Synthesizing Symmetric Lenses

ANDERS MILTNER, Princeton University, USA
SOLOMON MAINA, University of Pennsylvania, 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 updates to either their source or their target data to be transferred in both directions. Since their introduction by Foster *et al.*, lenses have been extensively studied, extended, and applied. Recent work has also demonstrated how techniques from *type-directed program synthesis* can be used to efficiently synthesize a 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 extend this synthesis algorithm to a much broader class of lenses, called *simple symmetric lenses*, including all bijective lenses, all of the popular category of "asymmetric" lenses, and a rich subset of the more powerful "symmetric lenses" proposed by Hofmann *et al.* Intuitively, simple symmetric lenses allow some information to be present on one side but not the other and vice versa. They 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 than 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, generally preferring lenses that propagate more information, as well as user annotations marking parts of the source and target data structures as either *irrelevant* or *essential*.

We describe an implementation of simple symmetric lenses and our synthesis procedure as extensions to the Boomerang language. We evaluate its performance on 48 benchmark examples drawn from Flash Fill, Augeas, the bidirectional programming literature, and electronic file format synchronization tasks. Our implementation can synthesize each of these lenses in under 30 seconds.

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

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

ACM Reference Format:
Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2019. Synthesizing Symmetric Lenses. *Proc. ACM Program. Lang.* 3, ICFP, Article 95 (August 2019), 28 pages. https://doi.org/10.1145/3341699

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

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/8-ART95
https://doi.org/10.1145/3341699

1 INTRODUCTION

In today's data-rich world, similar information is often stored in different places and in different formats. For instance, electronic calendars come in iCalendar, vCalendar, and h-event formats [Calendars 2016]; U.S. taxation data can be transmitted via Tax XML (used by the U.S. government for e-filing) or TXF (used by TurboTax) [Finance and Accounting 2016]; configuration files can be stored in ad-hoc or structured file formats [Lutterkort 2007] (for example, scheduled commands can be stored in a crontab or as Windows's Scheduled Tasks xml).

One convenient way to maintain consistency between such 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. Domain-specific languages for writing lenses can facilitate generation of principled data transformations. For example, Boomerang [Bohannon et al. 2008] is a language for expressing bidirectional string transformations.

Synthesizing lenses can make the task even easier. Indeed, lens languages like Boomerang are excellent targets for search-based synthesis, since their specialized, sub-Turing-complete syntax and their expressive type systems drastically limit the space of possible programs. The recent Optician synthesizer for Boomerang [Miltner et al. 2018a] has demonstrated the feasibility of synthesizing quite complex examples, albeit only for a restricted class of lenses—so-called bijective lenses. Given source and target formats plus a small number of examples of corresponding data, Optician's synthesis algorithm is guaranteed to find a bijective lens matching the examples, if one exists, often requiring no examples at all. While specifying regular expressions can be cumbersome, doing so is much easier than programming lenses – regular expressions are widely understood whereas few programmers are fluent in lenses, and writing lenses requires users to reason about how to align two formats at once. One reason that bijective lens synthesis can be so effective, even relative to successful synthesis tools in other domains, is that there are typically not very many bijections between a given pair of data formats. If the synthesis algorithm finds any bijection, it is fairly likely to be the intended one.

However, the set of real-world use cases for bijective lenses, where two data formats contain different arrangements of precisely the same information, is limited. Oftentimes, two data formats share just *some* of their information content. For instance, one ad-hoc system-configuration file might include some format-specific metadata, such as a date or a reference number, while the same configuration file on another operating system does not. Indeed among the benchmark problems described in Section 6, all of the benchmarks taken from Flash Fill [Gulwani 2011] and many of the benchmarks that synchronize between two ad hoc file formats have this characteristic.

Can Optician's basic synthesis procedure be extended to a richer class of lenses? Could we even imagine synthesizing all *symmetric lenses* [Hofmann et al. 2011]—a much larger class that includes bijective lenses, "asymmetric" lenses (where the transformation from a source structure to a target can throw away information that must then be restored when transferring from target to source), and even more flexible transformations that allow each side to throw away information?

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

The first of these is pragmatic. Symmetric lenses as presented by Hofmann et al. [2011] operate over three structures: a "left" structure X, a "right" structure Y and a "complement" C that contains the information not present in either X or Y. These complements must be stored and managed

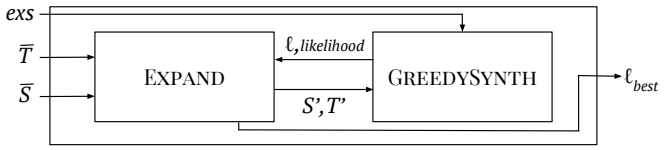

Fig. 1. Schematic diagram for the simple symmetric lens synthesis algorithm. The user provides regular expressions $\overline{S}$ and $\overline{T}$ and a set of examples *exs* as input. EXPAND first converts $\overline{S}$ and $\overline{T}$ to stochastic regular expressions S and T with default probabilities. 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.

somehow. More fundamentally, complements complicate synthesis *specifications*—instead of just giving single examples of source and target pairings, users would have to give longer "interaction sequences" to show how a lens should behave. To avoid these complexities, we define a restricted variant of symmetric lenses, called *simple symmetric lenses*. Intuitively, simple symmetric lenses are symmetric lenses that do not require external "memory" to recover data from past instances of X or Y when making a round trip. They only need the most recent instance.

