C-Lenses Explained: Bx Foundations for the Rest of Us

Anthony Anjorin[1], Hsiang-Shang Ko[2] and Erhan Leblebici

[1]IAV GmbH, Germany

[2]Institute of Information Science, Academia Sinica, Taiwan

Abstract

Bidirectional transformations (bx) are mechanisms for maintaining the consistency of multiple artefacts. Some of the challenges bx research aims to address include answering fundamental questions such as how best to precisely characterise consistency, "good" consistency maintainers, and the required input and assumptions to guarantee this good behaviour.

While substantial progress has been made towards unifying the different (variants of) formal frameworks for bx, many of these formal results are not yet easily accessible to more practically-oriented bx researchers, often due to a missing background in (advanced) category theory. This is an unfortunate situation, as many bx tools are developed and maintained by practically-oriented bx researchers, who would but currently cannot fully profit from formal results and insights. In particular, we are not aware of any practical implementation of C-lenses, not even preliminary discussions about its relevance to practical bx. We believe this is because of the inaccessibility of the categorical language, and write this paper to decipher C-lenses for bx researchers without any substantial background in category theory. Our goal is to spark broader interest in C-lenses and in discussions on their usefulness in practice and potential for improving existing, and inspiring new bx tools.

We start by reviewing the well-known state-based bx framework, already accessible to a broad audience due to its simplicity. We then explain how this framework can be elegantly generalised to the richer, delta-based setting of C-lenses using multiple examples and illustrative diagrams.

Keywords

Bidirectional Transformations, Formal Foundations, C-Lenses

CEUR-WS.org/Vol-2999/bxpaper1.pdf

1. Introduction

Bidirectional transformations (bx) are mechanisms for maintaining the consistency of multiple artefacts [1]. To provide support for consistency maintenance, a formal understanding of what constitutes a "good" consistency maintainer, as well as tooling support for automation are equally important. It is also essential that formal and practical bx go hand-in-hand: bx tools should faithfully instantiate a bx formal framework, and bx formal frameworks should cover aspects of practical relevance.

There has been considerable progress made regarding bx foundations. Johnson et al. [2, 3, 4] and Diskin et al. [5] have both suggested generalisations of the state-based lens bx framework [6] to a richer *delta-based* setting, addressing certain limitations inherent to a state-based setting.

Bx 2021: 9th International Workshop on Bidirectional Transformations, part of STAF, June 21, 2021

anthony.anjorin@iav.de (A. Anjorin); joshko@iis.sinica.edu.tw (H. Ko); erhan.leblebici@outlook.com (E. Leblebici)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

While there are relatively direct implementations of the state-based characterisation of bx such as Boomerang [7], GRoundTram [8], BiGUL [9] and HOBiT [10] (to name just a few), we are not aware of any bx tool that faithfully implements C-lenses according to Johnson et al. [3]. This is certainly not because practical bx tools are inherently state-based, or that working in a rich delta-based setting is often infeasible in practice. Indeed, bx tools from the Model-Driven Engineering (MDE) community are often able to record and work with deltas (see a recent comparison of bx tools for details [11]). Diskin and Hermann [12] have collaborated to discuss how Triple Graph Grammars (TGGs) [13] can be aligned with D-lenses according to Diskin et al. [5]. This is a helpful step in the right direction, as there are multiple TGG-based bx tools available. We claim that most results of Johnson et al. [3], concerning C-lenses as an elegant generalisation of state-based to delta-based bx, remain largely inaccessible to most practically-oriented bx researchers mainly due to a missing background in category theory.

Our goal in this paper is to contribute to the ongoing endeavour of bridging formal and practical bx. We aim to spark interest in C-lenses from a practical point of view, and enable discussions on their potential to improve existing bx tools and perhaps inspire a new generation of bx tools in the different bx sub-communities. To serve the primary purpose of the bx community — to enable communication and exchange of ideas between the various sub-communities — we target practical bx researchers without any substantial background in category theory, and aim to be as self-contained and understandable as possible. As the paper is intentionally kept lightweight in terms of technicality, we refer to and rely on Johnson et al. [2, 3, 4] for all formal details and proofs.

The paper starts with lenses in the state-based setting (Sect. 2) as a warm-up for the generalisation to C-lenses in our main, delta-based setting (Get and Put operations in Sect. 3 and their laws in Sect. 4). The main advantage of C-lenses is universality, which we explain and discuss in Sect. 5. Related work is in Sect. 6 and our conclusion in Sect. 7.

2. State-based bx

We start with the basic notions of consistency maintenance in a simple *state-based* setting, as a warm-up for our main *delta-based* setting in Sect. 3. Consider a simple consistency maintenance scenario (known as the composers example [14]) where we wish to synchronise multisets (sources) of composers with their name, date of birth, and nationality, with multisets (views) of composers with only their name and nationality. This minimal example exhibits *asymmetry*, a property of consistency maintenance scenarios that we will assume[1] throughout the paper: when a pair of source and view are consistent, the view is completely determined by the source. Assuming asymmetry, a *lens* [6] keeps a pair of source and view consistent using two functions: Get to obtain the view of a source, and Put to update an existing source given a new view. This can be formalised in a straightforward manner as follows:

$$Get: \quad S \to V, s \overset{Get}{\longmapsto} v \tag{1}$$

$$Put: \quad V \times S \to S, (v, s) \overset{Put}{\longmapsto} s' \tag{2}$$

[1]Johnson et al. [15] argue that this is not a fundamental restriction as symmetric lenses can be constructed as spans of asymmetric lenses.

S and V are sets of objects, Get and Put are functions, and $V \times S$ is the product set consisting of pairs (v, s) of objects with $v \in V$ and $s \in S$. Independent of the exact behaviour of a lens — a pair (Get, Put) — the challenge is to characterise "good" synchronisation behaviour in a general manner. The lens framework does this by requiring the three laws depicted in Fig. 1 for *very well-behaved* lenses. We have chosen a diagrammatic notation for the laws as this generalises well to the delta-based setting in Sect. 3. We now explain the meaning of these laws.

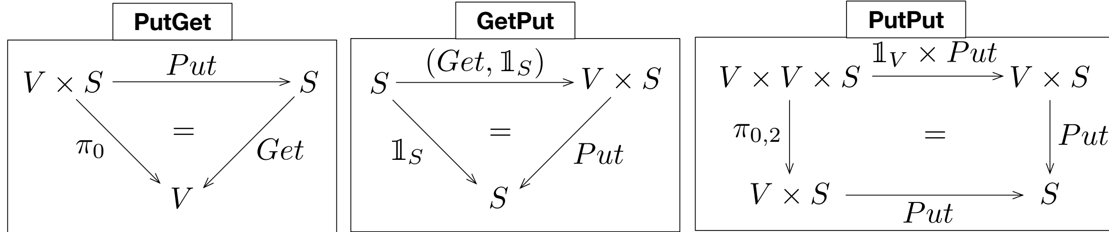

Figure 1: State-based lens laws for very well-behaved lenses

Starting with $PutGet$, the diagram to the far left: π_0 is the *projection* to the first (0th) dimension, i.e., to V in this case — meaning that π_0 maps pairs (v, s) to v. The equals sign "=" inside the diagram means that it must commute for the law to hold, i.e., $Put \,;\, Get = \pi_0$ where the operator ";" — read as "then" — composes functions as required. The intuition here is that if you **put** a view v into an old source s to obtain a new source s', the view you **get** from s' immediately after that should be exactly v. This ensures that Put synchronises v fully with s.

