Confluence in Lens Synthesis

Anders Miltner[1], Kathleen Fisher[2], Benjamin C. Pierce[3], David Walker[4], and Steve Zdancewic[5]

[1] Princeton University
amiltner@cs.princeton.edu
[2] Tufts University
kfisher@eecs.tufts.edu
[3] University of Pennsylvania
bcpierce@cis.upenn.edu
[4] Princeton University
dpw@cs.princeton.edu
[5] University of Pennsylvania
stevez@cis.upenn.edu

Abstract

A lens is a program that can be executed both forwards and backwards, from input to output and from output back to input again. Domain-specific languages for defining lenses have been developed to help users synchronize text files, and construct different "views" of databases, among other applications. Recent research has shown how string lenses can be synthesized from their types, which are pairs of regular expressions. However, guaranteeing that we can synthesize all possible lenses is quite tricky on these languages, due in large part to the many equivalences on regular expressions.

The proof that all string lenses are synthesizeable involves proving a confluence-like property, parameterized by a an additional binary relation R. We call this property R-confluence. In this model, standard confluence is the specific case where R is equality. In this paper, we show how existing techniques for demonstrating confluence do not work in the domain of R-confluence, and find that if the rewrite system is $=$-confluent and satisfies a commutativity property with R, then the system is R-confluent.

1 Introduction

Bidirectional transformations are pervasive in modern software systems, occuring as database views and view updaters, parsers and pretty-printers, data synchronization tools, and more. Instead of manually building the functions that comprise a bidirectional transformation, programmers can build them both "at once" using a bidirectional programming language. Bidirectional programming languages have been developed for creating view updaters [3], Linux configuration file editors [1], direct manipulation programming systems [11], and more [7, 5, 18]. Lenses are a particularly well-behaved class of bidirectional programs, where the underlying transformations are guaranteed to satisfy a number of "round-tripping" laws. Lens-based bidirectional programming languages often provide round-tripping guarantees through a set of typing rules; well-typed lens expressions are guaranteed to satisfy the round-tripping laws.

Optician [13], an extension of Boomerang, makes bidirectional programming easier by supporting synthesis of bidirectional string transformations. More specifically, it takes as input two regular expressions (R and S, which serve as the type of a Boomerang lens) and a set of examples specifying input-output behavior, and synthesizes a well-typed lens between the languages of those regular expressions. For brevity, we will not provide formal definitions for some aspects of Optician; the interested reader can find such definitions in the original Optician

paper [13]. Furthermore, examples detailing the use of synthesized lenses in practice (which we also elide for space) can be found in that paper and follow-up work [8, 14].

To indicate a lens l is well typed, and converts between the languages of R and S, we write $l : R \Leftrightarrow S$. For synthesis, given R and S, we must find a lens l such that $l : R \Leftrightarrow S$. In the context of lens synthesis, there are three sorts of lens typing rules: syntax-directed rules, composition, and type equivalence.

For syntax-directed rules, the syntax for the types closely mirrors the syntax of the expressions. As an example, consider the following rule for disjunction:

$$\textsc{Or Lens}\quad \frac{l_1 : R_1 \Leftrightarrow S_1 \qquad l_2 : R_2 \Leftrightarrow S_2 \qquad \mathcal{L}(R_1) \cap \mathcal{L}(R_2) = \emptyset \qquad \mathcal{L}(S_1) \cap \mathcal{L}(S_2) = \emptyset}{or(l_1, l_2) : R_1 \mid R_2 \Leftrightarrow S_1 \mid S_2}$$

Consider finding a lens of type $R_1 \mid R_2 \Leftrightarrow S_1 \mid S_2$. With only syntax-directed rules, the only lens that can be well-typed would be an or lens.

Composition sequentially composes two lenses.

$$\textsc{Composition}\quad \frac{l_1 : R_1 \Leftrightarrow R_2 \qquad l_2 : R_2 \Leftrightarrow R_3}{l_1 \; ; \; l_2 : R_1 \Leftrightarrow R_3}$$

Composition is difficult in the context of lens synthesis. If trying to synthesize a composition lens, one has to pull the central regular expression "out of thin air."

The last typing rule is type-equivalence. If two regular expressions are star-semiring equivalent to the type of a lens, those equivalent regular expressions also serve as the type of the lens.

$$\textsc{Type Equivalence}\quad \frac{l : R \Leftrightarrow S \qquad R \equiv^s R' \qquad S \equiv^s S'}{l : R' \Leftrightarrow S'}$$

This rule is difficult in the context of synthesis, as it forces a search through equivalent regular expressions. This rule can also be applied at any point in the derivation, which makes the search even harder.

