Coalgebraic Aspects of Bidirectional Computation

Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Coalgebraic Aspects of Bidirectional Computation

Faris Abou-Saleh[b] James McKinna[a] Jeremy Gibbons[b]

a. Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh

b. Department of Computer Science, University of Oxford

Abstract We have previously shown that several state-based bx formalisms can be captured using monadic functional programming, using the state monad together with possibly other monadic effects, giving rise to structures we have called monadic bx (mbx). In this paper, we develop a coalgebraic theory of state-based bx, and relate the resulting coalgebraic structures (cbx) to mbx. We show that cbx support a notion of composition coherent with, but conceptually simpler than, our previous mbx definition. Coalgebraic bisimulation yields a natural notion of behavioural equivalence on cbx, which respects composition, and essentially includes symmetric lens equivalence as a special case. Finally, we speculate on the applications of this coalgebraic perspective to other bx constructions and formalisms.

Keywords bidirectional transformation, lens, monads, effects, bisimulation

1 Introduction

Many scenarios in computer science involve multiple, partially overlapping, representations of the same data, such that whenever one representation is modified, the others must be updated in order to maintain consistency. Such scenarios arise for example in the context of model-driven software development, databases and string parsing [CFH+09]. Various formalisms, collectively known as *bidirectional transformations* (bx), have been developed to study them, including asymmetric and symmetric lenses [FGM+07, HPW11], relational bx [Ste10], and triple-graph grammars [SK08].

In recent years, there has been a drive to understand the similarities and differences between these formalisms [HSST11]; and a few attempts have been made to give a unified treatment. In previous work [CMS+14, ASCG+15] we outlined a unified theory, with examples, of various accounts of bx in the literature, in terms of computations defined monadically using Haskell's **do**-notation. The idea is to interpret a bx between two data sources A and B (subject to some consistency relation $R \subseteq A \times B$) relative to some monad M representing computational effects, in terms of monadic *get* and *set* operations which allow lookups and updates on both A and B while maintaining R. We

Faris Abou-Saleh, James McKinna, Jeremy Gibbons. *Coalgebraic Aspects of Bidirectional Computation*.

In Journal of Object Technology, vol. V, no. N, 2011, pages M:1–0. doi:10.5381/jot.201Y.VV.N.aN

defined state-based bx with effects in terms of these four operations, subject to several equations in line with the Plotkin–Power equational theory of state [PP02]. The key difference is that in a bidirectional context, the sources A and B are interdependent, or *entangled*: updating (or 'setting') A should in general affect B, and vice versa. Thus we must abandon some of the Plotkin–Power equations; in particular, one no longer expects A-updates to commute with B-updates. To distinguish our earlier monadic account of bx from the coalgebraic treatment to be developed in this paper, we will refer to them as *monadic bx*, or simply *mbx*.

We showed that several well-known bx formalisms may be described by monadic bx for the particular case of the *state monad*, $M_S\ X = S \to (X \times S)$, called *State S* in Haskell. We focused attention on the particular case of monadic bx for the *monad transformer* counterpart T_S^M to M_S (called *StateT S M* in Haskell), where $T_S^M\ X = S \to M\ (X \times S)$ builds on some monad M, such as I/O. We defined composition $t_1 \fatsemi t_2$ for such mbx with *transparent get* operations – i.e. *get*s which are effect-free and do not modify the state. The definition is in terms of *StateT*-monad morphisms derived from lenses (see Section 5), adapting work of Shkaravska [Shk05]. As with symmetric lenses, composition can only be well-behaved up to some notion of equivalence, due to the different state-spaces involved. The natural choice of equivalence in a monadic context is defined by monad morphisms, and encodes an isomorphism between the different state-spaces. We showed that our definition of composition was associative and had identities up to these state-space isomorphisms.

In this paper, we present a coalgebraic treatment of our earlier work on monadic bx, inspired by Power and Shkaravska's work on variable arrays and comodels [PS04] for the theory of mutable state, defined in terms of the costore comonad $S \times (-)^S$. This coalgebraic perspective on bx provides a conceptual clarification to the more high-level, monadic framework of our previous work on monadic bx. Firstly, all our instances of the earlier formalism have an underlying state-space (in particular, it is fiendishly subtle to define monadic bx composition enjoying the expected properties without this restriction, in addition to transparent *get*s as introduced above). By restricting attention to such models from the outset, the exposition of our ideas is simplified; there are natural definitions of bx initialisation and composition.

More importantly, it allows us to improve on our earlier notion of equivalence given by state-space isomorphism, appealing instead to the theory of coalgebraic bisimilarity [Rut00]. It is well known that bisimulation ('observational equivalence') is a better tool for reasoning about behaviour than state-space isomorphism ('implementation equivalence'). We illustrate this for effectful bx in Examples 3.1 and 3.8. Furthermore, coalgebraic bisimilarity enjoys a closer fit with the equivalence on symmetric lenses considered by Hofmann et al. [HPW11]; we show the precise relationship in Proposition 3.9 below. Finally, we have relaxed the set-based setting, and definition of mbx composition, into a more general categorical treatment.

The technical contributions and structure of this paper are as follows. Firstly, in Section 2 we motivate a coalgebraic perspective on bx, and identify a suitable categorical setting for this interpretation. In Section 3, we introduce an equivalence on coalgebras, namely pointed coalgebraic bisimulation, and demonstrate how this equivalence relates to that of symmetric lenses, and also allows us to model various bx scenarios incorporating effects. (Pointedness identifies initial states, with respect to which coalgebra behaviours are compared.) In Section 4 we give a detailed account of composition of coalgebraic bx in terms of pullbacks, which is both more direct and categorically general than our earlier definition [ASCG$^+$15], highlighting subtleties

in the definitions (such as Remark 4.8). We prove that our coalgebraic notion of composition is associative, and has identities, up to pointed bisimulation (Theorem 4.10). Finally, in Section 5 we show that coalgebraic bx composition is coherent with that for monadic bx [ASCG+15].

2 Coalgebraic bx

We begin by introducing coalgebras, motivating and sketching a coalgebraic view of bidirectional transformations, before the formal definitions in the following section.

2.1 Stateful Systems are Coalgebras

Coalgebras are a natural model of state-based systems in computer science [Rut00]. Typically, such systems exhibit some kind of behaviour observed by a user (or the ambient environment) as they interact with the system. One often wishes to abstract away from the internal details of such systems, and concentrate instead on modelling, and reasoning about, their observable behaviour – taking a 'black box' perspective. Coalgebra provides a high-level, abstract framework for modelling such systems.