Example 1. To make things more concrete, consider our composers example: Let S be a set of multisets of triples $(name, date, nat)$ representing composers, and V be a set of multisets of pairs $(name, nat)$ representing composers without their date of birth. This means that source and view objects here are multisets of triples and pairs, respectively. Let Get be the function mapping sources to views by mapping the triples $(name, date, nat)$ in a source to pairs $(name, nat)$ in the resulting view. Now let us define $Put(v, s) = s'$ as follows:

1. Set $s' = s$.
2. For every pair $(name, nat)$ in v, if there exists no triple of the form $(name, _, nat)$[2] in s' then add a new triple $(name, ???, nat)$ to s'.
3. For every triple $(name, date, nat)$ in s', if there exists no pair $(name, nat)$ in v then remove the triple $(name, date, nat)$ from s'.

While this choice of Get and Put might seem reasonable, the $PutGet$ law tells us that it is actually a bad idea: Consider $v = \{$ (*J.S. Bach*, *German*), (*J.S. Bach*, *German*) $\}$ and $s = \{$ (*J.S. Bach*, *31/03/1685*, *German*) $\}$. This is a valid input pair for Put — in particular, v is a valid view, i.e., $v \in V$. As Put does not care about duplicates, however, it will yield an unchanged s as the result s'. As $Get(s) = \{$ (*J.S. Bach*, *German*) $\} \neq v$, the $PutGet$ law is violated.

Intuitively, Put as we have defined it does not take v fully into account. Specifically, it ignores the fact that some pairs can be present multiple times in the view. As this information

[2] "_" means any value.

is, however, not ignored by Get, something is wrong. To avoid violating $PutGet$, Get can be adjusted to avoid duplicate pairs in the view.

For the second law $GetPut$ in the diagram of Fig. 1: $\mathbb{1}_S$ denotes the identity function $s \mapsto s$, $(Get, \mathbb{1}_S)$ denotes a pair of functions that maps s to the pair $(Get(s), \mathbb{1}_S(s)) = (v, s)$. As the diagram is to be commutative, the law demands that $(Get, \mathbb{1}_S)\ ; Put = \mathbb{1}_S$. Intuitively, this means **get**ting a view v from a source s and directly **put**ting the *same* view v back in s must do nothing to s. This ensures that Put does not fabricate random information that does not come from the view.

Example 2. Revisiting our composers example, let Put be defined as for Ex. 1. Let Get project triples to pairs as in Ex. 1 and additionally avoid duplicates so $PutGet$ is fulfilled. To explore a practical violation of $GetPut$, let us assume that Get is further adjusted to keep only last names, e.g., *J.S. Bach* becomes *Bach* in the view to provide a more compact list of composers. Without changing Put, the $PutGet$ law is still satisfied for this new Get. Consider the source containing *J.S. Bach* as a composer. Applying Get results in an abbreviated name in the view. Put as defined for Ex. 1 will not, however, recognise this and instead delete the composer *J.S. Bach* in the source and add the new composer *Bach* with a default date of birth. As this changes the source, we see that $(Get, \mathbb{1}_S)\ ; Put \neq \mathbb{1}_S$, violating $GetPut$.

The final law $PutPut$ is depicted to the right of Fig. 1. $V \times V \times S$ is a product set consisting of triples (v, v', s). $\pi_{0,2}$ is the projection to the first (0th) and third (2nd) components, i.e., maps triples (v, v', s) to pairs (v, s). The function $\mathbb{1}_V \times Put$ applies the identity to the first component of a triple, and Put to the second and third components, i.e., maps triples (v, v', s) to pairs $(v, Put(v', s)) = (v, s')$. As the diagram is to be commutative, this means that $\mathbb{1}_V \times Put\ ; Put$, i.e., applying Put to the first view v', then to the second v, is to be the same as $\pi_{0,2}\ ; Put$, i.e., skipping the first view v' and just applying Put to the second view v. Intuitively, this means that it should make no difference if intermediate views such as v' are **put** into the source or not. The last and final view v fully determines the resulting source.

Example 3. Consider our composers example with Put as defined in Ex. 1 and Get as the projection to names and nationalities avoiding duplicates. $PutPut$ can be shown to be violated as follows: Starting with a source $s = \{\, (\textit{J.S. Bach}, \textit{31/03/1685}, \textit{German}) \,\}$, let $v' = \{\}$ and $v = \{\, (\textit{J.S. Bach}, \textit{German}) \,\}$. As $Put(v, Put(v', s)) = Put(v, \{\}) = \{\, (\textit{J.S. Bach}, \textit{???}, \textit{German}) \,\} \neq Put(v, s) = s$, $PutPut$ is violated. Intuitively, the first view v' deletes the single entry in the source, while the second view v can be interpreted as attempting to recreate it. This does not work, however, as there is no way to reconstruct the date of birth of the composer from v.

Numerous authors including Johnson et al. [2] and Diskin et al. [5] use the $PutPut$ law to motivate a generalisation of the lens state-based framework to a delta-based setting. In some state-based scenarios, the $PutPut$ law is indeed unreasonable: it basically demands that **put**ting *arbitrary* intermediate views make no difference to the final result of **put**ting the final view.

Considering Ex. 3, the practical consequence of $PutPut$ would be to require tracking a complete history of all changes so that all previous values can be restored. This can be undesirable or even infeasible in practice due to limited memory and other practical constraints, and is also potentially confusing as there is no formal way to distinguish between (i) deleting and restoring a composer and (ii) deleting a composer and adding a *new* composer that just happens to have

the same name and nationality. In a richer, delta-based setting this distinction becomes possible, making the $PutPut$ law just as fundamental as $PutGet$ and $GetPut$.

3. Delta-based get and put

In this section and the next, we generalise the Get and Put functions and the (first two) well-behavedness laws to a *delta-based* setting, in which we can express not only what sources and views become but also how they are modified. In this setting, S is no longer simply a collection of all possible sources but is now a *category*, consisting not only of all sources as its objects $obj(S)$, but also of all possible *arrows* $arr(S)$ between all sources. As it helps (in a bx context) to think of these arrows as "deltas" that represent a means of obtaining (computing, transforming) one source from another, we shall often use the term "delta" instead of the more generic "arrow" in the rest of this paper. Analogously, V is also a category consisting of all possible views $obj(V)$ and view deltas $arr(V)$. While we shall use the notation $\delta_{ss'} : s \to s'$ to represent deltas, it is crucial to bear in mind that deltas are in general *not* fully determined by their endpoints. This means that unless explicitly stated otherwise, there can in general be multiple, *different* deltas $\delta_{ss'} \neq \delta^*_{ss'} : s \to s'$ between the same objects s and s'.

Example 4. Some objects and deltas in the source and view categories S and V for the composers example are shown in Fig. 2. Within the two categories (depicted as rectangles), objects are multisets/states (depicted as ellipses), and deltas are modifications from one multiset to another (depicted as bold arrows). Only three sources and three deltas are shown in S. The source s contains just one composer *J.S. Bach* (represented as a triple). The source delta $\delta_{ss'}$ adds a new composer with the same name and nationality but a different date of birth *08/03/1714* to result in s', while $\delta_{s's''}$ then edits this new composer by changing the name to *C.P.E. Bach*.