To address the difficulties with composition and type-equivalence rules, we synthesize lenses in an alternative language of disjunctive normal form (DNF) lenses. DNF lenses are in pseudonormal form, containing no composition operator, and so their synthesis never needs to pull regular expressions "out of thin air." The types of DNF lenses are pairs of regular expressions in a pseudonormal form, DNF regular expressions. Because DNF regular expressions are in a pseudonormal form, fewer equivalent regular expressions need to be searched through.

In our search algorithm, we only search through equivalent regular expressions once, before processing any syntax directed-rules. We formalize this in the typing of DNF regular expressions by only permitting the application of type-equivalence once, after all syntactic rules have been applied. This is enforced by having two typing judgements: one for the "rewriteless" type of the lens (meaning no type-equivalence rules were applied) and one of the "full" type of the lens. If $dl \; \tilde{:} \; DR \Leftrightarrow DS$, then dl is a DNF lens of rewriteless type $DR \Leftrightarrow DS$. If $dl : DR \Leftrightarrow DS$, then dl is a DNF lens of full type $DR \Leftrightarrow DS$. The following rule is used to get the full type of a DNF lens from the rewriteless type.

$$\textsc{Rewrite DNF Regex Lens}\quad \frac{DR' \rightarrow^* DR \qquad DS' \rightarrow^* DS \qquad dl \; \tilde{:} \; DR \Leftrightarrow DS}{dl : DR' \Leftrightarrow DS'}$$

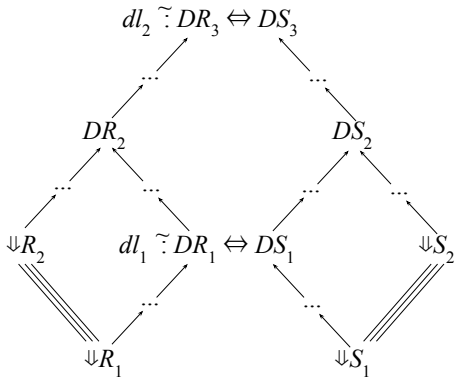

Figure 1: Diagram constructing a well-typed DNF lens between $\Downarrow R_2$ and $\Downarrow S_2$. Single lined arrows indicate rewrites ($\rightarrow$), and triple lines indicate equivalence ($\equiv_\rightarrow$).

Note that instead of using directionless equivalences, Rewrite DNF Regex Lens uses directioned rewrites. The relationship between the two is formalized by the following theorem:

Theorem 1. If $R \equiv^s S$, then $\Downarrow R \equiv_\rightarrow \Downarrow S$, where $\Downarrow R$ and $\Downarrow S$ are R and S in DNF form (respectively), and $\equiv_\rightarrow$ is the reflexive, transitive, and symmetric closure of $\rightarrow$.

Proving that our search procedure can generate any lens reduces to proving DNF lenses complete with respect to our standard lens language. In particular, we wish the prove:

Theorem 2. If $l : R \Leftrightarrow S$, then there exists a DNF lens, dl, such that $dl : \Downarrow R \Leftrightarrow \Downarrow S$ and the semantics of l and dl are equivalent.

We prove this propety by induction on the structure of the typing derivation. Particular difficulty lies in the lens equivalence rule. We begin this case below:

$$\frac{l : R_1 \Leftrightarrow S_1 \qquad R_1 \equiv^s R_2 \qquad S_1 \equiv^s S_2}{l : R_2 \Leftrightarrow S_2}$$

By induction assumption, there exists $dl : \Downarrow R_1 \Leftrightarrow \Downarrow S_1$, where the semantics of dl are equivalent to those of l. By inversion on the derivation of $dl : \Downarrow R_1 \Leftrightarrow \Downarrow S_1$, there exists DR_1 and DS_1 such that:

$$\frac{\Downarrow R_1 \rightarrow^* DR_1 \qquad \Downarrow S_1 \rightarrow^* DS_1 \qquad dl \mathbin{\tilde{?}} DR_1 \Leftrightarrow DS_1}{dl : \Downarrow R_1 \Leftrightarrow \Downarrow S_1}$$

To complete this case, we need to find a DNF lens $dl' : \Downarrow R_2 \Leftrightarrow \Downarrow S_2$ with equivalent semantics to l.

We first show that there exist DR_2 and DS_2 such that $\Downarrow R_2 \rightarrow^* DR_2$ and $\Downarrow DR_1 \rightarrow^* DR_2$ and $\Downarrow S_2 \rightarrow^* DS_2$ and $\Downarrow DS_1 \rightarrow^* DS_2$. To do this, we first prove that $\rightarrow$ is confluent. By Theorem 1, we know that $\Downarrow R_1 \equiv_\rightarrow \Downarrow R_2$, so confluence implies the existance of such a DR_2 and DS_2.