Formally, we characterize the expressiveness of simple symmetric lenses by proving that they are exactly the symmetric lenses that satisfy an intuitive property called *forgetfulness*. We also show they are expressive enough for many practical uses by adding simple symmetric lenses to the Boomerang language [Bohannon et al. 2008] and applying them to a range of real-world applications. This exercise also demonstrates that simple symmetric lenses can coexist with, and be extended by, 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].

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

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

With simple symmetric lenses and this SRE-based likelihood measure in hand, we propose a new algorithm for synthesizing likely lenses. At its core, the algorithm performs a type-directed search between descriptions of the data formats, measuring success using the likelihood measure.

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 in general depend on which of the possible representations are chosen for its source and target formats. Moreover, certain lenses may not be well-typed unless the format representations are replaced by equivalent ones: in general, the synthesis algorithm has to search through equivalent regular expression types to find the most likely lens. To tame this complexity, we divide the synthesis algorithm into two communicating search procedures (Figure 1), following Miltner et al. [2018a]. The first, EXPAND, uses rewriting rules to propose new

pairs of stochastic regular expressions equivalent to the original pair. The second, GreedySynth, uses a greedy, type-directed algorithm to find a simple symmetric lens between input SRE pairs, returning the lens and its likelihood score to Expand. The whole synthesis algorithm heuristically terminates when a sufficiently likely lens is found.

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

We evaluate our implementation, the effects of optimizations, and our inclusion of relevance annotations on a set of 48 lens synthesis benchmarks drawn from data cleaning, view maintenance, and file synchronization tasks. We find the system can synthesize simple symmetric lenses for all of the benchmarks in under 30 seconds (§6).

In summary, our contributions are as follows:

- We define *simple symmetric lenses*, a subclass of symmetric lenses with no hidden state, we propose a syntax for them—a collection of combinators for building simple symmetric lenses—and we describe how to calculate their likelihoods (§2–§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) and prove that our metrics correspond to the information loss of a given lens.
- We extend the Boomerang implementation with simple symmetric lenses and lens synthesis specifications, including types, examples, and relevance annotations. We evaluate our implementation, the effects of optimizations, and our inclusion of relevance annotations on a set of 48 lens synthesis benchmarks drawn from data cleaning, view maintenance, and file synchronization tasks. The system can synthesize simple symmetric lenses for all of the benchmarks in under 30 seconds (§6).
- We prove that simple symmetric lenses fall between asymmetric lenses [Foster et al. 2007] and full symmetric lenses [Hofmann et al. 2011] 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* [Lehmann 1977] on regular expressions remain valid when extended 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 full version of the paper [Miltner et al. 2018b].

2 OVERVIEW

We begin with an overview of simple symmetric lenses and our synthesis algorithm, using a simplified example drawn from a hypothetical 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

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

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

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 (we will just say "lens" when clear from context) between sets X and Y comprises four functions subject to four round-tripping laws.

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

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

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

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

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

The round-tripping laws guarantee that pushing an "unedited" value from one side (the result of a put or create) back through a lens in the other direction will get back to exactly where we began. For example, consider a lens synchronizing the employee data formats in Figure 2. Applying the createR function to a salary file creates an insurance file with the same employee names in the file, with "UNK" for each insurance company. The round-tripping laws guarantees that applying the putL function to the generated insurance file and the original salary file will return the original salary file, unmodified.

Simple symmetric lenses differ from "classical" symmetric lenses in that they do not involve a *complement*. We give a detailed comparison in §7.1.

2.2 Simple Symmetric Lens Combinators

In principle, programmers can define lenses just by writing four arbitrary functions by hand and showing they satisfy the round-tripping laws, but this can be tedious and error prone. A better idea is to provide a set of combinators—a domain-specific language—for building complex, law-abiding

lenses from simpler ones. Such bidirectional languages have been given for many different kinds of data, including database relations [Bohannon et al. 2006] algebraic data types [Hofmann et al. 2011], graphs [Schürr 1995], and more. 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 $\mathcal{L}(\overline{T})$. (We will use undecorated variables later for stochastic regular expressions, so we mark plain REs with overbars.) We illustrate some of these combinators by defining lenses on subcomponents of the employee data formats.

The simplest combinator is the identity lens id, which takes as an argument a regular expression $\overline{S}$ and propagates data unchanged in both directions.

```
id(name) : name ⇔ name
```

The identity lens moves data back and forth from source to target without changing it. Both the createR and createL functions are the identity function (createR $s = s$), and the put functions merely return the first argument (putR $s\ t = s$).

In contrast, disconnect(S, T, s, t) does not propagate any data at all from one format to the other. The disconnect lens takes four arguments: two regular expressions (S, T) and two strings (s, t). The regular expressions specify the formats on the two sides, while the strings provide default values.

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

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

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

```
ins(t) = disconnect("", t, "", t)
del(s) = disconnect(s, "", s, "")
```

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

Finally, there are a number of combinators that construct more complex lenses from simpler ones. These include concatenation (concat(ℓ_1, ℓ_2), often written $\ell_1.\ell_2$), variation (or(ℓ_1, ℓ_2)), and iteration (iterate(ℓ)). For example, we could use the lens

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

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

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

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

transforms a list of employees in employees in the Management format to a list of employees in the HR format and vice versa. These combinators are combined in figure 3 to construct a complete lens between the employee formats.

