Towards Bidirectional Synchronization between Communicating Processes and Session Types

Liye Guo[3,4], Hsiang-Shang Ko[3], Keigo Imai[1], Nobuko Yoshida[2], and Zhenjiang Hu[3,4,5]

[1] Gifu University, Japan [2] Imperial College London, UK [3] National Institute of Informatics, Japan
[4] SOKENDAI (The Graduate University for Advanced Studies), Japan [5] University of Tokyo, Japan

Email: guoly@nii.ac.jp, hsiang-shang@nii.ac.jp, keigoi@gifu-u.ac.jp, n.yoshida@imperial.ac.uk, hu@nii.ac.jp

***Abstract*—Session types are a type discipline for eliminating communication errors in concurrent computing. These types can be thought of as a representation of communication protocols implemented by communicating processes. One application scenario that can be naturally supported by session types is semantics-preserving transformation of processes in response to protocol changes due to optimization, evolution, refactoring, etc. Such transformation can be seen as a particular kind of synchronization problem that has long been studied by the bidirectional transformations (BX) community. This short paper offers a preliminary analysis of the process–type synchronization problem in terms of BX, describing the prospects and challenges.**

I. Introduction

The programming language community has a long tradition of using strong type disciplines to statically rule out programs with undesired behaviour, and concurrency is one of the areas for which type disciplines have been particularly successful in providing safety guarantees. A flourishing line of work is *session calculi* and *session types* [1]–[3], which are a family of concise languages and their type systems modelling message-passing operations and protocols used in communicating processes. Just like a type of an expression is an abstraction of the range of values the expression may evaluate to, a session typing (a definition of which is given in Section II) of a process describes abstractly a *communication protocol* implemented by the process, i.e., the order and types of messages sent and received by the process (while disregarding other aspects like what computation is performed by the process to determine what messages to send). It then becomes straightforward to determine whether multiple processes can be safely executed concurrently by checking whether they have compatible typings. Multiple variants of session type systems have been developed, and provide varying levels of safety guarantees, from the absence of mismatching sending and receiving actions to deadlock freedom. There have been implementations of session type systems for a wide range of programming languages, including C, Java, Erlang, Go, Haskell, and OCaml [4].

In this paper we will look at an application scenario that can be naturally supported by session types: semantics-preserving transformation of processes in response to protocol changes. Here we are interested only in reasonable and non-disruptive changes to a protocol—for example, merging two consecutive sending actions into one to reduce the number of messages [5]. Correspondingly, we want to transform a process to conform to the new protocol while retaining its behaviour—for example, when transforming a process to send fewer messages, the contents delivered by the messages should stay the same. Other similar scenarios include protocol evolution (e.g., adding a new authentication step) and refactoring (e.g., moving some functionalities from one server to another). In all these scenarios, existing programs have to be adapted to new protocols in a semantics-preserving way, and an automated solution would be immensely helpful in reducing programmer effort.

An important observation is that the above problem can be seen more abstractly as a *synchronization* problem where a process and a session typing are to be kept "synchronized". Here "being synchronized" means that the session typing correctly describes a protocol implemented by the process. Whenever the session typing changes, possibly breaking synchrony, we should transform the process to a new one that has the changed typing, re-establishing synchrony; likewise, changes to processes should also be propagated to the typing side. Synchronization problems of this kind have been systematically studied by the *bidirectional transformations* (BX) community [6], [7]. There has been a large body of work on BX, ranging from theoretical concepts and frameworks to practical languages and tools for constructing automatic synchronizers, and we conjecture that BX will be helpful in analysing the problem and implementing a solution.

This short paper aims to give a preliminary analysis of the problem of synchronizing communicating processes and session types in terms of BX, describing the prospects and challenges. This analysis will be offered in Section IV, before which we will provide some background knowledge about session types and BX respectively in Sections II and III.

