Synthesizing Bijective Lenses

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

Bidirectional transformations between different data representations occur frequently in modern software systems. They appear as serializers and deserializers, as parsers and pretty printers, as database views and view updaters, and as a multitude of different kinds of ad hoc data converters. Manually building bidirectional transformations—by writing two separate functions that are intended to be inverses—is tedious and error prone. A better approach is to use a domain-specific language in which both directions can be written as a single expression. However, these domain-specific languages can be difficult to program in, requiring programmers to manage fiddly details while working in a complex type system.

We present an alternative approach. Instead of coding transformations manually, we synthesize them from declarative format descriptions and examples. Specifically, we present *Optician*, a tool for type-directed synthesis of bijective string transformers. The inputs to Optician are a pair of ordinary regular expressions representing two data formats and a few concrete examples for disambiguation. The output is a well-typed program in Boomerang (a bidirectional language based on the theory of *lenses*). The main technical challenge involves navigating the vast program search space efficiently. In particular, and unlike most prior work on type-directed synthesis, our system operates in the context of a language with a rich equivalence relation on types (the theory of regular expressions). Consequently, program synthesis requires search in two dimensions: First, our synthesis algorithm must find a pair of "syntactically compatible types," and second, using the structure of those types, it must find a type- and example-compliant term. Our key insight is that it is possible to reduce the size of this search space *without losing any computational power* by defining a new language of lenses designed specifically for synthesis. The new language is free from arbitrary function composition and operates only over types and terms in a new disjunctive normal form. We prove (1) our new language is just as powerful as a more natural, compositional, and declarative language and (2) our synthesis algorithm is sound and complete with respect to the new language. We also demonstrate empirically that our new language changes the synthesis problem from one that admits intractable solutions to one that admits highly efficient solutions, able to synthesize intricate lenses between complex file formats in seconds. We evaluate Optician on a benchmark suite of 39 examples that includes both microbenchmarks and realistic examples derived from other data management systems including Flash Fill, a tool for synthesizing string transformations in spreadsheets, and Augeas, a tool for bidirectional processing of Linux system configuration files.

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

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

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

2475-1421/2018/1-ART1
https://doi.org/10.1145/3158089

ACM Reference Format:
Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing Bijective Lenses. *Proc. ACM Program. Lang.* 2, POPL, Article 1 (January 2018), 30 pages. https://doi.org/10.1145/3158089

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* [Foster et al. 2007] 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 two functions, *get* and *put*. The *get* function translates source data into the target format. If the target data is updated, the *put* function translates this edited data back into the source format. A benefit of lens-based languages is that they use a single term to express both *get* and *put*. Furthermore, well-typed lenses give rise to *get* and *put* functions guaranteed to satisfy desirable invertibility properties.

Lens-based languages are present in variety of tools and have found mainstream industrial use. Boomerang [Barbosa et al. 2010; Bohannon et al. 2008] lenses provide guarantees on transformations between *ad hoc* string document formats. Augeas [Lutterkort 2007], a popular tool that reads Linux system configuration files, uses the *get* 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 *put* to merge the edited results back into the original string format. Other lens-based languages and tools include GRoundTram [Hidaka et al. 2011], BiFluX [Pacheco et al. 2014], BiYacc [Zhu et al. 2015], Brul [Zan et al. 2016], BiGUL [Ko et al. 2016], bidirectional variants of relational algebra [Bohannon et al. 2006], spreadsheet formulas [Macedo et al. 2014], graph query languages [Hidaka et al. 2010], and XML transformation languages [Liu et al. 2007].

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. In short, lens languages provide their strong bidirectional guarantees by putting additional burdens on the programmer.

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 [Feng et al. 2017; Gulwani 2011a; Le and Gulwani 2014; Perelman et al. 2014], many guided by types [Feng et al. 2017; Feser et al. 2015; Frankle et al. 2015; Osera and Zdancewic 2015; Polikarpova et al. 2016].

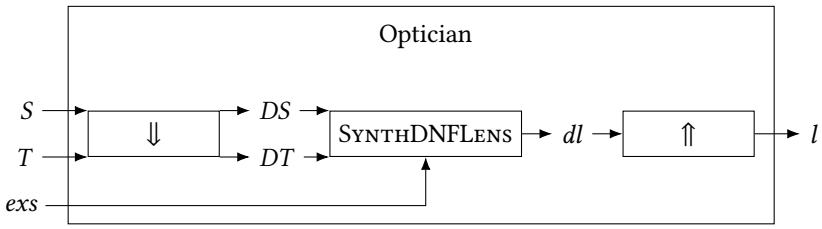


Fig. 1. Schematic Diagram for Optician. Regular expressions, S and T, and examples, *exs*, are given as input. First, the function ⇓ converts S and T into their respective DNF forms, DS and DT. Next, SynthDNFLens synthesizes a DNF lens, *dl*, from S, T, and *exs*. Finally, ⇑ converts *dl* into *l*, a lens in Boomerang that is equivalent to *dl*.

Specifically, Optician supports the synthesis of *bijective lenses*, a useful subset of Boomerang. While Boomerang can express more complex lens theories, like asymmetric lenses [Foster et al. 2007] and quotient lenses [Foster et al. 2008], we focus on synthesizing lenses encoding bijections. We theorize many of these more complex lenses have a bijective core, surrounded by structured transformations. For example, assymmetric lenses can provide mappings between formats where one format contains information the other does not. We expect that many asymmetric lenses are merely bijections composed with projections, where the projections remove information when operated left to right, and recover that information when operated from right to left. Quotient lenses can provide mappings between formats containing unnecessary information, like whitespace. We expect that many quotient lenses are merely bijections composed with functions that put the data into a canonical form. If the appropriate projections and canonizing functions are provided (either from the user or from another synthesis algorithm), Optician can be used as a drop-in component for synthesizing full terms in these more expressive theories.

