Synthesizing Symmetric Lenses

ANDERS MILTNER, Princeton University, USA
SOLOMON MAINA, 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

Lenses are programs that can be run both "front to back" and "back to front," allowing data and updates to be transformed in two directions. Since their introduction by Foster et al. [2007], lenses have been extensively studied and applied. Recent work [Miltner et al. 2018] has also demonstrated how techniques from *type-directed program synthesis* can be used to efficiently synthesize a very simple class of lenses—so-called *bijective lenses* over string data—given a pair of types (regular expressions) and a small number of examples.

We show how to extend this synthesis algorithm to a much wider class of lenses, which we call *simple symmetric lenses*, including both bijective lenses and the more widely used "asymmetric" lenses [Foster et al. 2007], as well as a rich subset of the full-blown "symmetric lenses" proposed by Hofmann et al. [2015]. Simple symmetric lenses are of independent theoretical interest, being the largest class of symmetric lenses that do not rely on persistent internal state.

Synthesizing simple symmetric lenses is substantially more challenging that synthesizing bijective lenses: Since some of the information on each side can be "disconnected" from the other side, there will in general be *many* lenses that agree with a given example. To guide the search process, we use *stochastic regular expressions* and ideas from information theory to estimate the amount of information propagated by a candidate lens.

We describe an implementation of simple symmetric lenses and our synthesis procedure as extensions to the Boomerang language [Bohannon et al. 2008]. We evaluate its performance on 48 benchmark examples drawn from Flash Fill [Gulwani 2011a], Augeus [Lutterkort 2008], the bidirectional programming literature, and electronic file format synchronization tasks. After modest tuning, which is encouraged by the interactive nature of the tool, our implementation can synthesize all these lenses in under 30 seconds.

1 INTRODUCTION

In today's big-data world, similar information is sometimes stored in different places and in different formats. For instance, electronic calendars come in iCalendar, vCalendar, and h-event formats [Calendars 2016]; US taxation data can be transmitted via Tax XML (used by the US government for e-filing) or TXF (used by TurboTax) [Finance and Accounting 2016]; and sewing machines can use embroidery files in formats with names like PES, DST, and several others [Machine Embroidery 2015].

One convenient way to maintain consistency between these varied data formats is to use a *lens* [Foster et al. 2007]—a bidirectional program that can transform updates to data represented in a format S to another format T and vice versa, providing "round tripping" guarantees that help ensure information is not lost or corrupted as it is transformed back and forth between different representations.

While domain-specific languages for writing lenses can facilitate building principled data transformations, *synthesizing* lenses can make the task even easier. Indeed, lens languages offer excellent targets for synthesis: their specialized, sub-Turing-complete syntax and expressive type systems drastically reduce the space of possible programs, making search-based synthesis easier. Recent

Authors' addresses: Anders Miltner, Princeton University, USA, amiltner@cs.princeton.edu; Solomon Maina, Princeton University, USA, smaina@cis.upenn.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.

work [Miltner et al. 2018] has exploited this observation, demonstrating that it is feasible to synthesize quite complex examples, albeit only for a restricted class of lenses—so-called bijective lenses. Specifically, given source and target formats plus a small number of examples of corresponding data, the bijective synthesis algorithm will find a bijective lens matching the examples, if one exists. Often, it can find an appropriate lens in less than a second with no examples at all. One reason that bijective lens synthesis is so effective, even relative to successful synthesis tools in other domains, is that there are typically not very many bijections between related data formats. Hence, if the synthesis algorithm can find any bijection at all, it is likely to be the right one.

However, many real-world data formats share just part of their information content with other representations. For instance, two related database tables might share four out of five columns, but each might have a different fifth column. Or, one ad-hoc system configuration file might include some format-specific meta data, such as a date or a reference number, whereas the same configuration file on another operating system does not. Indeed of the 48 benchmark problems described in Section 6, all of the benchmarks taken from Flash Fill [Gulwani 2011a] and many of the benchmarks that synchronize between two ad hoc file formats have this characteristic, 13 benchmarks in total. Consequently, synthesizing only strict bijections is often not useful in practice.

Could we extend synthesis tools to more kinds of lenses? In particular, what about *symmetric lenses* [Hofmann et al. 2015], a class that includes bijective lenses, "asymmetric" lenses where the transformation from S to T can throw away information that must then be restored when returning from T to S, and even more flexibile transformations that allow each side to retain information not present in the other?

One might hope that extending the algorithms from Miltner et al. [2018] to synthesize symmetric rather than bijective lenses would be relatively straightforward: Simply replace the bijective combinators with symmetric ones and search using similar heuristics. However, this naïve approach encounters two difficulties.

The first is pragmatic. "Classical" symmetric lenses as presented by Hofmann et al. [2015] operate over three structures: the source S, the target T and a complement C that contains the information not present in either S or T. Logistically, these complements must be stored and managed somehow. More fundamentially, complements complicate synthesis *specifications*—instead of just giving single examples of source and target pairings, users would have to give longer "interaction sequences" to show how a lens should behave. To avoid these complexities, we define a restricted variant of symmetric lenses, which we call *simple symmetric lenses*. Intuitively, simple symmetric lenses are the complete set of symmetric lenses that do not require external "memory" to recover data from past instances of S or T when making a round trip. They only need the most recent instance.

To characterize their expressive power formally, we show that simple symmetric lenses are exactly the symmetric lenses that satisfy an intuitive property called *forgetfulness*. To check that they are expressive enough in practice, we add simple symmetric lenses to the Boomerang language [Bohannon et al. 2008] and apply them to a range of real-world applications, showing in the process that simple symmetric lenses integrate smoothly with other advanced lens features provided by Boomerang, including quotient lenses [Foster et al. 2008; Maina et al. 2018] and matching lenses [Barbosa et al. 2010].

However, we are still left with a second difficulty with synthesizing symmetric lenses: While the number of bijective lenses between two related formats is typically tiny, the number of simple symmetric lenses is typically huge. Thus, if a naïve search algorithm selects the first simple symmetric lens it finds, this will generally *not* be the one we want. We need a new principle for identifying "more likely" lenses and a more sophisticated synthesis algorithm that uses this principles to search the space more intelligently.

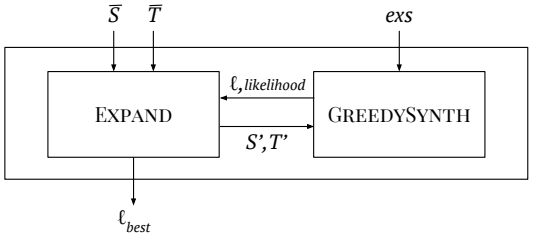

Fig. 1. Schematic diagram for the simple symmetric lens synthesis algorithm. The user provides regular expressions $\overline{S}$ and $\overline{T}$ and a set of examples *exs* as input. EXPAND first converts $\overline{S}$ and $\overline{T}$ to stochastic regular expressions S and T with default probabilities. It then finds pairs of stochastic regular expressions equivalent to S and T and iteratively proposes them to GREEDYSYNTH. GREEDYSYNTH finds a lens typed between the supplied SREs. When the algorithm finds a likely lens, it returns it.

For these, we turn to information theory. We consider a "likely" lens to be one that propagates "a lot" of information from source to target and vice versa. Conversely, an unlikely lens requires a large amount of additional information to recover the target given a source. This categorization follows from the intuition that users will typically want to synchronise formats with lots of shared information, and so lenses that propagate more information are preferable to ones that propagate less. This intuition is formalized using *stochastic regular expressions* (SREs) [Carrasco et al. 1996; Ross 2000], which simultaneously define a set of strings and a probability distribution over those strings. Using this probability distribution, we can calculate the shared information content of two formats and our measure of likelihood.

With simple symmetric lenses and our SRE-based likelihood measure in hand, we develop a new algorithm for synthesizing likely lenses. At its core, the algorithm performs a type-directed search between the descriptions of the source and target formats, measuring success using the likelihood measure. The interesting complications arise from the need to deal with regular-expression equivalences. There are infinitely many regular expressions equivalent to a given one, and the lens returned by a type-directed search will depend which of the infinitely many possible representations are chosen for its source and target formats. Moreover, there may be *no* lens connecting the source and target formats selected by the user: in general, the synthesis algorithm has to search to find equivalent ones for which a suitable lens exists. To tame this complexity, we divide the synthesis algorithm into two communicating search procedures (Figure 1), following Miltner et al. [2018]. The first, EXPAND, uses rewriting rules to iteratively propose new pairs of stochastic regular expressions equivalent to the original pair. The second, GREEDYSYNTH, takes a pair of SREs as input and uses a greedy, type-directed algorithm to find a simple symmetric lens between them, returning the lens and its likelihood score to EXPAND. The whole synthesis algorithm terminates when comparing the likilihood score of the best lens found so far with the complexity of newly generated SREs suggests that further searching is unlikely to be fruitful.

