AFRL-RI-RS-TR-2019-097

SYNTHESIZING DATA WRANGLERS

PRINCETON UNIVERSITY

APRIL 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ AIR FORCE MATERIEL COMMAND ■ UNITED STATES AIR FORCE ■ ROME, NY 13441

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-097 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN DRAGER
Work Unit Manager

/ S /
QING WU
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*: APRIL 2019
2. REPORT TYPE: FINAL TECHNICAL REPORT
3. DATES COVERED *(From - To)*: NOV 2016 – NOV 2018

4. TITLE AND SUBTITLE
SYNTHESIZING DATA WRANGLERS

5a. CONTRACT NUMBER: FA875-17-2-0028
5b. GRANT NUMBER: N/A
5c. PROGRAM ELEMENT NUMBER: 61101E
5d. PROJECT NUMBER: BRAS
5e. TASK NUMBER: PR
5f. WORK UNIT NUMBER: IN

6. AUTHOR(S)
Kathleen Fisher, Solomon Maina, Anders Miltner, Benjamin Pierce, David Walker, Steve Zdancewic

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Princeton University
35 Olden St
Princeton, NJ 08544

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

DARPA
675 North Randolph St
Arlington, VA 22203-2114

10. SPONSOR/MONITOR'S ACRONYM(S): AFRL/RI
11. SPONSOR/MONITOR'S REPORT NUMBER: AFRL-RI-RS-TR-2019-097

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Maintaining software systems that manage evolving representations of data is tedious and error prone. This study developed algorithms for automatically synthesizing lenses, which are software adaptors that convert between two different representations of the same information. These synthesis algorithms take as input (i) the type of each source as a regular expression, and (ii) a small collection of representative examples of the desired translation. They produce as output different varieties of string lenses. The products of this investigation include several published papers describing and evaluating the algorithms, as well as open-source code that implements those algorithms.

15. SUBJECT TERMS
Bi-directional programming, lenses, program synthesis, string transformations, regular expression types

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	19b. TELEPHONE NUMBER (*Include area code*)
U	U	U	UU	34	STEVEN DRAGER	N/A

Contents

List of Figures ii

1 Summary 1

2 Introduction 2

3 Methods, Assumptions and Procedures 4

3.1 Lenses 4

3.2 Bijective lens synthesis 6

3.3 Quotient lens synthesis 7

3.4 Simple symmetric lens synthesis 8

4 Results and Discussion 10

4.1 Bijective Synthesis Algorithms 10

4.2 Quotient Synthesis Algorithms 14

4.3 Simple Symmetric Synthesis Algorithms 16

5 Conclusions 22

6 References 23

7 List of Symbols, Abbreviations, and Acronyms 26

List of Figures

1 Hypothetical example data files and corresponding regular expressions used by management and HR at a US company to represent employee salaries and health insurance providers, respectively. 4

2 A lens that synchronizes management and HR employee files 6

3 Schematic Diagram for Optician. Regular expressions, S and T, and examples, *exs*, are given as input. First, the function $\Downarrow$ converts S and T into their respective DNF forms, DS and DT. Next, SynthDNFLens synthesizes a DNF lens, dl, from S, T, and *exs*. Finally, $\Uparrow$ converts dl into ℓ, a lens in Boomerang that is equivalent to dl. 7

4 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. 9

5 Sizes of Specifications. In (a), we show how many benchmarks are defined in our suite using a given number of AST nodes or fewer. In (b), we show how many benchmarks are defined in our suite using a given number of examples or fewer. 11

6 Average number of random examples required to synthesize benchmark programs. 12

7 Number of benchmarks that can be solved by a given algorithm in a given amount of time. Optician is our bijective synthesis algorithm including a set of optimizations we have implemented. **FlashExtract** is the existing FlashExtract system. **Flash Fill** is the existing Flash Fill system. **Naïve** is naïve type-directed synthesis on the bijective lens combinators. Our synthesis algorithm performs better than the naïve approach and other string transformation systems, and our optimizations speed up the algorithm enough that all tasks become solvable. 13

8 AST node measurements for each of the three approaches on each of the 10 non-bijective benchmark problems. Benchmarks are sorted in order of increasing complexity as measured by the number of AST nodes in the source and target format descriptions. QRE Synthesis requires far fewer AST nodes than the other two approaches. 15

9 Runtime measurements. In (a), we run Optician and QRE-enhanced Optician on the Optician benchmarks. We find that there is only a negligible performance overhead incurred by using QREs. In (b), we run QRE-enhanced Optician on the 10 Optician benchmarks previously edited to make them bijective, after removing those edits and then extending the synthesis specification to include QREs. (In other words, we restored them to their original state, added QREs, and then ran QRE-enhanced Optician). We find that QRE-enhanced Optician is able to synthesize all quotient lenses in under 10 seconds, and typically finishes in under 5 seconds. 17

10 Number of benchmarks that can be solved by a given algorithm in a given amount of time. **SS** is the full symmetric synthesis algorithm. **SSNC** is the symmetric synthesis algorithm without using a library of existing lenses. The symmetric synthesis algorithm is able to complete all benchmarks in under 30 seconds elapsed total time. Without compositional synthesis it is able to complete 26. Each benchmark specification includes source and target (potentially annotated) regular expressions, and between one and three sufficient examples. 19

11 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. 20

12 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. 21

1 Summary

Maintaining software systems that manage evolving representations of data is tedious and error prone. In this study, we have developed algorithms for automatically synthesizing *lenses*, which are software adaptors that convert between two different representations of the same information [Foster et al., 2007]. Lenses are a special form of programs that guarantee so-called *round-tripping properties* that ensure the transformations are well-behaved. Our synthesis algorithms take as input (i) the *type* of each source as a regular expression, and (ii) a small collection of representative *examples* of the desired translation. They produce as output different varieties of *string lenses*.

We have investigated three specific varieties of string lenses in detail: *bijective lenses*, *quotient lenses* and *simple symmetric lenses*. Bijective lenses address the simplest case, when the two formats we want to convert between have exactly the same information content, modulo rearrangement and reformatting. Quotient lenses apply when two formats are "morally" but not technically bijective. Examples include two formats that differ in their use of white space or in the order of sequences of elements. Simple symmetric lenses allow one side or the other to have information not present in the other; when converting between the formats, defaults take the place of missing information.

