligned}
$$

$\mathbb{H}^{\leftarrow}(R \mid \ell, S)$ is defined symmetrically. These functions bound the expected number of bits to recover one data format from a synchronized string in the other format. Note that we would be able to exactly calculate the conditional entropy, were it not for merge_left and merge_right. If $\mathsf{merge\_left}(\ell_1,\ell_2) : R \Leftrightarrow S_1 \mid_q S_2$, given a string in s, we need to determine if the synchronized string is in S_1 or S_2. However, this information content is dependent on how likely the synchronized string is to be in S_1 or S_2. Nevertheless, we typically calculate the conditional entropy exactly, as merges are relatively uncommon in practice; only 2 of the lenses synthesized in our benchmarks include merges.

The likelihood of a lens is the negative of its *cost*. The cost of a lens between two SREs $cost(\ell, R, S) = \max(\mathbb{H}^{\leftarrow}(R \mid \ell, S)) + \max(\mathbb{H}^{\rightarrow}(S \mid \ell, R))$ is the sum of the maximum expected number of bits to recover the left format from the right, and the right from the left. We have proven theorems demonstrating the calculated entropy corresponds to the actual conditional entropy to recover the data.

Theorem 11. Let $\ell : R \Leftrightarrow S$, where ℓ does not include composition, R and S are unambiguous, and neither R nor S contain any empty subcomponents.

1. $\mathbb{H}^{\rightarrow}(S \mid \ell, R)$ bounds the entropy of $\{t \mid t \in \mathcal{L}(S)\}$, given $\{s \mid s \in \mathcal{L}(R) \wedge \ell.\mathsf{putR}\, s\, t = t\}$

2. $\mathbb{H}^{\leftarrow}(R \mid \ell, S)$ bounds the entropy of $\{s \mid s \in \mathcal{L}(R)\}$, given $\{t \mid t \in \mathcal{L}(S) \wedge \ell.\mathsf{putL}\, t\, s = s\}$

Note that our definition of $\mathbb{H}^{\rightarrow}$ contains no case for sequential composition $\ell_1; \ell_2$ and our theorem excludes lenses that contain such compositions. Defining the entropy of lenses involving composition is challenging because ℓ_1 might, for instance, add some information that is subsequently projected away in ℓ_2. Such operations can cancel, leaving a zero-entropy bijection composed from two non-zero entropy transformations. However, detecting such cancellations directly is complicated and this property is difficult to determine merely from syntax. Fortunately, we are able to sidestep such considerations by synthesizing *DNF lenses*—simple symmetric lenses that inhabit a disjunctive normal form that does not include composition.

5.7 Synthesis Algorithm

Algorithm 5 presents our synthesis algorithm at a high level of abstraction. This algorithm searches for likely lenses in priority order one "class" at a time using a GreedySynth subroutine. Each class is the set of lenses that can by typed by a pair of regular expressions, modulo a set of simple axioms such as associativity, commutativity, and distributivity. The Expand subroutine generates new classes using the star-unrolling axioms ($Unrollstar_L$ and $Unrollstar_R$).

To summarize, the input regular expressions are first converted into stochastic regular expressions with ToStochastic. This pair of SREs is used to initialize a

Algorithm 5 SynthSymLens

```
1: function SynthSymLens(R̄, S̄, exs)
2:     R ← ToStochastic(R̄)
3:     S ← ToStochastic(S̄)
4:     pq ← PQ.Create(R, S)
5:     best ← None
6:     while Continue(pq, best) do
7:         (R, S) ← PQ.Pop(pq)
8:         ℓ ← GreedySynth(exs, R, S)
9:         if Cost(ℓ) < Cost(best) then
10:            best ← ℓ
11:        PQ.Push(pq, Expand(R, S))
12:    return best
```

priority queue (*pq*). The priority of a SRE pair is the number of rewrites needed to derive the pair from the originals. Next, SynthSymLens enters a loop that searches for likely lenses. The loop terminates when the algorithm believes it is unlikely to find a better lens than the best one it has found so far (a termination condition defined by Continue). Within each iteration of the loop, it:

- pops the next class (S, T) of lenses to search off of the priority queue (PQ.Pop),
- executes GreedySynth to find a best lens in that class if one exists (ℓ), using the examples *exs* to filter out potential lenses that do not satisfy the specification,
- replaces *best* with ℓ, if ℓ is more likely according to our information-theoretic metric, and
- adds the SREs derived from rewriting S and T (Expand(S, T)) to the priority queue.

When the loop terminates, the search returns the globally best lens found (*best*). Each subroutine of this algorithm will be explained in further depth in the following subsections.

5.7.1 Searching for (R, S) Candidate Classes

The first phase of the synthesis algorithm looks for pairs of SREs (R, S) to drive the GreedySynth algorithm. These pairs are generated using the star unrolling axioms:

$$\begin{aligned} R^{*_p} &\rightarrow \epsilon \mid_{1-p} (R \cdot R^{*_p}) \\ R^{*_p} &\rightarrow \epsilon \mid_{1-p} (R^{*_p} \cdot R) \end{aligned}$$

as well as the congruence rules that allow these rewrites to be applied on subexpressions. The priority queue yields stochastic regular expressions generated using fewer rewrites first. Only when there are no more proposed regular expressions derived from n rewrites will PQ.Pop propose regular expressions derived from $n + 1$ rewrites.

The procedure Continue terminates the loop based on the how long the search has been going, and how hard it expects the next class of problems to be. In particular, if PQ.Peek$(pq) = (R, S)$, that RE pair is at distance d, the number of pairs in pq at distance d is n, and the current best lens has cost c, then Continue(pq) continues the loop while $c < d + log_2(n)$. This termination condition is based around two primary principles: do not search overly deep (d), and do not tackle a frontier that would take too long to process $(log_2(n))$. The log of the frontier is included because the frontier grows much faster than lens costs grow. Lens cost grows with the log of the expected number of choices in a given lens (due to its information theoretic basis), so the frontier calculation does too.

When Continue(pq) terminates the loop, the algorithm stops proposing regular expression pairs, and instead returns to the user the best lens found thus far. If the algorithm finds a bijective lens, which has zero cost, it will immediately return.

5.7.2 Stochastic DNF Regular Expressions

The GreedySynth subroutine converts input SREs into a temporary pseudo-normal form, *Stochastic DNF regular expressions* (SDNF REs). SDNF REs normalize across

many of the star-semiring equivalences – if two regular expressions are equivalent modulo differences in associativity, commutativity, or distributivity, their corresponding SREs are syntactically equal.