We added this synthesis procedure to the Boomerang implementation and evaluated its effectiveness on a benchmark suite of 48 problems, including data-cleaning tasks taken from Flash Fill [Gulwani 2011b], view-maintenance tasks taken from Augeas [Lutterkort 2008], and ad-hoc synchronization tasks from a variety of other sources (mostly from the bidirectional programming literature). The "programmer" can control the search process via examples and a *tuning parameter* that governs how much effort is spent trying to improve on existing solutions.

In summary, our contributions are:

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

Fig. 2. Hypothetical example data files and corresponding regular expressions used by management and HR at a US company to represent employee salaries and health insurance providers, respectively.

- We define *simple symmetric lenses*, a subclass of symmetric lenses that do not maintain any hidden state (§2). We present a collection of combinators for building simple symmetric lenses and describe how to calculate their likelihoods (§4).
- We give an algorithm for synthesizing lenses built from these combinators, using a novel application of stochastic regular expressions to guide the search process (§5).
- We extend the Boomerang implementation with simple symmetric lenses, demonstrating that they integrate smoothly with other lens extensions such as quotient lenses [Foster et al. 2008] and matching lenses [Barbosa et al. 2010]. We evaluate our implementation and the effects of various optimizations and the tuning parameter on a set of 48 lens synthesis benchmarks drawn from data cleaning, view maintenance, and file synchronization tasks. With only modest interactive tuning, the system can synthesize simple symmetric lenses for all of the benchmarks in under 30 seconds (§6).
- We show that simple symmetric lenses fall between asymmetric lenses [Foster et al. 2007] and full symmetric lenses [Hofmann et al. 2015] in expressiveness. Indeed, the class of simple lenses can be characterized semantically as the subset of full symmetric lenses for which a simple "forgetfulness" property holds (§7).

Along the way, we show that *star-semiring equivalences* on regular expressions remain valid when applied to stochastic regular expressions, preserving probability distributions as well as languages (§3). We close with related work (§7) and concluding thoughts (§8). Elided proofs can be found in the appendices included with the supplemental material.

2 OVERVIEW

We begin with an overview of simple symmetric lenses and our synthesis algorithm, using a simplified example drawn from a hypothetical U.S. company. In this company, management and human resources (HR) store information about employees in separate text files: management stores the names of employees and their salaries while HR stores the names of employees and their health insurance providers. Figure 2 gives examples of the two file formats and regular expressions describing them.

The company uses a simple symmetric lens to keep these files synchronized. When management adds a new employee, say "Chris Roe: 32500", this lens adds the corresponding entry "Chris Roe, UNK" to HR's file. The default value "UNK" represents the fact that the employee's insurance company is currently unknown. A similar update happens if HR adds a new employee before management knows about them, in which case the sentinel value "unk" reflects unknown salary information on management's side. Furthermore, if HR corrects an error in an employee's name, say changing "John Public" to "Jon Public", the lens mirrors this change into management's file. Not all

the data is mirrored, however: management's file is not updated in response to insurance changes, and HR's is oblivious to salary changes. Simple symmetric lenses are appropriate for keeping these two files synchronized because they contain a mix of shared and unshared information.

2.1 Simple Symmetric Lenses

Semantically, a simple symmetric lens between sets X and Y comprises four functions that collectively satisfy four "round-tripping" laws:

$$\begin{array}{rcll} \mathsf{createR} : X \to Y & (1) & \mathsf{putL}\,(\mathsf{createR}\,x)\,x = x & (\textsc{CreatePutRL}) \\ \mathsf{createL} : Y \to X & (2) & \mathsf{putR}\,(\mathsf{createL}\,y)\,y = y & (\textsc{CreatePutLR}) \\ \mathsf{putR} : X \to Y \to Y & (3) & \mathsf{putL}\,(\mathsf{putR}\,x\,y)\,x = x & (\textsc{PutRL}) \\ \mathsf{putL} : Y \to X \to X & (4) & \mathsf{putR}\,(\mathsf{putL}\,y\,x)\,y = y & (\textsc{PutLR}) \end{array}$$

The two create functions are used to propagate shared information and fill in default values when introducing new data (*e.g.*, the "unk" salary entry is added alongside the name to the management file when HR inserts a new employee). The two put functions propagate edits from one format to the other by combining a new value from one with an old value from the other. The round-tripping laws guarantee that pushing an "un-edited" value from one side (the result of a put or create) back through a lens in the other direction will get back to exactly where we began.

This definition differs from classic symmetric lenses in that it does not involve a *complement*. We give a detailed comparison in §7.1.

2.2 Simple Symmetric Lens Combinators

Generally, we do not expect programmers to define simple symmetric lenses by writing four functions by hand and showing they satisfy the round-tripping laws. Instead, we provide a set of combinators—a domain-specific language—for building complex lenses from simpler ones. Such languages have been given for building lenses over many different kinds of data, including database relations [Bohannon et al. 2006] and algebraic data types [Hofmann et al. 2015]. We focus here on combinators for defining lenses between string data formats described by regular expressions.

If $\overline{S}$ and $\overline{T}$ are regular expressions, then $\ell : \overline{S} \Leftrightarrow \overline{T}$ indicates that ℓ is a simple symmetric lens between $\mathcal{L}(\overline{S})$ and $\mathcal{L}(\overline{T})$. (Later, we will use undecorated variables for stochastic regular expressions; we mark plain REs with overbars.) When clear from context, we shoren "simple symmetric lens" to just "lens."

We illustrate some of these combinators by defining a lens between the two employee data formats in Figure 2. The finished lens, which appears in Figure 3, has type emp_salaries ⇔ emp_insurance.

The simplest combinator is the identity lens id, which takes as an argument a regular expression $\overline{S}$ and performs an identity mapping on data matching $\overline{S}$.

```
id(name) : name ⇔ name
```

The identity lens fully connects the corresponding data in the two formats: when one is updated, the other gets the exact same value. Both the createR and createL functions are the identity function ($\mathsf{createR}\,s = s$), and the put functions merely return the first argument ($\mathsf{putR}\,s_1\,s_2 = s_2$). In our example, when a name is updated in one file, the identity lens ensures it is also updated in the other file.

In contrast, the disconnect lens does not propagate any edits from one format to the other. It takes four arguments: two regular expressions and two strings. The regular expressions specify the formats of the corresponding data on the two sides while the strings provide default values.

```
disconnect(salary, "", "unk", "") : salary ⇔ company
```

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

Fig. 3. A lens that synchronizes management and HR employee files

On creates, the input values are thrown away, and default values are returned (createR s = "unk"), and on puts, the second argument is used and the first is thrown away (putR s t = t). In our extended example, the salary field is only present in management files, so we use the disconnect lens to ensure salary edits do not cause updates to the HR file. The defaults are strings that indicate the information is missing.

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

$$\text{ins}(t) = \text{disconnect}(\text{"", } t, \text{ "", } t)$$
$$\text{del}(s) = \text{disconnect}(s, \text{ "", } s, \text{ ""})$$

Finally, there are a number of combinators that compositionally construct more complex lenses from simpler ones. These include concatenation (concat(ℓ_1, ℓ_2) or $\ell_1.\ell_2$), variation (or(ℓ_1, ℓ_2)), and iteration (iterate(ℓ)). For example, the composite lens name_lens in the running example

```
id(name) . id("␣") . id(name) . del(":␣") . ins(",")
```

will transform “Jane Doe: ” to “Jane Doe,” in the forward direction.

The iterate lens also appears in our running example, where it synchronizes data for a list of employees when given a lens employee_lens that synchronizes data for a single employee:

$$\text{iterate}(\text{id}(\text{"\textbackslash n"}) \text{ . employee\_lens}) : (\text{"\textbackslash n" . emp\_salary})^* \Leftrightarrow (\text{"\textbackslash n" . emp\_ins})^*$$

We provide typing rules for each of the simple symmetric combinators, most of which are syntax directed. However, regular expressions have a number of equivalences on them, and sometimes we must convert one regular expression type to another, equivalent, regular expression type. Our typing rules include type equivalence, so if a lens has type $\overline{S} \Leftrightarrow \overline{T}$, then it is also has type $\overline{S}' \Leftrightarrow \overline{T}'$, where $\overline{S}$ is equivalent to $\overline{S}'$, and $\overline{T}$ is equivalent to $\overline{T}'$. The flexibility this type equivalence rule provides is useful when a different representation of a regular expression would facilitate synthesis.