As inputs, Optician takes regular expressions specifying the source and target formats, plus a collection of concrete examples of the desired transformation. Format specifications are supplied as ordinary regular expressions. 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 Boomerang's types 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 language is equivalent to a natural, declarative specification of the bijective fragment of Boomerang.

Figure 1 shows a high-level, schematic diagram for Optician. First, Optician uses the function ⇓ 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.

Contributions. Optician makes bidirectional programming more accessible by obviating the need for programmers to write lenses by hand. We begin by briefly reviewing some background on regular expressions and core lens combinators (§2). After we motivate our problem with an extended, real-world example (§3), we offer the following technical contributions:

- We introduce a new lens language (DNF Lenses) that is suitable for synthesis (§4 and §5). We show how to convert ordinary regular expressions and lenses into the corresponding DNF forms, and we prove that DNF lenses are sound and complete with respect to the high-level bijective lens syntax.
- We present an efficient, type-directed synthesis algorithm for synthesizing lenses (§6). We prove that if there is a lens that satisfies the input specification, this algorithm will return such a lens.
- We evaluate Optician, its optimizations, and existing synthesis tools on 39 benchmarks, including examples derived from Flash Fill [Gulwani 2011b] and the Augeas [Lutterkort 2007] system (§7). We show that our optimizations are critical for synthesizing many of the complex lenses in our benchmark suite and that our full algorithm succeeds on all benchmarks in under 5 seconds.
- While we are not aware of any other systems for automatically synthesizing bijective transformations, we establish a baseline for our the effectiveness of our techniques by comparing our synthesis algorithm with the one used in Flash Fill [Gulwani 2011b], a well-known and influential synthesis system deployed in Microsoft Excel. Flash Fill only synthesizes transformations in one direction, but it was only able to complete synthesis of 3 out of 39 of our benchmarks. We conjecture that the extra information we supply the synthesis system via our regular format descriptions, allows it to scale to significantly more complex and varied formats than is possible in current string synthesis systems that do not use this information.

We close with related work (§8) and conclusions (§9).

2 PRELIMINARIES

Technical Report. Throughout the paper, we will state a number of theorems. We have omitted these theorems for space, and have included these details in a longer technical report [Miltner et al. 2017a].

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:

$$S, T \quad ::= \quad s \quad | \quad \emptyset \quad | \quad S^* \quad | \quad S_1 \cdot S_2 \quad | \quad S_1 \mid S_2$$

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

Unambiguity. The typing derivations of lenses require regular expressions to be written in a way that parses text unambiguously. S and T are *unambiguously concatenable*, written $S \cdot^! T$ if, for all strings $s_1, s_2 \in \mathcal{L}(S)$ and $t_1, t_2 \in \mathcal{L}(T)$, 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, S is *unambiguously iterable*, written $S^{*!}$ if, for all $n, m \in \mathbb{N}$ and for all strings $s_1, \ldots, s_n, t_1, \ldots, t_m \in \mathcal{L}(S)$, 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 S is *strongly unambiguous* if one of the following holds: (a) $S = s$, or (b) $\mathcal{L}(S) = \{\}$, or (c) $S = S_1 \cdot S_2$ with $S_1 \cdot^! S_2$, or (d) $S = S_1 \mid S_2$ with $S_1 \cap S_2 = \emptyset$, or (e) $S = (S')^*$ with $(S')^{*!}$. In the recursive cases, S_1, S_2, and S' must also be strongly unambiguous.

Equivalences. S and T are *equivalent*, written $S \equiv T$, if $\mathcal{L}(S) = \mathcal{L}(T)$. There exists an equational theory for determining whether two regular expressions are equivalent, presented by Conway [1971], and proven complete by Krob [1991].[1] Conway's axioms consist of the semiring axioms (associativity, commutativity, identities, and distributivity for $\mid$ and $\cdot$) plus the following rules for equivalences involving the Kleene star:

$$\begin{array}{rcll} (S \mid T)^* & \equiv & (S^* \cdot T)^* \cdot S^* & \textit{Sumstar} \\ (S \cdot T)^* & \equiv & \epsilon \mid (S \cdot (T \cdot S)^* \cdot T) & \textit{Prodstar} \\ (S^*)^* & \equiv & S^* & \textit{Starstar} \\ (S \mid T)^* & \equiv & ((S \mid T) \cdot T \mid (S \cdot T^*)^n \cdot S)^* \cdot (\epsilon \mid (S \mid T) \cdot ((S \cdot T^*)^0 \mid \ldots \mid (S \cdot T^*)^n)) & \textit{Dicyc} \end{array}$$

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* [Droste et al. 2009]. If two regular expressions are equivalent within this equational theory, they are *star semiring equivalent*, written $S \equiv^s T$. The star semiring axioms consist of the semiring axioms plus the following rules for equivalences involving the Kleene star:

$$\begin{array}{rcll} S^* & \equiv^s & \epsilon \mid (S \cdot S^*) & \textit{Unrollstar}_L \\ S^* & \equiv^s & \epsilon \mid (S^* \cdot S) & \textit{Unrollstar}_R \end{array}$$

In §3, we provide intuition for why synthesis with full regular expression equivalence is problematic and justify our choice of using star semiring equivalence instead.