Syntactically, stochastic DNF regular expressions (DS, DT) are lists of stochastic sequences. Stochastic sequences (SQ, TQ) themselves are lists of interleaved strings and stochastic atoms. Stochastic atoms (A,B) are iterated stochastic DNF regular expressions.

$$\begin{aligned} A,B &::= DS^{*_p} \\ SQ,TQ &::= [s_0 \cdot A_1 \cdot \ldots \cdot A_n \cdot s_n] \\ DS,DT &::= \langle (SQ_1, p_1) \mid \ldots \mid (SQ_n, p_n) \rangle \end{aligned}$$

Intuitively, stochastic DNF regular expressions are stochastic regular expressions with all concatenations fully distributed over all disjunctions. As such, the language of a stochastic DNF regular expression is a union of its subcomponents, the language of a stochastic sequence is the concatenation of its subcomponents, and the language of a stochastic atom is the iteration of its subcomponent. For $\langle (SQ_1, p_1) \mid \ldots \mid (SQ_n, p_n) \rangle$ to be a valid stochastic DNF regular expression, the probabilities must sum to one ($\sum_{i=0}^{n} p_i = 1$).

$$\begin{aligned} \mathcal{L}(DS^{*_p}) &= \{s_1 \cdot \ldots \cdot s_n \mid \forall i, s_i \in \mathcal{L}(DS)\} \\ \mathcal{L}([s_0 \cdot A_1 \cdot \ldots \cdot A_n \cdot s_n]) &= \{s_0 \cdot t_1 \cdots t_n \cdot s_n \mid t_i \in \mathcal{L}(A_i)\} \\ \mathcal{L}(\langle (SQ_1, p_1) \mid \ldots \mid (SQ_n, p_n) \rangle) &= \{s \mid s \in \mathcal{L}(SQ_i) \text{ and } i \in [1, n]\} \end{aligned}$$

As these DNF regular expressions are *stochastic*, they are annotated with probabilities to express a probability distribution, in addition to a language.

$$\begin{aligned} P_{DS^{*_p}}(s) &= \Sigma_n \Sigma_{s=s_1 \ldots s_n} p^n (1-p) \Pi_{i=1}^{n} P_{DS}(s_i) \\ P_{[s_0 \cdot A_1 \cdot \ldots \cdot A_n \cdot s_n]}(s') &= \Sigma_{s'=s_0 s'_1 \ldots s'_n s_n} \Pi_{i=1}^{n} P_{A_i}(s'_i) \\ P_{\langle (SQ_1, p_1) \mid \ldots \mid (SQ_n, p_n) \rangle}(s) &= \Sigma_{i=1}^{n} p_i P_{SQ_i}(s) \end{aligned}$$

$\odot_{SQ} : \mathit{Sequence} \to \mathit{Sequence} \to \mathit{Sequence}$
$[s_0 \cdot A_1 \cdot \ldots \cdot A_n \cdot s_n] \odot_{SQ} [t_0 \cdot B_1 \cdot \ldots \cdot B_m \cdot t_m] =$
$[s_0 \cdot A_1 \cdot \ldots \cdot A_n \cdot s_n \cdot t_0 \cdot B_1 \cdot \ldots \cdot B_m \cdot t_m]$

$\odot : \mathit{DNF} \to \mathit{DNF} \to \mathit{DNF}$
$\langle (SQ_1, p_1) \mid \ldots \mid (SQ_n, p_n) \rangle \odot \langle (TQ_1, q_1) \mid \ldots \mid (TQ_m, q_m) \rangle =$
$\langle (SQ_1 \odot_{SQ} TQ_1, p_1 q_1) \mid \ldots \mid (SQ_1 \odot_{SQ} TQ_m, p_1 q_m) \mid \ldots$
$\mid (SQ_n \odot_{SQ} TQ_1, p_n q_1) \mid \ldots \mid (SQ_n \odot_{SQ} TQ_m, p_n q_m) \rangle$

$\oplus_p : \mathit{DNF} \to \mathit{DNF} \to \mathit{DNF}$
$\langle (SQ_1, p_1) \mid \ldots \mid (SQ_n, p_n) \rangle \oplus \langle (TQ_1, q_1) \mid \ldots \mid (TQ_m, q_m) \rangle p =$
$\langle (SQ_1, p_1 p) \mid \ldots \mid (SQ_n, p_n p) \mid (TQ_1, q_1(1-p)) \mid \ldots \mid (TQ_m, q_m(1-p)) \rangle$

$\mathcal{D} : \mathit{Atom} \to \mathit{DNF}$
$\mathcal{D}(A) = \langle ([\epsilon \cdot A \cdot \epsilon], 1) \rangle$

Figure 5.5: Stochastic DNF Regular Expression Functions

The algorithm for converting a stochastic regular expressions R into its corresponding SDNF RE form, written $\Downarrow R$, is defined below. This conversion relies on operators defined in Figure 5.5.

$$\begin{aligned} \Downarrow s &= \langle ([s], 1) \rangle & \Downarrow (R_1 \cdot R_2) &= \Downarrow R_1 \odot \Downarrow R_2 \\ \Downarrow \emptyset &= \langle \rangle & \Downarrow (R_1 \mid_p R_2) &= \Downarrow R_1 \oplus_p \Downarrow R_2 \\ \Downarrow (R^{*_p}) &= \mathcal{D}((\Downarrow R)^{*_p}) \end{aligned}$$

After this syntactic conversion has taken place, the sequences are ordered (normalizing commutativity differences). This conversion respects languages and probability distributions.

Theorem 12. $P_R(s) = P_{\Downarrow R}(s)$ and $\mathcal{L}(R) = \mathcal{L}(\Downarrow R)$.

Entropy We have developed a syntactic means for finding the entropy of a stochastic DNF regular expression, like we have for stochastic regular expressions. This enables us to efficiently find the entropy without first converting a SDNF RE to a stochastic regular expression.

$$\begin{aligned}
\mathbb{H}(DS^{*_p}) &= \tfrac{p}{1-p}(\mathbb{H}(DS) - log_2 p) - log_2(1-p) \\
\mathbb{H}([s_0 \cdot A_1 \cdot \ldots \cdot A_n \cdot s_n]) &= \Sigma_{i=1}^{n} \mathbb{H}(A_i) \\
\mathbb{H}(\langle (SQ_1, p_1) \mid \ldots \mid (SQ_n, p_n) \rangle) &= \Sigma_{i=1}^{n} p_i(\mathbb{H}(SQ_i) + log_2 p_i)
\end{aligned}$$

Theorem 13. $\mathbb{H}(DS)$ is the entropy of P_{DS}.

ToStochastic With stochastic DNF regular expressions and $\Downarrow$defined, it is easier to explain ToStochastic, the function that converts regular expressions into stochastic regular expressions. If R is a stochastic regular expression generated by ToStochastic, then when put into SDNF RE form, $\Downarrow R = \langle (SQ_1, \frac{1}{n}) \mid \ldots \mid (SQ_n, \frac{1}{n}) \rangle$ for some sequences $SQ_1 \ldots SQ_n$, and every stochastic atom generated by ToStochasticis $DS^{*.8}$. In particular, $\Downarrow$ generates regular expressions whose DNF form gives equal probability to all sequence subcomponents of the SDNF REs, and gives a .8 chance for stars to continue iterating. In our experience generating random strings from regular expressions, these probabilities provide good distributions of strings—stars are iterated 4 times on average, and no individual choice in a series of disjunctions is chosen disproportionately often. It would be interesting future work to see how changing the probability of iterating, or changing the distribution of strings, impacts how well our algorithm performs.

5.7.3 Relevance Annotations

Even though our generated probability distribution works well in most situations, it is not perfect. Consider synthesizing a lens between the formats shown in Figure 5.2. Because salary information is present in emp_salaries, but not in emp_insurance, the algorithm might spend a long time (fruitlessly) trying to construct lenses that transform the salary to information present in emp_insurance even though that is impossible. In a similar but more elaborate example, such wasted processing effort may cause synthesis to fail to terminate in any reasonable amount of time.