We measured the effectiveness of our algorithms on a set of 39 benchmarks drawn from Augeas [Lutterkort, 2007], which is a system for transforming and editing Linux configuration files, and from FlashFill [Gulwani, 2011a], which is an extension to Microsoft Excel for synthesizing columns of spreadsheets. On these benchmarks, we find it is possible to synthesize lenses of comparable quality to hand-written lenses from format descriptions and a very small number of examples, leveraging the large amount of information embedded in the format descriptions. We further found that as the formats diverge and the class of lenses between them becomes richer, the synthesis problem becomes more difficult and additional heuristics need to be considered.

The products of this investigation include several published papers [Miltner et al., 2018a, Maina et al., 2018, Miltner et al., 2018e] describing and evaluating our algorithms, as well as open-source code [Miltner et al., 2018d, Miltner et al., 2018c, Miltner et al., 2019] that implements those algorithms in (several variants of) a prototype tool called Optician. In this report, we summarize our key methods, results, and findings, focusing on the core algorithms for bijective synthesis, quotient synthesis, and simple symmetric synthesis.

Approved for Public Release; Distribution Unlimited.

2 Introduction

Building information-processing systems and maintaining them over a long period of time is a tedious, labor-intensive process. One key challenge is that such systems must often interact with a large number of *ad hoc data sources*—partially structured data sources represented in non-standard formats. Ad hoc data sources include various different kinds of system log files as well as scientific data sources generated by experiments. These ad hoc data sources are often produced by other automated systems, and each requires custom tools. Over time, the data sources tend to evolve—fields are added, removed, or co-opted, variants are added, etc. Today, to manage these changes, engineers must manually re-code parsers and/or insert adaptors while maintaining the desired semantics. Such manual work is not only time-consuming, but exceedingly error-prone. Moreover, errors in environment-facing interfaces can not only lead to corruption of important data, but also to significant security vulnerabilities. In order to design and implement survivable, long-lived, complex software systems that are robust to changes in their operational environment—the vision of the Defense Advanced Research Project Agency's (DARPA) Building Resource Adaptive Software Systems (BRASS) program—it is necessary to develop new, easier-to-use and more robust programming systems for managing these ad hoc data sources as they evolve.

To help alleviate this problem, we studied algorithms for automatically synthesizing adaptors between related data sources, given (i) the *type* of each source, as well as (ii) a small collection of representative *examples* of the desired translation. The adaptors we synthesize are *bidirectional*, meaning that we synthesize transformations that may be applied both backwards and forwards (from source A to B as well as B back to A). Such bidirectional transformations may help faciliate evolution and maintainence of long-lived systems by making it possible to upgrade one component of a (possibly distributed) system, while it continues to interact correctly with other components and with its environment. The bidirectional transformations will be guaranteed to preserve strong invertability laws, thereby reducing the likelihood of inadvertant data corruption. In addition, parsing components will be synthesized by a compiler rather than being manually coded, thereby reducing the likelihood of the buffer overruns that lead to many security vulnerabilities.

The core result of our research is a new algorithm for the synthesis of *bijective string lenses*. Bijective string lenses (*i.e.,* bidirectional transformations) define a limited set of transformations between strings. The domain and range of such transformations are determined by regular expressions (*i.e.,* regular expressions serve as the types of these transformations). In addition, as their name suggests, these transformations are *bijections*. In other words, the information content of source A is preserved (though usually rearranged) when data is translated to target B, and vice versa when B is translated back to A. Such transformations can rearrange fields of a record or insert new kinds of syntactic separators (*e.g.,* replacing a comma with a vertical bar, or the name of one Hypertext Markup Language (HTML) tag with another) but they cannot implement more general transformations that elide irrelevant details, such as the amount of whitespace that separates two tokens.

Despite their limitations, bijections and bijective synthesis form a useful foundation on top

Approved for Public Release; Distribution Unlimited.

of which more general transformations between data sets may be built. Hence, in the second half of our of project, we extended the regular expression-based specifications of data types with *isomorphisms* between strings, also known as *quotients*. To synthesize transformations between sets of strings modulo isomorphisms, we generate canonizing functions followed by bijections (using the original bijective synthesis algorithm). The so-called *quotient lenses* [Foster et al., 2008] we generate are capable of handling "irrelevant" differences between structures such as white space or permutations of items in a row of data set and hence expand the set of transformations our system can define significantly. Finally, we explored the synthesis of *symmetric lenses* [Hofmann et al., 2011a], again building upon the bijective platform that we started with. Symmetric lenses are able to ignore arbitrary chunks of data in A when generating target data B, and vice versa. For example, a serial number specific to one data set may be ignored when we translate to a second. Symmetric lenses further expand the set of allowed transformations.

We measured the effectiveness of our algorithms on a set of 39 benchmarks drawn from the Augeas system [Lutterkort, 2007], system for transforming and editing Linux configuration files, and FlashFill [Gulwani, 2011a], an extension to Microsoft Excel for synthesizing columns of spreadsheets. Past synthesis tools, such as FlashFill [Gulwani, 2011a] were unable to translate most Augeas file formats. However, our synthesis toolkit was able to generate transforms for all 39 of the Augeas file formats that we analyzed.

We have written papers [Miltner et al., 2018a, Maina et al., 2018, Miltner et al., 2018e] describing and evaluating our algorithms and produced open-source code [Miltner et al., 2018d, Miltner et al., 2018c, Miltner et al., 2019] that implements those algorithms in (several variants of) a tool called Optician. In this report, we summarize some of our key methods, results and findings, focusing on the core algorithms for bijective synthesis and quotient synthesis.

Approved for Public Release; Distribution Unlimited.

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

Figure 1: Hypothetical example data files and corresponding regular expressions used by management and HR at a US company to represent employee salaries and health insurance providers, respectively.