Bijective Lenses. All bijections between languages are lenses. We define *bijective lenses* to be bijections created from the following Boomerang lens combinators, l.

$$\begin{array}{rl} l ::= & \textit{const}(s_1 \in \Sigma^*, s_2 \in \Sigma^*) \\ \mid & \textit{iterate}(l) \\ \mid & \textit{concat}(l_1, l_2) \\ \mid & \textit{swap}(l_1, l_2) \\ \mid & \textit{or}(l_1, l_2) \\ \mid & l_1 \ ; \ l_2 \\ \mid & \textit{id}_S \end{array}$$

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

$$\begin{array}{rcl} [\![\textit{const}(s_1, s_2)]\!] & = & \{(s_1, s_2)\} \\ [\![\textit{iterate}(l)]\!] & = & \{(s_1 \cdot \ldots \cdot s_n, t_1 \cdot \ldots \cdot t_n) \mid n \in \mathbb{N} \ \wedge \ \forall i \in [1, n], (s_i, t_i) \in [\![l]\!]\} \\ [\![\textit{concat}(l_1, l_2)]\!] & = & \{(s_1 \cdot s_2, t_1 \cdot t_2) \mid (s_1, t_1) \in [\![l_1]\!] \ \wedge \ (s_2, t_2) \in [\![l_2]\!]\} \\ [\![\textit{swap}(l_1, l_2)]\!] & = & \{(s_1 \cdot s_2, t_2 \cdot t_1) \mid (s_1, t_1) \in [\![l_1]\!] \ \wedge \ (s_2, t_2) \in [\![l_2]\!]\} \\ [\![\textit{or}(l_1, l_2)]\!] & = & \{(s, t) \mid (s, t) \in [\![l_1]\!] \vee (s, t) \in [\![l_2]\!]\} \\ [\![l_1 \ ; \ l_2]\!] & = & \{(s_1, s_3) \mid \exists s_2 (s_1, s_2) \in [\![l_1]\!] \ \wedge \ (s_2, s_3) \in [\![l_2]\!]\} \\ [\![\textit{id}_S]\!] & = & \{(s, s) \mid s \in \mathcal{L}(S)\} \end{array}$$

The simplest lens in the combinator language is the constant lens between strings s, and t, $\textit{const}(s, t)$. The lens $\textit{const}(s, t)$, when operated left-to-right, replaces the string s with t, and when operated right-to-left, replaces string t with s. The identity lens on a regular expression, $\textit{id}_S$, operates in both directions by applying the identity function to strings in $\mathcal{L}(S)$. The composition combinator,

[1] There are other complete axiomatizations for regular expression equivalence, such as those developed by Kozen [1994] and Salomaa [1966]. We focus on Conway's for the sake of specificity, but discuss alternative theories in §8.

$$\frac{s_1 \in \Sigma^* \qquad s_2 \in \Sigma^*}{const(s_1, s_2) : s_1 \Leftrightarrow s_2} \qquad \frac{l : S \Leftrightarrow T \qquad S^{*!} \qquad T^{*!}}{iterate(l) : S^* \Leftrightarrow T^*} \qquad \frac{\begin{array}{c} l_1 : S_1 \Leftrightarrow T_1 \\ l_2 : S_2 \Leftrightarrow T_2 \\ S_1 \cdot^! S_2 \qquad T_1 \cdot^! T_2 \end{array}}{concat(l_1, l_2) : S_1 S_2 \Leftrightarrow T_1 T_2}$$

$$\frac{\begin{array}{c} l_1 : S_1 \Leftrightarrow T_1 \\ l_2 : S_2 \Leftrightarrow T_2 \\ S_1 \cdot^! S_2 \qquad T_2 \cdot^! T_1 \end{array}}{swap(l_1, l_2) : S_1 S_2 \Leftrightarrow T_2 T_1} \qquad \frac{\begin{array}{c} l_1 : S_1 \Leftrightarrow T_1 \qquad l_2 : S_2 \Leftrightarrow T_2 \\ \mathcal{L}(S_1) \cap \mathcal{L}(S_2) = \emptyset \qquad \mathcal{L}(T_1) \cap \mathcal{L}(T_2) = \emptyset \end{array}}{or(l_1, l_2) : S_1 \mid S_2 \Leftrightarrow T_1 \mid T_2}$$

$$\frac{l_1 : S_1 \Leftrightarrow S_2 \qquad l_2 : S_2 \Leftrightarrow S_3}{l_1 \; ; \; l_2 : S_1 \Leftrightarrow S_3} \qquad \frac{S \text{ is strongly unambiguous}}{id_S : S \Leftrightarrow S}$$

$$\frac{l : S_1 \Leftrightarrow S_2 \qquad S_1 \equiv^s S_1' \qquad S_2 \equiv^s S_2'}{l : S_1' \Leftrightarrow S_2'}$$

Fig. 2. Lens Typing Rules

$l_1 \; ; \; l_2$, 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.

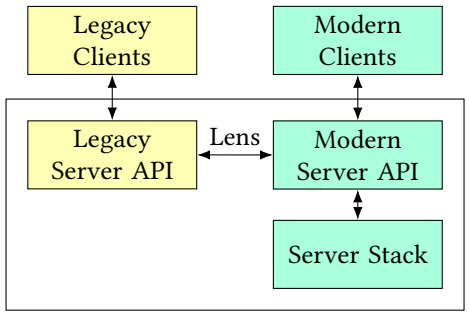
Each of the other lenses manipulates structured data. For instance, $concat(l_1, l_2)$ operates by applying l_1 to the left portion of a string, and l_2 to the right, and concatenating the results. The combinator $swap(l_1, l_2)$ does the same as $concat(l_1, l_2)$ but it swaps the results before concatenating. The combinator $or(l_1, l_2)$ operates by applying either l_1 or l_2 to the string. The combinator $iterate(l)$ operates by repeatedly applying l to subparts of a string. For example, the lens that transforms a last-name/first-name format (*e.g.* `"Kleene, Stephen Cole"`) to a first-name/last-name format (*e.g.* `"Stephen Cole Kleene"`) may be written as follows. [2]

```
swap
  (Id(name) . const(",","")
  ,(iterate (swap (Id(wsp) , Id_name))))
```

Unfortunately, even such a minimal example is hard to write! Writing this program required 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.