After this, we have $dl \mathbin{\tilde{?}} DR_1 \Leftrightarrow DS_1$ and $DR_1 \rightarrow^* DR_2$ and $DS_1 \rightarrow^* DS_2$. If we can prove a confluence-like property that would show the existance of some $dl' \mathbin{\tilde{?}} DR_3 \Leftrightarrow DS_3$, where $DR_2 \rightarrow^* DR_3$ and $DS_2 \rightarrow^* DS_3$, we would be done. This property is R-confluence for a properly chosen R (which we describe in §2). This case is diagrammed in Figure 1.

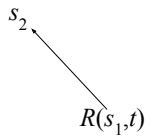

Figure 2: Rewrite system that satisfies the R-diamond property, but is not R-confluent.

2 R-Confluence Formulation

Let S be an underlying set, and $\rightarrow$ and R be binary relations. The rewrite system $(S, \rightarrow)$ is R-confluent, if for all $s_1, s_2 \in S$, if $R(s_1, s_2)$, $s_1 \rightarrow^* s_1'$, and $s_2 \rightarrow^* s_2'$, then there exist s_1'' and s_2'' such that $R(s_1'', s_2'')$, $s_1' \rightarrow^* s_1''$, and $s_2 \rightarrow^* s_2''$.

In the context of lens synthesis, S is the set of DNF REs, the rewrites are $\rightarrow^*$, and given a DNF lens dl, $R_{dl}(DR, DS)$ is true if there exists a DNF lens dl' such that $dl' \ \tilde{:} \ DR \Leftrightarrow DS$ and dl is equivalent to dl'.

Now that R-confluence has been formally defined, we can ask ourselves: "What is a good approach to proving R-confluence?" One approach is to prove that our rewrite system is locally confluent, which is equivalent to $=$-confluence in a terminating system [6]. Unfortunately, our rewrites are not terminating, so this approach does not work.

An approach pioneered by Tait and Martin-Löf [2] still works in non-terminating systems. This approach uses the *diamond property*: A rewrite system $(S, \rightarrow)$ satisfies the diamond property if $s_1 \rightarrow s_2$ and $s_1 \rightarrow s_3$ implies that there exists s_4 such that $s_2 \rightarrow s_4$, and $s_2 \rightarrow s_4$. If a rewrite system satisfies the diamond property, then it is also confluent. Unfortunately, this approach does not work, as the parameterized version of the diamond property does not imply R-confluence.

3 Proving $(S, \rightarrow^*)$ R-Confluent

In the Tait and Martin-Löf approach to proving $(S, \rightarrow)$ confluent, one must first prove $(S, \rightarrow)$ satisfies the diamond property. Consider a parameterized property analogous to the diamond property, the *R-diamond property*: A rewrite system $(S, \rightarrow)$ satisfies the R-diamond property if $s_1 \rightarrow s_2$ and $t_1 \rightarrow t_2$ and $R(s_1, t_1)$ implies that there exists s_3, t_3 such that $s_2 \rightarrow s_3$ and $t_2 \rightarrow t_3$ and $R(s_3, t_3)$.

However, satisfying the R-diamond property is not sufficient for R-confluence. Consider the simple rewrite system shown in Figure 2. In this rewrite system, there are 3 elements, s_1, s_2, and t. In this setup, $R(s_1, t)$ and $s_1 \rightarrow s_2$. R-confluence requires some s_3, t' such that $s_2 \rightarrow^* s_3$ and $t \rightarrow^* t'$ and $R(s_3, t')$, but no such values exist.

To get around this issue, we require a different set of properties.

1. $(S, \rightarrow)$ must be $=$-confluent.

2. R must be a bisimilulation relation for $(S, \rightarrow^*)$. In other words if $R(s_1, t_1)$, and $s_1 \rightarrow^* s_2$, then there exists t_2 such that $t_1 \rightarrow^* t_2$ and $R(s_2, t_2)$; and if $R(s_1, t_1)$, and $t_1 \rightarrow^* t_2$, then there exists s_2 such that $s_1 \rightarrow^* s_2$ and $R(s_2, t_2)$.

Theorem 3. Let $(S, \rightarrow)$ be $=$-confluent, and R be a bisimilulation relation for $(S, \rightarrow^*)$. If $R(s_1, t_1)$ and $s_1 \rightarrow^* s_2$ and $t_1 \rightarrow^* t_2$, then there exists s_3, t_3 such that $s_2 \rightarrow^* s_3$ and $t_2 \rightarrow t_3$ and $R(s_3, t_3)$.