One solution would be to cut off the search early. However, then one runs into the opposite problem: In other scenarios, salary information may be present in the other format, but it may take quite a bit of work to find a transformation that connects the salary in emp_salaries to the salary in the second format. Hence, early termination may cut off synthesis before the right lens is found.

We can solve both problems by allowing users to augment the specifications with *relevance annotations*. Sometimes, users have external knowledge that certain information appears exclusively in one format or the other, or they may know the information is present both formats. By communicating this knowledge to the synthesis algorithm through relevance annotations, users can force the synthesis of lenses that discard or retain certain information. The first annotation, skip(R), says the information of R appears only in R, and can safely be projected. The second annotation, require(R), says the information of R appears in the other format and cannot be discarded.

For example, users can easily recognize that salary information is not present in emp_insurance. By annotating the salary field as skip(salary), users can add this knowledge to the specification to optimize the search. In practice, we define the information content of skip(R) to be zero.

$$\mathbb{H}(\mathsf{skip}(R)) = 0$$

Similarly, users can recognize that employee names are present in both files. By annotating instances of name as require(name), users can add this knowledge to the specification to optimize the search and force the generated lens to retain name information. In practice, we make any lens that loses "required" information infinitely unlikely.

$$\mathbb{H}^{\rightarrow}(\mathsf{require}(S) \mid \ell, R) = \begin{cases} \infty & \text{if } \mathbb{H}^{\rightarrow}(\mathsf{require}(S) \mid \ell, R) \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbb{H}^{\rightarrow}(S \mid \mathsf{disconnect}(R, S, s, t), R) = \begin{cases} \infty & \text{if } S \text{ contains } \mathsf{require}(S') \text{ as a subexpression, where } \mathbb{H}(S') \neq 0 \\ [\mathbb{H}(S), \mathbb{H}(S)] & \text{otherwise} \end{cases}$$

By adding in relevance annotations, we override the default metric, and Theorem 11 no longer holds. This is expected and desired. We intentionally want to break the theorem, as minimizing entropy lost is no longer the first priority – the first priority becomes to retain required information, and to lose skipped information. After these priorities are achieved, the default metric determines what parts of the remaining information is lost, and what parts are retained.

5.7.4 Symmetric DNF Lenses

Symmetric DNF lenses are an intermediate synthesis target for GreedySynth. There are many fewer symmetric DNF lenses than symmetric regular lenses. In fact, if one does not use the star-unrolling axioms, there are only finitely many DNF lenses of a given type (though there are still many more symmetric DNF lenses than DNF bijective lenses).

The structure of symmetric DNF lenses mirrors that of SDNF REs. A symmetric DNF lens (sdl) is a union of symmetric sequence lenses, a symmetric sequence lens ($ssql$) is a concatenation of symmetric atom lenses, and a symmetric atom lens (sal) is an iteration of a symmetric DNF lens.

Just as we analyzed the information content of ordinary regular expressions, we can analyze the information content of DNF regular expressions. As before, we

use $\mathbb{H}^{\rightarrow}(DT \mid sdl, DS)$ to calculate bounds on the expected amount of information required to recover a string in DT from a string in DS, synchronized by sdl. We use the function $\mathbb{H}^{\leftarrow}(DS \mid sdl, DT)$ to calculate bounds on the expected amount of information required to recover a string in DS from a string in DT, synchronized by sdl.

The details of these definitions are syntactically tedious, but not intellectually difficult. We elide them here but include them in the full version of the original paper [44]. In Chapter 3, we proved DNF lenses are equivalent in expressiveness to standard lenses. While we conjecture symmetric DNF lenses are equivalent in expressivity to our standard symmetric lenses, we have not proven this equivalence.

5.7.5 GreedySynth

The synthesis procedure comprises three algorithms: one that greedily finds symmetric DNF lenses (GreedySynth), one that greedily finds symmetric sequence lenses (GreedySeqSynth), and one that finds symmetric atom lenses (GreedyAtomSynth). These three algorithms are hierarchically structured: GreedySynth relies on GreedySeqSynth, GreedySeqSynth relies on GreedyAtomSynth, and GreedyAtomSynth relies on GreedySynth. The structure of the algorithms mirrors the structure of symmetric DNF lenses and SDNF REs.

Symmetric DNF Lenses Algorithm 6 presents GreedySynth, which synthesizes symmetric DNF lenses. Its inputs are a suite of input-output examples and a pair of stochastic DNF regular expressions. First, NoMap determines whether there is no lens satisfying the examples, because either the examples are contradictory, or if the output parses have no corresponding input parses. If there are no such lenses, GreedySynth returns *None* immediately. Otherwise, GreedySynthfinds the best lenses (given the examples that match them) between all sequence pairs (SQ_i, TQ_j) drawn from

Algorithm 6 GreedySynth

```
1: function GreedySynth(exs, DS, DT)
2:     ⟨(SQ_1, p_1) | ... | (SQ_n, p_n)⟩ ← DS
3:     ⟨(TQ_1, q_1) | ... | (TQ_m, q_m)⟩ ← DT
4:     if NoMap(exs, ⟨(SQ_1, p_1) | ... | (SQ_n, p_n)⟩, ⟨(TQ_1, q_1) | ... | (TQ_m, q_m)⟩)
   then
5:         return None
6:     SQs ← [SQ_1; ...; SQ_n]
7:     TQs ← [TQ_1; ...; TQ_m]
8:     sls ← CartesianMap(GreedySeqSynth(exs), SQs, TQs)
9:     pq ← PQ.Create(sls)
10:    lb ← LensBuilder.Empty
11:    while PQ.IsNonempty(pq) do
12:        ssql ← PQ.Pop(pq)
13:        if LensBuilder.UsefulAdd(lb, ssql, exs) then
14:            lb ← LensBuilder.AddSeq(lb, ssql)
15:    return LensBuilder.ToDNFLens(pq)
```

the left and right DNF regular expressions. (The function CartesianMap maps its argument across the cross product of the input lists). A priority queue containing these sequence lenses, ordered by likelihood, is then initialized with PQ.Create. The symmetric lens is then built up iteratively from these sequence lenses, where the state of the partially constructed lens is tracked in the lens builder, *lb*.

GreedySynth loops until there are no more sequence lenses in the priority queue. Within this loop, a sequence lens is popped from the queue and, if it is "useful," is included in the final DNF lens. The lens is considered to be useful when its source (or target) is *not* already the source (or target) of an included sequence lens. If examples require that two sequences have a lens between them, such lenses are considered useful. The priorities of the sequence lenses update as the algorithm proceeds: if two sequence lenses have the same source, the second one to be popped gets a higher cost than it originally had; information must now be stored for including that source of non-bijectivity.

As an example, consider searching for a lens between "" | name.name* and "" | name. GreedySynth might first pop the sequence lens between the sequences

"" and "", because it is a bijective sequence lens between. As neither "" is involved in a sequence lens, this lens is considered useful. Next, the sequence lens between name.name* and name would be popped: while that lens is not bijective it is still better than the alternatives. As all sequences are now involved in sequence lenses, and there are no examples to make other lenses useful, no more sequence lenses would be added to the lens builder.