Lens Typing. The typing judgement for lenses has the form $l : S \Leftrightarrow T$, meaning l bijectively maps between $\mathcal{L}(S)$ and $\mathcal{L}(T)$. Figure 2 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 $l_1 : S_1 \Leftrightarrow T_1$, and $l_2 : S_2 \Leftrightarrow T_2$, and S_1 is not unambiguously concatenable with S_2, then there would exist $s_1, s_1' \in \mathcal{L}(S_1)$, and $s_2, s_2' \in \mathcal{L}(S_2)$ 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 $concat(l_1, l_2)$ 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'. Because

[2] For nested concatenations and disjunctions, the terms $concat(l_1, l_2)$ and $or(l_1, l_2)$ are syntactically heavy. Instead, these can be written in the infix style of $l_1.l_2$ (for concatenations) and $l_1|l_2$ (for disjunctions).



(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</Field>
</WorkItem>
	{
	Title: Return 400 on bad PUT request,
	Type: Bug,
	AssignedTo: J Smith,
	}

(b)

Fig. 3. 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).

any ambiguous RE can be replaced by an equivalent unambiguous one, these ambiguity constraints do not have an impact on the computational power of the language.

The typing rule for id_S requires a strongly unambiguous regular expression. This unambiguity allows the identity lens to be derivable from other lenses.[3] This requirement does not, however, reduce expressiveness, as any regular expression is equivalent to a strongly unambiguous regular expression [Book et al. 1971].

The last rule in Figure 2 is a type equivalence rule that lets the typing rules consider a lens $l : S_1 \Leftrightarrow S_2$ to have type $S_1' \Leftrightarrow S_2'$ so long as $S_1 \equiv^s S_1$ and $S_2 \equiv^s S_2'$. 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 equivalence in the next section.

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 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 [Harry 2014]. However, despite VSTS introducing new, modernized web APIs, they must still maintain the old, legacy web APIs for continued support of legacy clients [Microsoft Corporation 2017]. Instead of

[3]In practice, we allow regular expressions that aren't strongly unambiguous to appear in id_S, 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.

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 3a. 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 3b 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 *get* 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 *put* 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

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

while the modern client accepts inputs of the form

```
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` ⇔ `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

```
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 $S_1 \Leftrightarrow S_2$ also has type $S_1' \Leftrightarrow S_2'$. 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, $l_1 \; ; \; l_2$. 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 S_1 and S_3, a sound synthesizer must find an intermediate type S_2 and lenses with types $S_1 \Leftrightarrow S_2$ and $S_2 \Leftrightarrow S_3$. Searching for the correct regular expression S_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 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 4. 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 both highly nondeterministic and infinitely branching (due to the choice of n in the dyclicity axiom).

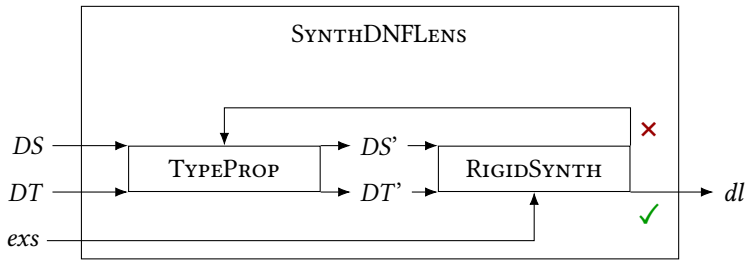


Fig. 4. Schematic Diagram for DNF Lens Synthesis Algorithm. DNF regular expressions, DS and DT, and a set of examples *exs* are given as input. The synthesizer, TypeProp, uses these input DNF regular expressions to propose a pair of equivalent DNF regular expressions, DS' and DT'. The synthesizer RigidSynth 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.

We also want DNF lenses to be closed under composition – if it is 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 $S \rightarrow S'$, then $\mathcal{L}(S) = \mathcal{L}(S')$. Second, if $S \rightarrow S'$ and S is strongly unambiguous, then S' 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 $dl \stackrel{\cdot}{:} DS \Leftrightarrow DT$ 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)**.** Whenever $dl_1 \stackrel{\cdot}{:} (\Downarrow S_1) \Leftrightarrow (\Downarrow T_1)$, if $S_1 \rightarrow S_2$ and $T_1 \rightarrow T_2$, there exist regular expressions S_3, and T_3 and a DNF lens dl_3, such that:

(1) $S_2 \rightarrow S_3$
(2) $T_2 \rightarrow T_3$
(3) $dl_3 \stackrel{\cdot}{:} (\Downarrow S_3) \Leftrightarrow (\Downarrow T_3)$
(4) $[\![dl_3]\!] = [\![dl_1]\!]$

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

Definition 2 (Bisimilarity)**.** Whenever $dl_1 \stackrel{\cdot}{:} (\Downarrow S_1) \Leftrightarrow (\Downarrow T_1)$ and $S_1 \rightarrow S_2$, there exist a regular expression T_2 and a DNF lens dl_2 such that

(1) $T_1 \rightarrow T_2$
(2) $dl_2 \stackrel{\cdot}{:} (\Downarrow S_2) \Leftrightarrow (\Downarrow T_2)$
(3) $[\![dl_2]\!] = [\![dl_1]\!]$

To be bisimilar, the symmetric property must also hold for $T_1 \rightarrow T_2$.

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 coarsest 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 is the

coarsest such relation possible, but it is sufficient to cover all the test cases in our benchmark suite (see §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.

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.

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

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 implementation, names of user-defined regular expressions are also atoms. However, we elide such definitions from our theoretical analysis.