2.3 Synthesizing Simple Symmetric Lenses

While our example is relatively simple for demonstration, writing such lenses for real-world formats is quite difficult, as these formats can be large and represent data in complex and unintuitive ways. To alleviate the difficulty of manually writing simple symmetric lenses, we wish to synthesize them instead. Given a pair of regular expressions and a set of input-output examples, we want to find a simple symmetric lens that is typed by the regular expression pair and satisfies the input/output examples. We call such a lens a *satisfying lens*. In our running example, we wish to synthesize a lens between emp_salaries and emp_insurance, using as an input-output example the data in Figure 2. The algorithm uses such examples to find suitable defaults for disconnect lenses and to figure out how to match data that could be synchronised in multiple ways.

A challenge is that the simple symmetric lens combinators permit many well-typed lenses between a given pair of regular expressions. For example, Figure 3 gives one possible lens between regular expressions emp_salaries and emp_insurance, but

disconnect(emp_salaries,emp_insurance,rep(emp_salaries),rep(emp_insurance))

is another valid lens (where rep($\overline{S}$) denotes some string in the language of $\overline{S}$). Furthermore, adding the input-output example in Figure 2 helps, but only a little: It may, in general, require *many* examples to rule out all possible occurrences of nested disconnect lenses, particularly in complex formats. While we could ask programmers to provide detailed and precise specifications for their synthesis tasks, we would prefer an algorithm that can intelligently choose a lens despite the underspecification. Instead of merely finding *any* satisfying lens, we wish to synthesize a satisfying lens that is likely to please the user.

But what is a "likely" satisfying lens? We propose the following heuristic: A satisfying lens is more likely if it connects more data between the two formats, *e.g.*, by using an identity rather than a disconnect lens. The likeliest satisfying lenses would connect as much data as possible. More formally, we define the likelihood of a satisfying lens as the expected number of bits required to recover data in one format from synchronized data in the other. Higher likelihoods correspond to lower numbers of bits. In more detail, we say that two strings s and t are *synchronized according to lens* l if l.putR $s\ t = t$ and l.putL $t\ s = s$.[1] We say that we can *recover* s from t using bits b and lens l if we can reconstruct s from t, b, and l. For example, given the id lens, we can recover s from t using no bits because, in this case, s is just t. In contrast, given the disconnect lens, we need enough bits to fully encode s in order to recover it from t because t contains no useful information. This calculation shows that if both id and disconnect are satisfying lenses, then id is more likely.

The expected number of bits required to recover data is the well-known information-theoretic concept of the *entropy* of the data [Shannon 1948]. Calculating entropy requires a probability distribution over the space of possible values for the data. Specifically, given a set S and a probability distribution $P : S \to \mathbb{R}$ over S, the entropy is $-\Sigma_{s \in S} P(s) log_2 P(s)$.

In our setting, we already have a way of expressing sets of data: regular expressions. What we need is a way to express probability distributions over that data. To that end, we adopt *stochastic regular expressions* [Carrasco et al. 1996; Ross 2000] (SREs), which are regular expressions in which each operator is annotated with a probability. A stochastic regular expression specifies both a set of strings and a probability distribution over that set, enabling us to calculate the entropy of the SRE.

This infrastructure gives us what we need to find likely satisfying lenses from a collection of satisfying lenses. All such lenses must synchronize data between the same two SREs S and T. For any such lens ℓ, we can calculate the expected number of bits required to recover a string t in $\mathcal{L}(T)$ from a synchronized string s in $\mathcal{L}(S)$. This expectation is the *conditional entropy of S given T and ℓ*, formally $\sum_{s \in S} P_S(s) \mathbb{H}(\{t \mid \ell.\text{putR}\ s\ t = t\})$. The likelihood we assign to ℓ is the sum of the conditional entropy of S given T and ℓ and the conditional entropy of T given S and ℓ. This metric assigns higher entropy (or cost) to lenses where knowing the string on one side provides little information about the string on the other side. It assigns zero entropy to bijections because given a string $s \in S$, the bijection exactly determines the corresponding string in T.

To obtain SREs from the plain regular expressions that we expect users to supply, we use a heuristic that attempts to assign probability annotations that give each string in the language equal probablity. Our algorithm could also take as input a stochastic regular expression rather than a plain one, which would allow the user to specify a data-specific probability distribution, either

[1] The notation l.putR extracts the putR function from simple symmetric lens l; similarly for putL.

manually calculated or inferred from a large dataset. We found our heuristic works well in practice, so we have not explored these options.

Now that we have a way to calculate likelihood, we need a way to search for likely lenses. Previous work on synthesis [Augustsson 2004; Feser et al. 2015; Frankle et al. 2015; Osera and Zdancewic 2015] has shown that we can use types to dramatically restrict the search space and improve the effectiveness of synthesis. In our setting, types are pairs of stochastic regular expressions where each regular expression is semantically equal to infinitely many others. We follow existing work on bidirectional program synthesis, and split our algorithm into two communicating synthesizers [Maina et al. 2018]. The first synthesizer, Expand, navigates the space of semantically equivalent regular expressions by applying rewrite rules that preserve both the semantics and the probability distributions of those stochastic regular expressions. Our algorithm ranks pairs of stochastic regular expressions by the number of rewrite rule applications required to obtain the pair from the stochastic regular expressions given as input. It passes the pairs off to the second synthesizer, GreedySynth, in rank order with the smallest first.

At a high level, GreedySynth looks for lenses between a given pair of stochastic regular expressions S and T that have the highest likelihood by performing a type-directed search. In more detail, GreedySynth first normalizes the stochastic regular expressions provided by Expand into *stochastic DNF regular expressions*. These stochastic DNF regular expressions are, intuitively, stochastic regular expressions with disjunctions distributed over concatenations, where associativity information has been removed. Then, GreedySynth uses the syntax of these n-ary DNF regular expressions to find highly normalized lenses, in a form we call *simple symmetric n-ary DNF lenses*. These lenses include neither a composition operator nor a type equivalence rule. These restrictions mean that each simple symmetric n-ary DNF lens is typed by a single pair of stochastic n-ary DNF regular expressions and GreedySynth's search space for a given pair of regular expressions is finite. The variations that GreedySynth considers include choosing between id and disconnect lenses and deciding how to match multiple occurrences of the same kind of data, *e.g.*, the two occurrences of name in the management and HR data formats (we want to map first names to first names and last names to last names). GreedySynth converts yields a simple symmetric lens by converting the n-ary forms into the binary forms provided in the surface language.

As an example of the partnership between Expand and GreedySynth, consider the lens shown in Figure 3. Inferring this lens requires Expand to use a star-semiring axiom to rewrite the regular expression ("\n" . emp_salary)* within emp_salaries into "" | (("\n" . emp_salary). ("\n" . emp_salary)*) so that GreedySynth can find the appropriate lens.

Our strategy of communicating synthesizers gives us a way to enumerate pairs of stochastic regular expressions of increasing rank and to efficiently search through them, but it poses a problem: when do we stop? We may have found a promising lens between a pair of stochastic regular expressions, but a different pair we haven't yet discovered may give rise to an even better lens. Or, we may have found a promising lens between a given pair of stochastic regular expressions, but the discovered lens is not acceptable to the users.

Our algorithm must resolve a three-way tension between the quality of the inferred lens, the number of examples the user must provide to eliminate unwanted lenses, and the amount of time it takes to return a result. For example, if the algorithm has already found the lens in Figure 3, we don't want to spend a lot of time searching for an even better lens.

To resolve this tension, our algorithm uses heuristics to judge whether to return the current best satisfying lens to the user or to pass the next set of unrollings to GreedySynth. The heuristics favor stopping if the current best satisfying lens has relatively high likelihood, indicating the lens is very promising. The heuristics also favor stopping if Expand has delivered to GreedySynth all pairs of stochastic regular expressions at a given rank and there is a large number of pairs at the

$$
\begin{array}{rcll}
S \mid_1 \emptyset & \equiv^s & S & +\ Ident \\
S \cdot \emptyset & \equiv^s & \emptyset & 0\ Proj_R \\
\emptyset \cdot S & \equiv^s & \emptyset & 0\ Proj_L \\
(S \cdot S') \cdot S'' & \equiv^s & S \cdot (S' \cdot S'') & \cdot\ Assoc \\
(S \mid_{p_1} S') \mid_{p_2} S'' & \equiv^s & S \mid_{p_1 p_2} (S' \mid_{\frac{(1-p_1)p_2}{1-p_1 p_2}} S'') & \mid\ Assoc \\
S \mid_p T & \equiv^s & T \mid_{1-p} S & \mid\ Comm \\
S \cdot (S' \mid_p S'') & \equiv^s & (S \cdot S') \mid_p (S \cdot S'') & Dist_R \\
(S' \mid_p S'') \cdot S & \equiv^s & (S' \cdot S) \mid_p (S'' \cdot S) & Dist_L \\
\epsilon \cdot S & \equiv^s & S & \cdot\ Ident_L \\
S \cdot \epsilon & \equiv^s & S & \cdot\ Ident_R \\
S^{*p} & \equiv^s & \epsilon \mid_{1-p} (S \cdot S^{*p}) & Unrollstar_L \\
S^{*p} & \equiv^s & \epsilon \mid_{1-p} (S^{*p} \cdot S) & Unrollstar_R
\end{array}
$$

Fig. 4. Stochastic Regular Expression Star-Semiring Equivalence

next rank because searching through all such pairs will take a long time. With this approach, the algorithm can quickly respond with a satisfying lens with relatively high likelihood. If users are unhappy with the result, they can either refine the search by supplying additional examples, which serve both to rule out the previously proposed lens(es) and to reduce the size of the search space by cutting down on the number of satisfying lenses, or supply a tuning parameter (discussed in §5.5) that adjusts how much effort is put into the search.