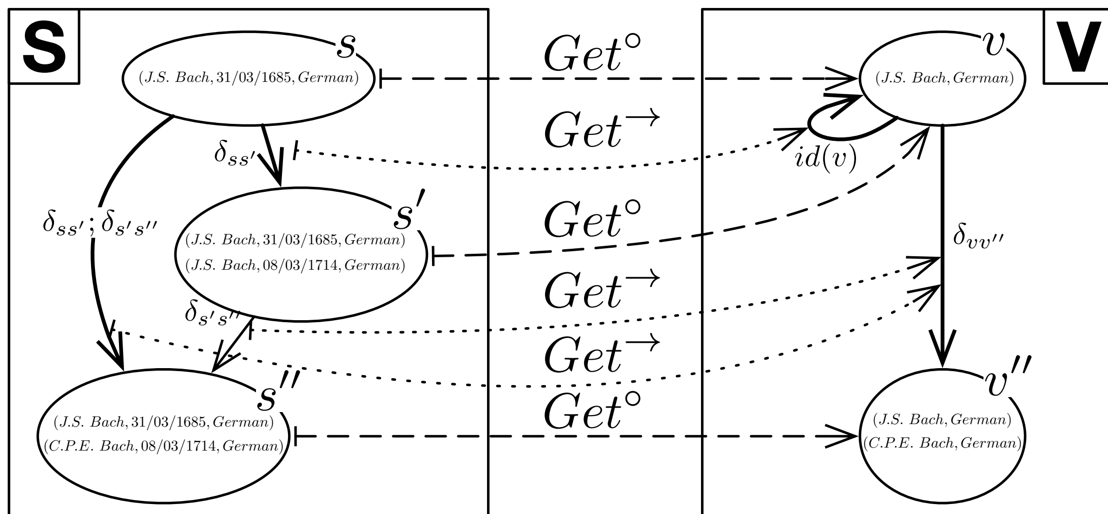

Figure 2: Excerpt of delta-based Get for the composers example

Categories must have a composition operator ";" with which deltas can be composed in an associative manner (without caring about the order in which a chain of deltas is composed), and an identity operator id that maps every object s to an identity delta $id(s) : s \to s$ that represents doing nothing to s. Identity is expected to be compatible with composition: doing nothing before or after doing something should make no difference, that is, $id(s)\ ; \delta_{ss'} = \delta_{ss'}$ and $\delta_{s's}\ ; id(s) = \delta_{s's}$ for all incoming deltas $\delta_{ss'} : s \to s'$ and outgoing deltas $\delta_{s's} : s' \to s$.

Example 5. Back to Fig. 2, the composition of both deltas, i.e., adding *C.P.E. Bach* directly, is depicted as the delta $\delta_{ss'};\delta_{s's''}$ leading directly from s to s''. Get° maps s to a view v in V containing *J.S. Bach* represented as a pair. The identity deltas are not shown in the figure, but there is one from each state back to itself, representing a no-op.

As S and V are now categories, $Get : S \to V$ has to operate on both objects and deltas, mapping source objects to view objects, and source deltas to view deltas. Get is now a so called *functor* with two parts: Get°, which maps sources to views, and $Get^\to$, which maps source deltas $\delta_{ss'} : s \to s'$ to corresponding view deltas $\delta_{vv'} : Get^\circ(s) \to Get^\circ(s')$. Such functors must respect the composition and identity operators in the source and target categories, i.e., $Get^\to(\delta_{ss'};\delta_{s's''}) = Get^\to(\delta_{ss'}); Get^\to(\delta_{s's''})$ and $Get^\to(id(s)) = id(Get^\circ(s))$.

Example 6. Figure 2 contains a small excerpt of a delta-based $Get : S \to V$ for the composers example. Mappings for Get° are depicted as dashed "$\mapsto$" arrows between sets, mappings for $Get^\to$ as dotted "$\mapsto$" arrows between deltas. The functor Get is given by the entirety of all dashed and dotted mappings. As we are using the variant of Get that avoids duplicates, Get° also maps s' to v. As the new composer becomes distinct in the view after his name is changed to *C.P.E. Bach*, s'' is mapped to a new set v'' now with both *J.S. Bach* and *C.P.E. Bach*. As Get is a functor, we also have to specify what happens to deltas: $\delta_{ss'}$ is mapped to $id(v)$ as it does not correspond to any change in the view. $\delta_{s's''}$ is mapped to $\delta_{vv''}$, i.e., in this case editing a composer in the source corresponds to *adding* a composer in the view if the corresponding composer becomes distinct. As functors must respect composition, $\delta_{ss'};\delta_{s's''}$ must be mapped to $Get^\to(\delta_{ss'};\delta_{s's''}) = Get^\to(\delta_{ss'}); Get^\to(\delta_{s's''}) = id(v); \delta_{vv''} = \delta_{vv''}$.

To generalise Put to a functor, let us start by discussing the input for Put°, the object part of Put. From the state-based Put, we can surmise that Put° has to take at least a new view v' and an old source s as input. But v' and s cannot be completely unrelated. The input will only make sense for Put° if the new view v' resulted from a delta $\delta_{vv'}$ applied to an old view v that corresponds to the old source s. This is formalised with the diagram depicted in Fig. 3. The new view v' must be the result of applying a view delta $\delta_{vv'}$ to v that must be $Get^\circ(s)$. This is crucially important as there are settings where it is simply impossible to get from some given v to another v', i.e., no such corner with v and v' exists. This cannot be expressed in a state-based framework. Furthermore, even if we are in a setting where it

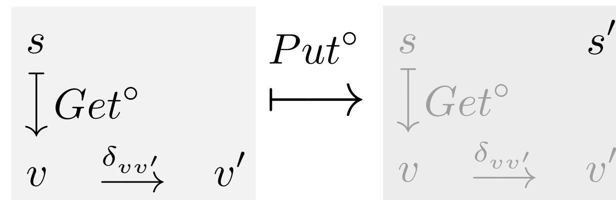

Figure 3: Input and output of Put°

is possible to get from every v to every other v' via some delta (or possibly multiple deltas), it is still important to note that Put can treat each of these deltas as *different* corners. In a state-based framework, the assumption is that it is possible to get from every v to every other v' in a *unique* way — think of the meaning of this delta as "discarding the original view and changing to the target view"; therefore there is no need to bother with how to handle this unique delta as it is fixed by the pair of connected objects.[3]

Example 7. Figure 4 provides a concrete example for Put°. The view v (containing *J.S. Bach*) is extended by adding *C.P.E. Bach* to produce v'. This corner is valid input for Put° according to Fig. 3, and Put° produces a new source s' consisting of *J.S. Bach* from the old source s and a new composer (*C.P.E. Bach*, *???*, *German*) corresponding to the newly added (*C.P.E. Bach*, *German*) in the view v'.

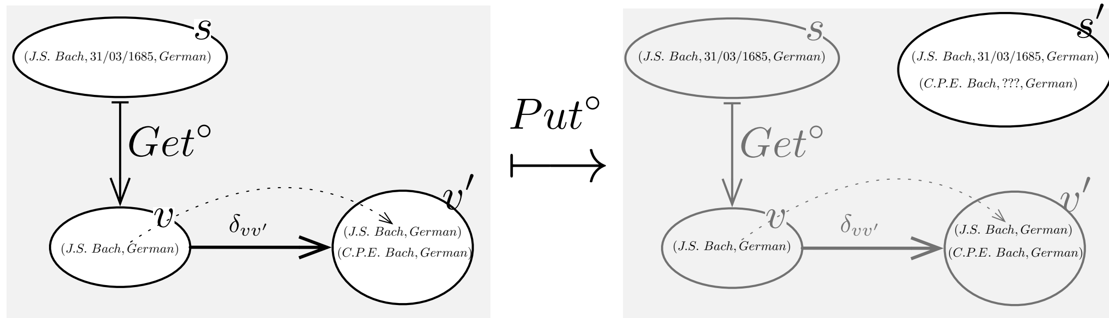

Figure 4: Example for Put°

On our way to C-lenses, the challenge is now to work out what $Put^{\rightarrow}$ could operate on. How would a category look like with "corner" objects shaped as the input for Put° in Fig. 3? How should we define deltas (arrows) between these corners? As chance would have it, categories with exactly such corner objects are already well-studied in category theory. This means we have a fully worked out suggestion that we can adopt to a bx context (and decide if it makes sense for us).