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

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

Lenses are typed by pairs of regular expressions and only a handful of rules result in a given type. Many of these rules provide tight relationships between terms and types, for example, the type of a concat lens is the concatenation of the regular expressions of the sub-lenses.

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

The typing rules maintain structural similarity between regular expression types and lens terms, with one exception: The type equivalence rule permits a well-typed lens to be well-typed among any equivalent regular expression pair.

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

This rule is needed for lenses like the final one in Figure 3 to be well-typed.

2.3 Ranking Simple Symmetric Lenses

For larger and more complex formats, it can become quite difficult to write lenses by hand, as we did above. Instead, we would like to synthesize them from types and examples. That is, 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 all the input/output examples. We call this a *satisfying lens*. For our running example, suppose we wish to synthesize a lens between emp_salaries and emp_insurance, using as sole input-output example the data in Figure 2.

One 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, s, t)

is another well-typed lens that satisfies the example in Figure 2 (where s and t represent the provided example strings). In general, *many* examples may be required to rule out all possible occurrences of disconnect lenses, particularly in complex formats. Instead of merely finding *any* satisfying lens, we wish to synthesize a satisfying lens that is likely to please the user.

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

in t gets thrown away. Thus, if both id and disconnect are satisfying lenses for a particular pair of formats, id will be more likely.

The expected number of bits required to recover a piece of data corresponds to the well-known information-theoretic concept of *entropy* [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 $\mathbb{H}(S, P)$ is $-\Sigma_{s \in S} P(s) log_2 P(s)$. In information theory, the *information content* of each element of $s \in S$ is the number of bits required to specify s in a perfect encoding scheme ($-log_2 P(s)$). The entropy of S is the expected number of bits required to specify an element drawn from S—the probability of each element times its information content. Entropy captures the intuition that, if a data source contains many possible elements and none have significantly higher probability than others, it will have high entropy. Data sources with just a few high probability elements have low entropy. When P is clear from context, we will often use the shorthand of $\mathbb{H}(S)$ for $\mathbb{H}(S, P)$.

In the present setting, we already have a way of expressing sets of data: regular expressions. To calculate entropy, what we need is a way to express probability distributions over those sets. To that end, we adopt *stochastic regular expressions* [Carrasco et al. 1996; Ross 2000] (SREs), which are regular expressions in which each operator is annotated with a probability (see §3). A stochastic regular expression thus specifies both a set of strings at the same time as a probability distribution over that set.

We use entropy to gauge the relative likelihood of lenses in our synthesis algorithm (see §4.1). For any 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 $\sum_{s \in S} P_S(s) \cdot \mathbb{H}(\{t \mid \ell.\mathsf{putR}\, s\, t = t\})$. The likelihood we assign to ℓ is the sum of the conditional entropy of S given T and ℓ and the conditional entropy of T given S and ℓ. This metric assigns higher entropy (or lower likelihood) to lenses where knowing the string on one side provides little information about the string on the other side. It assigns zero entropy to bijections because given a string $s \in S$, the bijection exactly determines the corresponding string in T.

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

Sometimes users already know that certain data should or should not be used to construct the other format. We introduce relevance annotations to SREs that enable users to specify whether a piece of data should be used to construct the other format (with require) or not (with skip). In our example, salary and company could safely be skipped (as they are disconnected), where name in emp_salary and emp_ins could be annotated as required (as they are converted with the identity lens). Doing so both constrains the problem (as names must be synchronized) and makes it easier (as the algorithm does not waste time looking for salary information in the insurance format). In this way, users can tweak the lens likelihoods with their external knowledge.

2.4 Searching for Likely Lenses

Now that we have a way to calculate likelihood, we need a way to search for likely lenses. Previous work [Augustsson 2004; Feser et al. 2015; Frankle et al. 2015; Osera and Zdancewic 2015] has shown that types can be used to dramatically restrict the search space and improve the effectiveness of synthesis. In our setting, types are pairs of stochastic regular expressions, and the fact that each regular expression is semantically equivalent to infinitely many others complicates the problem.

Following existing work on bidirectional program synthesis [Maina et al. 2018], we split our algorithm into two communicating search procedures. The first, Expand, navigates the space of semantically equivalent regular expressions by applying rewrite rules that preserve both semantics

and probability distributions. This algorithm ranks pairs of stochastic regular expressions by the number of rewrite rule applications required to obtain each pair from the one given as input. It passes the pairs off to the second search procedure, GreedySynth, in rank order with the smallest first.

GreedySynth looks for highest-likelihood lenses between a given pair of stochastic regular expressions S and T by performing a type-directed search. It first converts the stochastic regular expressions provided by Expand into *stochastic DNF regular expressions*—a constrained representation of stochastic regular expressions with disjunctions distributed over concatenations and with concatenations and disjunctions normalized to operating over lists. Then it uses the syntax of these n-ary DNF regular expressions to find normalized lenses in a form we call *simple symmetric n-ary DNF lenses*. These involve neither a composition operator nor a type equivalence rule. These restrictions mean that there are comparatively few simple symmetric n-ary DNF lenses that are well typed between a given pair of stochastic n-ary DNF regular expressions, so GreedySynth's search space for a given pair of regular expressions is finite. Finally, GreedySynth yields a simple symmetric lens by converting the n-ary syntax back to the binary forms provided in the surface language.