3 Methods, Assumptions and Procedures

Our approach to lens synthesis combines two strategies. First, we use a *domain-specific programming language*, Boomerang [Bohannon et al., 2008, Barbosa et al., 2010], which is designed for writing bidirectional string transformations. Second, we exploit *type- and example-directed program synthesis*, which uses enumerative techniques to search a space of candidate solutions. The search is constrained by the type information about the program, as well as by example instances provided by the user.

3.1 Lenses

Before surveying our methods and results on synthesis, we quickly sketch an example of what a Boomerang lens looks like, using a simplified situation drawn from a hypothetical U.S. company. In this company, management and human resources (HR) store information about employees in separate text files: management stores the names of employees and their salaries while HR stores the names of employees and their health insurance providers. Figure 1 gives examples of the two file formats and regular expressions describing them.

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

Simple symmetric lenses are appropriate for keeping these two files synchronized because they contain a mix of shared and unshared information.

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.

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

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

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.

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

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

In contrast, $\mathsf{disconnect}(S, T, s, t)$ 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.

$$\mathsf{disconnect}(\mathsf{salary}, \texttt{""}, \texttt{"unk"}, \texttt{""}) \ : \ \mathsf{salary} \Leftrightarrow \texttt{""}$$

On creates, the input values are thrown away, and default values are returned ($\mathsf{createL}\, t =$ "unk"), and on puts, the second argument is used and the first is thrown away ($\mathsf{putR}\, s\, t = t$). 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

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

Figure 2: A lens that synchronizes management and HR employee files

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 the Management format to a list of employees in the HR format and vice versa. These combinators are combined in Figure 2 to construct a complete lens between the employee formats.

3.2 Bijective lens synthesis

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

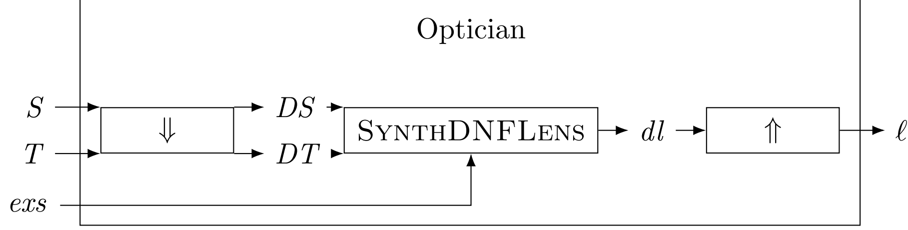


Figure 3: Schematic Diagram for Optician. Regular expressions, S and T, and examples, exs, are given as input. First, the function $\Downarrow$ converts S and T into their respective DNF forms, DS and DT. Next, SynthDNFLens synthesizes a DNF lens, dl, from S, T, and exs. Finally, $\Uparrow$ converts dl into ℓ, a lens in Boomerang that is equivalent to dl.

The goal of the synthesis algorithm is to find a lens $\ell : S \Leftrightarrow T$ that corresponds to a bijection from the language of a source regular expression S to a target language of a target regular expression T. Such a lens should satisfy the *bijective lens laws*:

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

Figure 3 shows a high-level, schematic diagram for Optician. First, Optician uses the function $\Downarrow$ to convert the input regular expressions into Disjunctive Normal Form (DNF) regular expressions. Next, SynthDNFLens performs type-directed synthesis on these DNF regular expressions and the input examples to synthesize a DNF lens. Finally, this DNF lens is converted back into a regular lens with the function $\Uparrow$, and returned to the user.

3.3 Quotient lens synthesis

Quotient lenses are lenses in which the lens laws are loosened so that they hold modulo an equivalence relation on the source and target data respectively; as above, we are concerned with *bijective quotient lenses* which are lenses for which Equation 1 holds modulo equivalence relations $\equiv_S$ and $\equiv_T$ defined on the source and target data respectively:

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

The inputs to the quotient lens synthesis problem, as in the bijective case, include source and target regular expressions S and T and a set of example instances. The equivalence relations $\equiv_S$ and $\equiv_T$ are also given as inputs to the synthesis algorithm. Definition of such equivalences are given via a new language of *quotient regular expressions* that we have defined. Quotient regular expressions describe both S (or T) and $\equiv_S$ (or $\equiv_T$) simulataneously in one compact notation.

Approved for Public Release; Distribution Unlimited.

3.4 Simple symmetric lens synthesis

Although it is very useful to be able to synthesize bijective or quotient lenses between data formats, such lenses do not fully solve the problem. Bijective and quotient lenses require the two data formats to have either precisely or morally the same information content. Many related data formats in practice have overlapping information: some information may be present in one format but not the other and vice versa.

Symmetric lenses [Hofmann et al., 2011b] address this limitation. General symmetric lenses introduce the notion of *complement* that stores the information necessary to fully reconstruct one format from the other. Because such complements would complicate the synthesis specificaiton problem, we instead define a more restricted language of *simple symmetric lenses*, which are the largest class of symmetric lenses that do not rely on persistent internal state to synchronise between two related data formats, a property called *forgetfulness*. (Instead of a complement, simple symmetric lenses rely on defaults to replace missing information.)

A challenge in adopting the type-directed synthesis approach to simple symmetric lenses is the number of such lenses. Whereas the number of bijective lenses between two 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. Symmetric synthesis requires 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 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. We also allow users to 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.

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.

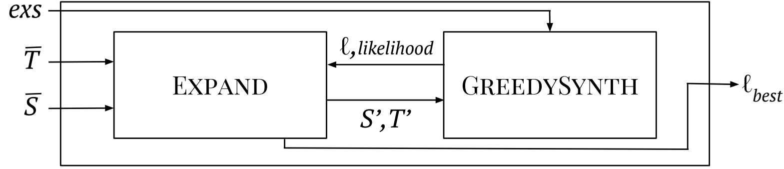


Figure 4: Schematic diagram for the simple symmetric lens synthesis algorithm. The user provides regular expressions $\overline{S}$ and $\overline{T}$ and a set of examples *exs* as input. Expand first converts $\overline{S}$ and $\overline{T}$ to stochastic regular expressions S and T with default probabilities. 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.

To tame this complexity, we divide the synthesis algorithm into two communicating search procedures (Figure 4), following [Miltner et al., 2018b]. 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.