3 STOCHASTIC REGULAR EXPRESSIONS

To characterize the likely lenses, we must compute the expected number of bits needed to recover a string in one data source from a synchronized string in the other data source. To do this, we must first develop a probabilistic language model for our language. We do this with *stochastic regular expressions*, regular expressions annotated with probability information. With this, we can jointly express a language and a probability distribution over that language. In the following grammar, s ranges over strings, and p ranges over real numbers between 0 and 1, exclusive.

$$S, T ::= \ s \mid \emptyset \mid S^{*p} \mid S_1 \cdot S_2 \mid S_1 \mid_p S_2$$

In the stochastic regular expression $S_1 \mid_p S_2$, the annotation p represents the likelihood the string is in S_1. In the stochastic regular expression S^{*p}, the annotation p how much more likely having a string involving more than n iterations is than having a string involving exactly n iterations. The semantics of SREs is defined by the function P below.

$$
\begin{array}{rcl}
P_s(s'') & = & \begin{cases} 1 & \text{if } s = s'' \\ 0 & \text{otherwise} \end{cases} \\
P_\emptyset(s) & = & 0 \\
P_{S_1 \cdot S_2}(s) & = & \Sigma_{s=s_1 s_2} P_{S_1}(s_1) P_{S_2}(s_2) \\
P_{S_1 \mid_p S_2}(s) & = & p P_{S_1}(s) + (1-p) P_{S_2}(s) \\
P_{S^{*p}}(s) & = & \Sigma_n \Sigma_{s=s_1 \ldots s_n} p^n (1-p) \Pi_{i=1}^n P_S(s_i)
\end{array}
$$

3.1 Stochastic Regular Expression Equivalences

Expand needs to be able to find equivalent stochastic regular expressions. However, existing work does not provide a means to reason about stochastic regular expression equivalence. We extend the *star-semiring* [Droste et al. 2009] equivalences to stochastic regular expressions (Figure 4).

Theorem 1. If $S \equiv^s T$ then $P_S(s) = P_T(s)$, for all strings $s \in \mathcal{L}(S)$.

Proof. The proofs of this theorem, and subsequent theorems in this paper, appears in the appendix present in the full version of the paper accompanying this submission. □

We prove these star-semiring equivalences sound, as those are the equivalences we either traverse (with Expand), or normalize across (as part of GreedySynth).

3.2 Stochastic Regular Expression Entropy

The entropy of a data source S is the expected number of bits required to describe a random element in S ($-\Sigma_{s \in S} P(s) log_2 P(s)$). The entropy of a stochastic regular expression can be computed directly from its syntax, when the stochastic regular expression is unambiguous and contains no empty subcomponents.

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

We have proven that this syntactic computation corresponds to the entropy of the stochastic regular expression.

Theorem 2. If S is unambiguous and does not contain $\emptyset$ as a subterm, $\mathbb{H}(S)$ is the entropy of S.

Without unambiguity constraints it is not easy to calculate entropy (and the method we have given is certainly incorrect). For example, consider the SRE "a" $|_{.5}$ "a". Using the above computation,s this SRE has an entropy of 1, but the entropy should be 0. These difficulties become particularlly complex when the languages are ambiguous in subtle ways, the algorithm must be correct when only some of the two languages overlap. Similar issues occur when trying to find the entropy when $\emptyset$ is a subterm (what should the entropy of $\emptyset^{*.5}$ be?), so we do not define entropy on $\emptyset$.

Fortunately, we already require unambiguous regular expressions as input to our synthesis procedure for other reasons, and we can easily preprocess emptysets out of the stochastic regular expressions that are themselves nonempty by traversing the appropriate star-semiring equivalences (*e.g.*, $\emptyset \mid_{.5}$ "s" $\equiv^S$ "s").

4 SIMPLE SYMMETRIC STRING LENSES

We now give a formal description of all the combinators in our symmetric lens language. While our algorithms perform synthesis on stochastic regular expressions, lenses themselves are typed between regular expressions. We can recover a regular expression from a stochastic regular expression by deleting probability annotations. In a slight abuse of notation, if S is a stochastic regular expression, $\overline{S}$ is the regular expression obtained by removing probability annotations. The typing judgment $\ell : \overline{S} \Leftrightarrow \overline{T}$ means that ℓ is a well typed lens between $\mathcal{L}(\overline{S})$ and $\mathcal{L}(\overline{T})$.

$$\begin{aligned} \ell ::= &\mathsf{id}(\overline{S}) \mid \mathsf{disconnect}(\overline{S}, \overline{T}, s, t) \mid \mathsf{iterate}(\ell) \mid \mathsf{concat}(\ell_1, \ell_2) \mid \mathsf{swap}(\ell_1, \ell_2) \mid \mathsf{or}(\ell_1, \ell_2) \mid \ell_1 \,;\, \ell_2 \mid \\ &\mathsf{merge\_right}(\ell_1, \ell_2) \mid \mathsf{merge\_left}(\ell_1, \ell_2) \mid \mathsf{invert}(\ell) \end{aligned}$$

We have already described a number of these lenses in §2. The compose lens $\ell_1; \ell_2$ composes two lenses sequentially, it transforms from one data format to the other by first transforming to an intermediate format, shared as the target of ℓ_1 and the source of ℓ_2. The merge_right lens combines two sublenses which share the same target. Which lens gets used depends on what data is in the left-hand format, if it is in the domain of ℓ_1 then ℓ_1 gets used, and similarly for ℓ_2. The merge_left lens is symmetric to merge_right, the two sublenses must share the same source. Finally, invert(ℓ) inverts a lens, createR becomes createL and putR becomes putL, and vice-versa.

We now provide the typing derivations and semantics for these lenses. We do not include the semantics for combinators equivalent to those presented in prior work [Bohannon et al. 2008; Hofmann et al. 2015]. Furthermore, we only present createR and putR when createL and putL are symmetric; full details can be found in an appendix (in the supplemental materials). For simplicity of presentation, we use unambiguous regular expressions everywhere, even though some lenses can actually be defined over ambiguous regular expressions (for example, the identity lens doesn't need any ambiguity constraints); we believe relaxing this restriction is unproblematic.

$$\frac{}{\mathsf{id}(\overline{S}) : \overline{S} \Leftrightarrow \overline{S}} \qquad \begin{aligned} \mathsf{createR}\ s &= s \\ \mathsf{putR}\ s_1\ s_2 &= s_1 \end{aligned}$$

Note that the identity lens ignores the second argument in the put functions. Because the two formats are fully synchronized, no knowledge of the prior data is needed.

$$\frac{s \in \mathcal{L}(\overline{S}) \qquad t \in \mathcal{L}(\overline{T})}{\mathsf{disconnect}(\overline{S}, \overline{T}, s, t) : \overline{S} \Leftrightarrow \overline{T}} \qquad \begin{aligned} \mathsf{createR}\ \ s' &= t \\ \mathsf{putR}\ \ s'\ \ t' &= t' \end{aligned}$$

Just as the identity lens ignores the second argument in puts, disconnect lenses ignore the first in both puts and creates. The data is unsynchronized in these two formats, information from one format doesn't impact the other.

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

Concat is similar to concatenation in existing string lens languages like Boomerang. For such terms, we do not provide the semantics, and merely refer readers to existing work. The swap combinator is similar to concat, though the second regular expression is swapped.

$$\frac{\ell_1 : \overline{S}_1 \Leftrightarrow \overline{T}_1 \qquad \ell_2 : \overline{S}_2 \Leftrightarrow \overline{T}_2}{\mathsf{or}(\ell_1, \ell_2) : \overline{S}_1 \mid \overline{S}_2 \Leftrightarrow \overline{T}_1 \mid \overline{T}_2}$$

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

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

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

$$\frac{\ell_1 : \overline{S}_1 \Leftrightarrow \overline{T} \qquad \ell_2 : \overline{S}_2 \Leftrightarrow \overline{T}}{\mathsf{merge\_right}(\ell_1, \ell_2) : \overline{S}_1 \mid \overline{S}_2 \Leftrightarrow \overline{T}}$$

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

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

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

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

The merge_right lens is interesting because it merges data where one data can be in two formats, and one data has only one format. In previous work [Bohannon et al. 2008], this was combined into or, where or could have ambiguous types, but we find it more clear to have explicit merge operators: it is easier to see what lens the synthesis algorithm is creating.

$$\frac{\ell_1 : \overline{S} \Leftrightarrow \overline{T}_1 \qquad \ell_2 : \overline{S} \Leftrightarrow \overline{T}_2}{\mathsf{merge\_left}(\ell_1, \ell_2) : \overline{S} \Leftrightarrow \overline{T}_1 \mid \overline{T}_2}$$

The merge_left lens is symmetric to merge_right.