II. A Simple Session Calculus and Its Types

For presentation, we shall consider a simple session calculus, focusing on session communication. We summarize the syntax in Figure 1. With this syntax, any single communication is always via a *channel* k. The syntax supports two kinds of interaction via a channel: One is data sending and receiving, the other label selecting and branching. Process $k!\langle e\rangle.P$ sends the value of expression e via channel k, and then continues as process P; dually, the data-receiving process

978-1-5386-7789-6/19/$31.00 ©2019 IEEE

$$
\begin{array}{rcll}
P & ::= & k!\langle e_1, \ldots, e_n \rangle.P & \text{data sending} \\
& | & k?(x_1, \ldots, x_n).P & \text{data receiving} \\
& | & k \triangleleft l.P & \text{label selecting} \\
& | & k \triangleright \{\, l_1 : P_1 \,[\!]\, \cdots \,[\!]\, l_n : P_n \,\} & \text{label branching} \\
& | & \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 & \text{conditional} \\
& | & \mathsf{inact} & \text{inaction} \\
e & ::= & x & \text{variable} \\
& | & n & \text{number} \\
& | & e_1\ op\ e_2 & \text{binary operation}
\end{array}
$$

Figure 1. Communicating processes

should have form $k?(x).P$, which receives a value via channel k and binds it to variable x, and then continues as process P, in which x may occur free. Process $k \triangleleft l.P$ sends *label* l via channel k, and then continues as process P; dually, process $k \triangleright \{\, l_1 : P_1 \,[\!]\, \cdots \,[\!]\, l_n : P_n \,\}$ receives a label l_i among $l_1, \ldots, l_n$ via channel k, and then continues as process P_i. All communications are synchronous. Besides, we include the standard conditional construct if-then-else and let inact denote the lack of action.

As a concrete example, we can define the following process, the *addition/negation server:*

$$
\begin{array}{l}
k?(x).k \triangleright \{\, \mathtt{add} : k?(y).k!\langle x + y \rangle.\mathsf{inact} \\
\qquad [\!]\, \mathtt{neg} : k!\langle 0 - x \rangle.\mathsf{inact} \,\}
\end{array}
$$

This process first receives via channel k a number x and one of the labels `add` and `neg`, then continues according to the label received: If the label is `add`, the server still needs another addend to finish the addition, so it will receive another number y and then send the sum of x and y back; on the other hand, nothing else is required to find the negation of x, so the server will simply send the negated value back if the label received is `neg`. We can also define two clients communicating with the addition/negation server: $k!\langle 42 \rangle.k \triangleleft \mathtt{add}.k!\langle 43 \rangle.k?(z).\mathsf{inact}$ asks for the sum of 42 and 43, and $k!\langle 42 \rangle.k \triangleleft \mathtt{neg}.k?(z).\mathsf{inact}$ the negation of 42.

Session types can ensure type-error and communication-error freedom for session calculi. For our session calculus, we introduce the session types as shown in Figure 2. The intuition here is that a session type reflects the process's behaviour in a concise fashion. We formalize this idea with the typing rules in Figure 3. In the type system, Γ is an *environment* that keeps track of the types of variables, and Δ a *session typing*, which keeps track of the session types of channels. (That is, session types are types of channels, and session typings are types of processes.) And $\cdot$ is used to extend both environments and session typings. There are two kinds of judgement: $\Gamma \vdash e : S$ is a judgement one would expect in a programming language's type system, and $\Gamma \vdash P \triangleright \Delta$ is read as "under environment Γ, channels to which process P refers are typed by Δ". Each typing rule in Figure 3 is read as "if all the judgements above the line hold, then the judgement below also holds".

$$
\begin{array}{rcll}
T & ::= & ![S_1, \ldots, S_n]; T & \text{data sending} \\
& | & ?[S_1, \ldots, S_n]; T & \text{data receiving} \\
& | & \oplus \{\, l_1 : T_1, \ldots, l_n : T_n \,\} & \text{label selecting} \\
& | & \& \{\, l_1 : T_1, \ldots, l_n : T_n \,\} & \text{label branching} \\
& | & \mathsf{end} & \text{inaction} \\
S & ::= & \mathsf{num} & \text{number} \\
& | & \mathsf{bool} & \text{boolean}
\end{array}
$$