4 Results and Discussion

We implemented our synthesis algorithms in the OCaml programming language where they are available as open source code [Miltner et al., 2018d, Miltner et al., 2018c, Miltner et al., 2019].

We analyzed our algorithms theoretically, proving various soundness and completeness results. We also analyzed these algorithms empirically, studying their impact on a range of real and synthetic benchmarks. Our results have been published in a series of conference papers and technical reports [Miltner et al., 2018a, Maina et al., 2018, Miltner et al., 2018e]. In the rest of this section, we summarize key results.

4.1 Bijective Synthesis Algorithms

We first show the effectiveness of the pure bijective synthesis algorithm by evaluating its performance on a set of 39 benchmark programs. All evaluations were performed on a 2.5 GHz Intel Core i7 processor with 16 GB of 1600 MHz DDR3 running macOS Sierra.

Benchmark Suite Construction We constructed our benchmarks primarily by adapting examples from Augeas [Lutterkort, 2007] and Flash Fill [Gulwani, 2011b]. Augeas is a configuration editing system for Linux that uses lens combinators similar to those in Boomerang. We derived 29 of the benchmark tests by adapting the first 27 lenses from the Augeas library in alphabetical order, as well as the lenses aug/xml − firstlevel and aug/xml that were referenced by the 'A' lenses. Furthermore, the 12 last synthesis problems derived from Augeas were tested after our synthesizer was finalized, demonstrating that the optimizations were not over-tuned to perform well on the testing data. Flash Fill is a system that allows users to specify common string transformations by example [Gulwani, 2011b]. We derived three benchmarks from the first few examples in this paper and one from the running example on extracting phone numbers.

Both Augeas and Flash Fill permit non-bijective transformations. To test our system on these benchmarks, we had to convert them into bijective transformations (by hand). Specifically, in many of the benchmarks, we normalized the whitespace in the documents so the amount of whitespace was equal in the source and target. In a few of the benchmarks, either the source or the target was missing information present in the other format. We modified those benchmarks by adding the missing information back into the file.

Finally, we added a few custom examples designed to highlight known weaknesses of our algorithm and to test situations for which we thought the tool would be particularly useful. These examples convert between work item formats, date formats, bibliography formats, and address formats, respectively.

Figure 5 shows the complexity of our type-based specifications as well as our example counts. An average benchmark has a type-based specification that can be represented using 310 Abstract Syntax Tree (AST) nodes, and requires 1.1 input/output examples. Our benchmarks vary from simple problems, like changing the representation of dates (with a specification size of 85, and a generated lens size of 79), to complex tasks, like transforming

Approved for Public Release; Distribution Unlimited.

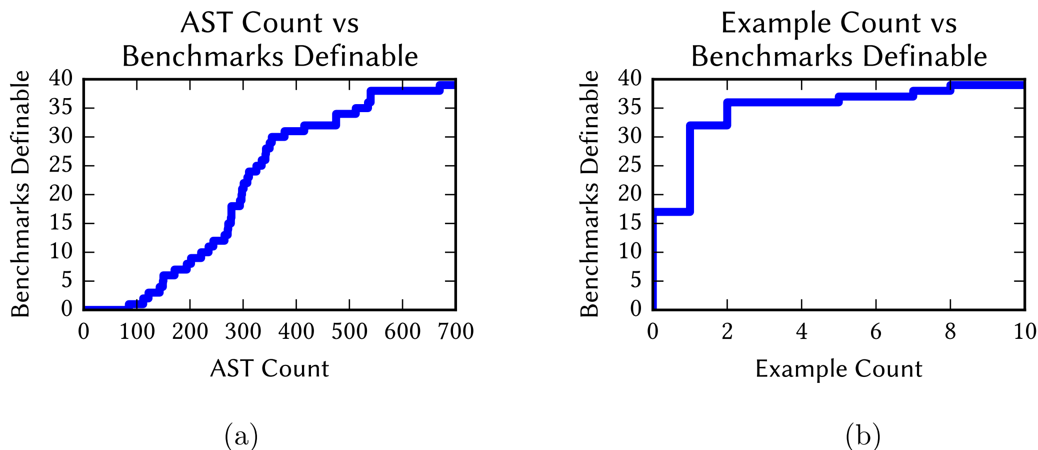


Figure 5: Sizes of Specifications. In (a), we show how many benchmarks are defined in our suite using a given number of AST nodes or fewer. In (b), we show how many benchmarks are defined in our suite using a given number of examples or fewer.

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

Importance of Examples To evaluate how many user-supplied examples the algorithm requires in practice, we *randomly* generated appropriate source/target pairs, mimicking what a naïve user might do. (We did not write the examples by hand out of concern that our knowledge of the synthesis algorithm might bias the selection.) Figure 6 shows the number of randomly generated examples it takes to synthesize the correct lens, averaged over ten runs. Strikingly, the synthesis algorithm almost never needs any examples at all: only 5 benchmarks need a nonzero number of examples to synthesize the correct lens and only one required more than 10 randomly generated examples. (A clever user may be able to reduce the number of examples further by selecting examples carefully; we were able to synthesize this last test case with only 8 examples.)

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

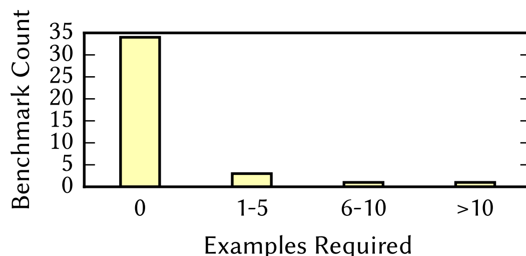


Figure 6: Average number of random examples required to synthesize benchmark programs.

fields can be omitted, a number of examples are needed to correctly identify the mapping between fields.

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

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

Comparison Against Other Tools Ours is the first tool to synthesize bidirectional transformations between data formats, so there is no tool to which we can make an apple-to-apples comparison. Instead, we compared against tools for generating unidirectional transformations. Figure 7 includes a comparison against two other well-known tools that synthesize text transformation and extraction functions from examples – Flash Fill and FlashExtract. For this evaluation, we used the version of these tools distributed through the PROSE project [PROSE, 2017].