This perspective offers several benefits. First of all, it is very concise, largely specified by a few ingredients. Secondly, it is general; one may apply coalgebraic methods to a wide class of systems, simply by adjusting these ingredients accordingly. Thirdly, it provides pre-existing concepts and tools for reasoning about the behaviour of these systems – such as natural candidate definitions of what it means for two systems to exhibit 'the same behaviour', and formal methods for proving this equivalence.

The key elements of our coalgebraic model are as follows. A *behaviour functor* F^M_{AB} specifies the kind of behaviour we expect systems to have – their 'public interface'. Each system has a hidden *state-space* X; then $F^M_{AB}\ X$ describes the possible behaviours we can observe over that state-space X. A particular system is described by a *coalgebra-structure* $X \to F^M_{AB}\ X$ indicating the observable behaviour of each internal state. All of this description is relative to an underlying *category* $\mathbb{C}$, which provides concrete meanings to the above symbols – in particular, the kind of *objects* X under study, and the *morphisms* $X \to Y$ between them (and hence indirectly the functor F^M_{AB}).

Each behaviour functor F^M_{AB} comes with a natural notion of *behavioural equivalence*, identifying what it means for pairs of states in X and Y (state-spaces of two given coalgebras) to be indistinguishable to any observer. This correspondence is captured by *coalgebraic bisimulations* – pairs of morphisms $X \leftarrow R \to Y$, picking out pairs of indistinguishable states. We ensure the morphisms match equivalent behaviours by requiring them to be *coalgebra morphisms*.

We now apply this perspective to the bx formalism studied in our previous work [ASCG+15]. For simplicity, we consider the effect-free case first, taking $M = Id$. A *monadic bx* between two given data sources A and B, with state-space X, has the following public interface. In any particular state of the bx, a user may request to observe or *get* the current values of A or B; they may also *update* the value of either A or B. Updates moreover should restore the consistency relation R between A and B, by computing a new bx state in X.

The *get* operations are described by morphisms $X \to A$ and $X \to B$, indicating for each state X what the observed value of A or B will be. We call these get_L and get_R respectively. The updates correspond to functions $X \times A \to X$ and $X \times B \to X$, indicating for each initial state and 'new' A or B, what the 'new' state X will be. It

is convenient to curry these functions into the form $X \to X^A$ and $X \to X^B$; we call these set_L and set_R respectively.

In practice, during updates some computation may be required to restore consistency to the state X. Such computations may be represented using a *monad* M (see Section 2.3). Thus the types of the update functions become $X \to (MX)^A$ and $X \to (MX)^B$; rather than merely returning a 'new' state X, they now return a computation MX that yields such a state.

The four operations get_L, get_R, set_L, set_R may now be combined into a single function, describing a particular example of such behaviour:

$$\alpha : X \to A \times B \times (MX)^A \times (MX)^B$$

and the right-hand side describes the behaviour functor F^M_{AB} (see Definition 2.3).

Note that in this presentation, the choice of $A \times B$, rather than $MA \times MB$, makes clear that the *get* operations are pure functions of the coalgebra state – this is the essence of "transparent" monadic bx as previously introduced by us [ASCG+15].

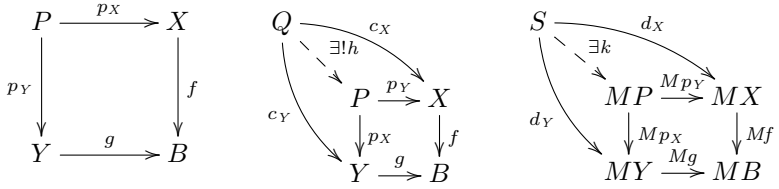
2.2 Categorical Prerequisites

Many semantic models of bx are set-theoretic; in our terminology, they are framed in the category Set, where the objects X are sets, and morphisms $X \to Y$ are functions. However, more refined models such as delta-based bx, are likely to require richer categories [DXC11, JR13]. Hence, we do not assume a fixed category $\mathbb{C}$, but rather identify the structure $\mathbb{C}$ will require for the ensuing definitions to work. The reader may consult Moggi [Mog91] and Rutten [Rut00] for more categorical and coalgebraic detail respectively.

1. We will assume $\mathbb{C}$ is a Cartesian closed category – i.e. with finite products $X \times Y$, and exponentials Y^X (generalising from Set to arbitrary categories the idea of a function-space from X to Y) – on which a strong monad M is defined. This allows us to make free use of the equational theory of **do**-notation as the internal language of the Kleisli category $\mathsf{Kl}(M)$ [GH11, for further detail]. Rather than using pointfree strength and associativity isomorphisms, **do**-notation is convenient for representing value-passing between functions using pointwise syntax.

2. In order to define coalgebraic bx composition in Section 4, we further require that $\mathbb{C}$ have *pullbacks*, and that M *weakly preserve them* [Gum01]; we say "M is *wpp*", for short. The following diagrams make this more explicit. Recall that a pullback of two arrows $f : X \to B$ and $g : Y \to B$ is an object P and span of arrows $p_X : P \to X$, $p_Y : P \to Y$ making the below-left diagram commute, such that for any object Q and span c_X, c_Y making the outermost face in the middle diagram commute, there is a *unique* arrow h making the whole diagram commute. Finally, the *wpp* property (for M) asserts that the span Mp_X, Mp_Y forms a *weak* pullback of Mf and Mg: for any object S and span d_X, d_Y making the outermost face in the right-hand diagram commute, there is an arrow k, *not*

necessarily unique, making the whole diagram commute.



3. For technical reasons (in Section 4), we assume that the *return* of the monad M is *monic*, or a *mono(morphism)* (left-cancellable): $return \circ f = return \circ g$ implies $f = g$. This allows us to observe the value x wrapped in an effect-free computation $return\ x$. Most computational monads have this property: *e.g.* global state, I/O, non-determinism, and exceptions [Mog91]; we are unaware of natural counterexamples.

4. Lastly, we assume an object I, such that arrows $x : I \to X$ represent 'elements' x of an object X. Typically, as in Set, I will be the terminal object 1.

Remark 2.1. *The wpp condition lets us consider (at least for $\mathbb{C} = \mathsf{Set}$) monads M of computational interest such as (probabilistic) non-determinism [Sok11, Gum09], which are wpp but do not preserve pullbacks; more generally, we can include I/O, exceptions, and monoid actions, by appealing to a simple criterion to check that wpp holds for such M [Gum01, Theorem 2.8].*

2.3 Bx as Pointed Coalgebras

We now give a coalgebraic description of bx, i.e. as state-based systems. We begin by noting that many bx formalisms, such as (a)symmetric lenses and relational bx, often involve an *initialised* state. The behaviours of two such bx are compared relative to their initial states. Hence, to reason about such behaviour, throughout this paper we concern ourselves with *pointed* coalgebras with designated initial state. Coalgebras with the same structure, but different initial states, are considered distinct [AMMS13, for more general considerations]. Corollary 4.13 makes explicit the categorical structure of bx represented by such structures.