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:

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

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

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

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

Fig. 5. DNF Regular Expression Functions

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

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 S is *unambiguously iterable*, written $S^{*!}$ if, for all $n, m \in \mathbb{N}$ and for all strings $s_1, \ldots, s_n, t_1, \ldots, t_m \in \mathcal{L}(S)$, 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 5. 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 $\Downarrow S$, is defined below.

$$\begin{aligned} \Downarrow s &= \langle [s] \rangle \\ \Downarrow \emptyset &= \langle \rangle \\ \Downarrow (S^*) &= \mathcal{D}((\Downarrow S)^*) \\ \Downarrow (S_1 \cdot S_2) &= \Downarrow S_1 \odot \Downarrow S_2 \\ \Downarrow (S_1 \mid S_2) &= \Downarrow S_1 \oplus \Downarrow S_2 \end{aligned}$$

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]⟩*·","]
  | [""] ⟩
```

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

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 6 defines a collection of rewrite rules on DNF regular expressions designed to search the space of equivalent DNF REs. This directed

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

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

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

Fig. 6. DNF Regular Expression Rewrite Rules

rewrite system helps limit 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.

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.

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

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 *or*ed sequence lenses. DNF lenses also contain a permutation 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:"` $\cdot$ `text_char` $\cdot$ `""` $\cdot \langle$[`text_char`]$\rangle^* \cdot$ `","`] 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

$$\frac{dl \mathrel{\tilde{:}} DS \Leftrightarrow DT \qquad DS^{*!} \qquad DT^{*!}}{iterate(dl) \mathrel{\tilde{:}} DS^{*} \Leftrightarrow DT^{*}}$$

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

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

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

Fig. 7. DNF Lens Typing

`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 *get* and *put* 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_char` and `⟨[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.

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

Type Checking. Figure 7 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 domain $(SQ_1 \ldots SQ_n)$ and the range $(TQ_{\sigma(1)} \ldots TQ_{\sigma(n)})$. Permutations are used in a similar way in the typing rule for sequence 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, S and T, such that $\Uparrow dl : S \Leftrightarrow T$ and $\Downarrow S = DS$ and $\Downarrow T = DT$ and $[\![\Uparrow dl]\!] = [\![dl]\!]$.

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

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.

6 ALGORITHM

Synthesis Overview. Algorithm 1 presents the synthesis procedure. SynthLens takes the source and target regular expressions S and T, and a list of examples *exs*, as input. First, SynthLens validates the unambiguity of the input regular expressions, S and T, and confirms that they parse the input/output examples, *exs*. Next, the algorithm converts S and T 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.

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 4), 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.

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

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
```

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.

ExpandRequired has three major components: ExpandRequired, FixProblemElts, and ExpandOnce, which we discuss in turn. ExpandRequired performs expansions 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 ⟨[`legacy_title`]⟩ and ⟨[`modern_title`]⟩. 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 {(`modern_title`, 0), (`text_char`, 0), (`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 ⟨[`legacy_title`]⟩ has been expanded to

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

and ⟨[`modern_title`]⟩ has been expanded to

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

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 Unrollstar$_L$, and one from an application of Atom Unrollstar$_R$.

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.

Algorithm 3 RigidSynth

```
1:  function RigidSynthAtom(A, B, exs)
2:      match (A, B) with
3:          | (U, V) →
4:              if U ≤^exs_Atom V ∧ V ≤^exs_Atom U then
5:                  return Some id_U
6:              else
7:                  return None
8:          | (DS*, DT*) →
9:              match RigidSynth(DS, DT, exs) with
10:                 | Some dl → return iterate(dl)
11:                 | None → return None
12:         | _ → return None

13: function RigidSynthSeq(SQ, TQ, exs)
14:     [s_0 · A_1 · ... · A_n · s_n] ← SQ
15:     [t_0 · B_1 · ... · B_m · t_m] ← TQ
16:     if n ≠ m then
17:         return None
18:     σ_1 ← sorting(≤^exs_Atom, [A_1 · ... · A_n])
19:     σ_2 ← sorting(≤^exs_Atom, [B_1 · ... · B_n])
20:     σ ← σ_1^{-1} ∘ σ_2
21:     ABs ← Zip([A_1 · ... · A_n], [B_σ(1) · ... · B_σ(n)])
22:     alos ← Map(RigidSynthAtom(exs), ABs)
23:     match AllSome(alos) with
24:         | Some [al_1 · ... · al_n] → return Some([(s_0, t_0) · al_1 · ... · al_n · (s_n, t_n)], σ^{-1})
25:         | None → return None

26: function RigidSynth(DS, DT, exs)
27:     ⟨SQ_1 | ... | SQ_n⟩ ← DS
28:     ⟨TQ_1 | ... | TQ_m⟩ ← DT
29:     if n ≠ m then
30:         return None
31:     σ_1 ← sorting(≤^exs_Seq, [SQ_1 | ... | SQ_n])
32:     σ_2 ← sorting(≤^exs_Seq, [TQ_1 | ... | TQ_m])
33:     σ ← σ_1^{-1} ∘ σ_2
34:     STQs ← Zip([SQ_1 | ... | SQ_n], [TQ_σ(1) | ... | TQ_σ(n)])
35:     sqlos ← Map(RigidSynthSeq(exs), STQs)
36:     match AllSome(sqlos) with
37:         | Some [sql_1 | ... | sql_n] → return Some(⟨sql_1 | ... | sql_n⟩, σ^{-1})
38:         | None → return None
```

Searching for aligning permutations requires care, as naïvely considering all permutations between two 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 $sql \mathrel{:} SQ \Leftrightarrow TQ$ if, and only if, $SQ \leq_{Seq} TQ$ and $TQ \leq_{Seq} SQ$. Through these orderings, aligning the components reduces to merely sorting and zipping lists. Furthermore, through composing the permutations required to sort the lists, the algorithm discovers the permutation used in the lens.