To generate specifications for Flash Fill, we generated input/output specifications by generating random elements of the source language, and running the lens on those elements

Approved for Public Release; Distribution Unlimited.

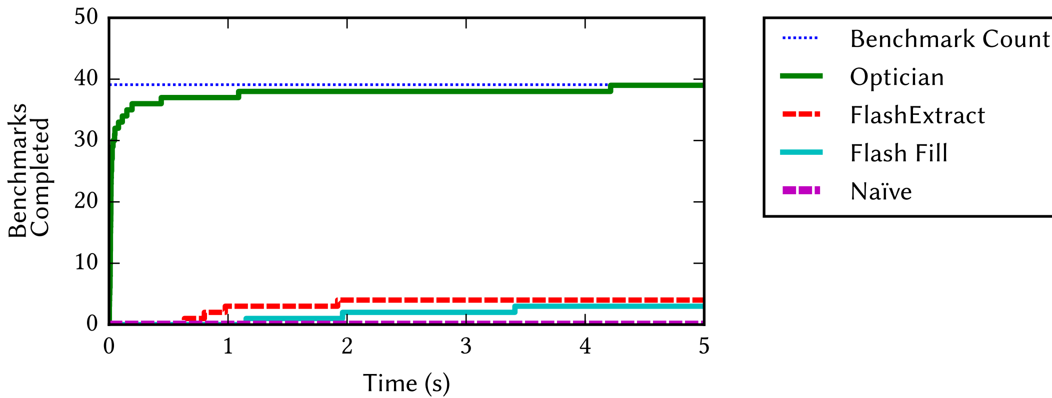


Figure 7: Number of benchmarks that can be solved by a given algorithm in a given amount of time. Optician is our bijective synthesis algorithm including a set of optimizations we have implemented. **FlashExtract** is the existing FlashExtract system. **Flash Fill** is the existing Flash Fill system. **Naïve** is naïve type-directed synthesis on the bijective lens combinators. Our synthesis algorithm performs better than the naïve approach and other string transformation systems, and our optimizations speed up the algorithm enough that all tasks become solvable.

Approved for Public Release; Distribution Unlimited.

to generate elements of the target language. These were then fed to Flash Fill.

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

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

Flash Fill and FlashExtract have two tasks: to determine how the data is transformed, they must also infer the structure of the data, a difficult job for complex formats. In particular, neither Flash Fill nor FlashExtract was able to synthesize transformations or extractions where the regular expressions describing the formats involved two nested iteration (Kleene-star) operators—a type of format that is notoriously hard to infer. However, this kind of nested iteration is pervasive in Linux configuration file formats, making Flash Fill and FlashExtract ill suited for many of the synthesis tasks in our test suite.

Conversely, since they deal with only unidirectional transformations, Flash Fill and FlashExtract can offer a more expressive set of operators. To guarantee bidirectionality, our syntax must be highly restrictive, providing a smaller search space to traverse.

4.2 Quotient Synthesis Algorithms

Our next thread of work was extending the bijective synthesis tool to synthesize quotient lenses. In this subsection, we will use "QRE-enhanced Optician" to denote our extended version of Optician, and just plain "Optician" to denote the pre-quotient version of Optician. The synthesis algorithm produces Boomerang lens values, so Boomerang gives synthesized lenses the same first-class status as hand-written ones. All evaluations were performed on a 2.5 GHz Intel Core i7 processor with 16 GB of 1600 MHz DDR3 running macOS High Sierra.

Benchmark Suite Construction We analyzed the same 39 lens synthesis tasks as we did for the original Optician system. We also experimented using our tool to synthesize quotient lenses between XML, Resource Description Framework (RDF) and Javascript Object Notation (JSON) formats using data from the data.gov database; the data consisted of census statistics, demographic statistics, wage comparison data, and crime index data). Recall that, to use bijective synthesis, we had to modify some of the benchmarks (10 of 39) to make them bijective—for instance, by normalizing whitespace. Because quotient lenses are more expressive, we were able avoid such modifications. This experience alone points to the benefits that quotient lens synthesis brings to the table.

Evaluating Programmer Effort To evaluate the impact of QRE lens synthsis on programmer effort, we focus our attention on the 10 problems in the benchmark suite that are

Approved for Public Release; Distribution Unlimited.

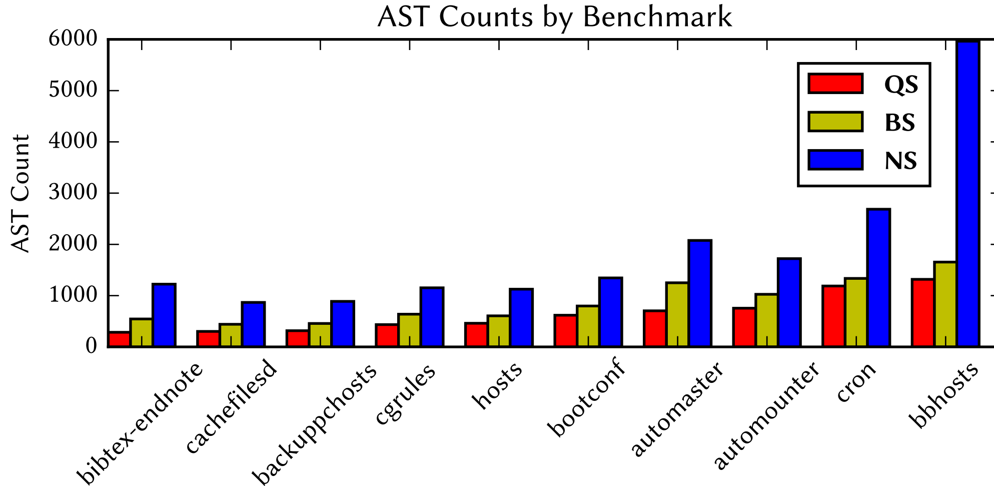


Figure 8: AST node measurements for each of the three approaches on each of the 10 non-bijective benchmark problems. Benchmarks are sorted in order of increasing complexity as measured by the number of AST nodes in the source and target format descriptions. QRE Synthesis requires far fewer AST nodes than the other two approaches.

not bijective and hence require non-trivial canonizers. (Optician already handles the other problems with minimal programmer effort.)