Definition 2.2. *For any endofunctor F on a category $\mathbb{C}$, a* pointed F-coalgebra *is a triple $(X, \alpha, \epsilon_\alpha)$ consisting of: an object X of $\mathbb{C}$, the* carrier *or* state-space*; an arrow $\alpha : X \to FX$, its* structure map *or simply* structure*; and an arrow $\epsilon_\alpha : I \to X$, picking out a distinguished initial state. We abuse notation, often writing α as a synecdoche for the pointed coalgebra itself.*

Now we define the behaviour functors we use to describe bx coalgebraically; as anticipated above, we incorporate a monad M into the definition from the outset to capture effects, although for many example classes, such as symmetric lenses, it suffices to take $M = Id$, the identity monad. Here are several example settings, involving more interesting monads, to which we return in Section 3.2. For simplicity, we assume the examples are in the category Set; to model such behaviour in other categories $\mathbb{C}$, one would have to assume structure relevant to the example (for instance, existence of the greatest fixpoint defined in the case of interactive I/O, and binary coproducts for failure).

- **Interactive I/O.** In previous work [ASCG+15] we gave an example of an mbx which notifies the user whenever an update occurred. Extending this example further, after an update to A or B, in order to restore consistency (as specified e.g. by a consistency relation $R \subseteq A \times B$) the bx might prompt the user for a new B or A value respectively until a consistent value is supplied. Such behaviour may be described in terms of a simplified version of the *I/O monad* $MX = \nu Y \,.\, X + (O \times Y) + Y^I$ given by a set O of observable output labels, and a set of inputs I that the user can provide. Note that the use of the greatest (ν) fixpoint permits possibly non-terminating I/O interactions.
- **Failure.** Sometimes it may be simply impossible to propagate an update on A across to B, or vice versa; there is no way to restore consistency. In this case, the update request should simply fail; and we may model this with the *Maybe* monad $MX = 1 + X$.
- **Non-determinism.** There may be more than one way of restoring consistency between A and B after an update. In this case, rather than prompting the user at every such instance, it may be preferable for a non-deterministic choice to be made. We may model this situation by taking the monad M to be the (finitary) *powerset monad*.

Definition 2.3. *For objects A and B, we define the behaviour functor*

$$F^M_{AB}\ (-) = A \times B \times (M\ (-))^A \times (M\ (-))^B.$$

By taking projections of a structure map $\alpha : X \to F^M_{AB}\ X$, we recover the bx operations outlined in Section 2.1: $get_L : X \to A$, $get_R : X \to B$, $set_L : X \to (MX)^A$, and $set_R : X \to (MX)^B$.

Convention 2.4. *Given $\alpha : X \to F^M_{AB}\ X$, we write $\alpha.get_L : X \to A$, and $\alpha.set_L : X \to (MX)^A$, for the corresponding projections, called 'left'- or L-operations, and similarly $\alpha.get_R : X \to B$, $\alpha.set_R : X \to (MX)^B$ for the other projections, called R-operations. Where α may be inferred, and we wish to draw attention to the carrier X, we also write $x.get_L$ for $\alpha.get_L\ x$, and similarly for the other L-, R-operations.*

To ensure that pointed F^M_{AB}-coalgebras provide sensible implementations of reading and writing to A and B, we impose laws restricting their behaviour. We call such well-behaved coalgebras *coalgebraic bx*, or cbx.

Definition 2.5. *A* coalgebraic bx *is a pointed F^M_{AB}-coalgebra $(X, \alpha : X \to F^M_{AB}\ X, \epsilon_\alpha)$ for which the following laws hold at L (writing $x.get_L$ for $\alpha.get_L\ x$, etc.):*

$$\begin{aligned}
&(\mathsf{CGetSet}_L)\ (\alpha): && x.set_L\ (x.get_L) = return\ x \\
&(\mathsf{CSetGet}_L)\ (\alpha): && \mathbf{do}\ \{x' \leftarrow x.set_L\ a; return\ (x', x'.get_L)\} \\
& && = \mathbf{do}\ \{x' \leftarrow x.set_L\ a; return\ (x', a)\}
\end{aligned}$$

and the corresponding laws $(\mathsf{CSetGet}_R)$ and $(\mathsf{CGetSet}_R)$ hold at R.

We typically refer to a cbx by its structure map, and simply write $\alpha : A \nrightarrow_X B$, where we may further omit explicit mention of X.

These laws are the analogues of the (GS), (SG) laws [ASCG+15] which generalise those for well-behaved lenses [FGM+07, see also Section 5.2 below]. The generalisation

is that the ($\mathsf{CSetGet}_L$) law explicitly returns pairs (x', a) of the new state x' and the a that was set (and similarly for the R law); this extra information is required in the proof of Proposition 5.1 establishing the relationship between cbx and our earlier mbx formalism. An analogous strengthening (Remark 4.8) is also required to correctly define cbx composition. The laws also correspond to a subset of the laws for coalgebras of the costore comonad $S \times (-)^S$, but excluding the analogue of 'Put-Put' or very-well-behavedness of lenses [GJ12].

Here is a simple example of a cbx, which will provide identities for cbx composition as defined in Section 4 below. Note that there is a separate identity cbx for each *pair* of an object A and initial value $e : A$, and that the definition is for any choice of M.

Example 2.6. *Given $(A, e : A)$, there is a trivial cbx structure $\iota(e) : A \nrightarrow_A A$ defined by $\epsilon_{\iota(e)} = e$; $a.get_L = a = a.get_R$; $a.set_L\ a' = return\ a' = a.set_R\ a'$.*

Remark 2.7. *Our definition does not make explicit any consistency relation $R \subseteq A \times B$ on the observable A and B values; however, one obtains such a relation from the get functions applied to all possible states, viz. $R = \{(a, b) : \exists x\,.\, get_L\ x = a \wedge get_R\ x = b\}$. One may then show that well-behaved cbx do maintain consistency with respect to R.*

3 Behavioural Equivalence and Bisimulation

In this section, we introduce the notion of pointed coalgebraic bisimulation, which defines a behavioural equivalence $\equiv$ for pointed cbx. In Section 3.1 we compare this equivalence to the established notion of equivalence for symmetric lenses. We then discuss in Section 3.2 the behavioural equivalences induced for the classes of effectful cbx described in Section 2.3: interactive I/O, failure, and non-determinism.

We begin with a simple illustration that state-space isomorphism is not adequate for comparing cbx behaviour, before giving the definitions leading up to bisimulation.