Figure 2. Session types

$$
\frac{\Gamma \vdash e_i : S_i \ (i = 1, \ldots, n) \qquad \Gamma \vdash P \triangleright \Delta \cdot k : T}{\Gamma \vdash k!\langle e_1, \ldots, e_n \rangle.P \triangleright \Delta \cdot k : ![S_1, \ldots, S_n]; T}\ \textsc{Send}
$$

$$
\frac{\Gamma \cdot x_1 : S_1 \cdots x_n : S_n \vdash P \triangleright \Delta \cdot k : T}{\Gamma \vdash k?(x_1, \ldots, x_n).P \triangleright \Delta \cdot k : ?[S_1, \ldots, S_n]; T}\ \textsc{Rcv}
$$

$$
\frac{\Gamma \vdash P \triangleright \Delta \cdot k : T_i}{\Gamma \vdash k \triangleleft l_i.P \triangleright \Delta \cdot k : \oplus \{\, l_1 : T_1, \ldots, l_n : T_n \,\}}\ \textsc{Sel}
$$

$$
\frac{\Gamma \vdash P_i \triangleright \Delta \cdot k : T_i \ (i = 1, \ldots, n)}{\begin{array}{l}\Gamma \vdash k \triangleright \{\, l_1 : P_1 \,[\!]\, \cdots \,[\!]\, l_n : P_n \,\} \\ \quad \triangleright \Delta \cdot k : \& \{\, l_1 : T_1, \ldots, l_n : T_n \,\}\end{array}}\ \textsc{Br}
$$

$$
\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash P_1 \triangleright \Delta \qquad \Gamma \vdash P_2 \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \triangleright \Delta}\ \textsc{If}
$$

$$
\frac{}{\Gamma \vdash \mathsf{inact} \triangleright \Delta}\ \textsc{Inact}
$$

Figure 3. Type system

One can see the *duality* of sending–receiving and label selecting–branching actions in the definition of processes, and this duality can be cleanly expressed in session types. Indeed, a pair of processes that can communicate with each other ought to have session typings that are dual to each other. For example,

$$
\begin{array}{l}
k : ?[\mathsf{num}]; \& \{\, \mathtt{add} : ?[\mathsf{num}]; ![\mathsf{num}]; \mathsf{end}, \\
\qquad\qquad \mathtt{neg} : ![\mathsf{num}]; \mathsf{end} \,\}
\end{array}
$$

and

$$
\begin{array}{l}
k : ![\mathsf{num}]; \oplus \{\, \mathtt{add} : ![\mathsf{num}]; ?[\mathsf{num}]; \mathsf{end}, \\
\qquad\qquad \mathtt{neg} : ?[\mathsf{num}]; \mathsf{end} \,\}
\end{array} \tag{1}
$$

are session typings of the addition/negation server and its clients defined above, respectively. The converse is also true, which is guaranteed by the type system. The type system does not, however, guarantee some other important properties (such as deadlock freedom) in concurrent computing.

As a final remark, the session calculus defined in this section together with its session types is simplified, compared to session calculi normally defined in the literature (e.g., Honda et al.'s original paper [2]). The session calculus we consider does not, for example, allow creation of new channels. All channels used are presumed to be created beforehand and

universally accessible. Also, our session calculus forbids transferring channels from a process to another. This operation, usually called *delegation* in session calculi, requires a different treatment from data sending and receiving, and are thus omitted to simplify the presentation. Finally, our session calculus lacks constructs supporting recursion, making it impossible to express some more realistic processes such as one that may stay live and communicating with others forever.

III. Bidirectional Transformations (BX)

In this paper, we use specifically *asymmetric lenses* as our model of bidirectional transformations. An asymmetric lens consists of two unidirectional transformations: The forward transformation, dubbed get, maps a set $\mathcal{S}$ to another set $\mathcal{V}$, and the backward transformation, dubbed put, maps $\mathcal{V} \times \mathcal{S}$ to $\mathcal{S}$. Elements of $\mathcal{S}$ and of $\mathcal{V}$ are called *sources* and *views*, respectively. The forward and backward transformations are required to satisfy the two *well-behavedness laws*:

$$get(put(v, s)) = v \qquad \text{(Correctness)}$$
$$put(get(s), s) = s \qquad \text{(Hippocraticness)}$$