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

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

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

BS: The sum of (1) the number of AST nodes in $W(q)$ for each QRE q in the source and target formats, (2) the number of AST nodes in canonize(q) for each QRE q requiring

Approved for Public Release; Distribution Unlimited.

a non-trivial canonizer ("canonizers" as transformations that project a string to its equivalence class representative; they are written as programs in Boomerang), and (3) the number of AST nodes in the examples. We use (1) to estimate the cost of describing the external source and target formats and (2) to estimate the cost of writing the requisite canonizers by hand. We count the nodes in the examples because they would be fed to the bijective synthesizer. These counts are an approximation, as both $W(q)$ and canonize(q) are automatically generated from the corresponding QRE q, and it is possible that a human-written version might be smaller.

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

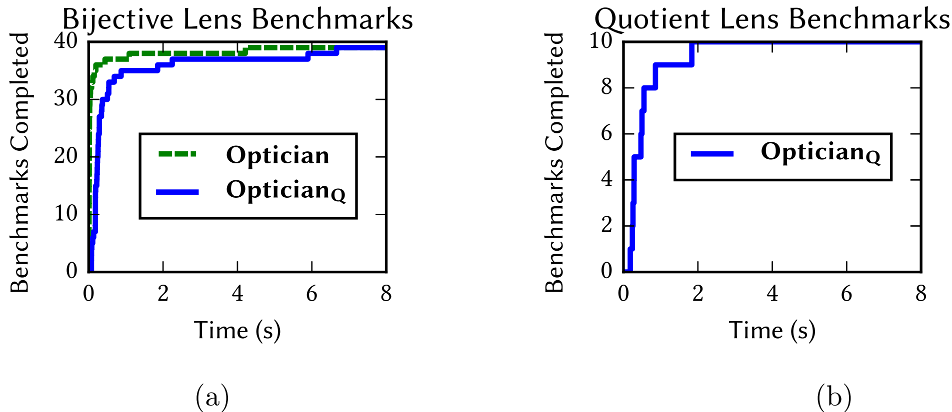
Figure 8 shows each of these measures for the 10 non-bijective problems in the benchmark suite. On average (using a geometric mean), **BS** used 38.5% more AST nodes than **QS**, requiring an average of 214 more AST nodes. On average, **NS** used 180% more AST nodes than **QS**, requiring an average of 998 more AST nodes. These figures suggest that introducing QREs saves programmers significant effort compared to both Optician and basic Boomerang.

Quotient Lens Performance To assess the performance of quotient synthesis, we are interested in two different questions. First, how does the performance of QRE-enhanced Optician compare to the performance of Optician on benchmarks that do not require quotients? The answer to this question tells us how much overhead we have introduced by adopting the more general mechanism. Figure 9(a) shows that QRE-enhanced Optician was able to synthesize all of the Optician benchmarks at a speed competitive with the old version. There is a small amount of additional overhead introduced by QREs in calculating equivalences, resulting in a slight decrease in performance.

Second, how much time does it take for QRE-enhanced Optician to synthesize a quotient lens when running on a non-bijective benchmark problem? Figure 9(b) shows the amount of time required to infer a lens for each of the 10 benchmark programs with nontrivial quotients. We find that QRE-enhanced Optician is able to synthesize all quotient lenses in under 10 seconds, and typically finishes in under 5 seconds.

4.3 Simple Symmetric Synthesis Algorithms

In addition to the results above, we implemented simple symmetric lenses as an extension to Boomerang [Bohannon et al., 2008]. This allows 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.



(a) (b)

Figure 9: Runtime measurements. In (a), we run Optician and QRE-enhanced Optician on the Optician benchmarks. We find that there is only a negligible performance overhead incurred by using QREs. In (b), we run QRE-enhanced Optician on the 10 Optician benchmarks previously edited to make them bijective, after removing those edits and then extending the synthesis specification to include QREs. (In other words, we restored them to their original state, added QREs, and then ran QRE-enhanced Optician). We find that QRE-enhanced Optician is able to synthesize all quotient lenses in under 10 seconds, and typically finishes in under 5 seconds.

Benchmark Suite Construction Our benchmarks for this experiment are drawn from three different sources.

1. We adapted 8 data cleaning benchmarks from Flash Fill [Gulwani, 2011a]. Flash Fill data cleaning tasks are either derived from online help forums or taken from the Excel product team. 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]. Because these benchmarks merely transform 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 Representational State Transfer (REST) and JSON web resource descriptions to synchronizing BibTeX and EndNote citation descriptions. Five of these benchmarks were not bijective lenses.

Synthesizing Correct Simple Symmetric Lenses To determine whether the system can synthesize desired lenses, we ran it interactively on all 48 benchmark tasks, working with the system to create sufficient examples and provide useful relevance annotations (*i.e.*, *essential* and *irrelevant* markings, written as require and skip, respectively). 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. Providing relevance annotations was needed in only 8 of the 48 tasks. In practice, we found that adding such annotations was quite easy: if manual inspection of the lens showed there were too few ids, and too many disconnects[1] or merges, we would add require annotations. If synthesis took too long, we would add skip annotations. We verified that the default running mode of our synthesis tool (**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**.

Effectiveness of Optimizations in Simple Symmetric 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.

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 10 summarizes the results of these tests. We found that our algorithm is able to synthesize all of the benchmarks in under 30 seconds. Without compositional synthesis, the synthesis algorithm is able to solve 26 out of 48 problem instances.

Slowdown Compared to Bijective Synthesis To compare to the existing bijective synthesis algorithm, we ran 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]Recall that disconnect is a lens combinator that marks a place where information appears in one format but not the other: there is a *disconnect* between the two formats.

[2]We had to slightly alter four of these benchmarks, either by providing additional examples or by adding require annotations. Without these alterations, symmetric synthesis yielded a lens that fit the specification but was not bijective and hence was different from the one generated by Optician.

Approved for Public Release; Distribution Unlimited.