This architecture of communicating synthesizers gives us a way to enumerate pairs of stochastic regular expressions of increasing rank and to efficiently search through them, but it poses a problem: when should Expand stop proposing new SRE pairs? We might have found a promising lens between a pair of stochastic regular expressions, but a different pair we haven't yet discovered may give rise to an even better lens. The search algorithm must resolve a tension between the quality of the inferred lens and the amount of time it takes to return a result. For example, if the algorithm has already found the lens in Figure 3, we don't want to spend a lot of time searching for an even better lens. To resolve this tension, the algorithm uses heuristics to judge whether to return the current best satisfying lens to the user or to pass the next set of equivalent SREs to GreedySynth. The heuristics favor stopping if the current best satisfying lens is very likely, indicating the lens is very promising (for example, if the satisfying lens loses no information, the algorithm should terminate for no other lens will be more likely). The heuristics also favor stopping if Expand has delivered to GreedySynth all pairs of stochastic regular expressions at a given rank and there is a large number of pairs at the next rank, because searching through all such pairs will take a long time (*i.e.*, if a satisfying lens loses only a little information and searching for a better one will take a long time, the discovered lens is returned). With this approach, the algorithm can quickly return a satisfying lens with relatively high likelihood. If the user is unhappy with the result, they can either refine the search by supplying additional examples, which serve both to rule out previously proposed lenses and to reduce the size of the search space by cutting down on the number of satisfying lenses, or they can supply annotations on the source and target SREs indicating that certain information either must be retained or must be discarded by the lens (see §5.3).

3 STOCHASTIC REGULAR EXPRESSIONS

To characterize likely lenses, we must compute the expected number of bits needed to recover a string in one data source from a synchronized string in the other data source. To do this, we first develop a probabilistic model for our language using *stochastic regular expressions* (SREs)—regular expressions annotated with probability information [Carrasco et al. 1996; Ross 2000]—that jointly express a language and a probability distribution over that language.

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

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

$$\begin{array}{rcll} \emptyset \cdot S & \equiv & \emptyset & \text{0-Proj}_L \\ \text{""} \cdot S & \equiv & S & \cdot \mathit{Ident}_L \\ S \mid_1 \emptyset & \equiv & S & \mid \mathit{Ident} \\ S \cdot (S' \mid_p S'') & \equiv & (S \cdot S') \mid_p (S \cdot S'') & \mathit{Dist}_R \\ S^{*p} & \equiv & \text{""} \mid_{1-p} (S \cdot S^{*p}) & \mathit{Unroll}_L \end{array} \qquad \begin{array}{rcll} S \cdot \emptyset & \equiv & \emptyset & \text{0-Proj}_R \\ S \cdot \text{""} & \equiv & S & \cdot \mathit{Ident}_R \\ S \mid_p T & \equiv & T \mid_{1-p} S & \mid \mathit{Comm} \\ (S' \mid_p S'') \cdot S & \equiv & (S' \cdot S) \mid_p (S'' \cdot S) & \mathit{Dist}_L \\ S^{*p} & \equiv & \text{""} \mid_{1-p} (S^{*p} \cdot S) & \mathit{Unroll}_R \end{array}$$

$$\begin{array}{rcll} (S \cdot S') \cdot S'' & \equiv & S \cdot (S' \cdot S'') & \cdot \mathit{Assoc} \\ (S \mid_{p_1} S') \mid_{p_2} S'' & \equiv & S \mid_{p_1 p_2} (S' \mid_{\frac{(1-p_1)p_2}{1-p_1 p_2}} S'') & \mid \mathit{Assoc} \end{array}$$

Fig. 4. Stochastic Regular Expression Star-Semiring Equivalence

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

One may think of SREs as string generators. Under this interpretation, the constant SRE s always generates the string s and never any other string. The SRE $S_1 \mid_p S_2$ generates a string from S_1 with probability p and generates a string from S_2 with probability $1-p$. The SRE S^{*p} generates strings in S^n with probability $p^n(1-p)$. For example, S^{*p} will generate a string in S with probability $p(1-p)$ and a string in $S \cdot S$ with probability $p^2(1-p)$.

3.1 Stochastic Regular Expression Equivalences

The Expand algorithm enumerates SREs that are "equivalent" to a given one. However, existing work does not define any notion of stochastic regular expression equivalence. Figure 4 shows how we extend the *star-semiring* [Lehmann 1977] equivalences (a *finer* notion of equivalence than semantic equivalence) to SREs.

Theorem 1. If $S \equiv T$, then $P_S = P_T$.

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

3.2 Stochastic Regular Expression Entropy

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

$$\begin{array}{rcl} \mathbb{H}(s) & = & 0 \\ \mathbb{H}(S^{*p}) & = & \frac{p}{1-p}(\mathbb{H}(S) - log_2 p) - log_2(1-p) \\ \mathbb{H}(S \cdot T) & = & \mathbb{H}(S) + \mathbb{H}(T) \\ \mathbb{H}(S \mid_p T) & = & p(\mathbb{H}(S) - log_2(p)) + (1-p)(\mathbb{H}(T) - log_2(1-p)) \end{array}$$

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

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

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

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

4 SIMPLE SYMMETRIC STRING LENSES

Simple symmetric string lenses are bidirectional programs that convert string data between formats. These lenses operate over regular expressions, not SREs. If S is a stochastic regular expression, $\overline{S}$ is the regular expression obtained by removing probability annotations.