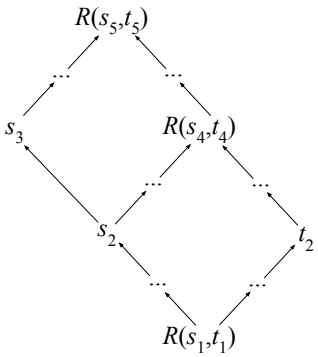

Figure 3: Diagram showing how we prove the inductive case of R-confluence of the transitive closure of a rewrite system. $R(s_4, t_4)$ comes from the inductive hypothesis. The existance of s_5 is guaranteed through $=$-confluence. $R(s_5, t_5)$ is guaranteed because R is a bisimulation relation.

Proof. By induction length of the reduction of $s_1 \to^* s_2$

Case 1 (Base Case). Let $R(s_1, t_1)$ and $t_1 \to^* t_2$. As R is a bisimulation relation for $(S, \to^*)$, there exists s_2 such that $s_1 \to^* s_2$ and $R(s_2, t_2)$, as desired.

Case 2 (Inductive Case). Let $R(s_1, t_1)$ and $s_1 \to^* s_2$ and $s_2 \to s_3$ and $t_1 \to^* t_2$. By the induction hypothesis, there exists s_4, t_4 such that $R(s_4, t_4)$ and $s_2 \to^* s_4$ and $t_2 \to t_4$.

Because $(S, \to)$ is $=$-confluent, there exists s_5 such that $s_3 \to^* s_5$ and $s_4 \to^* s_5$. As $(S, \to^*)$ is a bisimilulation relation on $(S, \to^*)$, there exists t_5 such that $t_4 \to^* t_5$ and $R(s_5, t_5)$, as desired. This case is diagrammed in Figure 3.

□

4 Related Work

The concept of R-confluence is related to the notion of *confluence modulo* $\sim$ [6]. The definition of confluent modulo $\sim$ is almost the same as $\sim$-confluence, the only difference is that confluence modulo $\sim$ requires $\sim$ to an equivalence relation. Conditions that suffice to prove a rewrite system confluent modulo $\sim$ are not generally sufficient to prove R-confluence (and vice versa). Furthermore, our bisimulation relations are closely related to local coherence modulo $\sim$.

Bisimulation relations come from concurrency theory [15], but a related notion, commuting rewrites [16], appear in the confluence literature. We require a single rewrite of R to commute with an arbitrary number of rewrites of $\to$, which commuting rewrites do not express.

The full proof of completeness is contained in the appendix of the full version of the original optician paper [12]. The original proof of R-confluence for the transitive closure of $\to$ required additional assumptions. These unnecessary assumptions have been identified and removed in this paper. Future work used the proof of completeness over our lens language to show that quotient bijective lenses are also synthesizeable [8]. Lastly, while synthesizeability was not proven for symmetric lenses [14], such a proof would likely have proven R-confluence in a similar manner.

This work continues a trend in making programming easier through synthesis [4]. While synthesis is one approach to make bidirectional programming easier, it is not the only approach. Work has gone into building lenses without requiring a point-free combinator style [10]. Other work has found applicative [9] and monadic [17] approaches to compositionally building lenses.

References

[1] Augeas - A configuration API. http://augeas.net/index.html.

[2] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984. http://www.cs.ru.nl/ henk/Personal Webpage.

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

[4] Sumit Gulwani, Alex Polozov, and Rishabh Singh. *Program Synthesis*, volume 4. NOW, August 2017.

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

[6] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, October 1980.

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

[8] Solomon Maina, Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing quotient lenses. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018.

[9] Kazutaka Matsuda and Meng Wang. Applicative bidirectional programming with lenses. *SIGPLAN Not.*, 50(9):62–74, August 2015.

[10] Kazutaka Matsuda and Meng Wang. Hobit: Programming lenses without using lens combinators. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 31–59, Cham, 2018. Springer International Publishing.

[11] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. Bidirectional evaluation with direct manipulation. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[12] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses, 2017. https://arxiv.org/abs/1710.03248.

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

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

[15] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, USA, 2011.

[16] Yoshihito Toyama. Commutativity of term rewriting systems. *Programming of future generation computers II*, pages 393–407, 1988.

[17] Li-yao Xia, Dominic Orchard, and Meng Wang. Composing bidirectional programs monadically. In Luís Caires, editor, *Programming Languages and Systems*, pages 147–175, Cham, 2019. Springer International Publishing.

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