Finally, after all sequences have been popped, the partial DNF lens *lb* is converted into a symmetric DNF lens. This is only possible if all sequences are involved in some sequence lens: if they are not, LensBuilder.ToDNFLens instead returns *None*.

Symmetric Sequence lenses Algorithm 7 presents GreedySeqSynth, which synthesizes symmetric sequence lenses using an algorithm whose structure is similar to GreedySynth's. It calls AtomSynth, which synthesizes atom lenses by iterating a DNF lens between its subcomponents.

The inputs to GreedySeqSynth are a suite of input-output examples and a pair of lists of stochastic atoms. As in GreedySynth, GreedySeqSynth returns *None*early if there is no possible lens. Afterward, GreedySeqSynth finds the best lenses between each atom pair of the left and right sequences, and organizes them into a priority queue ordered by likelihood with PQ.Create. The symmetric sequence lens is built up iteratively from these atom lenses, where the state of the partially built lens is tracked in the sequence lens builder, *slb*.

GreedySeqSynth loops until there are no more atom lenses in the priority queue. In the loop, a popped atom lens is considered "useful" if adding it to the sequence will lower the cost of the generated sequence lens, or if examples show that one of its atoms must not be disconnected. Each atom can be part of only one lens at a time, so the algorithm must sometimes remove a previously chosen atom lens in order to connect one that must not be disconnected. The algorithm succeeds when all atoms that must

Algorithm 7 GreedySeqSynth

```
1: function AtomSynth(exs, DS*p, DT*q)
2:     if NoMap(exs, DS*p, DT*q) then
3:         return None
4:     else
5:         return GreedySynth(exs, DS, DT)*
6: function GreedySeqSynth(exs, [s0 · A1 · ... · An · sn], [t0 · B1 · ... · Bm · sm])
7:     if NoMap(exs, [s0 · A1 · ... · An · sn], [t0 · B1 · ... · Bm · sm]) then
8:         return None
9:     als ← CartesianMap(GreedyAtomSynth(exs), [A1; ...; An], [B1; ...; Bm])
10:    pq ← PQ.Create(als)
11:    slb ← SLensBuilder.Empty
12:    while PQ.IsNonempty(pq) do
13:        sal ← PQ.Pop(pq)
14:        if SLensBuilder.UsefulAdd(slb, sal, exs) then
15:            pq ← SLensBuilder.AddAtom(pq, sal)
16:    return SLensBuilder.ToDNFLens(pq)
```

not be disconnected are involved in an atom lens; SLensBuilder.ToDNFLens returns *None* otherwise.

5.7.6 Optimizations

Our implementation includes a number of optimizations not described above: annotations that guide DNF conversion; an expansion inference algorithm; and compositional synthesis. The optimizations make the system performant enough for interactive use.

Open and Closed Regular Expressions While GreedySynth acts relatively efficiently, it can suffer from an exponential blowup when converting SREs to DNF form. This problem can be mitigated by avoiding the conversion of some SREs to DNF form and by performing the conversion lazily when necessarily. More specifically, Expand labels some unconverted SREs as "closed," which means the type-directed GreedySynth algorithm treats them as DNF atoms and does not dig into them recursively. In other words, given a pair of closed SREs, GreedySynth can either construct the identity lens between them (it will do this if they are the same SRE), or

it can construct a disconnect lens between them. Regular expressions that are not annotated as closed are considered "open."

Distinguishing between open and closed regular expressions improves the efficiency of GreedySynth, but forces Expand to decide which closed expressions to open. At the start of synthesis, all regular expressions are closed, and Expand rewrites selected closed regular expressions to open ones (thereby triggering DNF normalization).

Open an closed regular expressions are a generalization of user-defined regular expressions introduced in Chapter 3. By using open and closed regular expressions, where the user uses variables no longer has an impact on performance.

Expansion Inference These additional rewrites make the search through possible regular expressions harder. Our algorithm identifies when certain closed regular expressions can *only* be involved in a disconnect lens (unless opened). Such regular expressions will automatically be opened. The full details of expansion inference are explained in Chapter 3.

Compositional Synthesis We port the compositional synthesis algorithm from Chapter 3 to the context of symmetric lenses.

5.8 Evaluation

We implemented simple symmetric lenses as an extension to Boomerang [6]. In doing so, we reimplemented Boomerang's asymmetric lens combinators using a combination of the symmetric combinators presented in this chapter and symmetric versions of asymmetric extensions (like matching lenses [5] and quotient lenses [18]) already present in Boomerang. We also integrated our synthesis engine into Boomerang, allowing users to write synthesis tasks alongside lens combinators, incorporate synthesis results into manually-written lenses, and reference previously defined lenses during synthesis.

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

In this evaluation, we aim to answer four primary questions:

1. Can the algorithm (with suitable examples and annotations) find the correct lens?

2. Is the synthesis procedure efficient enough to be used in everyday development?

3. How much slower is our tool on bijective lens synthesis benchmarks than prior work customized for bijective lenses [42]?

4. How effective is the information-theoretic search heuristic and how do our annotations affect the results?

5.8.1 Benchmark Suite

Our benchmarks are drawn from three different sources.

1. We use 8 data cleaning benchmarks from Flash Fill [20] (3 of which were present in Chapter 3, but had their alterations removed). None of these benchmarks were bijective.

2. We use the 29 benchmarks from Augeas [35].

3. We created 11 additional benchmarks derived from real-world examples and/or the bidirectional programming literature, 7 of which were present in Chapter 3. These tasks range from synchronizing REST and JSON web resource descriptions to synchronizing BibTeX and EndNote citation descriptions. Five of these benchmarks were not bijective lenses.

5.8.2 Synthesizing Correct Lenses

To determine whether the system can synthesize desired lenses, we ran it interactively on all 48 tasks, working with the system to create sufficient examples and provide useful relevance annotations. In all cases, the desired lens was obtained. The majority of the tasks required only a single example and none required more than three examples to synthesize the desired lens.[1]

Providing relevance annotations was needed in only 8 of the 48 tasks. In practice, we found that adding such annotations quite easy: if manual inspection of the lens showed there were too few ids, and too many disconnects or merges, we would add require annotations. If synthesis took too long, we would add skip annotations. Section 5.8.5 studies the effects of removing such annotations.

We verified that our default running mode (**SS**) generated the correct lenses the way programmers often validate their programs: we manually inspected the code and ran unit tests on the synthesized code. To determine whether the synthesis procedure generated the correct lens when running in modes other than **SS**, we compared generated lens to the lens synthesized by **SS**.

5.8.3 Effectiveness of Compositional Synthesis

Having determined appropriate examples and annotations for the 48 benchmarks, we evaluate the performance of the system by measuring the running time of our algorithm in two modes:

SS: Run the symmetric synthesis algorithm with all optimizations enabled.

SSNC: Run the symmetric synthesis algorithm, with no compositional synthesis enabled.

Recall that compositional synthesis allows users to break a benchmark into a series of smaller synthesis tasks, whose solutions are utilized in more complex synthesis

[1] In one benchmark, we supplied a fourth example that was later discovered to be unnecessary.