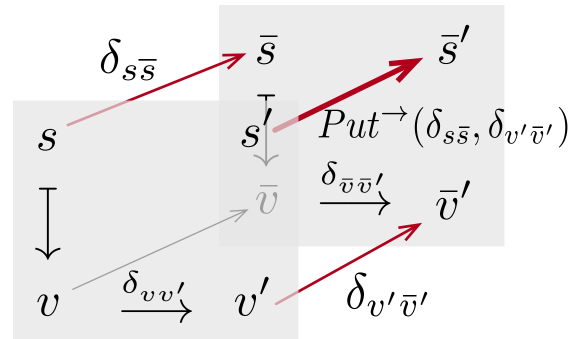

Figure 5: Input and output for $Put^{\rightarrow}$

The basic idea of the *comma category* construction is to create new categories by handling arrows as objects. So in our case, the arrows that we want to treat as objects are the view deltas $\delta_{vv'}$. The construction generalises this idea by using functors to choose the exact arrows we are interested in. In the most general form, one functor is used to choose the start of the arrow, while a second functor chooses the

[3] In category theory the state-based setting corresponds to *codiscrete* categories and not *discrete* categories as one might falsely assume.

end of the arrow. Applying this idea to our "corners", we can characterise our corner objects as exactly those view deltas $\delta_{vv'}$, whose starting point is fixed by Get (remember Get is a functor), and whose ending point is basically free. We can express this freeness formally by using $\mathbb{1}_V$ (the identity functor from V to V) to "fix" the ending point of $\delta_{vv'}$.

With this we can now look up the standard definition of the comma category $(Get, \mathbb{1}_V)$ and obtain the shape for its objects and arrows as depicted in Fig. 5. An "arrow" in $(Get, \mathbb{1}_V)$ is a pair of arrows $(\delta_{s\bar{s}}, \delta_{v'\bar{v}'})$ between two corners $(s, \delta_{vv'} : v \to v')$ and $(\bar{s}, \delta_{\bar{v}\bar{v}'} : \bar{v} \to \bar{v}')$ in light grey squares. So this special "corner delta" in our category of corners is a pair of a source delta $\delta_{s\bar{s}}$ changing the old source s, and a view delta $\delta_{v'\bar{v}'}$ changing the new view v'. As Get is a functor, $\delta_{s\bar{s}}$ already fixes the corresponding view delta on the old view v, namely $Get^{\circ}(\delta_{s\bar{s}})$. As the bottom square must commute (according to the standard comma category construction), we only accept such $\delta_{v'\bar{v}'}$ that "preserve" our corner structure, i.e., there must exist a suitable $\delta_{\bar{v}\bar{v}'}$.

Intuitively, think of the corner delta as a change to an input for Put° that produces again a valid input for Put°, i.e., that produces a valid corner. The corner delta serves as input for the functor $Put^{\to}$ that now has to map the pair of source and view changes $(\delta_{s\bar{s}}, \delta_{v'\bar{v}'})$ to a corresponding change $Put^{\to}(\delta_{s\bar{s}}, \delta_{v'\bar{v}'})$ of the updated source s in the "pre-square" to obtain the new updated source $\bar{s}'$, which completes the "post-square". So we can think of $Put^{\to}$ in Fig. 5 as being able to handle a "small" perturbation of the bottom-left corner (s and v') of a consistent pre-square, by computing a corresponding update of the top-right part (s') of the pre-square required to directly obtain a consistent post-square.

Example 8. A concrete example demonstrating input and output for $Put^{\to}$ is depicted in Fig. 6. The two grey rectangles contain *squares* that can both be interpreted as the combined input (corner objects in $(Get, \mathbb{1}_V)$) and output (objects in S) for Put°. In the pre-square a new composer (*C.P.E. Bach*, *German*) is added to the view and put back into the old source to result in a new composer (*C.P.E. Bach*, *???*, *German*) with an unknown date of birth. What the example now demonstrates is that if the old source s is changed by adding a composer (*J.S. Bach*, *08/03/1714*, *German*) via $\delta_{s\bar{s}}$, and the new view v' is changed by removing (*J.S. Bach*, *German*) via $\delta_{v'\bar{v}'}$, these two changes can be passed to $Put^{\to}$ to directly compute the delta required to change the new source s' to $\bar{s}'$. As Put is required to be a functor, the computed delta $Put^{\to}(\delta_{s\bar{s}}, \delta_{v'\bar{v}'})$ must produce a source $\bar{s}'$ that matches the post-square derived by applying Put° to the post corner.

While Put° is clearly useful and can be intuitively understood as the straightforward delta-based version of Put, it is not so clear if $Put^{\to}$ is a good fit for bx. At first glance, Fig. 5 is probably unexpected. Why do we have to handle a *pair* of changes to both the source and the view? The reader's intuition at this point is probably to map the view delta $\delta_{vv'}$ to a source delta $\delta_{ss'}$ using $Put^{\to}$. While this would certainly be simpler, the problem is that the choice of $Put^{\to}(\delta_{ss'})$ would not be "unique", "special", "minimal", "least controversial", or – as category theorists would say in a general setting – *universal* when compared to other possible choices. We shall see in Sect. 5 that providing the additional information of how to handle all possible ways of changing corners (old source and corresponding new view) helps us arrive at a particular Put behaviour that is unique in a specific sense and highly desirable in some scenarios.