Example 3.1. *In* Set*, take M to be the (finitary) powerset monad, for non-determinism. Consider the following cbx $l_1, l_2 : \mathbb{Z} \nrightarrow \mathbb{Z}$ on the integers. l_1 is essentially the identity cbx of Example 2.6, with state-space $X = \mathbb{Z}$, initial state 0, and operations:*

$$x.get_L = x.get_R = x \qquad x.set_L\ x' = x.set_R\ x' = return\ x'$$

l_2 is similar, but with state-space $Y = (\mathbb{Z} \times \mathbb{B})$ incorporating a boolean $\mathbb{B} = \{0, 1\}$. The initial state is $(0, 0)$, and the operations are:

$$(x, b).get_L = (x, b).get_R = x \qquad (x, b).set_L\ x' = (x, b).set_R\ x' = \{(x', 0), (x', 1)\}$$

The set operations overwrite the integer and non-deterministically change the boolean. Clearly the state-spaces X and Y are not isomorphic, and yet there is no way to distinguish the behaviour of l_1 and l_2. We will prove they are bisimilar in Example 3.8.

Definition 3.2. *A* pointed $(F$-)coalgebra morphism *h between pointed coalgebras $(X, \alpha, \epsilon_\alpha)$ and $(Y, \beta, \epsilon_\beta)$ is a map $h : X \to Y$ such that $\beta \circ h = Fh \circ \alpha$ and $h \circ \epsilon_\alpha = \epsilon_\beta$.*

Remark 3.3. *In terms of* **do** *notation, $h : X \to Y$ being an F^M_{AB}-coalgebra morphism between α and β is equivalent to the following laws (where we again write $x.set_L$ for $\alpha.set_L\ x$, and so on), and a pair of similar laws for the R-operations:*

$$\begin{aligned}
&(\mathsf{CGetP}_L)(h): && x.get_L = (h\ x).get_L \\
&(\mathsf{CSetP}_L)(h): && \mathbf{do}\ \{x' \leftarrow x.set_L\ a; return\ (h\ x')\} \\
& && = \mathbf{do}\ \{\mathbf{let}\ y = (h\ x); y' \leftarrow y.set_L\ a; return\ y'\}
\end{aligned}$$

We now present a modest generalisation to $\mathbb{C}$ of the standard Set-based definition of (coalgebraic) bisimulation relations [TP97]. (Since we are concerned only with the *existence* of bisimulations between X and Y, we may consider them to be given non-uniquely by some jointly monic pair, as follows.)

Definition 3.4. *A* bisimulation between pointed coalgebras α and β *is a tuple* (ζ, p, q) *of a pointed coalgebra* ζ *and pointed coalgebra morphisms* $p : \zeta \to \alpha$, $q : \zeta \to \beta$ *which is* jointly monic *(in* $\mathbb{C}$*): i.e.* $p \circ f = p \circ f'$ *and* $q \circ f = q \circ f'$ *implies* $f = f'$.

Pointed bisimulation is trivially symmetric. It is also transitive, because of our assumptions on F^M_{AB} preserving weak pullbacks [Rut00, Theorem 5.4]. Intuitively, a pointed bisimulation provides explicit correspondences between pairs of observationally indistinguishable states of two cbx – at least, states reachable from the initial state. This is the notion of behavioural equivalence adopted in this paper. In general, there are other possibilities – for instance, *coalgebraic bisimilarity* is given by the *largest* bisimulation relation (suitably formulated), which identifies as many pairs of indistinguishable states as possible. Another possibility is so-called *observational equivalence* of coalgebras α and β, which requires two coalgebra morphisms *into* some coalgebra ζ (i.e. the reverse of Definition 3.4, without joint monicity). However, all of these notions are equivalent if the behaviour functor, like F^M_{AB}, preserves weak pullbacks, so we do not dwell on this point further.

As a concrete example, Definition 3.4 characterises bisimulation for F^{Id}_{AB}-coalgebras (i.e. $M = Id$) in the setting $\mathbb{C} = \mathsf{Set}$ as follows. Abusing notation, in Set we write $(\epsilon_\alpha, \epsilon_\beta)$ for the element of R picked out by the arrow $\langle \epsilon_\alpha, \epsilon_\beta \rangle$.

Proposition 3.5. *A pointed* F^{Id}_{AB}*-bisimulation* (ζ, p, q) *on a pair of coalgebraic bxs* α, β *is equivalent to a relation* $R \subseteq X \times Y$ *such that* $(\epsilon_\alpha, \epsilon_\beta) \in R$*, and* $(x, y) \in R$ *implies, for all* $a : A$ *and* $b : B$*,*

- $x.get_L = y.get_L$ *and* $x.get_R = y.get_R$*;*
- $(x.set_L\ a, y.set_L\ a) \in R$*, and* $(x.set_R\ b, y.set_R\ b) \in R$*.*

Proof. First, note that in Set, a relation $R \subseteq (X \times Y)$ induces a jointly monic span $p : X \leftarrow S \to Y : q$ – simply by taking its projections into X and Y, and taking $S = R$. Conversely, such a span induces a relation $R \subseteq (X \times Y)$, where we define $(x, y) \in R$ iff there exists $s \in S$ such that $p\ s = x$ and $q\ s = y$. Now consider a pointed bisimulation (ζ, p, q), and suppose that $(x, y) \in R$, or equivalently, there is some $s \in S$ such that $p\ s = x$ and $q\ s = y$. Definition 3.2 ($M = Id$) tells us that for all a,

$$\begin{array}{ll} s.get_L = (p\ s).get_L = x.get_L & p\ (s.set_L\ a) = (p\ s).set_L\ a = x.set_L\ a \\ s.get_L = (q\ s).get_L = y.get_L & q\ (s.set_L\ a) = (q\ s).set_L\ a = y.set_L\ a \end{array}$$

from which we deduce that $x.get_L = y.get_L$, and that $(x.set_L\ a, y.set_L\ a) \in R$. The right-hand operations are similar. □

Definition 3.6. *We say that two cbx* α, α' *are* behaviourally equivalent*, and write* $\alpha \equiv \alpha'$*, if there exists a pointed coalgebraic bisimulation* (ζ, p, q) *between* α *and* α'*.*

The following fact will be useful (e.g. proving Theorem 4.10 identities, associativity).