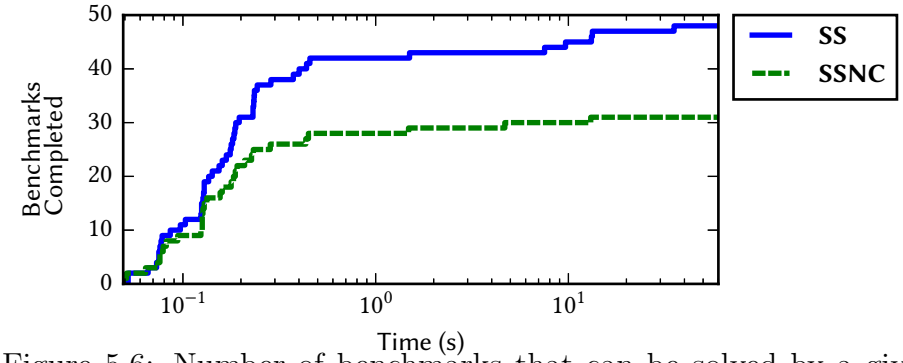


Figure 5.6: Number of benchmarks that can be solved by a given algorithm in a given amount of time. **SS** is the full symmetric synthesis algorithm. **SSNC** is the symmetric synthesis algorithm without using a library of existing lenses. The symmetric synthesis algorithm is able to complete all benchmarks in under 30 seconds elapsed total time. Without compositional synthesis it is able to complete 31. Each benchmark specification includes source and target (potentially annotated) regular expressions, and between one and three sufficient examples.

procedures. Compositional synthesis (**SS** mode) allows our system to scale to arbitrarily large and complex formats; measuring it shows the responsiveness of the system when used as intended. **SSNC** mode, which synthesizes a complete lens all at once, provides a useful experimental stress test for the system.

For each benchmark in the suite and each mode, we ran the system with a timeout of 60 seconds, averaging the result over 5 runs. Figure 5.6 summarizes the results of these tests. We find that our algorithm is able to synthesize all of the benchmarks in under 30 seconds. Without compositional synthesis, the synthesis algorithm is able to solve 31 out of 48 problem instances. In total, 73 existing lenses were used in compositional synthesis (about 1.5 per benchmark on average).

5.8.4 Slowdown Compared to Bijective Synthesis

To compare to the existing bijective synthesis algorithm, we run our symmetric synthesis algorithm on the original Optician benchmarks, comprised of 39 bijective synthesis tasks.[2]

To perform this comparison, we synthesized lenses in two modes:

BS: The existing bijective synthesis algorithm with all optimizations enabled.

SS: The symmetric synthesis algorithm with all optimizations enabled.

For each benchmark, we ran it in both modes with a timeout of 60 seconds and averaged the result over 5 runs. Figure 5.7 summarizes the results of these tests. On average, **SS** took 1.3 times (0.5 seconds) longer to complete than **BS**. The slowest completed benchmark for both synthesis algorithms is `xml_to_augeas.boom`, a benchmark that converts arbitrary XML up to depth 3 into a serialized version of the structured dictionary representation used in Augeas. This benchmark takes 18.9 seconds for the symmetric synthesis algorithm to complete, and 9.3 seconds the bijective synthesis algorithm to complete.

Both the bijective synthesis and the symmetric synthesis engines use a pair of collaborating synthesizers that (1) search for a compatible pair of regular expressions and (2) search for a lens given those regular expressions. Bijective synthesis is faster than symmetric synthesis because part (2) is much faster. Specifically, a bijection must translate all data on the left into data on the right, and this fact constrains the search. By contrast, a symmetric synthesis problem has a choice of which data on the left to translate into data on the right. This choice gives rise to additional choices, and symmetric synthesis must consider all of them. However, with memoization, this slowdown is not too substantial, and all our benchmarks still terminate in under 30 seconds.

[2]We had to slightly alter four of these benchmarks, either by providing additional examples or by adding in require annotations. Without these alterations, symmetric synthesis yielded a lens that fit the specification but that was undesired.

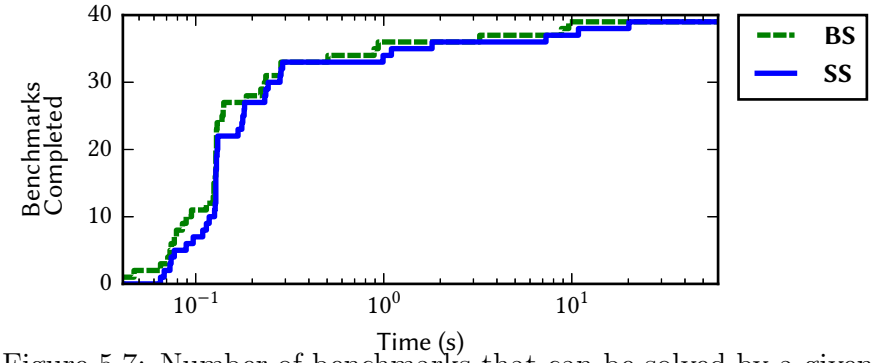


Figure 5.7: Number of benchmarks that can be solved by a given algorithm in a given amount of time. **SS** is the full symmetric synthesis algorithm. **BS** is the full bijective lens synthesis algorithm.

5.8.5 The Effects of Heuristics and Relevance Annotations

We evaluate the usefulness of (1) our information-theoretic metric, (2) our termination heuristic and (3) our relevance annotations. To this end, we run our program in several different modes:

Any: Ignore the information-theoretic preference metric (*i.e.*, all valid lenses have cost 0).

FL: Return the first highest ranked lens GreedySynth returns (*i.e.*, ignore the termination heuristic).

DC: Replace our information-theoretic cost metric with one where the cost of the lens is the number of disconnects plus the number of merges.

NS: Ignore all skip annotations in the SRE specifications.

NR: Ignore all require annotations in the SRE specifications.

We experimented with the **DC** mode to determine whether the complexity of the information-theoretic measure is really needed. Related work on string transformations has often used simpler measures such as "avoid constants" that align with, but are simpler than our measures [20, 32, 53]. The **DC** mode is an example of such a simple measure—it operates by counting disconnects, which put a complete stop to information transfer, and merges, which eliminate the information in a union.

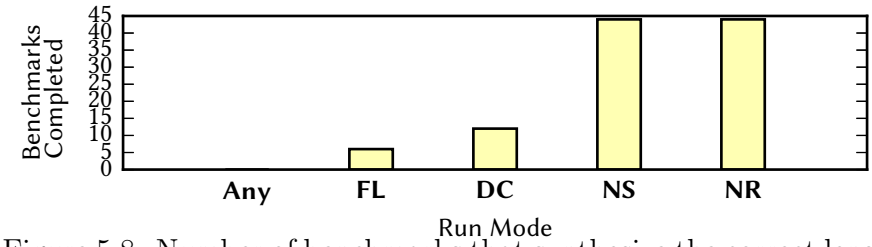


Figure 5.8: Number of benchmarks that synthesize the correct lens by a given algorithm. **Any** provides no notion of cost, and merely returns the first lens it finds that satisfies the specification. **FL** provides a notion of cost to GreedySynth, but once a satisfying lens is greedily found, that lens is returned. **DC** synthesizes lenses, where the cost of a lens is the number of disconnects plus the number of merges. **NS** ignores all skip annotations while running the algorithm. **NR** ignores all require annotations while running the algorithm.