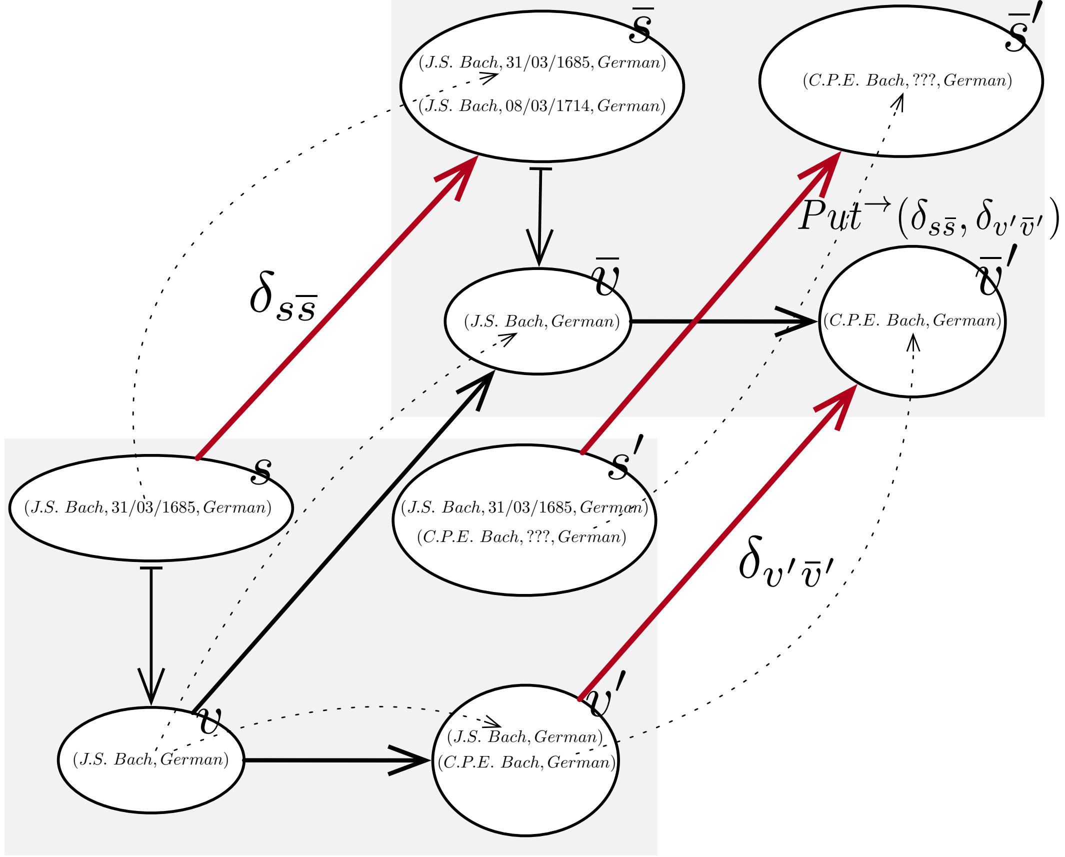

Figure 6: Example for $Put^{\rightarrow}$

4. Delta-based lens laws

Before we explain the advantages of $Put^{\rightarrow}$ in Sect. 5, we still need to generalise the lens laws depicted as commutative diagrams in Fig. 1. We will do so for $PutGet$ and $GetPut$; as the third law $PutPut$ is not directly required for the rest of the paper, we shall skip it and refer the interested reader to Johnson et al. [2, 3] for the rather technical details.

To provide delta-based versions of $PutGet$ and $GetPut$ we need two simple functors: $Q_V : (Get, \mathbb{1}_V) \rightarrow V$ to extract normal views from our comma category, and $\eta_{Get} : S \rightarrow (Get, \mathbb{1}_V)$ to inject sources into the comma category. As these are functors, we have to think in terms of both objects and arrows as depicted in Fig. 7. Q_V° extracts the new view v' from a corner object, while $Q_V^{\rightarrow}$ extracts the delta between views of two connected corners. η_{Get}° injects a source s into the comma category by extending it to a corner object using Get° and id. $\eta_{Get}^{\rightarrow}$ operates similarly on a source delta, using Get°, $Get^{\rightarrow}$ and id to extend it to a pair of connected corners (an arrow between corners).

Using Q and η_{Get} we can now formulate delta-based versions of $PutGet$ and $GetPut$ as depicted in Fig. 8. Compared with the state-based diagrams in Fig. 1, the only difference here is that $V \times S$ is now generalised to our comma category $(Get, \mathbb{1}_V)$. Consequently, extracting the

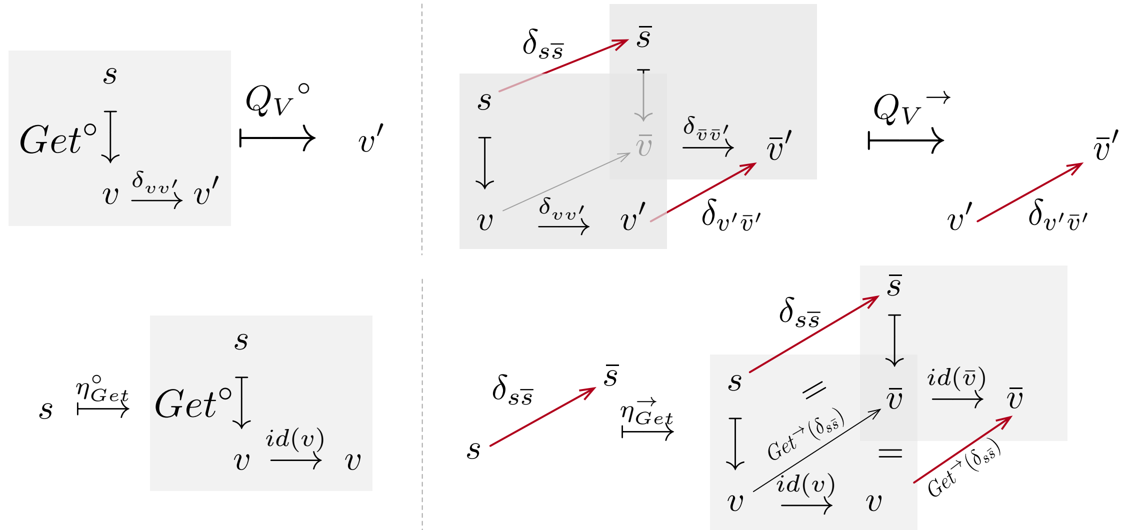

Figure 7: View extraction and injection

view in $PutGet$ now requires Q_V as a generalisation of π_0 and lifting the source uses η_{Get} as a generalisation of the function $(Get, \mathbb{1}_S)$ used in Fig. 1. When "reading" the diagrams, note that all arrows are functors with an object *and* a corresponding delta component. By tracing the commutative paths in the diagrams, equations for both objects and deltas can be compiled.

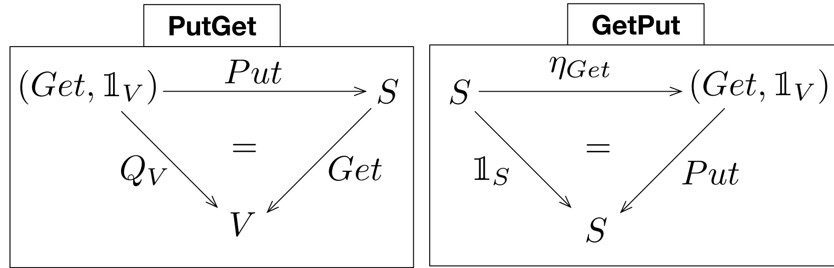

Figure 8: Delta-based lens laws

Example 9. Here we verify a concrete instance of $PutGet$ for the composers example by tracing the $PutGet$ diagram in Fig. 8 — both its object and delta parts — on the specific input shown in Fig. 4 and 6. For the object part, we start with the corner $(s, \delta_{vv'})$ on the left of Fig. 4, which is mapped to s' on the right of Fig. 4 by Put°. We then apply Get° to s', yielding $\{$ (*J.S. Bach*, *German*), (*C.P.E. Bach*, *German*) $\}$. If $PutGet$ holds, we should also get this state if we apply Q_V° to the input corner in Fig. 4, and indeed, what Q_V° does is extract the new view component v', which is the same as the state we arrive at through the Put ; Get path. For the delta part, the input is the pair of deltas $(\delta_{s\bar{s}}, \delta_{v'\bar{v}'})$ between the corner objects $(s, \delta_{vv'})$ and $(\bar{s}, \delta_{\bar{v}\bar{v}'})$ in Fig. 6. This delta pair is first mapped by $Put^{\rightarrow}$ to the source delta $Put^{\rightarrow}(\delta_{s\bar{s}}, \delta_{v'\bar{v}'})$ between $s' = Put^{\circ}(s, \delta_{vv'})$ and $\bar{s}' = Put^{\circ}(\bar{s}, \delta_{\bar{v}\bar{v}'})$, deleting *J.S. Bach*. This source delta is then mapped

by $Get^{\rightarrow}$ to the view delta between $Get^{\circ}(s') = \{ (\textit{J.S. Bach}, \textit{German}), (\textit{C.P.E. Bach}, \textit{German}) \}$ and $Get^{\circ}(\bar{s}') = \{ (\textit{C.P.E. Bach}, \textit{German}) \}$, also deleting *J.S. Bach*. As this view delta is precisely $\delta_{v'\bar{v}'}$, the result of applying $Q_V^{\rightarrow}$ to the input, the delta part of $PutGet$ is also verified.