We can regard asymmetric lenses as solutions to a particular kind of synchronization problem between sources and views. In BX terminology, a (binary) synchronization problem is specified by a *consistency relation* between two pieces of data, and solved by *synchronizers* that, whenever one piece of data changes, update the other to restore the consistency relation [8]. In the asymmetric setting, when the source changes, we recompute the view with get to restore consistency; when the corresponding view v of a source s has changed to v', the change in the view may be propagated back to the source by put, that is, we obtain a new source $s' = put(v', s)$. This backward propagation should, of course, restore the source–view consistency, which means that $get(s') = v'$ should hold and is guaranteed by the correctness law. When the change in the view is nil, that is, $v' = v$, there is nothing to be propagated back. In this special case, we expect $s' = s$, which is guaranteed by the Hippocraticness law. Asymmetric lenses can be seen as a framework for coping with synchronization problems in which the consistency relation is *functional* [9], which means that there cannot be multiple views consistent with a single source. One important property of asymmetric lenses and thus of this problem-solving framework is that there is always at most one forward transformation with which an arbitrary backward transformation can form an asymmetric lens [10]. In other words, an asymmetric lens is determined solely by put.

If we let sources be processes defined in Section II and views session typings, the forward transformation should naturally infer typings for processes. More importantly, when session typings have been altered, the changes may be propagated back to processes by the backward transformation. Asymmetric lenses as a framework can thus offer a new means to reason about process–type relationships.

IV. Analysis of the Synchronization Problem

We are ready to discuss the synchronization between communicating processes and session types now. Below let us consider two kinds of protocol change studied by Sivaramakrishnan et al. [5] for reducing the number of messages.

- The first transformation is *batching*: Two (or more) consecutive sending actions on the same channel can be combined into a single one that transmits all data at once. This kind of protocol change can be described by changing the type of the channel, for example, from $![\mathsf{num}]; ![\mathsf{num}]; T$ to $![\mathsf{num}, \mathsf{num}]; T$.
- The second transformation is *choice lifting*: When a sending action is followed by a label selecting action on the same channel, the latter action can be performed first. For example, typing (1) can be changed to:

$$k : \oplus \{\, \mathtt{add} : ![\mathsf{num}]; ![\mathsf{num}]; ?[\mathsf{num}]; \mathsf{end}, \\ \mathtt{neg} : ![\mathsf{num}]; ?[\mathsf{num}]; \mathsf{end} \,\}$$

  Choice lifting by itself does not reduce the number of messages, but may lead to more opportunities for batching. For example, in the `add` branch above, the originally separated sending actions become adjacent and amenable to batching, and we can further optimize the above typing to:

$$k : \oplus \{\, \mathtt{add} : ![\mathsf{num}, \mathsf{num}]; ?[\mathsf{num}]; \mathsf{end}, \\ \mathtt{neg} : ![\mathsf{num}]; ?[\mathsf{num}]; \mathsf{end} \,\} \qquad (2)$$

Sivaramakrishnan et al. approached the problem by analysing the session typing of a process for optimization opportunities, and then compiling the process to run on top of a runtime system, which handles the actual communication with the environment in accordance with the optimized protocol, and relays the messages to the process in accordance with the original protocol. With BX, we pursue a different approach: We will construct a synchronizer to modify the process to follow the optimized protocol directly. This approach can help to avoid the cost and complication of a runtime system, eliminate the need to maintain the old process that follows the original protocol, and offer correctness guarantees (BX well-behavedness in particular). On the other hand, since this kind of transformation changes the protocol, we have to modify (the code of) all processes involved, which may or may not be possible. For example, after optimizing the client typing to (2), we also need to invoke the synchronizer on the server and the dual typing of (2)

$$k : \& \{\, \mathtt{add} : ?[\mathsf{num}, \mathsf{num}]; ![\mathsf{num}]; \mathsf{end}, \\ \mathtt{neg} : ?[\mathsf{num}]; ![\mathsf{num}]; \mathsf{end} \,\}$$

so that the server and clients can keep working together.