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

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

and

```
⟨ ["Title:"·text_char·""·⟨[text_char]⟩*·","]
| [""] ⟩
```

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

$$([(\texttt{"<FieldId=2></Field>"}, \texttt{""})], id)$$

Next, the sequences `["<Field Id=2>"·text_char·""·⟨[text_char]⟩*·"<Field Id=2>"]` and `["Title="·text_char··""·⟨[text_char]⟩*·","]` would be sent to RigidSynthSeq. In RigidSynthSeq, the atoms would not be reordered, aligning `text_char` with `text_char`, and `⟨[text_char]⟩*` with `⟨[text_char]⟩*`. Immediately, RigidSynthAtom finds the identity transformation on `text_char`, and will recurse to find the identity transformation for `⟨[text_char]⟩*`:

$$iterate((\langle([(\texttt{""}, \texttt{""}) \cdot id_{\texttt{text\_char}} \cdot (\texttt{""}, \texttt{""}), id])\rangle, id))$$

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 expressions, but also that the lens will satisfy the examples. For example if $SQ \leq^{exs}_{Seq} TQ$ and $TQ \leq^{exs}_{Seq} SQ$ (where $\leq^{exs}_{Seq}$ is the ordering incorporating example information) 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"`. The details of these orderings are formalized in the extended version of this paper [Miltner et al. 2017a].