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

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

Composing is interesting in the put functions. Because puts require intermediary data, we recreate that intermediary data with creates.

$$\frac{\ell : \overline{S} \Leftrightarrow \overline{T}}{\mathsf{iterate}(\ell) : \overline{S}^* \Leftrightarrow \overline{T}^*} \qquad \frac{\ell : \overline{S} \Leftrightarrow \overline{T}}{\mathsf{invert}(\ell) : \overline{T} \Leftrightarrow \overline{S}}$$

The iterate lens deals with iterated data, while inverting reverses the direction of a lens: creating on the right becomes creating on the left and vice versa, and putting on the right becomes putting on the left and vice versa. The invert combonator is particularly useful when chaining many compositions together, as it can be used to align the central types.

$$\frac{\ell : \overline{S} \Leftrightarrow \overline{T} \qquad \overline{S} \equiv^s \overline{S}' \qquad \overline{T} \equiv^s \overline{T}'}{\ell : \overline{S}' \Leftrightarrow \overline{T}'}$$

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

4.1 Lens Costs

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

$$\begin{aligned}
\mathbb{H}^{\rightarrow}(T \mid \mathsf{id}(T), T) &= [0,0] \\
\mathbb{H}^{\rightarrow}(T \mid \mathsf{disconnect}(S,T,s,t), S) &= [\mathbb{H}(T), \mathbb{H}(T)] \\
\mathbb{H}^{\rightarrow}(T^{*q} \mid \mathsf{iterate}(\ell), S^{*p}) &= \tfrac{p}{1-p}\mathbb{H}^{\rightarrow}(T \mid \ell, S) \\
\mathbb{H}^{\rightarrow}(T_1 \cdot T_2 \mid \mathsf{concat}(\ell_1,\ell_2), S_1 \cdot S_2) &= \mathbb{H}^{\rightarrow}(T_1 \mid \ell_1, S_1) + \mathbb{H}^{\rightarrow}(T_2 \mid \ell_2, S_2) \\
\mathbb{H}^{\rightarrow}(T_2 \cdot T_1 \mid \mathsf{swap}(\ell_1,\ell_2), S_1 \cdot S_2) &= \mathbb{H}^{\rightarrow}(T_2 \mid \ell_1, S_2) + \mathbb{H}^{\rightarrow}(T_1 \mid \ell_1, S_1) \\
\mathbb{H}^{\rightarrow}(T_1 \mid_q T_2 \mid \mathsf{or}(\ell_1,\ell_2), S_1 \mid_p S_2) &= p\mathbb{H}^{\rightarrow}(S_1 \mid \ell_1, T_1) + (1-p)\mathbb{H}^{\rightarrow}(S_2 \mid \ell_2, T_2) \\
\mathbb{H}^{\rightarrow}(T \mid \mathsf{merge_right}(\ell_1,\ell_2), S_1 \mid_p S_2) &= p\mathbb{H}^{\rightarrow}(T \mid \ell_1, S_1) + (1-p)\mathbb{H}^{\rightarrow}(T \mid \ell_2, S_2) \\
\mathbb{H}^{\rightarrow}(T_1 \mid_q T_2 \mid \mathsf{merge_left}(\ell_1,\ell_2), S) &= (0, \mathbb{H}^{\rightarrow}(T \mid \ell_1, S_1) + \mathbb{H}^{\rightarrow}(T \mid \ell_2, S_2) + 1) \\
\mathbb{H}^{\rightarrow}(S \mid \mathsf{invert}(\ell), T) &= \mathbb{H}^{\leftarrow}(S \mid \ell, T)
\end{aligned}$$

$\mathbb{H}^{\leftarrow}(S \mid \ell, T)$ is defined symmetrically. These functions bound the expected number of bits to recover one data format from a synchronized string in the other format. Note that we would be able to exactly calculate the conditional entropy, were it not for merge_left. If $\mathsf{merge\_left}(\ell_1,\ell_2) : S \Leftrightarrow T_1 \mid_q T_2$, given a string in s, we need to determine if the synchronized string is in T_1 or T_2. However, this information content is dependent on the how likely the synchronized string is to be in T_1 or T_2. Nevertheless, we typically calculate the conditional entropy exactly, as merges are relatively uncommon on practice; only 2 of the lenses sythesized in our benchmarks include merges.

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

Theorem 3. Let $\ell : S \Leftrightarrow T$, where ℓ does not include composition, S and T are unambiguous, and neither S nor T contain any empty subcomponents.

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

Note that our definition of $\mathbb{H}$ contains no case for sequential composition $\ell_1; \ell_2$ and our theorem excludes lenses that contain such compositions. Unfortunately, defining the entropy of lenses involving composition is challenging because ℓ_1 might, for instance, add some information that is subsequently projected away in ℓ_2. Such operations can cancel, leaving a zero-entropy bijection composed from two non-zero entropy transformations. However, detecting such cancellations directly is complicated and this property is difficult to determine merely from syntax. Fortunately, we are able to avoid such complication altogether by synthesizing *DNF lenses*—simple symmetric lenses that inhabit a disjunctive normal form. DNF lenses have a number of properties that facilitate synthesis: they cut down the program search space dramatically and they do not contain instances of composition.

5 SYNTHESIS

Algorithm 1 presents our synthesis algorithm at a high level of abstraction. The input regular expressions are first converted into stochastic regular expressions with ToStochastic. This pair of SREs is used to initialize a priority queue (*pq*). The priority of a SRE pair is the number of rewrites needed to derive the pair from the originals. Next, SynthSymLens enters a loop that searches for likely lenses. The loop terminates when the algorithm believes it is unlikely to find a better lens than the best one it has found so far (a termination condition defined by Continue). Within each iteration of the loop, it:

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

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

Stochastic DNF regular expressions are critical to the success of our algorithm. Stochastic DNF regular expressions (SDNF REs) are a pseudo-normal form for stochastic regular expressions. Of all the star-semiring equations required to relate two SDNF REs, only those relating to Kleene star (*e.g.*, $\mathit{Unrollstar}_L$ and $\mathit{Unrollstar}_R$) are needed to relate two SDNF REs; other axioms, like associativity, commutativity or distributivity, are unneeded. This important property makes it possible to construct an efficient algorithm (GreedySynth), directed by the syntax of two SDNF REs, to synthesize any lenses in the same "class" (*i.e.*, any lenses that have the same type when the Kleene star axioms are unused). This property, and the resulting efficiency of GreedySynth is key to the effectiveness of our core algorithms.

5.1 Searching for (S, T) Candidate Classes

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

$$S^{*p} \quad \rightarrow \quad \epsilon \mid_p (S \cdot S^{*p})$$
$$S^{*p} \quad \rightarrow \quad \epsilon \mid_p (S^{*p} \cdot S)$$

Algorithm 1 SynthSymLens

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

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

The procedure Continue returns false based on the how long the search has been going, and how hard it expects the next class of problems to be. In particular, if PQ.Peek$(pq) = (S, T)$, that RE pair is at distance d, the number of pairs in pq at distance d is n, and the current best lens has cost c, then Continue(pq) checks whether $c > d + log_2(n)$.

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

5.2 Stochastic DNF Regular Expressions

Just as SREs are REs with probability annotations, SDNF REs are SDNF REs with probability annotations. As with SREs and regular expressions, if DS is a stochastic DNF regular expression, $\overline{DS}$ is the DNF regular expression generated by erasing probability annotations.

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

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

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

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

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

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

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

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

Fig. 5. Stochastic DNF Regular Expression Functions

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

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

The algorithm for converting a stochastic regular expressions into its corresponding stochastic DNF regular expressions, written $\Downarrow S$, is defined below.

$$\begin{aligned} \Downarrow s &= \langle ([s], 1) \rangle & \Downarrow (S_1 \cdot S_2) &= \Downarrow S_1 \odot \Downarrow S_2 \\ \Downarrow \emptyset &= \langle \rangle & \Downarrow (S_1 \mid_p S_2) &= \Downarrow S_1 \oplus_p \Downarrow S_2 \\ \Downarrow (S^{*p}) &= \mathcal{D}((\Downarrow S)^{*p}) \end{aligned}$$

This conversion relies on operators defined in Figure 5. We have proved that this conversion respects languages and probability distributions.