Let us assume that the synchronizer we are going to construct is an asymmetric lens, for which the source set is processes and the view set is session typings. (We will revisit this assumption at the end of this section.) It suffices to consider the put direction of this lens, which takes a (possibly optimized) session typing and a process as input and returns a

transformed process. Sometimes what the put transformation does can be straightforward—for example, when the view is

$$\Delta \cdot k : ![\mathsf{num}, \mathsf{num}]; T \tag{3}$$

and the source is

$$k!\langle 42\rangle.k!\langle 43\rangle.P \tag{4}$$

(for some k, P, Δ, and T), then it is clear that we should perform a batching to transform the source into $k!\langle 42, 43\rangle.P$, and then continue to synchronize P with $\Delta \cdot k : T$.

However, processes in general have more complicated structures than session types, and we should be able to deal with patterns other than direct correspondences like the above one. For example, given the same view (3), the source that we want to synchronize may instead be

$$k!\langle 42\rangle.\mathsf{if}\ b\ \mathsf{then}\ k!\langle 43\rangle.P_1\ \mathsf{else}\ k!\langle 44\rangle.P_2$$

In this case, we should first distribute $k!\langle 42\rangle$ into the two branches of the if-expression

$$\mathsf{if}\ b\ \mathsf{then}\ k!\langle 42\rangle.k!\langle 43\rangle.P_1\ \mathsf{else}\ k!\langle 42\rangle.k!\langle 44\rangle.P_2$$

before we can perform batching on both branches and go on to transform P_1 and P_2. This distribution of sending operations into branches of if-expressions is also important for choice lifting, since processes usually make label selections dynamically with if-expressions. For example, given a source

$$k!\langle 42\rangle.\mathsf{if}\ b\ \mathsf{then}\ k \triangleleft l_1.P_1\ \mathsf{else}\ k \triangleleft l_2.P_2$$

and its view

$$\Delta \cdot k : ![\mathsf{num}]; \oplus\{\, l_1 : T_1, l_2 : T_2 \,\}$$

which is changed to

$$\Delta \cdot k : \oplus\{\, l_1 : ![\mathsf{num}]; T_1, l_2 : ![\mathsf{num}]; T_2 \,\}$$

we see that what we need to perform is a choice lifting, which is done essentially by swapping the order of $k!\langle\ldots\rangle$ and $k \triangleleft \ldots$, but this swapping cannot take place before $k!\langle 42\rangle$ is distributed into the branches of the if-expression. When we scale up and try to deal with more realistic process languages, we will need more auxiliary transformations of this kind, and these auxiliary transformations will be more complicated.

There is another more aggressive kind of auxiliary transformation that we can consider but must be more cautious about: rearranging actions on different channels. Consider batching again. Given the source

$$k!\langle 42\rangle.k'?(x).k!\langle 43\rangle.P_1 \tag{5}$$

and the view (3), it might be tempting to swap $k!\langle 42\rangle$ with $k'?(x)$ so that the former can be merged with $k!\langle 43\rangle$. But this may be too aggressive since the swapping may not be able to be propagated to other processes—this process may interact with, say, the following process

$$k?(x).k'!\langle x\rangle.k?(y).P_2$$

in which we cannot swap the corresponding receiving and sending operations. This example reveals two problems: First, an invocation of the synchronizer may not succeed—we have seen that, with the same view (3), the synchronizer can succeed for source (4) but not source (5). Therefore we need to switch to a revised definition of well-behaved lenses that takes partiality into account [11], and to make the behaviour of the synchronizer predictable, we should give a characterization of the synchronizer's domain, i.e., those pairs of source and view the synchronizer can successfully handle. Second, multiple invocations of the synchronizer to bring a set of processes to follow a new protocol must be coordinated to achieve more aggressive transformations. In the example above, if we know that there is no dependency between the receiving and sending actions in the other process, then we can perform the swapping on both processes. This will complicate the characterization of the synchronizer's domain though.

All these transformations should preserve the original functionality of the process in some sense; formally stating this preservation turns out to be another interesting problem. Preservation in general is some kind of equivalence—we want to say that the original and transformed processes have the same behaviour in some suitable sense. The usual way to say two processes have the same behaviour is the notion of bisimulation [12], which basically says that both processes can continue to match each other's moves with identical moves. This is too strong in our case, however, since batching, choice lifting, and rearranging actions on different channels all make a transformed process send and receive messages in a different way from the original process. We thus need to explore a weaker version of bisimulation, and prove that the synchronizer establishes this weaker bisimulation between the original and transformed processes.