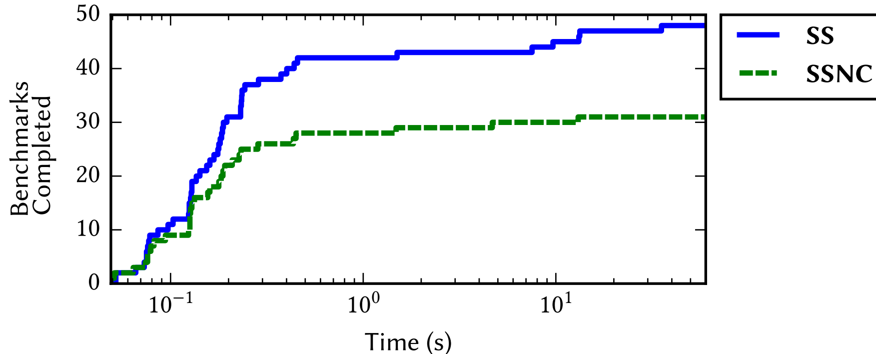


Figure 10: Number of benchmarks that can be solved by a given algorithm in a given amount of time. **SS** is the full symmetric synthesis algorithm. **SSNC** is the symmetric synthesis algorithm without using a library of existing lenses. The symmetric synthesis algorithm is able to complete all benchmarks in under 30 seconds elapsed total time. Without compositional synthesis it is able to complete 26. Each benchmark specification includes source and target (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 11 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 for the bijective synthesis algorithm to complete.

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 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.

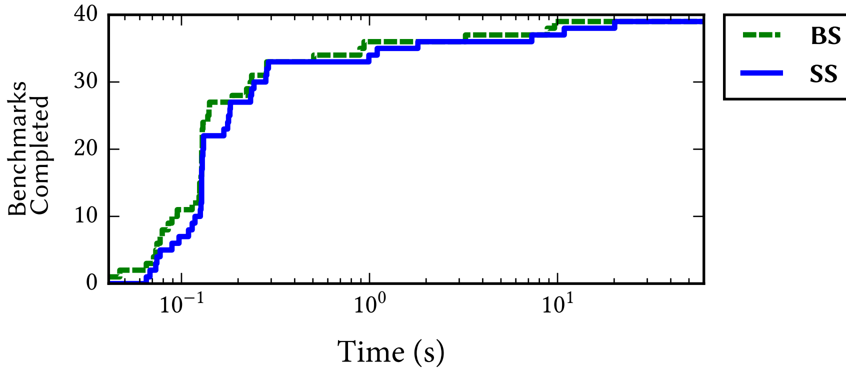


Figure 11: Number of benchmarks that can be solved by a given algorithm in a given amount of time. SS is the full symmetric synthesis algorithm. BS is the full bijective lens synthesis algorithm.

We experimented with the DC mode to determine whether the complexity of the information-theoretic measure was really needed. Related work on string transformations has often used simpler measures such as "avoid constants" 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 12 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; the other eight required relevance annotations to find the correct lens.

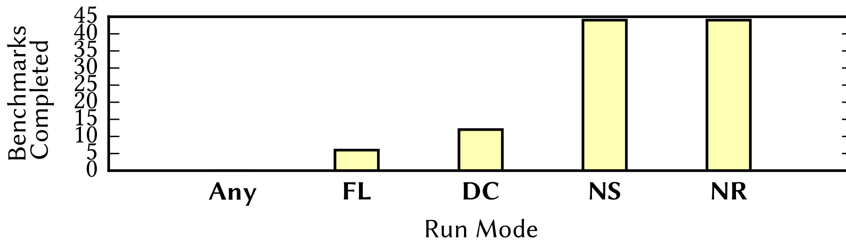


Figure 12: Number of benchmarks that synthesize the correct lens by a given algorithm. **Any** provides no notion of cost, and merely returns the first lens it finds that satisfies the specification. **FL** provides a notion of cost to GreedySynth, but once a satisfying lens is greedily found, that lens 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.

Approved for Public Release; Distribution Unlimited.

5 Conclusions

In this project, we designed, analyzed and implemented algorithms that synthesize three classes of bidirectional transformations: (1) pure bijective transformations, (2) bijections modulo equivalences classes (quotient lenses) and (3) bijections modulo projections (simple symmetric lenses). Our synthesis algorithms take as inputs a format specification (which includes specification of equivalence classes) and a collection of examples. As the class of transformations becomes richer, the number of potential programs grows dramatically. As a result, the naïve inference algorithm slows and its ability to guess the transformation desired by the user decreases, leading to reduced accuracy. We overcame these challenges by applying heuristics that use information theory to help guide the search for program transformations. In addition, to scale our prototypes to larger transformation tasks, we found it necessary to work compositionally, by breaking down complex synthesis tasks into smaller, more mangeable subtasks. The resulting system, embodied by (variants of) our prototype Optician implementation, out perform existing state-of-the-art tools for synthesis tasks in this domain.

Approved for Public Release; Distribution Unlimited.

6 References

[Barbosa et al., 2010] Barbosa, D. M. J., Cretin, J., Foster, N., Greenberg, M., and Pierce, B. C. (2010). Matching lenses: Alignment and view update. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland.*

[Bohannon et al., 2008] Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A., and Schmitt, A. (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.

[Bohannon et al., 2006] Bohannon, A., Vaughan, J. A., and Pierce, B. C. (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.

[Carrasco et al., 1996] Carrasco, R. C., Forcada, M. L., and Santamaría, L. (1996). Inferring stochastic regular grammars with recurrent neural networks. In Miclet, L. and de la Higuera, C., editors, *Grammatical Interference: Learning Syntax from Sentences*, pages 274–281, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Foster et al., 2007] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2007). Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17.

[Foster et al., 2008] Foster, J. N., Pilkiewicz, A., and Pierce, B. C. (2008). Quotient lenses. *SIGPLAN Not.*, 43(9):383–396.

[Gulwani, 2011a] Gulwani, S. (2011a). Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46. ACM.

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

[Hidaka et al., 2010] Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., and Nakano, K. (2010). Bidirectionalizing graph transformations. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 205–216.

[Hidaka et al., 2011] Hidaka, S., Hu, Z., Inaba, K., Kato, H., and Nakano, K. (2011). Groundtram: An integrated framework for developing well-behaved bidirectional model transformations. In *Automated Software Engineering (ASE).*

Approved for Public Release; Distribution Unlimited.