Theorem 4. $P_S(s) = P_{\Downarrow S}(s)$ and $\mathcal{L}(S) = \mathcal{L}(\Downarrow S)$.

ToStochastic. With stochastic DNF regular expressions and $\Downarrow$ defined, it is easier to explain ToStochastic, the function that converts regular expressions into stochastic regular expressions. If users provided stochastic regular expressions themselves (perhaps inferred from example data), this step could be skipped, though our implementation does not (yet) permit users to provide such stochastic regular expressions. If S is a stochastic DNF regular expression generated by ToStochastic, then $\Downarrow S = \langle (SQ_1, \frac{1}{n}) \mid \ldots \mid (SQ_n, \frac{1}{n}) \rangle$ for some sequences $SQ_1 \ldots SQ_n$, and every stochastic atom generated by ToStochastic is $DS^{*.8}$. In particular, $\Downarrow$ generates regular expressions whose DNF form gives equal probability to all sequence subcomponents of the DNF regular expression., and gives iterations a .8 chance to continue iterating. In our experience generating random strings from regular expressions, these probabilities provide good distributions of strings—stars are iterated 4 times on average, and no individual choice in a series disjunctions is chosen disproportionately often. As we have found these distributions generate reasonable strings, we believe they describe the shape of data well.

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

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

Theorem 5. $\mathbb{H}(DS)$ is the entropy of P_{DS}.

5.3 Symmetric DNF Lenses

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

Intuitively, a symmetric DNF lens *sdl* is a union of symmetric sequence lenses, $ssql_1 \ldots ssql_p$, a symmetric DNF sequence lens is a concatenation of symmetric atom lenses, $sal_1 \ldots sal_n$, and a symmetric atom lens *sal* is an iteration of a symmetric DNF lens.

Just as we analyzed the information content of ordinary regular expressions, we can analyze the information content of DNF regular expressions. As before, we use $\mathbb{H}^{\rightarrow}(DT \mid sdl, DS)$ to calculate bounds of the expected amount of information required to recover a string in DT from a string in DS, synchronized by *sdl*. We use the function $\mathbb{H}^{\leftarrow}(DS \mid sdl, DT)$ to calculate bounds on the expected amount of information required to recover a string in DS from a string in DT, synchronized by *sdl*.

The details of these definitions are syntactically tedious, but not intellectually difficult. We elide them here but include them in the appendix.

5.4 GreedySynth

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

Symmetric DNF Lenses. Algorithm 2 presents GreedySynth, which synthesizes symmetric DNF lenses. Its inputs are a suite of input-output examples and a pair of stochastic DNF regular expressions. First, CannotMap determines whether there is no lens satisfying the examples, and if so, GreedySynth returns *None* immediately. Otherwise, GreedySynth finds the best lenses (given the examples that match them) between all sequence pairs (SQ_i, TQ_j) drawn from the left and right DNF regular expressions. (The function CartesianMap maps its argument across the cross product of the input lists). A priority queue containing these sequence lenses, ordered by cost, is then initialized with PQ.Create. The symmetric lens is then built up iteratively from these sequence lenses, where the state of the partially constructed lens is tracked in the lens builder, *lb*.

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

As an example, consider searching for a lens between "" | name.name* and "" | name. GreedySynth might first pop the sequence pair "" and "", because there is a bijective sequence lens between

Algorithm 2 GreedySynth

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

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

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

Symmetric Sequence lenses. Algorithm 3 presents GreedySeqSynth, which synthesizes symmetric sequence lenses using an algorithm whose structure is similar to GreedySynth's. It calls AtomSynth, which synthesizes atom lenses by iterating a DNF lens between its subcomponents.

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

GreedySeqSynth loops until there are no more atom lenses in the priority queue. In the loop, a popped atom lens is considered "useful" if adding it to the sequence will lower the cost of the generated sequence lens, or if examples show that one of its atoms must not be disconnected. Each atom can be part of only one lens at a time, so the algorithm must sometimes remove a previously chosen atom lens in order to connect one that must not be disconnected. The algorithm succeeds when all atoms that must not be disconnected are involved in an atom lens; SLensBuilder.ToDNFLens returns *None* otherwise.

5.5 Optimizations and Termination Tuning

Our implementation includes a number of optimizations not described above: annotations that guide DNF conversion; an expansion inference algorithm; hash-consing and memoization; and compositional synthesis. The implementation also provides for an additional specification parameter that customizes the termination condition for the synthesis procedure. The optimizations make the system performant enough for interactive use, and the tuning parameter gives the user a "knob" by which the algorithm can be directed to try harder to find a good lens.

Algorithm 3 GreedySeqSynth

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

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

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

Expansion Inference. These additional rewrites make the search through possible regular expressions harder. Our algorithm identifies when certain closed regular expressions can *only* be involved in a disconnect lens. Such regular expressions will automatically be opened.

Hash-consing and Memoization. We further improve GreedySynth by hash-consing stochastic DNF regular expressions. This optimization enables fast equality checks and saves memory.

Compositional Synthesis. Compositional synthesis allows us to use previously defined lenses, whether they are previously defined by users or by synthesis. As the synthesizer processes a Boomerang file, it accumulates lens definitions and their types. It tackles synthesis problems one after another and there may be many such problems in a given program. During synthesis, if an existing lens has the right type and agrees with the examples. GreedySynth will use it. Consequently, if a large synthesis problem cannot be solved all at once, a user can first generate subproblems for parts of a data format, and the larger problem can subsequently use solutions to those subproblems. In our experience, this is a very powerful tool that allows synthesis to tackle problems of arbitrary complexity.

Termination Customization. In this work, there are three competing constraints. The tool should (1) return lenses quickly to users, (2) return the highest quality lenses, and (3) not require users to write a large number of examples. Our termination heuristic works well for most situations, but there are cases where a large number of expansions must be performed to find the correct lens. Users can work around such a problem by providing a large number of examples, but constructing examples is a chore, especially when formats are complicated. We have often found it simpler to allow users to change the synthesis termination condition to demand SynthSymLens be more judgemental when it comes to deciding it has found a "good enough" lens. Conversely, if the user's desired transformation has a particularly high cost, the algorithm may spend a long time searching in vain for lenses with lower cost. By tuning SynthSymLens to be more permissive, such a search can be short-circuited.

For these situations, users can provide a number t that affects the strictness of the termination condition by adjusting the Continue formula as follows. When PQ.Peek$(pq) = (S, T)$, S and T are at distance d, the number of pairs in pq at distance d is n, and the current best lens has cost c,

$$\textsc{Continue}(pq) := c > \min(d + log_2(n) - t, 0).$$

With a positive t, the termination condition is stricter, and the algorithm will spend longer searching for more likely lenses. With a negative t, the termination condition is less strict, and the algorithm will spend less time searching. The default t is 0.

6 EVALUATION

We have implemented symmetric DNF lenses as an extension to Boomerang. Specifically, we added symmetric lens combinators to Boomerang, and implemented all of Boomerang's asymmetric lens combinators using a combination of the symmetric lens combinators presented in this paper and symmetric versions of the asymmetric extensions (like matching lenses and quotient lenses) already present in Boomerang. We also integrated our synthesis engine into Boomerang. This integration allows users to write synthesis tasks alongside lens combinators, incorporate synthesis results into manually-written lenses, and reference previously defined lenses in the synthesis of new lenses.

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

(1) Can the algorithm (with suitable examples and termination tuning) find the correct lens?
(2) Is the synthesis procedure efficient enough to be used in everyday development?
(3) How much slower is our tool on bijective lens synthesis benchmarks than the existing Optician system customized for bijective lenses?
(4) How effective is the information-theoretic search heuristic and how does termination tuning affect the results?

6.1 Benchmark Suite

Our benchmarks are drawn from three different sources.

(1) We adapted 8 data cleaning benchmarks from Flash Fill. Note that our tool produces bidirectional transformations rather than one-way transformers like Flash Fill.
(2) We adapted 29 benchmarks from Augeas. These benchmarks turn the ad hoc data present in Linux configuration files into a structured dictionary representation.
(3) We created 11 additional benchmarks derived from real-world examples and/or the bidirectional programming literature. These tasks range from synchronizing REST and JSON web resource descriptions to synchronizing BibTeX and EndNote citation descriptions.

All three sources provide realistic problems suitable for answering the research questions: Flash Fill data cleaning tasks are either derived from online help forums or taken from the Excel product

team; Augeas is a utility that transforms Linux configuration files into a manageable dictionary form; and many of the remaining benchmarks arise from synchronizing real-world data formats such as BibTeX and EndNote. Each task consists of a source and target regular expression.

6.2 Synthesizing Correct Lenses

Because our tool is intended to be interactive—the user typically starts with zero or one examples, runs the tools, determines whether the output is the desired one and, if not, either adds an example or adjusts the termination metric and then tries again—it is hard to cleanly evaluate the net user experience. The primary concern is whether it is usable to generate synthesize correct lenses.