5. Universal updates

We are now ready to argue from a practical point of view why the delta-based bx framework according to Johnson et al. could be a suitable formal basis for building bx tools. One point we have left open at the end of Sect. 3 is why the apparent complexity of $Put^{\rightarrow}$ is justified. Indeed, instead of just requiring mappings of view deltas to source deltas as one might expect, the framework requires information about pairs of view *and source* deltas as depicted in Fig. 5 and Fig. 6. One justification for this additional specification effort is that the resulting source deltas for the provided view deltas are *universal* in the sense that any other valid source delta (with respect to functoriality and the lens laws) can be obtained by composing the universal delta with another uniquely determined delta. Due to the uniqueness condition, universal deltas are essentially unique in the sense that they are all just different representations of the same change, so a bx tool is free to produce any representation of a universal delta without worrying about accidentally implementing a different change.

To explain this in more detail, let us *complete the square* formed by *Put*°. Up until now, we have refrained from connecting the new s' with the old source s and new view v' (see Fig. 3 and Fig. 4). We now fill in the missing arrows in the following (we refer the interested reader to Johnson et al. [3] for all technical details we skip). As depicted in Fig. 9 it is helpful to start with the corner $s \mapsto v \overset{id(v)}{\rightarrow} v$. As this corner is exactly how s is injected into the corner category by η_{Get}°, we can apply $GetPut$ on objects and surmise that the result of applying Put° to the corner must be s (as indicated in the figure). As this is the same s as before, we know that it maps via Get° to v. We now apply our actual view delta of interest $\delta_{vv'}$ to v to obtain v'. The corner $s \mapsto v \overset{\delta_{vv'}}{\rightarrow} v'$ maps to s' via Put°. To obtain information about the connection between s and s' we now extend $\delta_{vv'}$ to a pair of deltas between corners by pairing it with doing nothing to s, i.e., with $id(s)$ as depicted in Fig. 9. Providing this as input to $Put^{\rightarrow}$ yields $Put^{\rightarrow}(id(s), \delta_{vv'})$ connecting s and s'. After rotating the "cube" as indicated in the figure, we obtain our actual square of interest now with the explicit connection between s and s'. As a final step, we can argue with $PutGet^{\circ}$ that Get° applied to s' must yield v', as this is what is extracted via Q_V° from the input corner $s \mapsto v \overset{\delta_{vv'}}{\rightarrow} v'$ that yielded s' via Put°. This all leads to the final square depicted to the far right of Fig. 9. Indeed for Get and Put that obey all three delta-based lens laws, also referred to as "C-lenses" [3], one is free to think in terms of such squares representing how view deltas $\delta_{vv'}$ are to be *put* to source deltas $\delta_{ss'}$ computed using the provided Put as $Put^{\rightarrow}(id(s), \delta_{vv'})$.

So why do we need to know how to handle *pairs* of source and view deltas to corners as depicted in Fig. 5? Why not simply focus on the view deltas that are relevant for bx and only provide an implementation of $Put^{\rightarrow}$ for $\delta_{vv'}$ always paired with an unchanged source, i.e., $id(s)$? The answer is related to the quest for constructively/computationally justifying the choices of source deltas produced by Put. As depicted in Fig. 10, the source delta $Put^{\rightarrow}(id(s), \delta_{vv'})$ is *the*

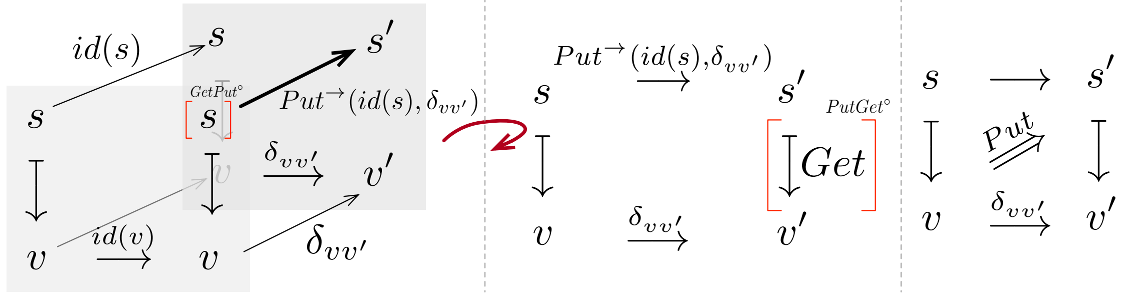

Figure 9: Filling in all arrows in the square

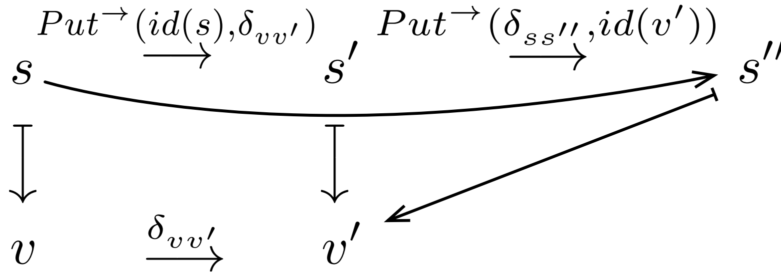

Figure 10: Source updates are least change updates

universal source delta that we (as bx developers) have decided to fix for our current choice of Put. This choice is justified in the sense that it still fully retains the possibilities for the user to get other valid results (in unique ways), because every other valid source delta $\delta_{ss''}$ (with respect to the lens laws) that also completes the square can be factored into the universal source delta and then $Put^{\rightarrow}(\delta_{s's''}, id(v'))$, where the latter represents explicitly the unique additional source delta that the user can apply to the updated source s' produced by the tool to get the source s'' that the user actually wants. As with any implementation of a universal property, the extra work of computing all other possibilities provides a computational "proof" for why a certain valid candidate is not *the* choice we have made. This does not only provide for formal clarity, but can also be used as the formal basis for *explaining* to users why their synchronisers have made certain choices and not others.

We should emphasise that universality is not always useful — whether and how it is useful depends crucially on the setting (in particular the allowed source deltas) in which we instantiate the abstract categorical definition of C-lenses. In some cases universality may guarantee nothing at all, whereas in some other cases universality can give rise to a highly intuitive and desirable property guaranteeing that a bx tool produces the *least* source change needed for a view change.

One setting where universality is not useful is the state-based setting, which, as we briefly mentioned in Sect. 3, can be seen as a special case of the delta-based setting where there is exactly one delta from any source/view to another, whose meaning is "discarding the original source/view and changing to the target source/view". In this setting, any source delta produced by Put is universal because whichever source is produced, it can always be further updated (in

a unique way) to any other source, in particular the one that the user actually wants.[4] Therefore, what universality guarantees is already implied by the underlying assumption of the setting, and becomes trivial.