[Hofmann et al., 2011a] Hofmann, M., Pierce, B. C., and Wagner, D. (2011a). Symmetric lenses. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas*.

[Hofmann et al., 2011b] Hofmann, M., Pierce, B. C., and Wagner, D. (2011b). Symmetric lenses. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Austin, Texas*.

[Ko et al., 2016] Ko, H., Zan, T., and Hu, Z. (2016). BiGUL: A formally verified core language for putback-based bidirectional programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 61–72.

[Liu et al., 2007] Liu, D., Hu, Z., and Takeichi, M. (2007). Bidirectional interpretation of XQuery. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*, pages 21–30.

[Lutterkort, 2007] Lutterkort, D. (2007). Augeas: A Linux configuration API. Available from `http://augeas.net/`.

[Macedo et al., 2014] Macedo, N., Pacheco, H., Sousa, N. R., and Cunha, A. (2014). Bidirectional spreadsheet formulas. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*, pages 161–168.

[Maina et al., 2018] Maina, S., Miltner, A., Fisher, K., Pierce, B., Walker, D., and Zdancewic, S. (2018). Synthesizing quotient lenses. *Proc. ACM Program. Lang.*, 2(ICFP):80:1–80:29.

[Miltner et al., 2018a] Miltner, A., Fisher, K., Pierce, B. C., Walker, D., and Zdancewic, S. (2018a). Synthesizing bijective lenses. In *Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2018.

[Miltner et al., 2018b] Miltner, A., Fisher, K., Pierce, B. C., Walker, D., and Zdancewic, S. (2018b). Synthesizing bijective lenses. In *Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2018.

[Miltner et al., 2018c] Miltner, A., Maina, S., Fisher, K., Pierce, B. C., Walker, D., and Zdancewic, S. (2018c). boomerang/lenssynth at eval-pt2 . solomonaduol-maina/boomerang. `https://github.com/SolomonAduolMaina/boomerang/tree/eval-pt2/lenssynth`.

[Miltner et al., 2018d] Miltner, A., Maina, S., Fisher, K., Pierce, B. C., Walker, D., and Zdancewic, S. (2018d). boomerang/optician at master . boomerang-lang/boomerang. `https://github.com/boomerang-lang/boomerang/tree/master/optician`.

[Miltner et al., 2018e] Miltner, A., Maina, S., Fisher, K., Pierce, B. C., Walker, D., and Zdancewic, S. (2018e). Synthesizing symmetric lenses. Technical Report arXiv:1810.11527, arXiv. Available at https://arxiv.org/abs/1810.11527.

[Miltner et al., 2019] Miltner, A., Maina, S., Fisher, K., Pierce, B. C., Walker, D., and Zdancewic, S. (2019). boomerang/optician at test_symmetric . miltnoid/. https://github.com/Miltnoid/boomerang/tree/test_symmetric/optician.

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

[PROSE, 2017] PROSE, M. (2017). Microsoft Program Synthesis using Examples SDK.

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

[Zan et al., 2016] Zan, T., Liu, L., Ko, H., and Hu, Z. (2016). Brul: A putback-based bidirectional transformation library for updatable views. In *Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016.*, pages 77–89.

[Zhu et al., 2015] Zhu, Z., Ko, H., Martins, P., Saraiva, J., and Hu, Z. (2015). Biyacc: Roll your parser and reflective printer into one. In *Proceedings of the 4th International Workshop on Bidirectional Transformations co-located with Software Technologies: Applications and Foundations, STAF 2015, L'Aquila, Italy, July 24, 2015.*, pages 43–50.

Approved for Public Release; Distribution Unlimited.

7 List of Symbols, Abbreviations, and Acronyms

$\Downarrow$	An operation that converts regular expressions into DNF form
$\Uparrow$	An operation that converts a DNF lens to a lens
$\equiv_S$, $\equiv_T$	Equivalence relations on data matching regular expressions S and T respectively
Any	Mode for simple symmetric synthesis that ignores the information-theoretic metric
AST	Abstract syntax tree
BRASS	Building Resource Adaptive Software Systems
BS	Bijective lens synthesis algorithm
DARPA	Defense Advanced Research Project Agency
DC	Mode for simple symmetric synthesis that replace our information-theoretic cost metric with one where the cost of the lens is the number of disconnects plus the number of merges.
dl	Lens in disjunctive normal form
DNF	Disjunctive normal form
DS, DT	Regular expression in disjunctive normal form
exs	Examples
EXPAND	A component of the synthesis algorithm that proposes candidate types equivalent to the input
FL	Mode for simple symmetric synthesis that returns the first highest ranked lens found by GREEDYSYNTH
GREEDYSYNTH	Synthesis procedure that searches greedily for candidate solutions
HR	Human Resources
HTML	Hypertext Markup Language
JSON	JavaScript Object Notation—a standard file format used to encode data objects
ℓ	A lens
$\mathcal{L}(\overline{S})$	The language of strings for the regular expression $\overline{S}$
NR	Mode for simple symmetric synthesis that ignores all require annotations in the SRE specifications.
NS	Mode for simple symmetric syntehsis that ignores all skip annotations in the SRE specifications.
Optician	Our prototype implementation of the lens synthesis algorithms
Optician_Q	Optician enhanced with support for QRE synthesis
QRE	Quotient regular expression
QS	Number of nodes in the abstract syntax of QRE specifications
RDF	Resource Description Framework—used to model web resources
RE	Regular Expression
REST	Representational State Transfer
SRE	Stochastic regular expression

Approved for Public Release; Distribution Unlimited.

$\overline{S}$ and $\overline{T}$	Regular expressions provided by the user
S, T	Regular expressions
$S \Leftrightarrow T$	Type of a lens that maps between data described by regular expressions S and T
SS	Full symmetric synthesis algorithm
SSNC	Symmetric synthesis algorithm without using a library of existing lenses.
SynthDNFLens	A type-directed lens synthesis algorithm
XML	Extensible Markup Language—used in web services

Approved for Public Release; Distribution Unlimited.