Figure 5.8 summarizes the result of these experiments. The data reveal that the information-theoretic metric is critical for finding the correct lens: Only 10 of the benchmarks succeeded when running in **DC** mode. The termination condition is also quite important. When running in **FL** mode, the algorithm only discovers 5 lenses, which shows that the first class that contains a satisfying lens is rarely the correct class. However, our algorithm is not perfect and fails when either it is very difficult to find the desired lens (necessitating require) or when a large amount of data is projected (necessitating skip). Without any annotations, our algorithm finds the correct lens for 40 of our 48 benchmarks; eight required relevance annotations to find the correct lens. In total, there are 12 uses of require, and 4 uses of skip in our benchmark suite.

Chapter 6

Related Work

In this thesis, we have designed and implemented lens synthesis techniques for bijective, quotient, and symmetric lenses. We compare this line of work to existing lens formulations, and work towards making lens programming easier (§6.1), and we compare to previous work in synthesis (§6.2).

6.1 Lenses

6.1.1 QRE Lenses vs Quotient Lenses

Our work in synthesizing quotient lenses builds on the work of Foster et al [18] who introduced the theory of quotient lenses and implemented quotient lenses as a refinement of the bidirectional string processing language Boomerang [6]. As we mentioned in Section 4.4.4, 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 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 $B/\equiv_B$ to C to be a pair of functions $q.\mathsf{canonize} : B \longrightarrow C$ and $q.\mathsf{choose} : C \longrightarrow B$ such that for every $b \in B$:

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

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 defined as follows,

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

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

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

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

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 (asymmetric) 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. Lens synthesis can help define these user defined functions.

6.1.2 Simple Symmetric Lenses vs Symmetric Lenses

The concept of symmetric lenses was originally introduced in Hofman et al. [25]. However, general symmetric lenses cause issues with synthesis, so we introduced *simple symmetric lenses*.

In this section, we analyze the relationship between our simple symmetric lenses and classical symmetric lenses. Proofs about this relationships are included in the full version of the symmetric lens paper [44].

A classical symmetric lens[25] ℓ between X and Y consists of 4 components: a complement C, a designated element $init \in C$, and two functions, $putr : X \times C \to Y \times C$ and $putl : Y \times C \to X \times C$, that propagate changes in one format to the other.

In this formulation, data unique to each side are stored in the complement. When one format is edited, the putR or putL function stitches together the edited data with data stored in the complement. The *init* element is the initial value of C and specifies default behavior when data is missing. For instance, to implement the scenario in Figure 5.2, the complement would consist of a list of pairs of salary and company name. Classical symmetric lenses satisfy the following equational laws.

$$\frac{putr\,(x, c) = (y, c')}{putl\,(y, c') = (x, c')} \qquad \frac{putl\,(y, c) = (x, c')}{putr\,(x, c') = (y, c')}$$

Two classical symmetric lenses are equivalent if they output the same formats given any sequence of edits. Formally, given a lens ℓ between X and Y, an *edit* for ℓ is a member of $X + Y$. Consider the function *apply*, which, given a lens and an element of that lens's complement, is a function from sequences of edits to sequences of edits. If $apply(\ell, c, es) = es'$, then given complement c and edit es_i, the lens ℓ generates es'_i.

$$\frac{}{apply(\ell, c, []) = []} \qquad \frac{\ell.putr(x, c) = (y, c') \qquad apply(\ell, c', es) = es'}{apply(\ell, c, (inl\,x) :: es) = (inr\,y) :: es'}$$

$$\frac{\ell.putl(y, c) = (x, c') \qquad apply(\ell, c', es) = es'}{apply(\ell, c, (inr\,y) :: es) = (inl\,x) :: es'}$$

Two lenses, ℓ_1 and ℓ_2, are equivalent if $apply(\ell_1, \ell_1.init, es) = apply(\ell_2, \ell_2.init, es)$ for all es.

To compare classical and simple symmetric lenses, we define an *apply* function on simple symmetric lenses as well. If $apply(\ell, None, es) = es'$, then starting with no prior data, after edit es_i, the lens ℓ generates es'_i (the right format if $es_i = inl\, x$, and the left format if $es_i = inr\, y$). If $apply(\ell, Some\, (x, y), es) = es'$, then starting with data x and y on the left and right, respectively, after edit es_i, the lens ℓ generates es'_i.

$$\frac{}{apply(\ell, xyo, []) = []} \qquad \frac{\ell.\mathsf{createR}\, x = y \qquad apply(\ell, Some\, (x, y), es) = es'}{apply(l, None, inl\, x :: es) = inr\, y :: es'}$$

$$\frac{\ell.\mathsf{createL}\, y = x \qquad apply(\ell, Some\, (x, y), es) = es'}{apply(l, None, inr\, y :: es) = inl\, x :: es'}$$

$$\frac{\ell.\mathsf{putR}\, x'\, y = y' \qquad apply(\ell, Some\, (x', y'), es) = es'}{apply(l, Some\, (x, y), inl\, x' :: es) = inr\, y' :: es'}$$

$$\frac{\ell.\mathsf{putL}\, y'\, x = x' \qquad apply(\ell, Some\, (x', y'), es) = es'}{apply(l, Some\, (x, y), inr\, y' :: es) = inl\, x' :: es'}$$

Next, we define *forgetful symmetric lenses* to be symmetric lenses that satisfy the following additional laws.

$$\frac{\begin{array}{ll}\ell.putr(x, c_1) = (\_, c'_1) & \ell.putl(y, c'_1) = (\_, c''_1) \\ \ell.putr(x, c_2) = (\_, c'_2) & \ell.putl(y, c'_2) = (\_, c''_2)\end{array}}{c''_1 = c''_2} \qquad (\textsc{ForgetfulRL})$$

$$\frac{\begin{array}{ll}\ell.putl(y, c_1) = (\_, c'_1) & \ell.putr(x, c'_1) = (\_, c''_1) \\ \ell.putl(y, c_2) = (\_, c'_2) & \ell.putr(x, c'_2) = (\_, c''_2)\end{array}}{c''_1 = c''_2} \qquad (\textsc{ForgetfulLR})$$

Intuitively, these equations state that complements are uniquely determined by the most recent input x and y. Such lenses correspond exactly with simple symmetric lenses, where all state is maintained by the x and y data.

Theorem 14. Let ℓ be a classical symmetric lens. The lens ℓ is equivalent to a forgetful lens if, and only if, there exists a simple symmetric lens ℓ' where $apply(\ell, \ell.init, es) = apply(\ell', None, es)$, for all put sequences es.

The proof of this theorem is present in the appendix of the original "Synthesizing Symmetric Lenses" paper [44].

Simple symmetric lenses are a strict subset of classical symmetric lenses, but quite a useful one. For instance, all asymmetric lenses are expressible as simple symmetric lenses. The primary loss is the loss of "memory" within the complement. In classical symmetric lenses, disjunctive (or) lenses retain information about both possible formats. If a user edits a format from one disjuncted format to the other, the information contained in that first disjunct is retained within the complement. Simple symmetric lenses have no such complement, so they mimic the forgetful disjunctive lens of classical symmetric lenses.