Remark 3.7. *A pointed coalgebra morphism* h *from* α *to* α' *yields a bisimulation, by taking* $(\zeta, p, q) = (\alpha, id, h)$*, and hence* $\alpha \equiv \alpha'$*.*

Example 3.8. *Consider the non-deterministic cbx l_1, l_2 of Example 3.1. We show that l_1 and l_2 are bisimilar. By Remark 3.7, it suffices to show that $\pi_1 : \mathbb{Z} \times \mathbb{B} \to \mathbb{Z}$ is a pointed coalgebra morphism from l_2 to l_1. Firstly, π_1 respects initial states: $\pi_1\ (0,0) = 0$. Now we appeal to Remark 3.3 and show $(\mathsf{CGetP}_L)(\pi_1)$, $(\mathsf{CSetP}_L)(\pi_1)$. The R-laws are similar.*

$$(\mathsf{CGetP}_L)(\pi_1): \quad (x,b).get_L = x = x.get_L = (\pi_1\ (x,b)).get_L$$

$(\mathsf{CSetP}_L)(\pi_1)$:
do $\{(x',b') \leftarrow x.set_L\ x'; \mathit{return}\ (\pi_1\ (x',b'))\}$
$=$ $[\![$ *definitions of* $l_2.set_L$ *and projection* π_1 $]\!]$
do $\{(x',b') \leftarrow \{(x',0),(x',1)\}; \mathit{return}\ x'\}$
$=$ $[\![$ *laws of non-determinism* $]\!]$
return x'
$=$ $[\![$ *redundant* **let** $]\!]$
do $\{\mathbf{let}\ x_0 = \pi_1\ (x,b); \mathit{return}\ x'\}$
$=$ $[\![$ *definition of* $l_1.set_L$ $]\!]$
do $\{\mathbf{let}\ x_0 = \pi_1\ (x,b); x_0.set_L\ x'\}$

3.1 Relationship with Symmetric Lens Equivalence

In this subsection, we describe symmetric lenses (SL) [HPW11] in terms of cbx, and relate pointed bisimilarity between cbx and symmetric lens (SL-)equivalence [HPW11, Definition 3.2]. First of all, it is straightforward to describe as a cbx a symmetric lens between A and B with complement C – given by a pair of functions $putr : A \times C \to B \times C$, $putl : B \times C \to A \times C$ and initial state ϵ_C together satisfying two laws: take $M = Id$ and state-space $X = A \times C \times B$, encapsulating the current value of the lens complement C, as well as those of A and B (*cf.* [CMS+14, Section 4]). We now define the analogues of the SL-operations for a cbx between A and B:

$$x.put_L : A \to (B \times X) \qquad x.put_L\ a = \mathbf{let}\ x' = x.set_L\ a\ \mathbf{in}\ (x'.get_R, x')$$
$$x.put_R : B \to (A \times X) \qquad x.put_R\ b = \mathbf{let}\ x' = x.set_R\ b\ \mathbf{in}\ (x'.get_L, x')$$

(Note that this is the opposite *L-R* convention from that of Hofmann et al. [HPW11].)

Proposition 3.9. *Taking $\mathbb{C} = \mathsf{Set}$, a pointed F_{AB}^{Id}-bisimulation between cbxs α, β is equivalent to a relation $R \subseteq X \times Y$ such that $(\epsilon_\alpha, \epsilon_\beta) \in R$, and $(x,y) \in R$ implies:*

- *$x.get_L = y.get_L$ and $x.get_R = y.get_R$;*
- *for all $a : A$, $x.put_L\ a = (b', x')$ and $y.put_L\ a = (b', y')$ for some b' and $(x', y') \in R$;*
- *for all $b : B$, $x.put_R\ b = (a', x')$ and $y.put_R\ b = (a', y')$ for some a' and $(x', y') \in R$.*

Proof. We show that Proposition 3.5 implies the above; the converse is similar, and we omit it. Suppose $(x,y) \in R$. Proposition 3.5 immediately tells us that $x.get_L = y.get_L$ and $x.get_R = y.get_R$, giving the first bullet. To show the second bullet, let $x.put_L\ a = (x', b_1)$ and $y.put_L\ a = (y', b_2)$. By definition, $x' = x.set_L\ a$, $b' = x'.get_L$, and $y' = y.set_L\ a$. Proposition 3.5 again tells us that $(x', y') \in R$, and $b_1 = x'.get_L = y'.get_L = b_2$. The third bullet is exactly symmetric. □

This shows that a bisimulation between cbx is effectively the same as an equivalence between SLs, whose definition only differs from R in Proposition 3.9 by not mentioning *get* operations. In a cbx, we are able to observe the 'current' values of A and B in any given state, via the *get* functions. This information is implicitly present in SL-equivalence, where a sequence of *putr* or *putl* operations amounts to a sequence of *set*s to A and B, but where we cannot observe which values have been set. Here, the *get* operations make this information explicit. We say more about the categorical relationship between cbx and SLs in Corollary 4.14 below.

3.2 Coalgebraic Bisimilarity with Effects

By introducing effects through M, our coalgebraic definition of behavioural equivalence applies to a wide class of effectful behaviours in a uniform manner, and we illustrate with the examples anticipated in Section 2.3. As mentioned before, for concreteness we consider the special case $\mathbb{C} = \mathsf{Set}$, although the examples apply in any other categories with the necessary structure for the definitions.

3.2.1 Interactive I/O

We take $MX = \nu Y \,.\, X + (O \times Y) + Y^I$, where O is a given set of observable outputs, and I inputs the user can provide. The components of the disjoint union induce monadic $\mathsf{return} : X \to MX$ and algebraic operations $\mathsf{out} : O \times MX \to MX$ and $\mathsf{in} : (I \to MX) \to MX$ (*cf.* [PP02]). In the context of cbx that exhibit I/O effects in this way, an operation like $set_L : X \to (MX)^A$ maps a state $x : X$ and an A-value $a : A$ to a value $m : MX$, where m describes some path in an (unbounded) tree of I/O actions, either terminating eventually and returning a new state in X, or diverging, depending on the user's input.

One may characterise pointed bisimulations on such cbx as follows. Intuitively, behaviourally equivalent states must 'exhibit the same observable I/O activity' during updates set_L and set_R, and subsequently arrive at behaviourally equivalent states. To formalise this notion of I/O activity, we need an auxiliary definition (which derives from the greatest-fixpoint definition of M):

Definition 3.10. *With respect to an I/O monad M and a relation $R \subseteq X \times Y$, the I/O-equivalence relation $\sim_R \subseteq MX \times MY$ induced by R is the greatest fixpoint of the operation Φ mapping a relation $S \subseteq MX \times MY$ to the relation $\Phi(S)$ such that $(m, n) \in \Phi(S)$ if and only if any of the following hold:*

- *$m = \mathsf{return}\ x$, $n = \mathsf{return}\ y$, and $(x, y) \in R$ for some x, y; or*
- *$m = \mathsf{out}\ (o, m')$ and $n = \mathsf{out}\ (o, n')$ for some $o : O$ and $(m', n') \in S$; or*
- *$m = \mathsf{in}\ (\lambda i \to m(i))$ and $n = \mathsf{in}\ (\lambda i \to n(i))$, where $(m(i), n(i)) \in S$ for all $i : I$.*

One may now show that a pointed F^M_{AB}-bisimulation R on a pair of such cbxs α, β is equivalent to a relation $R \subseteq X \times Y$ such that $(\epsilon_\alpha, \epsilon_\beta) \in R$, and $(x, y) \in R$ implies

- $x.get_L = y.get_L$ and $x.get_R = y.get_R$;
- for all $a : A$ and $b : B$, $(x.set_L\ a) \sim_R (y.set_L\ a)$ and $(x.set_R\ b) \sim_R (y.set_R\ b)$.