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

We have already described a number of these combinators in §2. Briefly (full definitions are below), the 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 that operate on disjoint source formats and share the same target. The lens that gets used depends on what data is in the left-hand format, if the data 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 give typing rules and semantics for (some of) these combinators. We write $\ell : \overline{S} \Leftrightarrow \overline{T}$ to indicate that ℓ is a well typed lens between the language of $\overline{S}$ and the language of $\overline{T}$. We present just createR and putR in cases where createL and putL are symmetric; full details can be found in the full version of the paper. For simplicity of presentation, we use unambiguous regular expressions everywhere, even though the identity and disconnect lenses can actually be defined over ambiguous regular expressions; relaxing this restriction is not problematic.

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

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 argument 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.

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

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

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

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. This differs from the or lens of classical symmetric lenses, as those can retain such information in the complement (see §7).

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

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

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

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

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

The merge_right lens is interesting because it merges data where one piece of 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.

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

The merge_left lens is symmetric to merge_right.

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

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

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

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

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 combinator is particularly useful when chaining many compositions together, as it can be used to align the central types.

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

Type equivalence enables a lens of type $S \Leftrightarrow T$ to be used as a lens of type $S' \Leftrightarrow T'$ if S is 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 Likelihoods

Our likelihood metric is based on the expected amount of information required to recover a string in one data format from the other. We use the function $\mathbb{H}^{\rightarrow}(T \mid \ell, S)$ to calculate bounds on the expected amount of information required to recover a string in T from a string in S, synchronized by ℓ. Similarly, we use the function $\mathbb{H}^{\leftarrow}(S \mid \ell, T)$ to calculate bounds on the expected amount of information required to recover a string in S from a string in T, synchronized by ℓ. We write $a[b,c]$ to mean $[ab, ac]$, and $[a,b] + [c,d]$ to mean $[a+c, b+d]$.

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

$\mathbb{H}^{\leftarrow}(S \mid \ell, T)$ 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 and merge_right. 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 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 in practice; only 2 of the lenses synthesized in our benchmarks include merges.

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

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

5 SYNTHESIS

Algorithm 1 presents our synthesis algorithm at a high level of abstraction. This algorithm searches for likely lenses in priority order one "class" at a time using a GreedySynth subroutine. Each class is the set of lenses that can by typed by a pair of regular expressions, modulo a set of simple axioms

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:          ℓ ← GreedySynth(exs, S, T)
9:          if Cost(ℓ) < Cost(best) then
10:             best ← ℓ
11:         PQ.Push(pq, Expand(S, T))
12:     return best
```

such as associativity, commutativity, and distributivity. The Expand subroutine generates new classes using the star-unrolling axioms ($Unroll_L$ and $Unroll_R$).

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

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

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

5.1 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:

$$\begin{array}{lcl} S^{*_p} & \longrightarrow & \text{""} \mid_{1-p} (S \cdot S^{*_p}) \\ S^{*_p} & \longrightarrow & \text{""} \mid_{1-p} (S^{*_p} \cdot S) \end{array}$$

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

The procedure Continue terminates the loop based on the how long the search has been going, and how hard it expects the next class of problems to be. In particular, if PQ.Peek$(pq) = (S, T)$, 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) continues the loop while $c < d + log_2(n)$. This termination condition is based around two primary principles: do not search overly deep (d), and do not tackle a frontier that would take too long to process ($log_2(n)$). The log of the frontier is included because the frontier

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

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

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

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

Fig. 5. Stochastic DNF Regular Expression Functions

grows much faster than lens costs grow. Lens cost grows with the log of the expected number of choices in a given lens (due to its information theoretic basis), so the frontier calculation does too.

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

5.2 Stochastic DNF Regular Expressions

The GreedySynth subroutine converts input SREs into a temporary pseudo-normal form, *Stochastic DNF regular expressions* (SDNF REs). SDNF REs normalize across many of the star-semiring equivalences – if two regular expressions are equivalent modulo differences in associativity, commutativity, or distributivity, their corresponding SREs are syntactically equal.

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

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

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 ($\sum_{i=0}^{n} p_i = 1$).

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

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

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

The algorithm for converting a stochastic regular expressions S into its corresponding SDNF RE form, written $\Downarrow S$, is defined below. This conversion relies on operators defined in Figure 5.

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

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

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

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

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

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

ToStochastic. With stochastic DNF regular expressions and $\Downarrow$ defined, it is easier to explain ToStochastic, the function that converts regular expressions into stochastic regular expressions. If S is a stochastic regular expression generated by ToStochastic, then when put into SDNF RE form, $\Downarrow S = \langle(SQ_1, \frac{1}{n}) \mid \ldots \mid (SQ_n, \frac{1}{n})\rangle$ 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 SDNF REs, and gives a .8 chance for stars to continue iterating. In our experience generating random strings from regular expressions, these probabilities provide good distributions of strings—stars are iterated 4 times on average, and no individual choice in a series of disjunctions is chosen disproportionately often.

5.3 Relevance Annotations

Even though our generated probability distribution works well in most situations, it is not perfect. Consider synthesizing a lens between the formats shown in Figure 2. Because salary information is present in emp_salaries, but not in emp_insurance, the algorithm might spend a long time (fruitlessly) trying to construct lenses that transform the salary to information present in emp_insurance even though that is impossible. In a similar but more elaborate example, such wasted processing effort may cause synthesis to fail to terminate in any reasonable amount of time.

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

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

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

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

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

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

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

5.4 Symmetric DNF Lenses

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

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

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

The details of these definitions are syntactically tedious, but not intellectually difficult. We elide them here but include them in the full version of the paper. In the original Optician paper [Miltner et al. 2018a], DNF lenses were proven equivalent in expressiveness to standard lenses. While we conjecture symmetric DNF lenses are equivalent in expressivity to our standard symmetric lenses, we have not proven this equivalence.

5.5 GreedySynth

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

Symmetric DNF Lenses. Algorithm 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

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)
```