Finally, let us get back to the assumption that the synchronizer is an asymmetric lens. As we explained in Section III, to use an asymmetric lens to solve a synchronization problem, the consistency relation should be functional. For our particular synchronization problem, this means that every process has at most one typing. However, this is not the case for the standard typing relation (Figure 3). For example, the process $k \triangleleft l.\mathsf{inact}$ can have any typing of the form $\Delta \cdot k : \oplus\{\, l : \mathsf{end}, \ldots \}$. Here are some possible ways out: We can enrich the syntax of session types such that it is capable of expressing "most general types", which are types that can be specialized to any other types that a channel can have; this will complicate the type structure, though. We can insist that a process can only have a "most precise" typing with no redundant information—$k : \oplus\{\, l : \mathsf{end} \,\}$ for the example above; this, however, will require the introduction of subtyping. We can instead treat the synchronization problem as a symmetric one [13], [14], where the consistency relation does not need to be functional; however, bidirectional programming in the symmetric setting is much less explored, with the most reliable way still being constructing a symmetric lens in terms of two or more asymmetric lenses [15], [16], which can be cumbersome. Some initial experiments will be necessary for determining which is the best way to go.

REFERENCES

[1] K. Takeuchi, K. Honda, and M. Kubo, "An interaction-based language and its typing system," in *International Conference on Parallel Architectures and Languages Europe*, ser. Lecture Notes in Computer Science, vol. 817. Springer, 1994, pp. 398–413.

[2] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language primitives and type discipline for structured communication-based programming," in *European Symposium on Programming*, ser. Lecture Notes in Computer Science, vol. 1381. Springer, 1998, pp. 122–138.

[3] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," *Journal of the ACM*, vol. 63, no. 1, pp. 9:1–9:67, 2016.

[4] S. Gay and A. Ravara, Eds., *Behavioural Types: from Theory to Tools*. River Publishers, 2017.

[5] K. Sivaramakrishnan, M. Qudeisat, L. Ziarek, K. Nagaraj, and P. Eugster, "Efficient sessions," *Science of Computer Programming*, vol. 78, no. 2, pp. 147–167, 2013.

[6] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *International Conference on Model Transformation*, ser. Lecture Notes in Computer Science, vol. 5563. Springer, 2009, pp. 260–283.

[7] F. Abou-Saleh, J. Cheney, J. Gibbons, J. McKinna, and P. Stevens, "Introduction to bidirectional transformations," in *International Summer School on Bidirectional Transformations (Oxford, UK, 25–29 July 2016)*, ser. Lecture Notes in Computer Science. Springer, 2018, vol. 9715, ch. 1, pp. 1–28.

[8] L. Meertens, "Designing constraint maintainers for user interaction," 1998. [Online]. Available: http://www.kestrel.edu/home/people/meertens/pub/dcm.ps

[9] P. Stevens, "Bidirectional model transformations in QVT: Semantic issues and open questions," *Software and Systems Modeling*, vol. 9, pp. 7–20, 2010.

[10] Z. Hu, H. Pacheco, and S. Fischer, "Validity checking of putback transformations in bidirectional programming," in *International Symposium on Formal Methods*, ser. Lecture Notes in Computer Science, vol. 8442. Springer, 2014, pp. 1–15.

[11] H. Pacheco, Z. Hu, and S. Fischer, "Monadic combinators for "putback" style bidirectional programming," in *Workshop on Partial Evaluation and Program Manipulation*. ACM, 2014, pp. 39–50.

[12] R. Milner, *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[13] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas, "From state- to delta-based bidirectional model transformations: the symmetric case," in *International Conference on Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science. Springer, 2011, pp. 304–318.

[14] M. Hofmann, B. Pierce, and D. Wagner, "Symmetric lenses," in *Symposium on Principles of Programming Languages*. ACM, 2011, pp. 371–384.

[15] M. Johnson and R. Rosebrugh, "Cospans and symmetric lenses," in *International Workshop on Bidirectional Transformations*. ACM, 2018, pp. 21–29.

[16] ——, "Symmetric delta lenses and spans of asymmetric delta lenses," *Journal of Object Technology*, vol. 16, no. 1, pp. 2:1–32, 2017.