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

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

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

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 *concat*s and *or*s, (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("","")
      . iterate(const("","") . Id(text_char) . const("",""))
      . const("","")
      . const("</Field>",", "))
```

where with minimization, the title field transformation is:

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

Compositional Synthesis. Most synthesis problems can be divided into subproblems. For example, if the format $S_1 \cdot S_2$ must be converted into $T_1 \cdot T_2$, one might first work on the $S_1 \Leftrightarrow T_1$ and $S_2 \Leftrightarrow T_2$ subproblems. After those subproblems have been solved, the lenses they generate can be combined into a solution for $S_1 \cdot S_2 \Leftrightarrow T_1 \cdot T_2$.

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.

7 EVALUATION

We have implemented 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 [Miltner et al. 2017b].

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 [Lutterkort 2007] and Flash Fill [Gulwani 2011b] 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 [Gulwani 2011b]. 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 8 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 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:

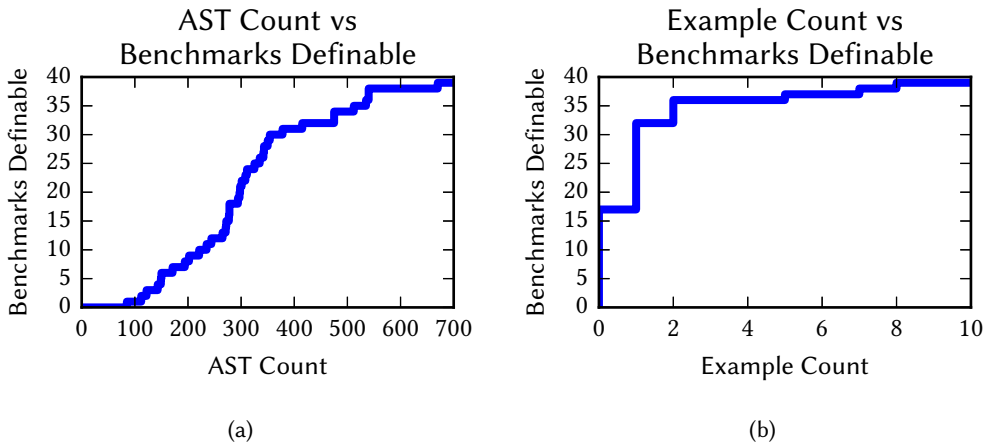


Fig. 8. Sizes of Specifications. In (a), we show how many benchmarks are defined in our suite using a given number of AST nodes or fewer. In (b), we show how many benchmarks are defined in our suite using a given number of examples or fewer.

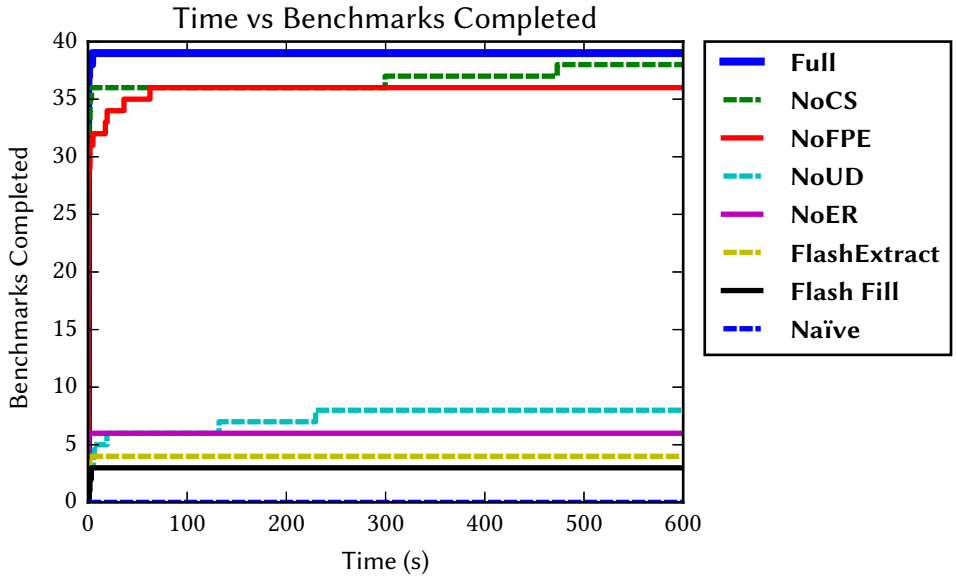


Fig. 9. Number of benchmarks that can be solved by a given algorithm in a given amount of time. **Full** is the full synthesis algorithm. **NoCS** is the synthesis algorithm using all optimizations but without using a library of existing lenses. **NoFPE** is the core DNF synthesis algorithm augmented 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.

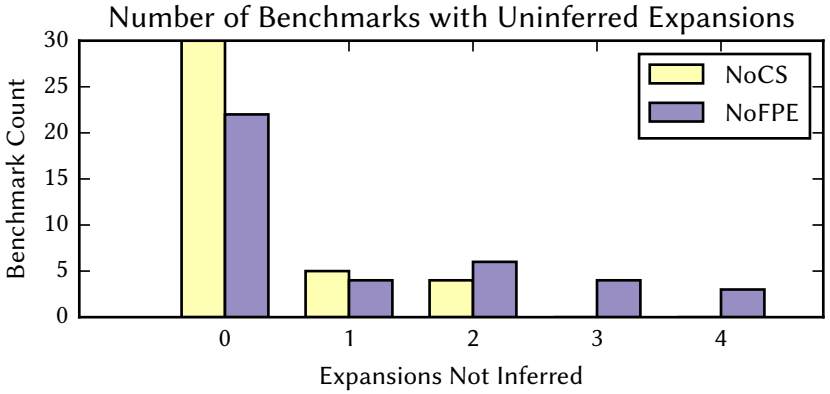


Fig. 10. Number of expansions found using enumerative search for tasks requiring a given number of expansions. **NoCS** is using the full inference algorithm. **NoFPE** only counts forced inferences as found by the ExpandRequired function. Both systems are able to infer the vast majority of expansions. Full inference only rarely requires enumerative search.

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.

We ran Optician in each mode over our benchmark suite. Figure 9 summarizes the results of these tests. **Full** synthesized all 39 benchmarks, **NoCS** synthesized 48 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 10 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 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

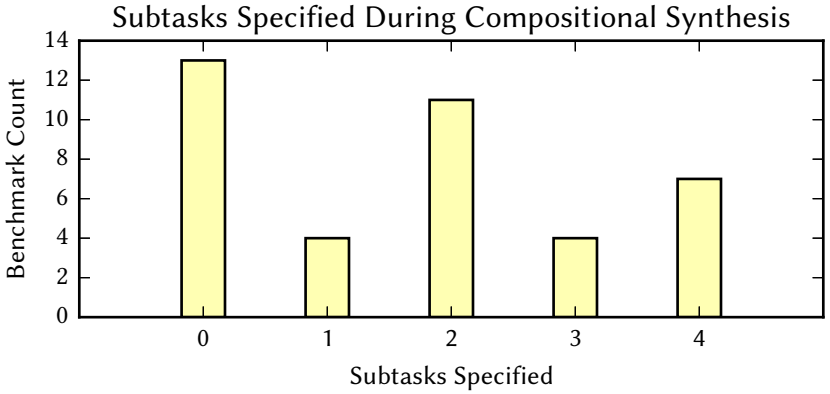


Fig. 11. Number of subtasks specified during compositional synthesis. Splitting the task into just a few subtasks provides huge performance benefits at the cost of a small amount of additional user work.

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 11. 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

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

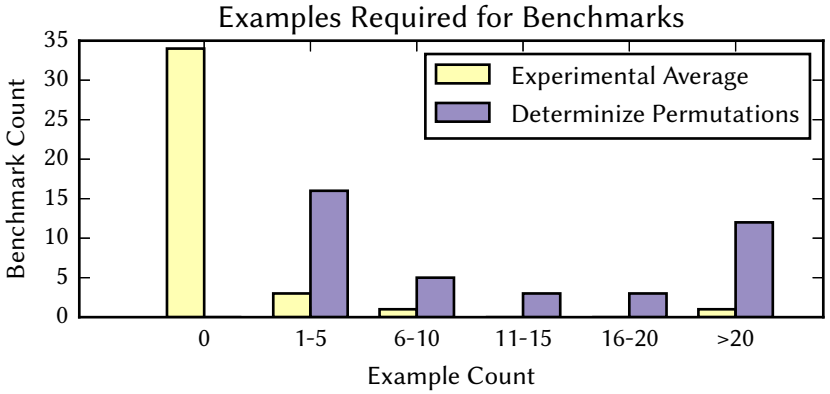


Fig. 12. Average number of random examples required to synthesize benchmark programs. **Experimental Average** is the average number of randomly generated examples needed to correctly synthesize the lens. **Determinize Permutations** is the theoretical number of examples required to determinize the choice 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.

algorithm might bias the selection. Figure 12 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 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 12 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 9 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 [PROSE 2017].

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.

8 RELATED WORK

In searching for equivalent regular expressions, we focused on Conway's equational theory rather than alternative axiomatizations such as Kozen's [Kozen 1994] and Salomaa's [Salomaa 1966]. 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, we focused on Conway's equational theory.

The literature on bidirectional programming languages and on lens-like structures is extensive. We discussed highlights in the introduction; readers can also consult a (slightly dated) survey [Czarnecki et al. 2009] and recent theoretical perspectives [Abou-Saleh et al. 2016; Fischer et al. 2015].

Recently, symbolic transducers have been used to infer program inverses [Hu and D'Antoni 2017], 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 §2.

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 [Gulwani 2011b; Le and Gulwani 2014; Perelman et al. 2014; Singh 2016; Singh and Gulwani 2012]. We compared our system to two of these unidirectional string transformers, Flash Fill [Gulwani 2011b] and FlashExtract [Le and Gulwani 2014]. 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 which occurred under two iterations – for efficiency reasons Flash Fill does not synthesize programs with nested loops [Gulwani 2011b]. 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 §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. Furthermore, by focusing on bidirectional transformations, we limit the space of synthesized functions to bijective ones, reducing the search space, and the expressiveness of the search space.

There are many other recent results showing how to synthesize functions from type-based specifications [Augustsson 2004; Feser et al. 2015; Frankle et al. 2015; Osera and Zdancewic 2015; Polikarpova et al. 2016; Scherer and Rèmy 2015]. These systems enumerate programs of their target language, orienting their search procedures to process only terms that are well-typed. Our system is distinctive in that it synthesizes terms in a language with many type equivalences. Perhaps the most similar is InSynth [Gvero et al. 2013], a system for synthesizing terms in the simply-typed lambda calculus that addresses equivalences on types. Instead of trying to directly synthesize terms of the simply-typed lambda calculus, InSynth synthesizes a well-typed term in the succinct calculus, a language with types that are equivalent "modulo isomorphisms of products and currying" [Gvero et al. 2013]. 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 [Feng et al. 2017] is another synthesis system that uses two communicating synthesizers to generate programs. In both Morpheus and Optician, one synthesizer provides an outline for the program, and the other fills in that outline with program details that satisfy the user's specifications. This approach works well in large search spaces, 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 [Gulwani 2011a; Le and Gulwani 2014; Solar-Lezama 2008; Yaghmazadeh et al. 2016]. 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.