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

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

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

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

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

GreedySeqSynth loops until there are no more atom lenses in the priority queue. In the loop, a popped atom lens is considered "useful" if adding it to the sequence will lower the cost of the generated sequence lens, or if examples show that one of its atoms must not be disconnected. Each atom

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.Create(als)
11:     slb ← SLensBuilder.Empty
12:     while PQ.IsNonempty(pq) do
13:         sal ← PQ.Pop(pq)
14:         if SLensBuilder.UsefulAdd(slb, sal, exs) then
15:             pq ← SLensBuilder.AddAtom(pq, sal)
16:     return SLensBuilder.ToDNFLens(pq)
```

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.

5.6 Optimizations

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

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

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

Expansion Inference. These additional rewrites make the search through possible regular expressions harder. Our algorithm identifies when certain closed regular expressions can *only* be involved in a disconnect lens (unless opened). Such regular expressions will automatically be opened. The full details of expansion inference are explained in Miltner et al. [2018a].

Compositional Synthesis. Compositional synthesis allows GreedySynth to use previously defined (by users or synthesis) lenses. As the synthesizer processes a Boomerang file, it accumulates lens

definitions and 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. If a large synthesis problem cannot be solved all at once, a user can generate subproblems for parts of a data format, and the larger problem can subsequently use solutions to those subproblems. For example, when given the synthesis task:

```
let l_1 = synth R <=> S
let l_2 = synth R' <=> S'
```

the synthesis algorithm use the previously generated l_1 in the synthesis of l_2. In our experience, this is a very powerful tool that allows synthesis to tackle problems of arbitrary complexity.

6 EVALUATION

We implemented simple symmetric lenses as an extension to Boomerang [Bohannon et al. 2008]. In doing so, we reimplemented Boomerang's asymmetric lens combinators using a combination of the symmetric combinators presented in this paper and symmetric versions of asymmetric extensions (like matching lenses [Barbosa et al. 2010] and quotient lenses [Foster et al. 2008]) already present in Boomerang. We also integrated our synthesis engine into Boomerang, allowing users to write synthesis tasks alongside lens combinators, incorporate synthesis results into manually-written lenses, and reference previously defined lenses during synthesis. All experiments were performed on a 2.5 GHz Intel Core i7 processor with 16 GB of 1600 MHz DDR3 running macOS Mojave.

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

(1) Can the algorithm (with suitable examples and annotations) find the correct lens?
(2) Is the synthesis procedure efficient enough to be used in everyday development?
(3) How much slower is our tool on bijective lens synthesis benchmarks than prior work customized for bijective lenses [Miltner et al. 2018a]?
(4) How effective is the information-theoretic search heuristic and how do our annotations 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 [Gulwani 2011]. Flash Fill data cleaning tasks are either derived from online help forums or taken from the Excel product team. The 8 we chose are merely the first 8 found in the Flash Fill paper. Note that our tool produces bidirectional transformations rather than one-way transformers like Flash Fill. We ensure one direction of our bidirectional transformers performs the same unidirectional transformation as Flash Fill—the other direction is determined from the round-tripping laws. None of these benchmarks were bijective.
(2) We adapted 29 benchmarks from Augeas [Lutterkort 2007]. Augeas is a utility that bidirectionally converts between Linux configuration files and an in-memory dictionaries. In our benchmarks, we synthesize lenses between Linux configuration files and serialized versions of Augeas dictionaries. We selected the first 28 benchmarks from Augeas in alphabetical order, and included one additional benchmark we considered particularly difficult, `xml_to_augeas.boom`. Because these benchmarks merely transformed the information into a structured form, synthesized lenses were all bijective.
(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. Five of these benchmarks were not bijective lenses.

Each task consists of a source and target regular expression and a set of examples. We manually programmed the regular expressions. However, when writing these regular expressions, we did not massage them to be amenable to synthesis. The synthesizer implements the star-semiring regular expression equivalence axioms, so that users do not have to worry about the way they write expressions themselves. We are synthesizing transformations between real formats, so these tasks are can be quite large – on average, the size of each RE in the specification is 1637 and the size of the synthesized lens is 3254. For each of these tasks, we selected tasks until we did not feel additional examples would be elucidating.

6.2 Synthesizing Correct Lenses

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

Providing relevance annotations was needed in only 8 of the 48 tasks. In practice, we found that adding such annotations quite easy: if manual inspection of the lens showed there were too few ids, and too many disconnects or merges, we would add require annotations. If synthesis took too long, we would add skip annotations. Section 6.5 studies the effects of removing such annotations.

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

6.3 Effectiveness of Compositional Synthesis

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

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

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

For each benchmark in the suite and each mode, we ran the system with a timeout of 60 seconds, averaging the result over 5 runs. Figure 6 summarizes the results of these tests. We find that our algorithm is able to synthesize all of the benchmarks in under 30 seconds. Without compositional synthesis, the synthesis algorithm is able to solve 31 out of 48 problem instances. In total, 73 existing lenses were used in compositional synthesis (about 1.5 per benchmark on average).

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.[2]

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

[1] In one benchmark, we supplied a fourth example that was later discovered to be unnecessary.
[2] We had to slightly alter four of these benchmarks, either by providing additional examples or by adding in require annotations. Without these alterations, symmetric synthesis yielded a lens that fit the specification but that was undesired.

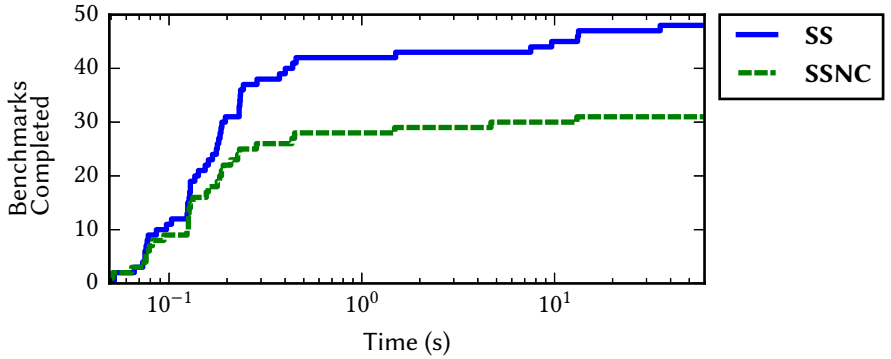

Fig. 6. Number of benchmarks that can be solved by a given algorithm in a given amount of time. SS is the full symmetric synthesis algorithm. SSNC is the symmetric synthesis algorithm without using a library of existing lenses. The symmetric synthesis algorithm is able to complete all benchmarks in under 30 seconds elapsed total time. Without compositional synthesis it is able to complete 31. Each benchmark specification includes source and target (potentially annotated) regular expressions, and between one and three sufficient examples.

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

For each benchmark, we ran it in both modes with a timeout of 60 seconds and averaged the result over 5 runs. Figure 7 summarizes the results of these tests. On average, SS took 1.3 times (0.5 seconds) longer to complete than BS. The slowest completed benchmark for both synthesis algorithms is `xml_to_augeas.boom`, a benchmark that converts arbitrary XML up to depth 3 into a serialized version of the structured dictionary representation used in Augeas. This benchmark takes 18.9 seconds for the symmetric synthesis algorithm to complete, and 9.3 seconds the bijective synthesis algorithm to complete.

Both the bijective synthesis and the symmetric synthesis engines use a pair of collaborating synthesizers that (1) search for a compatible pair of regular expressions and (2) search for a lens given those regular expressions. Bijective synthesis is faster than symmetric synthesis because part (2) is much faster. Specifically, a bijection must translate all data on the left into data on the right, and this fact constrains the search. By contrast, a symmetric synthesis problem has a choice of which data on the left to translate into data on the right. This choice gives rise to a large set of other choices, and symmetric synthesis must consider all of them.

6.5 The Effects of Heuristics and Relevance Annotations

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

Any: Ignore the information-theoretic preference metric (*i.e.*, all valid lenses have cost 0).