Such an equivalence guarantees that, following any sequence of updates in α or β, the user experiences exactly the same sequence of I/O actions; and when the sequence is complete, they observe the same values of A and B for either cbx. Thus, pointed bisimulation asserts that α, β are indistinguishable from the user's point of view.

3.2.2 Failure

Here we take $MX = 1+X$, and write None and $\mathsf{Just}\ x$ for the corresponding components of the coproduct. This induces a simple equivalence on pairs of cbx, asserting that sequences of updates to either cbx will succeed or fail in the same way. More formally, a pointed bisimulation R on a pair of coalgebraic bxs α, β is equivalent to a relation $R \subseteq X \times Y$ such that $(\epsilon_\alpha, \epsilon_\beta) \in R$, and $(x, y) \in R$ implies for all $a : A$ and $b : B$,

- $x.get_L = y.get_L$ and $x.get_R = y.get_R$;
- if $x.set_L\ a = \mathsf{Just}\ x'$, then $y.set_L\ a = \mathsf{Just}\ y'$ for some y' with $(x', y') \in R$; and if $y.set_L\ a = \mathsf{Just}\ y'$, then $x.set_L\ a = \mathsf{Just}\ x'$ for some x' with $(x', y') \in R$;
- analogous clauses, where a and set_L are replaced with b and set_R.

Note that the second condition implies $x.set_L\ a = \mathsf{None}$ if and only if $y.set_L\ a = \mathsf{None}$; but not conversely, because of the additional requirement that $(x', y') \in R$.

3.2.3 Non-determinism

Taking M to be the finitary powerset monad, the resulting behavioural equivalence on cbx comes close to the standard notion of strong bisimulation on labelled transition systems – and as we will see, shares its excessively fine granularity. A pointed F^M_{AB}-bisimulation R on a pair of cbxs α, β is equivalent to a relation $R \subseteq X \times Y$ such that $(\epsilon_\alpha, \epsilon_\beta) \in R$, and $(x, y) \in R$ implies that for all $a : A$ and $b : B$,

- $x.get_L = y.get_L$ and $x.get_R = y.get_R$;
- for all $a : A$ and $x' \in x.set_L\ a$, there is some $y' \in y.set_L\ a$ with $(x', y') \in R$;
- for all $a : A$ and $y' \in y.set_L\ a$, there is some $x' \in x.set_L\ a$ with $(x', y') \in R$;
- analogous clauses, replacing (B, b, set_R) with (A, a, set_L).

In contrast with the case of user I/O, this equivalence may be too fine for comparing cbx behaviours, as it exposes too much information about when non-determinism occurs. Here is a prototypical scenario: consider the effect of two successive L-updates. In one implementation, suppose an update $set_L\ a$ changes the system state from s to t, and a second update $set_L\ a'$ changes it to either u or u'. Each state-change is only observable to the user through the values of get_L and get_R; so suppose $u.get_R = u'.get_R = b$. (Note that $u.get_L = u'.get_L = a'$ by ($\mathsf{CSetGet}_L$).) This means u and u' cannot be distinguished by their get values.

In a different implementation, suppose $set_L\ a$ instead maps s to one of two states t' or t'' (both with the same values of get_R and get_L as state t above), and then $set_L\ a'$ maps these respectively to u and u' again. The states called s in both implementations, although indistinguishable to any user by observing their get values, are not bisimilar. In such situations, a coarser 'trace-based' notion of equivalence [HJS07] may be more appropriate.

4 Coalgebraic bx Composition

A cbx $\alpha : A \nleftarrow_X B$ describes how changes to a data source A are propagated across X to B, and vice versa. It is then natural to suppose, given another such $\beta : B \nleftarrow_Y C$, that we may propagate these changes to C (and vice versa), raising the question of

whether there exists a composite cbx $\alpha \bullet \beta : A \nrightarrow_Z C$ for some Z. Here, we give a more general, categorical definition of cbx composition than our previous account for mbx [ASCG+15].

4.1 Defining cbx Composition via Pullbacks

First, we introduce some necessary technical details regarding weak pullback preserving (wpp) functors. Wpp functors are closed under composition, and we also exploit the following fact (recall the definitions of Section 2.2):

Lemma 4.1. *A wpp functor preserves monomorphisms [PW98, Lemma 4.4].*

Remark 4.2. *The following technical observation will also be useful for reasoning about F^M_{AB}-coalgebras. As M is wpp (assumption 2 of Section 2.2), so too is F^M_{AB}, using the fact that $A \times (-)$ and $(-)^A$ preserve pullbacks, and hence are wpp. Then by Lemma 4.1, F^M_{AB} also preserves monos.*

The following is also useful in proofs, where $k\ x\ a$ is a **do**-block referring to x and a:

Lemma 4.3. $(\mathsf{CSetGet}_L)$ *and* $(\mathsf{CGetSet}_L)$ *are equivalent to the 'continuation' versions*

$$\begin{aligned}(\mathsf{CGetSet}_L)\ (\alpha): &\quad \mathbf{do}\,\{\mathbf{let}\ a = x.get_L; x' \leftarrow x.set_L\ a; k\ x'\ a\}\\ &= \mathbf{do}\,\{\mathbf{let}\ a = x.get_L; k\ x\ a\}\\ (\mathsf{CSetGet}_L)\ (\alpha): &\quad \mathbf{do}\,\{x' \leftarrow x.set_L\ a; k\ x'\ (x'.get_L)\}\\ &= \mathbf{do}\,\{x' \leftarrow x.set_L\ a; k\ x'\ a\}\end{aligned}$$

Similarly, there are continuation versions of the coalgebra-morphism laws $(\mathsf{CGetP}_L)(h)$, $(\mathsf{CSetP}_L)(h)$, etc. in Remark 3.3, which we omit. We are now ready to define cbx composition; we do this in four stages.

(i) Defining a State-space for the Composition of α and β

The state-spaces X, Y of coalgebraic bx $\alpha : A \nrightarrow_X B, \beta : B \nrightarrow_Y C$ both contain information about B, in addition to A and C respectively. We define the state-space Z of the composite as consisting of those pairs $(x, y) : X \times Y$ which are 'B-consistent', in that $x.get_R = y.get_L$. We must also identify an initial state in Z; the obvious choice is the pairing of initial states $\langle \epsilon_\alpha, \epsilon_\beta \rangle : I \to X \times Y$ from α and β. To lie in Z, the pair itself must be B-consistent: $\epsilon_\alpha.get_R = \epsilon_\beta.get_L$. We may only compose cbx whose initial states are B-consistent in this way.