Though classical symmetric lenses are more expressive, they introduce drawbacks for synthesis: because each lens has a custom complement, one can no longer specify the put functions through input/output examples alone. One alternative would be to enrich specifications with edit sequences; another would be to specify the structure of complements explicitly (though the latter would be somewhat akin to specifying the internal state of a program). In either case, the complexity of the specifications increases.

6.1.3 Simple Symmetric Lenses vs Symmetric Lenses

Formally, an *asymmetric lens* $\ell : S \Leftrightarrow V$ is a triple of functions $\ell.get : S \longrightarrow V$, $\ell.put : V \longrightarrow S \longrightarrow S$ and $\ell.create : V \longrightarrow S$ satisfying the following laws [17]:

$$\ell.get\ (\ell.put\ s\ v) = v \qquad \text{(PUTGET)}$$

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

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

Simple symmetric lenses are strictly more expressive than classical asymmetric lenses.

Theorem 15. Let ℓ be an asymmetric lens. ℓ is also a simple symmetric lens, where:

$$\ell.\mathsf{createL}\ y = \ell.create\ y \qquad \ell.\mathsf{createR}\ x = \ell.get\ x$$

$$\ell.\mathsf{putL}\ y\ x = \ell.put\ y\ x \qquad \ell.\mathsf{putR}\ x\ y = \ell.get\ x$$

The proof of this theorem is present in the appendix of the original "Synthesizing Symmetric Lenses" paper [44].

6.1.4 Other Lens Formulations

The literature on bidirectional programming languages and on lens-like structures is extensive. Some lens-like languages and tools include GRoundTram [24], BiYacc [63], Brul [62], BiGUL [29], bidirectional variants of relational algebra [7], spreadsheet formulas [37], graph query languages [23], and XML transformation languages [34].

Some tools use quotient lens-like structures, like XSugar [9], biXid [28], FliPpr [39], BiFluX [49], and X/Inv [27, 46, 47]. We refer the reader to the related work section in Foster et al. [18] for an extended comparison of these works to quotient lenses.

For further detail on the bidirectional programming literature, readers can consult a (slightly dated) survey [12] and more recent theoretical perspectives [1, 16].

6.2 Data Transformation Synthesis

6.2.1 Invertible Function Synthesis

Recently, symbolic transducers have been used to infer program inverses [26], providing invertibility guarantees for functions expressible in extended symbolic finite transducers. This task differs from the one we tackle in that a programmer must supply a program that performs a transformation in one direction and they get back a program that performs the transformation in the inverse direction, whereas we specify data formats, and obtain programs in both directions at once. Furthermore, this tool does not work on many of the programs we are interested in: swap cannot be expressed in full generality using these transducers. As extended symbolic finite transducers have only finite lookahead, they cannot rearrange data of arbitrary length, making them unable to express lenses like the name-swapping lens shown in Chapter 1.

6.2.2 String Transformation Synthesis

While we do not know of any previous efforts to synthesize both directions of bidirectional transformations, there is a good deal of other recent research on synthesizing unidirectional string transformations [20, 32, 50, 58, 59]. We compared our system to two of these unidirectional string transformers, Flash Fill [20] and FlashExtract [32]. We found that these tools were unsuccessful in synthesizing the complex transformations we are performing – both these tools synthesized under 5 of our 39 examples. Furthermore, neither of these tools were able to infer transformations that occurred under two or more iterations – for efficiency reasons Flash Fill does not synthesize programs with nested loops [20]. Much of this work assumes, like us, that the synthesis engine is provided with a collection of examples. Our work differs in that we assume the programmer supplies both examples *and* format descriptions in the form of regular expressions. There is a trade-off here. On the one hand, a user must have some

programming expertise to write regular expression 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, as shown in §3.7. 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.

6.2.3 DSL and Type-Directed Synthesis

Over the past decade, the programming languages community has explored the synthesis of programs from a wide variety of angles. One of the key ideas is typically to narrow the program search space by focusing on a specific domain, and to impose constraints on syntax [2], typing [3, 15, 19, 21, 48, 56], or both.

There are many other recent results showing how to synthesize functions from type-based specifications [3, 15, 19, 48, 51, 56]. These systems enumerate programs of their target language, orienting their search procedures to process only terms that are well-typed. Our system is distinctive in that it synthesizes terms in a language with many type equivalences. Perhaps the most similar is InSynth [21], 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" [21]. Our type structure is significantly more complex. In particular, because our types do not have full canonical forms, we use a pseudo-canonical form, which captures part of the equivalence relation over types. To preserve completeness, we push some of the remaining parts of the type equivalence relation into a set of rewriting rules and other parts into the RigidSynth algorithm itself.

Morpheus [14] 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, which require some enumerative search. Our systems differ in that an outline for Morpheus is a sketch—an *expression* containing holes—whereas an outline for Optician is a pair of DNF regular expressions, i.e., a *type*. Moreover, in order to implement an efficient search procedure, we had to create both a new type language and a new term language for lenses. Once we did so, we proved our new, more constrained language designed for synthesis was just as expressive as the original, more flexible and compositional language designed for human programmers.

Many synthesis algorithms work on domain-specific languages custom built for synthesis [20, 32, 60, 61]. We too built a custom domain-specific language for synthesis – DNF lenses. We provide the capabilities to convert specifications in an existing language, Boomerang, to specifications as DNF regular expressions, and provide the capabilities to convert our generated DNF lenses to Boomerang lenses. But we go further than merely providing a converter to Boomerang, we also provide completeness results stating exactly which terms of Boomerang we are able to synthesize.

Chapter 7

Conclusions

In this work, we showed how to synthesize bijective, quotient, and symmetric lenses from format specifications. We focus on string lenses, and synthesize Boomerang expressions from lens types – pairs of regular expressions. Our core algorithm works via two communicating synthesizers, one that generates a series of semantically equivalent lens types from the initial specification, and another that searches for transformations between the proposed lens types. We evaluated our algorithm on a number of benchmarks, 39 for Bijective and Quotient lenses and 48 for Symmetric lenses, and found that we were able to synthesize all of them in under 5 minutes.

While we focused on string lenses, I believe our core algorithm can be repurposed to lenses in other domains (like lenses over algebraic data types, relational algebras, and regular trees) and other domains where the type systems have rich equivalence classes. If the type system for the language can be partitioned into type-directed rules, and a rule for type-equivalence, our high-level approach should work. Users should be able to have two synthesizers – one that traverses equivalences, and one that searches through syntax-directed rules. Proving this process complete may be difficult, as it entails showing the lenses have a normal form where all equivalences are traversed "at once." In the string lens domain, we did this with the DNF lens

typing derivation, which first processes all rewrites, then processes syntactic rules. While other synthesizers would not necessarily need to synthesize in a DNF form, they would require such a two-sorted typing derivation.

One of my favorite parts of this work is how we use richer specifications to synthesize transformations that were previously intractable. Previously, all string transformation synthesis engines only used input-output examples as specifications. Synthesis systems that only use examples have a very difficult task, they must recognize characteristics of the data formats – essentially learning the data formats, and then learn how to transform inputs from one learned data format to the other. Because they must learn the data formats, these tools have particular difficulty in synthesizing transformations with nested iterations, and disjunctions under iterations, which are also particularly difficulty for data format inference systems.