To determine whether the system can synthesize the desired lenses, we ran it interactively on all 48 tasks, working with the system to create sufficient examples and adjust the termination parameter. In all cases, the desired lens was obtained, which we verified by manually inspecting the generated lens programs. The majority of the tasks required only a single example and none required more than three examples in order to synthesize the desired lens.[2]

Adjusting the termination parameter was needed in only 9 of the 48 tasks. In practice, we found that customizing the termination metric to be quite easy: if manual inspection of the lens showed there were too few ids, and too many disconnects or merges, we would increase the termination parameter. If synthesis took too long, we would decrease it. Section 6.5 studies the effect of fixing the termination metric.

6.3 Effectiveness of Compositional Synthesis

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

- **SS**: The symmetric synthesis algorithm with all optimizations enabled.
- **SSNC**: The symmetric synthesis algorithm, with no compositional synthesis enabled.

Recall that compositional synthesis allows users to break a benchmark into a series of smaller synthesis tasks, whose solutions are utilized in more complex synthesis procedures. Compositional synthesis (**SS** mode) allows our system to scale to arbitrarily large and complex formats; measuring it shows of the responsiveness of the system when used as intended. **SSNC** mode, which synthesizes a complete lens all at once, provides a useful experimental stress test for the system.

For each benchmark in the suite and each mode, we ran the system with a timeout of 60 seconds, averaging the result over 5 runs. Figure 6 summarizes the results of these tests. We find that our algorithm is able to synthesize all of the benchmarks in under 30 seconds. Without compositional synthesis, the synthesis algorithm is able to solve 26 problem instances.

6.4 Slowdown Compared to Bijective Synthesis

To compare to the existing bijective synthesis algorithm, we run our symmetric synthesis algorithm on the original Optician benchmarks, comprised of 39 bijective synthesis tasks.[3]

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

- **SS**: The symmetric synthesis algorithm with all optimizations enabled.
- **BS**: The existing bijective synthesis algorithm with all optimizations enabled.

For each benchmark, we ran it in both modes with a timeout of 60 seconds and averaged the result over 5 runs. Figure 7 summarizes the results of these tests. On average, **SS** took 1.3 times

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

[3] We had to slightly alter four of these benchmarks, either by providing additional examples or by tuning termination parameters. Without these alterations, symmetric synthesis yielded a lens that fit the specification but that was undesired.

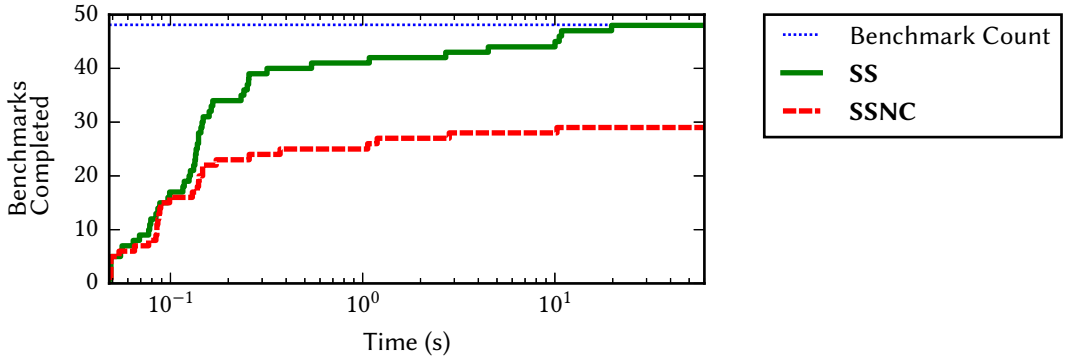

Fig. 6. Number of benchmarks that can be solved by a given algorithm in a given amount of time. SS is the full symmetric synthesis algorithm. SSNC is the symmetric synthesis algorithm without using a library of existing lenses. The symmetric synthesis algorithm is able to complete all benchmarks in under 30 seconds elapsed total time. Without compositional synthesis it is able to complete 26. Each benchmark specification includes source and target regular expressions, between one and three sufficient examples, and, if necessary, the timing parameter determined to be sufficient for correctness during a prior interactive session.

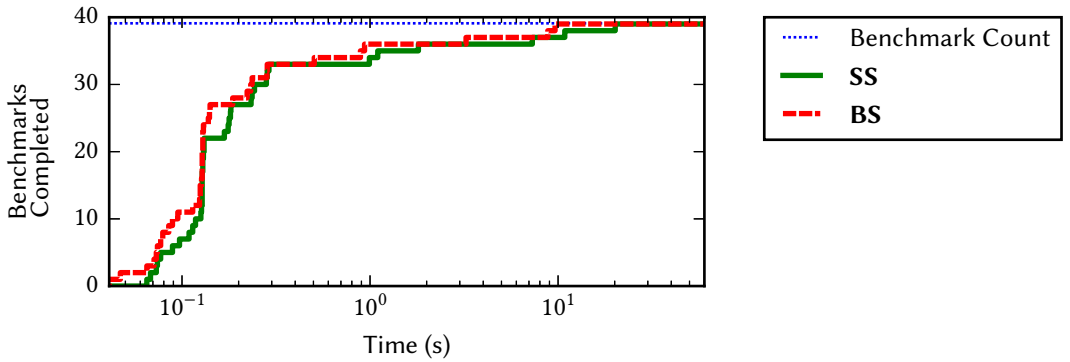

Fig. 7. Number of benchmarks that can be solved by a given algorithm in a given amount of time. SS is the full symmetric synthesis algorithm. BS is the full bijective lens synthesis algorithm.

(0.5 seconds) longer to complete than BS. The slowest completed benchmark for both synthesis algorithms is `xml_to_augeas.boom`, a benchmark that converts arbitrary XML up to depth 3 into a serialized version of the structured dictionary representation used in Augeas. This benchmark takes 18.9 seconds for the symmetric synthesis algorithm to complete, and 9.3 seconds the bijective synthesis algorithm to complete.

6.5 The Effects of Our Heuristics

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

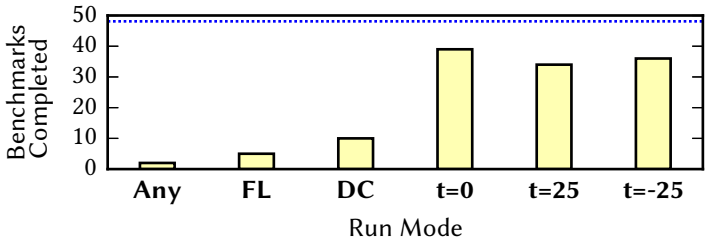

Fig. 8. Number of benchmarks that synthesize the correct lens by a given algorithm. **Any** provides no notion of cost, and merely returns the first lens it finds that satisfies the specification. **FL** provides a notion of cost to GreedySynth, but once a satisfying lens is greedily found, that lens is returned. **DC** synthesizes lenses, where the cost of a lens is the number of disconnects plus the number of merges. **t=0** runs the algorithm with the termination parameter universally set to 0 (default). **t=25** runs the algorithm with the termination parameter universally set to 25. **t=-25** runs the algorithm with the termination parameter universally set to -25.

Any: We ignore the information-theoretic preference metric entirely, returning the first lens GreedySynth finds without using this information.

FL: GreedySynth uses the information-theoretic preference metric to search, but returns immediately upon finding a satisfying lens, instead of returning based on Continue.

DC: We replace our information-theoretic cost metric with one where the cost of the lens is the number of disonnects plus the number of merges.

t=0: **t=0** runs the algorithm with the termination parameter universally set to 0 (default).

t=25: **t=25** runs the algorithm with the termination parameter universally set to 25.

t=-25: **t=-25** runs the algorithm with the termination parameter universally set to -25.

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

Figure 8 summarizes the result of these experiments. The data reveal that the information-theoretic metric is critical for finding the correct lens: Only 10 of the benchmarks succeeded when running in **DC** mode. The termination condition is also quite important. When running in **FL** mode, the algorithm only discovers 5 lenses, which shows that the first cluster that contains a satisfying lens is rarely the correct cluster. However, our termination condition is not perfect. Sometimes users must tell the algorithm to "keep going" or "stop early." Without termination parameters, our algorithm finds the correct lens for 39 of our 48 benchmarks; nine required tuning to find the correct lens. Nevertheless, our default parameter sits in a good middle-ground. Both **t=25** and **t=-25** synthesized the correct lens less often than **t=0**. **t=25** demands too good a lens, so the search procedure keeps spinning, while **t=-25** terminates early, not always trying hard enough to find the correct lens.

7 RELATED WORK

In this paper, we have designed and implemented a new, pragmatic formulation of symmetric lenses as well as new program synthesis techniques. In this section, we analyze the relationship between our simple symmetric lenses and two previous lens languages: classical symmetric lenses and asymmetric lenses. We also comment on related program synthesis techniques.