We can argue a bit more generally that universality may become trivial when deltas can be undone by other deltas.[5] Intuitively, universality requires that a bx tool commit source changes cautiously so that it is still possible to get to other valid sources. If committed changes can be easily undone, however, then the tool does not need to be cautious — it can produce any valid delta $\delta_{ss'}$ and claim that it is universal, because when the user actually wants some other delta $\delta_{ss''}$ and invokes universality to request an additional delta $\delta_{s's''}$, the tool can simply produce a delta that undoes $\delta_{ss'}$ and then applies $\delta_{ss''}$. In the state-based setting, every delta can be seen as undoing (discarding) the effect of all previous deltas, so the above argument applies.

On the other hand, if we turn the above argument around, we can identify settings where universality is useful. For example, in a setting where we only allow deltas representing insertion into a database (and not deletion or any other operations that could undo the effect of insertion), a bx tool guaranteeing universality will have to produce exactly the absolutely necessary source insertions, i.e., the least change — if the tool inserted some redundant entry and the user wanted an updated database without that entry, then it would be impossible for the tool to produce a delta to delete the entry; so, to satisfy universality, the tool must not insert the entry in the first place. Real-world situations can be more delicate though, as the following example indicates.

Example 10. Back to the composers example, consider the instance in Fig. 4, where the view delta $\delta_{vv'}$ to be handled is an insertion. We can fill in the missing source delta $\delta_{ss'}$ between s and s' by setting $\delta_{ss'} = Put^{\rightarrow}(id(s), \delta_{vv'})$, which represents the insertion of (*C.P.E. Bach*, *???*, *German*) into the source multiset. As both $\delta_{vv'}$ and $\delta_{ss'}$ are insertions, this instance is valid in the insertion-only setting. Is $\delta_{ss'}$ universal in this setting? Let us try some other possible updated sources s'' and deltas $\delta_{ss''}$ and see if $\delta_{ss''}$ can be factored through $\delta_{ss'}$. One possibility of $\delta_{ss''}$ is redundantly inserting *C.P.E. Bach* twice, yielding the updated source $s'' = \{$ (*J.S. Bach*, *31/03/1685*, *German*), (*C.P.E. Bach*, *???*, *German*), (*C.P.E. Bach*, *???*, *German*) $\}$. In this case $\delta_{ss''}$ can indeed be factored as $\delta_{ss'}$; $\delta_{s's''}$ where $\delta_{s's''}$ represents the (redundant) insertion of the second *C.P.E. Bach*. So far so good, but we will see that the situation is in fact more delicate if we consider another possible source delta $\delta_{ss'''}$ that inserts (*C.P.E. Bach*, *08/03/1714*, *German*) which includes the specific date of birth, leading to the updated source $s''' = \{$ (*J.S. Bach*, *31/03/1685*, *German*), (*C.P.E. Bach*, *08/03/1714*, *German*) $\}$. This time $\delta_{ss'''}$ cannot be factored into $\delta_{ss'}; \delta_{s's'''}$ for any insertion delta $\delta_{s's'''}$, so $\delta_{ss'}$ is in fact not universal in the insertion-only setting. However, if we allow one additional kind of source delta that modifies a date of birth in our setting, then we can choose $\delta_{s's'''}$ to be the one that changes *C.P.E. Bach*'s date of birth from *???* to *08/03/1714*, making $\delta_{ss'}$ universal (in this instance). The Put behaviour is not fully determined by universality even in this amended setting though — we could have filled in any

[4] In a bit more detail: suppose that, in response to a view change, Put decides to change an original source s to a new source s', represented by the delta $\delta_{ss'}$ ("change to s'"); for any other delta $\delta_{ss''}$ ("change to s''") where s'' is also a valid result for the view change, we have $\delta_{ss''} = \delta_{ss'}; \delta_{s's''}$ ("changing to s'' is the same as changing to s' and then to s''"), where $\delta_{s's''}$ is automatically the unique way of updating s' to s'' in the state-based setting, so $\delta_{ss'}$ is universal.

[5] This may-statement is deliberately conservative because there is the subtle issue about the uniqueness condition in the definition of universality, which we do not touch from now on.

specific date of birth when we inserted *C.P.E. Bach* into s' and still remained universal, because that specific date can always be changed to something else later. While determining such choices *only up to isomorphism* (i.e., not caring about the specific initial date) can be advantageous, we could fully determine the Put behaviour by restricting the date-of-birth modification deltas to only those that change *???* to a specific date. To satisfy universality, Put must then use *???* for the dates of birth in newly-inserted entries, because *???* is the only date that can be changed to any other dates the user actually wants. For a more general handling of such "missing information" in structures, we refer the interested reader to Johnson et al. [16, 17] for the categorical modelling of databases with incomplete data.

6. Related work

According to Johnson and Rosebrugh [4], there are essentially three classes of lenses: state-based (set-based) lenses, delta-based lenses, and edit-based [18] lenses. We have discussed in this paper how state-based lenses can be viewed as a special case of delta-based lenses (for codiscrete categories). Edit lenses can also be represented as delta lenses via a suitable functor [4].

As state-based and edit lenses originate from the programming language community, they typically have a strong connection to practical tooling and integration in programming languages. In the following, we discuss the situation for delta lenses and the existing work that can be compared to our goal of bridging C-lenses and practical bx.

Diskin et al. [5] introduce D-lenses as a generalisation of the state-based bx framework to a delta-based setting. Compared to C-lenses, D-lenses do exactly what one would expect, namely map view deltas to source deltas. This means that C-lenses turn out to be essentially *special* D-lenses with the universal property discussed in Sect. 5; every C-lens is a D-lens but not every D-lens is a C-lens [4].

Concerning the practical relevance of D-lenses, Hermann et al. [12] have explored the relation between D-lenses [5] and Triple Graph Grammars (TGGs) [13]. This relation is further explored in a tutorial-like fashion by Anjorin [19] as part of the 2016 bx summer school [20]. The result of this line of work indicates that TGGs can be regarded as an implementation of D-lenses under a series of additional assumptions. In essence, a TGG rule can be directly interpreted as a pair of corresponding source and target deltas. An important point is that TGG rules are applied by checking for matches of their preconditions as graph *patterns*. This means that a relatively compact specification consisting of a few TGG rules actually describes an infinite number of concrete delta mappings for Put. While we do not (yet) see how TGG rules can be directly used to specify C-lenses, one could take an analogous rule-based approach that leverages graph pattern matching to enable a compact, finite specification of Get and Put. To specify how all possible pairs of deltas are to be handled for a specific corner, *amalgamated* TGG rules [21] could be explored; the fixed *kernel rule* would specify the universal choice, while multiple *complementary rules* would be used to describe how to handle all possible source deltas.

The challenging of guaranteeing *least* change synchronisers has been studied in detail by Cheney et al. [22]. While C-lenses certainly do not completely solve the least change problem, we believe they provide a helpful formal framework in which least change requirements can be systematically studied and precisely characterised.

Anjorin and Cheney [23] have started exploring connections between provenance and bx, with TGGs as a concrete bx language. While they identify *why*-provenance with TGG correspondence graphs, and *how*-provenance with the derivation sequence produced when applying TGG rules, *why-not*-provenance [24] is left open. As the universal property of C-lenses can be used to *explain* why a certain source delta is *not* produced by Put, this could be viewed as a means of providing *why-not* provenance.