By using types as inputs, our inference algorithm does not have to identify format information, the formats are already available! Optician does not have any difficulties with nested iterations; many of the benchmarks in our benchmark suite have deeply nested iterations. Optician can scale to these complex formats because it only has to focus on one thing – generating the lens.

The approach of enriching specifications to synthesize more complex transformations helps synthesize quotient lenses in Chapter 4. The key contributions of Chapter 4 lie in the design of QREs and our theorem demonstrating lenses with QREs at the edges are equivalent in expressivity to full quotient lenses. This chapter is unique, as it's fundamentally a language design chapter, but the language being designed is a only really used as specifications for a synthesizer. QREs in isolation don't really make a programmer's job easier, they are basically just more restrictive forms of quotient lens canonizers. But when combined with a bijective synthesis algorithm, QREs permit Optician to synthesize all quotient lenses with only some lightweight annotations.

Finally, while extending Optician to work on symmetric lenses, we quickly hit difficulties due to the disconnect lens. As a disconnect lens is not usually the lens a user wants but can have any type, we needed some extra form of specification. By including probability annotations in the form of SREs, disconnect lenses can be avoided. However, sometimes users *want* lenses with large amounts of information lost. By including additional specifications in the form of relevance annotations, users override Optician's default rankings. By letting users control the rankings, and having a default ranking that worked well in many cases, users can quickly find their desired lenses.

In addition to synthesizing multiple types of string lenses, Optician makes intellectual contributions that are more broadly applicable. Our general algorithm can be used for synthesizing lenses in other domains, and could even be useful for type-directed synthesis in non-lens domains where the types have rich equivalences. Our approaches showed the benefits of using complex user specifications. There is an inherent tension that we've exposed – by asking users for more complex specifications, the user's job is harder, but the synthesis task is made simpler. Prior work on program sketching [60] also explores these boundaries – program sketches are heavyweight specifications, but complex programs can be synthesized using them. In the future, I think currently intractable synthesis tasks, like synthesizing long programs in a general-purpose language, can become tractable by employing more complex types of human interaction.

Bibliography

[1] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Reflections on monadic lenses. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 1–31, 2016.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–17, October 2013.

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

[4] Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: Alignment and view update. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, September 2010.

[5] Davi M.J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: Alignment and view update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 193–204, New York, NY, USA, 2010. ACM.

[6] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08. ACM, 2008.

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

[8] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Trans. Comput.*, 20(2), February 1971.

[9] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Inf. Syst.*, 33(4-5), June 2008.

[10] Rafael C. Carrasco, Mikel L. Forcada, and Laureano Santamaría. Inferring stochastic regular grammars with recurrent neural networks. In Laurent Miclet and Colin de la Higuera, editors, *Grammatical Interference: Learning Syntax from Sentences*, pages 274–281, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[11] J. H. Conway. *Regular Algebra and Finite Machines*. Printed in GB by William Clowes & Sons Ltd, 1971.

[12] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In Richard F. Paige, editor, *ICMT*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer, 2009.

[13] Manfred Droste, Werner Kuich, and Heiko Vogler, editors. *Semirings and Formal Power Series*, pages 3–28. Springer Berlin Heidelberg, 2009.

[14] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 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, 2017.

[15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[16] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. The essence of bidirectional programming. *SCIENCE CHINA Information Sciences*, 58(5):1–21, 2015.

[17] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007.

[18] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. *SIGPLAN Not.*, 43(9):383–396, September 2008.

[19] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: A type-theoretic interpretation (extended version). Technical Report MS-CIS-15-12, University of Pennsylvania, 2015.

[20] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11. ACM, 2011.

[21] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[22] Brian Harry. A new api for visual studio online, 2014.

[23] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 205–216, 2010.

[24] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. Groundtram: An integrated framework for developing well-behaved bidirectional model transformations. In *Automated Software Engineering (ASE)*, 2011.

[25] Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas*, January 2011.

[26] Qinheping Hu and Loris D'Antoni. Automatic program inversion using symbolic transducers. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 376–389, New York, NY, USA, 2017. ACM.

[27] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. 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, pages 178–189, New York, NY, USA, 2004. ACM.

[28] Shinya Kawanaka and Haruo Hosoya. biXid: A bidirectional transformation language for XML. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06. ACM, 2006.

[29] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. BiGUL: A formally verified core language for putback-based bidirectional programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 61–72, 2016.

[30] D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2), 1994.

[31] Daniel Krob. Complete systems of b-rational identities. *Theor. Comput. Sci.*, 89(2), October 1991.

[32] Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14. ACM, 2014.

[33] Daniel Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4:59–76, 02 1977.

[34] Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. Bidirectional interpretation of XQuery. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*, pages 21–30, 2007.

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

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

[37] Nuno Macedo, Hugo Pacheco, Nuno Rocha Sousa, and Alcino Cunha. Bidirectional spreadsheet formulas. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*, pages 161–168, 2014.

[38] Solomon Maina, Anders Miltner, Kathleen Fisher, Benjamin Pierce, Dave Walker, and Steve Zdancewic. Synthesizing quotient lenses. 2018.

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

[40] Microsoft Corporation. *Requirements and compatibility — Team Foundation Server Setup, Update and Administration*, 2017.

[41] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses, 2017.

[42] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses. In *Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2018, 2018.

[43] Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Optician-tool. `https://github.com/Optician-Tool/Optician-Tool`, 2017.

[44] Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing symmetric lenses. *CoRR*, abs/1810.11527, 2018. Extended technical report.

[45] Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing symmetric lenses. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[46] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. *An Algebraic Approach to Bi-directional Updating*, pages 2–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[47] Shin-Cheng MU, Zhenjiang HU, and Masato TAKEICHI. Bidirectionalizing tree transformation languages: A case study. *Computer Software*, 23(2):129–141, 2006.

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

[49] Hugo Pacheco, Tao Zan, and Zhenjiang Hu. Biflux: A bidirectional functional update language for XML. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 147–158, 2014.

[50] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, 2014.

[51] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16. ACM, 2016.

[52] Microsoft PROSE. Microsoft Program Synthesis using Examples SDK, 2017.

[53] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 404–415. IEEE Press, 2017.

[54] Brian J. Ross. Probabilistic pattern matching and the evolution of stochastic regular expressions. *Applied Intelligence*, 13(3):285–300, November 2000.

[55] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1), January 1966.

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

[57] Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 7 1948.

[58] Rishabh Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proc. VLDB Endow.*, 9(10), June 2016.

[59] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *Proceedings of the VLDB Endowment*, 5(8), 2012.

[60] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

[61] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16. ACM, 2016.

[62] Tao Zan, Li Liu, Hsiang-Shang Ko, and Zhenjiang Hu. Brul: A putback-based bidirectional transformation library for updatable views. In *Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016.*, pages 77–89, 2016.

[63] Zirun Zhu, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. Biyacc: Roll your parser and reflective printer into one. In *Proceedings of the 4th International Workshop on Bidirectional Transformations co-located with Software Technologies: Applications and Foundations, STAF 2015, L'Aquila, Italy, July 24, 2015.*, pages 43–50, 2015.