7.1 Relationship with Classical Lens Languages

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

In this formulation, data unique to each side are stored in the complement. When one format is edited, the putR or putL function stitches together the edited data with data stored in the complement. The *init* element is the initial value of C and specifies default behavior when data is missing. For instance, to implement the scenario in Figure 2, the complement would consist of a list of pairs of salary and company name. Classical symmetric lenses satisfy the following equational laws.

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

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

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

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

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

While classical symmetric lenses are technically more expressive than simple symmetric lenses, it is unclear whether this additional expressiveness is useful in practice. In our judgement, the complications they introduce into synthesis did not justify the potential additional expressiveness: because each lens has a custom complement, one can no longer specify the put functions through input/output examples alone. One alternative would be to enrich specifications with edit sequentials; another would be to specify the structure of complements explicitly (though the latter would be somewhat akin to specifying the internal state of a program). In either case, the complexity of the specifications increases.

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

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

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

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

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

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

$$\frac{\ell.putr(x, c_1) = (\_, c_1') \qquad \ell.putl(y, c_1') = (\_, c_1'')}{\ell.putr(x, c_2) = (\_, c_2') \qquad \ell.putl(y, c_2') = (\_, c_2'')} \qquad \text{(ForgetfulRL)}$$
$$c_1'' = c_2''$$

$$\frac{\ell.putl(y, c_1) = (\_, c_1') \qquad putr(x, c_1') = (\_, c_1'')}{\ell.putl(y, c_2) = (\_, c_2') \qquad \ell.putr(x, c_2') = (\_, c_2'')} \qquad \text{(ForgetfulLR)}$$
$$c_1'' = c_2''$$

Intuitively, these equations state that complements are uniquely determined by the most recently input x and y. Hence, such lenses correspond exactly with simple symmetric lenses, where all state is maintained by the current state of the x and y data. Forgetful symmetric lenses express exactly the same *apply* function as simple symmetric lenses.

Theorem 6. Let ℓ be a classical symmetric lens. The lens ℓ is equivalent to a forgetful lens if, and only if, there exists a simple symmetric lens ℓ' where $apply(\ell, \ell.init, es) = apply(\ell', None, es)$, for all put sequences es.

Asymmetric Lenses. Formally, an *asymmetric lens* $\ell : S \Leftrightarrow V$ is a triple of functions $\ell.\mathsf{get} : S \longrightarrow V$, $\ell.\mathsf{put} : V \longrightarrow S \longrightarrow S$ and $\ell.\mathsf{create} : V \longrightarrow S$ satisfying the following laws [Foster et al. 2007]:

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

Despite being less expressive than symmetric lenses, simple symmetric lenses are strictly more expressive than classical asymmetric lenses.

Theorem 7. Let ℓ be an asymmetric lens. ℓ is also a simple symmetric lens, where:

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

7.2 Data Transformation Synthesis

Over the past decade, the programming languages community has explored the synthesis of programs from a wide variety of angles. One of the key ideas is typically to narrow the program search space by focusing on a specific domain, and imposing constraints on syntax [Alur et al. 2013] or typing [Augustsson 2004; Feser et al. 2015; Frankle et al. 2015; Gvero et al. 2013; Osera and Zdancewic 2015; Scherer and Rèmy 2015], or both.

Automation of string transformations, in particular, has been the focus of much prior attention. For example, Gulwani and others work on FlashFill generates one-way spreadsheet transformations from input/output examples [Gulwani 2011a; Le and Gulwani 2014]. On the one hand, FlashFill is easier to use because one need not specify the type of the data being transformed. On the other hand, this type information makes it possible to transform more complex formats—Flash Fill does not synthesize programs with nested loops, for instance, and hence is incapable of synthesizing the majority of our benchmarks, even in one direction [Gulwani 2011a].

All pragmatic synthesis algorithms use heuristics of one kind or another. One of the goals of the current paper is to try to ground those heuristics in a broader theory, information theory, with the hope that this theory may inform future design decisions and help us understand heuristics

crafted in other tools and possibly in other domains. For instance, we speculate that some of the heuristics used in FlashFill (to take one well-documented example) may be connected to some of the principles laid out here. For instance, Flash Fill prioritizes the substring constructor over the constant string constructor when ranking possible programs. Such a choice is consistent with our information-theoretic viewpoint as the constant function throws away a great deal of information about the source string being transformed. Likewise, Flash Fill prefers "wider" character classes over "narrower" ones. Again such a choice is implied by information theory – the wider class preserves more information during translation. More broadly, we hope our information-theoretic analysis provides a basis for understanding the heuristic choices made in related work.

As discussed in the introduction, the Optician tool [Maina et al. 2018; Miltner et al. 2018] was a building block for our work. Optician synthesized bijective transformations [Miltner et al. 2018] and bijections modulo quotients [Maina et al. 2018], but could not synthesize more complex bidirectional transformations where one format contains important information not present in the other—a common situation in the real world. From a technical perspective, the first key novelty in our work involves the definition, theory, analysis, and implementation of a new class of simple symmetric lenses, designed for synthesis. The second key technical innovation involves the use of stochastic regular expressions and information theory to guide the search for program transformations. As mentioned earlier, we believe such information-theoretic techniques may have broad utility in helping us understand how to formulate a search for a data transformation function.

While our tool uses types and examples to specify invertable transformations, other tools have been shown to synthesize a backwards transformation from ordinary code designed to implement the forwards transformation. For instance, Hu and D'Antoni [2017] show how to invert transformations using symbolic finite automata, and Voigtländer [2009] demonstrates how to construct reverse transformations from forwards transformations by exploiting parametricity properties. These kinds of tools are very useful, but in different circumstances from ours (namely, when one already has the code to implement one direction of the transformation).

More broadly, information theory and probabilistic languages are common tools in natural language understanding, machine translation, information retrieval, data extraction and grammatical inference (see Pereira [2000], for an introduction). Indeed, our work was inspired, in part, by the PADS format inference tool [Fisher et al. 2008], which was in turn inspired by earlier work on probabilistic gramatical inference and information extraction [Arasu and Garcia-Molina 2003; Kushmerick 1997]. PADS did not synthesize data transformers, and we are not aware of the use of information-theoretic principles in type-directed or syntax-guided synthesis of DSL programs.

8 CONCLUSION

We have developed a synthesis algorithm for synthesizing synchronization functions between data formats that may not be in bijective correspondence. More specifically, we identify a subset of symmetric lenses, simple symmetric lenses, develop new combinators for them, and show how to synthesize them from regular expression specifications. In order to guide the search for "likely" lenses, we introduce new search principles based on information theory. Based on these principles, we designed, implemented, and evaluated a new tool for lens synthesis.

ACKNOWLEDGMENTS

REFERENCES

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

Arvind Arasu and Hector Garcia-Molina. 2003. Extracting Structured Data from Web Pages. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. 337–348.

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 *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/1863543.1863572

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.

Calendars 2016. Calendars. http://fileformats.archiveteam.org/wiki/Calendars. (2016).

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

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

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)*.

Finance and Accounting 2016. Finance and Accounting. http://fileformats.archiveteam.org/wiki/Finance_and_Accounting. (2016).

Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. 2008. From Dirt to Shovels: Fully Automatic Tool Generation From Ad Hoc Data.

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. *SIGPLAN Not.* 43, 9 (Sept. 2008), 383–396. https://doi.org/10.1145/1411203.1411257

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

Sumit Gulwani. 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)*.

Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. 2015. Symmetric Lenses. *J. ACM* (2015). To appear; extended abstract in POPL 2011.

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

Nicholas Kushmerick. 1997. *Wrapper Induction for Information Extraction*. Ph.D. Dissertation. Seattle, WA, USA.

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.

David Lutterkort. 2008. Augeas–A Configuration API. In *Linux Symposium, Ottawa, ON*. 47–56. http://augeas.net/index.html

Machine Embroidery 2015. Machine Embroidery. http://fileformats.archiveteam.org/wiki/Machine_Embroidery. (2015).

Solomon Maina, Anders Miltner, Kathleen Fisher, Benjamin Pierce, Dave Walker, and Steve Zdancewic. 2018. Synthesizing Quotient Lenses. To appear.

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing Bijective Lenses. In *Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2018)*.

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.

Fernando Pereira. 2000. Formal grammar and information theory: Together again? *Philosophical transactions of the royal society* 358 (2000), 1239–1253.

Brian J. Ross. 2000. Probabilistic Pattern Matching and the Evolution of Stochastic Regular Expressions. *Applied Intelligence* 13, 3 (Nov. 2000), 285–300. https://doi.org/10.1023/A:1026524328760

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)*.

Claude Elwood Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27, 3 (7 1948), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

Janis Voigtländer. 2009. Bidirectionalization for Free! (Pearl). In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/1480881.1480904