We now give the categorical formulation of these ideas, in terms of pullbacks:

Definition 4.4. *Given two pointed cbx $\alpha : A \nrightarrow_X B$ and $\beta : B \nrightarrow_Y C$, we define a* state-space *for their composition $\alpha \bullet \beta$ to be the pullback $P_{\alpha,\beta}$ in the below-left diagram. It is straightforward to show that this also makes the below-right diagram (also used in Step ((iii)) below) into a pullback, where $e_{\alpha,\beta}$ is defined to be $\langle p_\alpha, p_\beta \rangle$.*

$$\begin{array}{ccc} P_{\alpha,\beta} & \xrightarrow{p_\beta} & Y \\ {\scriptstyle p_\alpha}\downarrow & & \downarrow{\scriptstyle \beta.get_L} \\ X & \xrightarrow{\alpha.get_R} & B \end{array} \qquad \begin{array}{ccc} P_{\alpha,\beta} & \xrightarrow{e_{\alpha,\beta}=\langle p_\alpha, p_\beta\rangle} & X \times Y \\ {\scriptstyle p_\alpha}\downarrow & & \downarrow{\scriptstyle \alpha.get_R \times \beta.get_L} \\ X & \xrightarrow{\langle \alpha.get_R, \alpha.get_R\rangle} & B \times B \end{array}$$

For instance, in the category Set, these definitions may be interpreted as follows:

$$P_{\alpha,\beta} = \{(x,y) \mid x.get_R = y.get_L\} = \{(x,y) \mid (x.get_R, y.get_L) = (x.get_R, x.get_R)\}$$

and $\epsilon_{\alpha\bullet\beta}$ is the pair of initial states $(\epsilon_\alpha, \epsilon_\beta)$, assuming $\epsilon_\alpha.get_R = \epsilon_\beta.get_L$.

Remark 4.5. *We note that $e_{\alpha,\beta}$ is also the* equalizer *of the parallel pair of arrows $\alpha.get_R \circ \pi_1, \beta.get_L \circ \pi_2 : X \times Y \to B$. Hence, $e_{\alpha,\beta}$ is monic (i.e. left-cancellable), and thus by Lemma 4.1, so is its image under the wpp functors M and F^M_{AB}.*

This remark forms a key technique in our proofs about things defined by pullbacks. To prove an equation, we typically proceed indirectly, by proving its postcomposition with a mono f (or monos derived from f) which may then be left-cancelled from both sides to yield the required result. This technique will aid in proving properties of the composition $\alpha \bullet \beta$ in Section 4.2. It also allows us to pick out an initial state for Z, by noting that the arrow $\langle \epsilon_\alpha, \epsilon_\beta \rangle : I \to X \times Y$ equalizes the parallel pair of morphisms in Remark 4.5; universality then gives the required arrow $\epsilon_{\alpha\bullet\beta} : I \to Z$.

(ii) Defining Pair-based Composition $\alpha \diamond \beta$

Definition 4.6. *$(X \times Y, \alpha \diamond \beta)$ is an F^M_{AC}-coalgebra with L-operations (similarly R):*

$$\begin{aligned}(x,y).get_L &= x.get_L \\ (x,y).set_L\ a &= \mathbf{do}\ \{x' \leftarrow x.set_L\ a; y' \leftarrow y.set_L\ (x'.get_R); return\ (x', y')\}\end{aligned}$$

(iii) Inducing the Coalgebra $\alpha \bullet \beta$ on the Pullback

We now prove that the *set* operations of $\alpha \diamond \beta$ produce B-consistent pairs – even if the input pairs (x, y) were not B-consistent (because the *set* operations involve retrieving a B-value from one cbx, and setting the same value in the other). Note that this implies $\alpha \diamond \beta$ will generally fail to be a coalgebraic bx, as it will not satisfy the coalgebraic bx law (CGetSet): getting and then setting A or C in a B-inconsistent state will result in a different, B-consistent state – in effect, the B-inconsistent states are 'unreachable' after a *set* operation – which contradicts the law's requirement that the state should not change.

Lemma 4.7. *The following equation $(\dagger_L)$ holds at L for the set_L operation of Definition 4.6, and a corresponding property $(\dagger_R)$ for set_R. (The last two occurrences of $x'.get_R$ may equivalently be replaced with $y'.get_L$.)*

$$\begin{aligned} & \mathbf{do}\ \{(x',y') \leftarrow (x,y).set_L\ a; return\ (x'.get_R, y'.get_L)\} \\ = \quad & \mathbf{do}\ \{(x',y') \leftarrow (x,y).set_L\ a; return\ (x'.get_R, x'.get_R)\} \end{aligned} \qquad (\dagger_L)$$

Proof. We prove $(\dagger_L)$; the argument for $(\dagger_R)$ is symmetric.

$$\begin{aligned} & \mathbf{do}\ \{(x',y') \leftarrow (x,y).set_L\ a; return\ (x'.get_R, y'.get_L)\} \\ = \quad & [\![\ \text{definition of } (x,y).set_L\ ]\!] \\ & \mathbf{do}\ \{x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; return\ (x'.get_R, y'.get_L)\} \\ = \quad & [\![\ (\mathsf{CSetGet}_L)\ (\beta), \text{ where } k\ y'\ b \text{ is } return\ (x'.get_R, b) \text{ (it doesn't use } y')\ ]\!] \\ & \mathbf{do}\ \{x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; return\ (x'.get_R, b)\} \\ = \quad & [\![\ \text{inlining of } \mathbf{let}\ b = x'.get_R\ ]\!] \\ & \mathbf{do}\ \{x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; return\ (x'.get_R, x'.get_R)\} \\ = \quad & [\![\ \text{definition of } (x,y).set_L\ ]\!] \\ & \mathbf{do}\ \{(x',y') \leftarrow (x,y).set_L\ a; return\ (x'.get_R, x'.get_R)\} \end{aligned}$$