For state-based tools/languages developed in the programming languages community, the formal guarantee about the behaviour of synchronisation programs is usually just well-behavedness, following the precedent of Foster et al.'s lens combinators [6]. However, it is clear that well-behavedness alone does not guarantee that users get the synchronisation behaviour they desire, as analysed by, for example, [9]. Consequently, these tools usually strive to provide a good range of features so as to be as expressive as possible, with one latest attempt being Matsuda and Wang's [10]; complementarily, Ko and Hu [9] developed a program logic with which users can reason about and determine the bidirectional behaviour of their programs. The overall direction is giving users adequate expressive power to write down the synchronisation programs they have in mind. For specific problem domains, however, one could argue that this expressive power is too low-level and becomes more a burden, and that users should instead be provided with higher-level or more *declarative* solutions where most, if not all, of the users' effort is devoted to describing just the synchronisation problems/requirements, as opposed to designing the synchronisation programs and managing the implementation details. But to be declarative, the tools must guarantee stronger properties from which users can deduce that the synchronisation behaviour is good enough for their purposes without looking into the implementation details. The universality of C-lenses could be a candidate for such properties: while universality can be trivial for e.g., codiscrete categories, it might be possible to impose more structure and restrictions on the source category in a way similar to Ex. 10. In such a setting where universality is meaningful, we could start all over again and work out a new generation of lens combinators that guarantee universality compositionally.

On a final note, some indication for the ubiquity of C-lenses and their potential for practical bx tooling can be observed by investigating existing applications that have essentially converged to C-lenses without being aware of this (see Johnson et al. [25] for a discussion of this). For example, the common view updating mechanisms used in the database world depend on so-called "constant complement updates" which can be viewed in a bx context as the Puts of C-lenses [26].

7. Conclusion and future work

In this paper, we provide a practical introduction to C-lenses for readers without any substantial background in category theory. We start with the accessible state-based approach to bx and generalise it step-by-step to a delta-based setting using multiple examples. We provide explanations and a practical discussion of the main concepts to help establish an intuitive understanding for C-lenses and their potential benefit as a formal basis for a new generation of bx tools.

While we already mention some preliminary ideas for how C-lenses could be specified in a practical bx tool, an open challenge to be addressed in the future is how the lens laws can be

automatically checked by such a tool. Finally, our primary focus in this paper was how formal results can be leveraged in a practical implementation. Conversely, there are also extensions to formal foundations that are yet to be worked out in full detail but are already required by practical approaches such as handling non-determinism and resolving conflicts in the context of *concurrent* synchronisation.

References

[1] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, J. Terwilliger, Bidirectional Transformations: A Cross-Discipline Perspective, in: R. F. Paige (Ed.), ICMT 2009, volume 5563 of *LNCS*, Springer, 2009, pp. 260–283.

[2] M. Johnson, R. D. Rosebrugh, Lens put-put laws: monotonic and mixed, Electron. Commun. Eur. Assoc. Softw. Sci. Technol. 49 (2012).

[3] M. Johnson, R. D. Rosebrugh, Delta lenses and opfibrations, Electron. Commun. Eur. Assoc. Softw. Sci. Technol. 57 (2013).

[4] M. Johnson, R. D. Rosebrugh, Unifying set-based, delta-based and edit-based lenses, in: A. Anjorin, J. Gibbons (Eds.), Bx 2016, volume 1571 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2016, pp. 1–13.

[5] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, F. Orejas, From state- to delta-based bidirectional model transformations: The symmetric case, in: J. Whittle, T. Clark, T. Kühne (Eds.), MODELS 2011, volume 6981 of *LNCS*, Springer, 2011, pp. 304–318.

[6] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt, Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, ACM Transactions on Programming Languages and Systems 29 (2007) 17:1–65.

[7] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, A. Schmitt, Boomerang: resourceful lenses for string data, in: G. C. Necula, P. Wadler (Eds.), POPL 2008, ACM, 2008, pp. 407–419.

[8] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Nakano, GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations, Progress in Informatics 10 (2013) 131–148.

[9] H.-S. Ko, Z. Hu, An axiomatic basis for bidirectional programming, Proceedings of the ACM on Programming Languages 2 (2018) 41:1–29.

[10] K. Matsuda, M. Wang, HOBiT: Programming lenses without using lens combinators, in: ESOP, volume 10801 of *LNCS*, Springer, 2018, pp. 31–59.

[11] A. Anjorin, T. Buchmann, B. Westfechtel, Z. Diskin, H. Ko, R. Eramo, G. Hinkel, L. Samimi-Dehkordi, A. Zündorf, Benchmarking bidirectional transformations: theory, implementation, application, and assessment, Softw. Syst. Model. 19 (2020) 647–691.

[12] F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, Y. Xiong, S. Gottmann, T. Engel, Model synchronization based on triple graph grammars: correctness, completeness and invertibility, Softw. Syst. Model. 14 (2015) 241–269.

[13] A. Schürr, Specification of graph translators with triple graph grammars, in: E. W. Mayr, G. Schmidt, G. Tinhofer (Eds.), WG ’94, volume 903 of *LNCS*, Springer, 1994, pp. 151–163.

[14] P. Stevens, J. McKinna, J. Cheney, Composers v0.1 in Bx Examples Repository, http://bx-community.wikidot.com/examples:home, 2010. Date retrieved: July 9, 2021.

[15] M. Johnson, R. D. Rosebrugh, Symmetric delta lenses and spans of asymmetric delta lenses, J. Object Technol. 16 (2017) 2:1–32.

[16] M. Johnson, R. Rosebrugh, Three approaches to partiality in the sketch data model, in: CATS, volume 78 of *ENTCS*, Elsevier, 2003, pp. 82–99.

[17] M. Johnson, S. Kasangian, A relational model of incomplete data without NULLs, in: CATS, volume 109 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society, 2010, pp. 89–94.

[18] M. Hofmann, B. C. Pierce, D. Wagner, Edit lenses, in: J. Field, M. Hicks (Eds.), POPL 2012, ACM, 2012, pp. 495–508.

[19] A. Anjorin, An introduction to triple graph grammars as an implementation of the delta-lens framework, in: J. Gibbons, P. Stevens (Eds.), Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures, volume 9715 of *LNCS*, Springer, 2016, pp. 29–72.

[20] J. Gibbons, P. Stevens (Eds.), Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures, volume 9715 of *LNCS*, Springer, 2018.

[21] E. Leblebici, A. Anjorin, A. Schürr, G. Taentzer, Multi-amalgamated triple graph grammars: Formal foundation and application to visual language translation, J. Vis. Lang. Comput. 42 (2017) 99–121.

[22] J. Cheney, J. Gibbons, J. McKinna, P. Stevens, On principles of least change and least surprise for bidirectional transformations, J. Object Technol. 16 (2017) 3:1–31.

[23] A. Anjorin, J. Cheney, Provenance meets bidirectional transformations, in: TaPP 2019, USENIX Association, 2019.

[24] A. Chapman, H. V. Jagadish, Why not?, in: U. Çetintemel, S. B. Zdonik, D. Kossmann, N. Tatbul (Eds.), SIGMOD 2009, ACM, 2009, pp. 523–534.

[25] M. Johnson, R. D. Rosebrugh, R. J. Wood, Lenses, fibrations and universal translations, Math. Struct. Comput. Sci. 22 (2012) 25–42.

[26] M. Johnson, R. Rosebrugh, Constant complements, reversibility and universal view updates, in: J. Meseguer, G. Roşu (Eds.), Algebraic Methodology and Software Technology, Springer Berlin Heidelberg, 2008, pp. 238–252.