FL: Return the first highest ranked lens GreedySynth returns (*i.e.*, ignore the termination heuristic).
DC: Replace our information-theoretic cost metric with one where the cost of the lens is the number of disconnects plus the number of merges.
NS: Ignore all skip annotations in the SRE specifications.
NR: Ignore all require annotations in the SRE specifications.

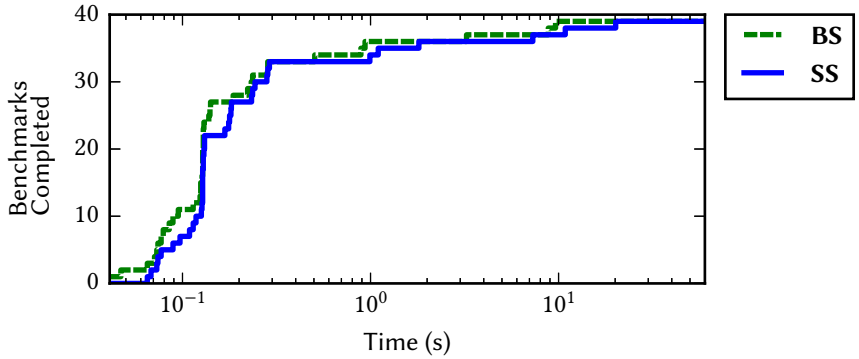

Fig. 7. Number of benchmarks that can be solved by a given algorithm in a given amount of time. **SS** is the full symmetric synthesis algorithm. **BS** is the full bijective lens synthesis algorithm.

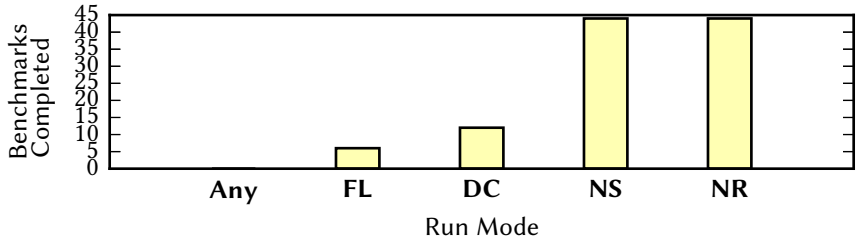

Fig. 8. Number of benchmarks that synthesize the correct lens by a given algorithm. **Any** provides no notion of cost, and merely returns the first lens it finds that satisfies the specification. **FL** provides a notion of cost to GreedySynth, but once a satisfying lens is greedily found, that lens is returned. **DC** synthesizes lenses, where the cost of a lens is the number of disconnects plus the number of merges. **NS** ignores all skip annotations while running the algorithm. **NR** ignores all require annotations while running the algorithm.

We experimented with the **DC** mode to determine whether the complexity of the information-theoretic measure is really needed. Related work on string transformations has often used simpler measures such as "avoid constants" that align with, but are simpler than our measures. 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 class that contains a satisfying lens is rarely the correct class. However, our algorithm is not perfect and fails when either it is very difficult to find the desired lens (necessitating require) or when a large amount of data is projected (necessitating skip). Without any annotations, our algorithm finds the correct lens for 40 of our 48 benchmarks; eight required relevance annotations to find the correct lens. In total, there are 12 uses of require, and 4 uses of skip in our benchmark suite.

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. Proofs about these relationships are included in the full version of the paper. We also comment on related program synthesis techniques.

7.1 Relationship with Classical Lens Languages

Symmetric Lenses. A classical symmetric lens[Hofmann et al. 2011] 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 ℓ between X and Y, an *edit* for ℓ is a member of $X + Y$. Consider the function *apply*, which, given a lens and an element of that lens's complement, is a function from sequences of edits to sequences of edits. If $apply(\ell, c, es) = es'$, then given complement c and edit es_i, the lens ℓ generates es'_i.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Theorem 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 $\ell.\text{get} : S \longrightarrow V$, $\ell.\text{put} : V \longrightarrow S \longrightarrow S$ and $\ell.\text{create} : V \longrightarrow S$ satisfying the following laws [Foster et al. 2007]:

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

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

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

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:

$$\begin{array}{rclcrcl}\ell.\mathsf{createL}\ y & = & \ell.create\ y & \qquad & \ell.\mathsf{createR}\ x & = & \ell.get\ x \\ \ell.\mathsf{putL}\ y\ x & = & \ell.put\ y\ x & & \ell.\mathsf{putR}\ x\ y & = & \ell.get\ x\end{array}$$

Other Lens Formulations. Bidirectional programming has an extensive literature, with many extensions onto basic lenses, like quotient lenses [Foster et al. 2008] and matching lenses [Barbosa et al. 2010]. Readers can consult Czarnecki et al. [2009] for a survey of lenses and lens-like structures.

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], 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's and others' work on FlashFill generates one-way spreadsheet transformations from input/output examples [Gulwani 2011; 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. For example, Flash Fill does not synthesize programs with nested loops [Gulwani 2011], and hence is incapable of synthesizing the majority of our benchmarks, even in one direction. A comparison of Optician (on bijective benchmarks) to these synthesis tools is included in Miltner et al. [2018a].

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. 2018a] was a building block for our work. Optician synthesized bijective transformations [Miltner et al. 2018a] 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 invertible 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 grammatical 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. More recently, the principles used in Fisher et al. [2008] have been applied in unsupervised learning [Ellis et al. 2015]. This work learns compact descriptions of a single data set. Ideally, such descriptions are compact and information theory is used as a measure of the compactness of the description learned. In contrast, we are attempting to learn a translation from one data set to another. However, there are many different candidate translations. To select amongst the candidate translations, we choose the translation that preserves as much information from source to target (and vice versa) as possible.

8 CONCLUSION

We described a synthesis algorithm for synthesizing synchronization functions between data formats that may not be in bijective correspondence. We identified 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 and allow users to control this search via relevance annotations. To evaluate the effectiveness of these ideas, we designed and implemented a new tool for symmetric lens synthesis and integrated it into the Boomerang lens programming framework. Our experiments on 48 benchmarks demonstrate that we can synthesize complex lenses for real-life formats in under 30 seconds.

ACKNOWLEDGMENTS

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

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.

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

Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2015. Unsupervised Learning by Program Synthesis. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'15)*. MIT Press, Cambridge, MA, USA, 973–981. http://dl.acm.org/citation.cfm?id=2969239.2969348

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. 2011. 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. 2011. Symmetric Lenses. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas.*

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.

Daniel Lehmann. 1977. Algebraic structures for transitive closure. *Theoretical Computer Science* 4 (02 1977), 59–76. https://doi.org/10.1016/0304-3975(77)90056-1

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

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. 2018a. Synthesizing Bijective Lenses. In *Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2018).*

Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018b. Synthesizing Symmetric Lenses. *CoRR* abs/1810.11527 (2018). arXiv:1810.11527 http://arxiv.org/abs/1810.11527 Extended technical report.

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).*

Andy Schürr. 1995. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–163.

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