9 CONCLUSION

Data processing systems often need to convert data back-and-forth between different formats. Domain-specific languages for generating bidirectional programs help prevent data corruption in such contexts, but are unfamiliar and hard to use. To simplify the development of bidirectional applications, we have created the first synthesizer of a bidirectional language, generating lenses from data format specifications and input/output examples. To reduce the size of the synthesis search space, our system introduces a new language of DNF lenses, which are typed by DNF regular expressions. We have proven our new language sound and complete with respect to a declarative specification. We also describe effective optimizations for efficiently searching through the refined space of lenses.

We evaluated our system on a range of practical examples drawn from other systems in the literature including Flash Fill and Augeas. In general, we found our system to be robust, to require few examples, and to finish in seconds, even on complex data formats. We found that our type-directed synthesis algorithm is able to generate data transformations too complex for both existing example-directed systems and for a naïve type-directed algorithm, succeeding on 35 more benchmarks than the tested existing alternatives. We attribute its success to a combination of (1) the information provided by format specifications, (2) the structure induced by user-specified names, and (3) the inferences used to guide search. The approaches we used are generalizable both to other bidirectional languages, as well as to other domain-specific languages with large numbers of equivalences on the types.

ACKNOWLEDGMENTS

We thank our anonymous reviewers, Loris D'Antoni, Fritz Henglein, Neil Jones, and Nate Foster for their useful feedback and discussions. We thank Solomon Maina and Nate Foster for their extensive assistance in integrating Optician into Boomerang. This research has been supported in part by DARPA award FA8750-17-2-0028 and ONR 568751 (SynCrypt).

REFERENCES

Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. 2016. 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*. 1–31.

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

Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. 2010. Matching Lenses: Alignment and View Update. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*.

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

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

R. Book, S. Even, S. Greibach, and G. Ott. 1971. Ambiguity in Graphs and Expressions. *IEEE Trans. Comput.* 20, 2 (Feb. 1971).

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

Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective. In *ICMT (Lecture Notes in Computer Science)*, Richard F. Paige (Ed.), Vol. 5563. Springer, 260–283.

Manfred Droste, Werner Kuich, and Heiko Vogler (Eds.). 2009. *Semirings and Formal Power Series.* Springer Berlin Heidelberg, 3–28. http://dx.doi.org/10.1007/978-3-642-01492-5_1

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

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

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

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

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. 2008. Quotient Lenses. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, Canada.*

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

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

Sumit Gulwani. 2011b. 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.

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

Brian Harry. 2014. A new API for Visual Studio Online. (2014).

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

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

Qinheping Hu and Loris D'Antoni. 2017. Automatic Program Inversion Using Symbolic Transducers. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 376–389. https://doi.org/10.1145/3062341.3062345

Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. 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*. 61–72.

D. Kozen. 1994. A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. *Information and Computation* 110, 2 (1994). http://www.sciencedirect.com/science/article/pii/S0890540184710376

Daniel Krob. 1991. Complete Systems of B-rational Identities. *Theor. Comput. Sci.* 89, 2 (Oct. 1991). http://dx.OMITdoi.org/10.1016/0304-3975(91)90395-I

Vu Le and Sumit Gulwani. 2014. 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.

Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. 2007. 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*. 21–30.

David Lutterkort. 2007. Augeas: A Linux Configuration API. (Feb. 2007). Available from `http://augeas.net/`.

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

Microsoft Corporation 2017. *Requirements and compatibility | Team Foundation Server Setup, Update and Administration.* Microsoft Corporation.

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2017a. Synthesizing Bijective Lenses. (2017). arXiv:arXiv:1710.03248 https://arxiv.org/abs/1710.03248

Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2017b. Optician-Tool. https://github.com/Optician-Tool/Optician-Tool. (2017).

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

Hugo Pacheco, Tao Zan, and Zhenjiang Hu. 2014. BiFluX: A Bidirectional Functional Update Language for XML. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*. 147–158.

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

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

Microsoft PROSE. 2017. Microsoft Program Synthesis using Examples SDK. (2017). https://microsoft.github.io/prose/

Arto Salomaa. 1966. Two Complete Axiom Systems for the Algebra of Regular Events. *J. ACM* 13, 1 (Jan. 1966). http://doi.acm.org/10.1145/321312.321326

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

Rishabh Singh. 2016. BlinkFill: Semi-supervised Programming by Example for Syntactic String Transformations. *Proc. VLDB Endow.* 9, 10 (June 2016).

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

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. University of California, Berkeley.

Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing Transformations on Hierarchically Structured Data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM.

Tao Zan, Li Liu, Hsiang-Shang Ko, and Zhenjiang Hu. 2016. 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*. 77–89.

Zirun Zhu, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. 2015. 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*. 43–50.