□

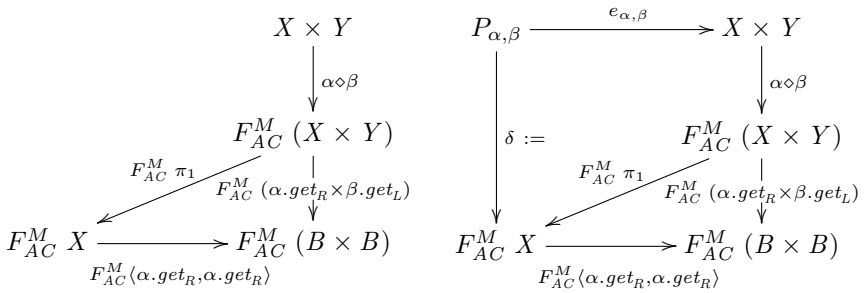
Remark 4.8. *In general, this is a stronger constraint than the corresponding equation*

$$\begin{array}{rl} & \mathbf{do}\ \{(x',y') \leftarrow (x,y).set_L\ a; return\ (x'.get_R)\} \\ = & \mathbf{do}\ \{(x',y') \leftarrow (x,y).set_L\ a; return\ (y'.get_L)\} \end{array} \qquad (\dagger_L^*)$$

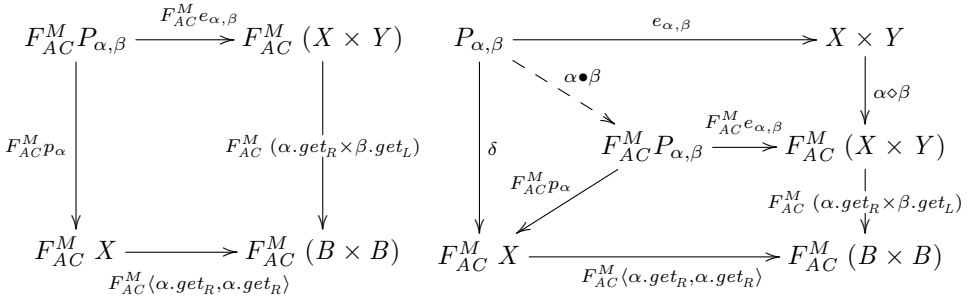
although it is equivalent if the monad M preserves jointly monic pairs (Definition 3.4). To illustrate the difference, suppose $B = \{0,1\}$ and consider a non-deterministic setting, where M is the (finitary) powerset monad on Set *(and indeed, that choice of M* does not *preserve jointly monic pairs). In state (x,y), suppose that $(set_L\ a)$ can land in either of two new states (x_1,y_1) or (x_2,y_2), where $x_2.get_R = y_1.get_L = 0$ and $x_1.get_R = y_2.get_L = 1$. Then $(\dagger_L^*)$ holds at (x,y) as both sides give $\{0,1\}$, but $(\dagger_L)$ does not, because the left side gives $\{(0,1),(1,0)\}$ and the right gives $\{(0,0),(1,1)\}$. We require the stronger version $(\dagger_L)$ to correctly define composition below.*

Our goal is to show that the properties $(\dagger_L)$ and $(\dagger_R)$ together are sufficient to ensure that the operations of $\alpha \diamond \beta : X \times Y \to F_{AC}^M\ (X \times Y)$, restricted to the B-consistent pairs $P_{\alpha,\beta}$, induce well-defined operations $P_{\alpha,\beta} \to F_{AC}^M P_{\alpha,\beta}$ on the pullback.

To do this, it is convenient to cast the properties $(\dagger_L)$, $(\dagger_R)$ in diagrammatic form, as shown in the left-hand diagram below. (It also incorporates two vacuous assertions, $(x,y).get_L = (x,y).get_L$ and similarly at R, which we may safely ignore.) Then, we precompose this diagram with the equalizer $e_{\alpha,\beta}$ as shown below-right, defining δ to be the resulting arrow $P_{\alpha,\beta} \to F_{AC}^M\ X$ given by the composition $F_{AC}^M\ \pi_1 \circ (\alpha \diamond \beta) \circ e_{\alpha,\beta}$.



Under the assumption that M is wpp, so is F_{AC}^M. Hence, the image under F_{AC}^M of the 'alternative' pullback characterisation of $P_{\alpha,\beta}$ (the right-hand diagram in Definition 4.4) is a weak pullback; it is shown below-left. Now the above-right diagram contains a cone over the same span of arrows; hence (by definition) we obtain a mediating morphism $P_{\alpha,\beta} \to F_{AC}^M\ (P_{\alpha,\beta})$ (not *a priori* unique) as shown below-right. We take this to be the coalgebra structure $\alpha \bullet \beta$ of the composite cbx.



Although this does not explicitly define the operations of the composition $\alpha \bullet \beta$, it does relate them to those of $\alpha \diamond \beta$ via the monic arrow $F^M_{AC}\, e_{\alpha,\beta}$ (Remark 4.5) – allowing us to reason in terms of B-consistent pairs $(x, y) : X \times Y$, appealing to left-cancellability of monos. Moreover, in spite of only *weak* pullback preservation of F^M_{AC}, the coalgebra structure $\alpha \bullet \beta$ is canonical: there can be at most one coalgebra structure $\alpha \bullet \beta$ such that $e_{\alpha,\beta}$ is a coalgebra morphism from $\alpha \bullet \beta$ to $\alpha \diamond \beta$. This is a simple corollary of Lemma 4.11 below.

(iv) Proving the Composition is a Coalgebraic bx

Proposition 4.9. (CGetSet) $(\alpha \bullet \beta)$ *and* (CSetGet) $(\alpha \bullet \beta)$ *hold at* L *and* R.

Proof. We focus on the L case (the R case is symmetric). As anticipated in Remark 4.5, we prove the laws post-composed with the monos $Me_{\alpha,\beta}$ and $M\ (e_{\alpha,\beta} \times id)$ respectively; left-cancellation completes the proof. (The laws $(\mathsf{CGetP}_L)(e_{\alpha,\beta})$ and $(\mathsf{CSetP}_L)(e_{\alpha,\beta})$ are given in Remark 3.3.) Here is $(\mathsf{CGetSet}_L)\ (\alpha \bullet \beta)$ postcomposed with $Me_{\alpha,\beta}$:

$$
\begin{array}{ll}
 & \mathbf{do}\ \{\mathbf{let}\ a = z.get_L; z' \leftarrow z.set_L\ a; return\ (e_{\alpha,\beta}\ (z'))\} \\
= & \quad [\![\ (\mathsf{CSetP}_L)(e_{\alpha,\beta})\ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ a = z.get_L; \mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (z); (x', y') \leftarrow (x, y).set_L\ a; return\ (x', y')\} \\
= & \quad [\![\ \text{swapping } \mathbf{let}\text{s, and using } (\mathsf{CGetP}_L)(e_{\alpha,\beta})\ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (z); \\
 & \qquad \mathbf{let}\ a = (x, y).get_L; (x', y') \leftarrow (x, y).set_L\ a; return\ (x', y')\} \\
= & \quad [\![\ \text{definitions of } (x, y).get_L \text{ and } (x, y).set_L\ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (z); \mathbf{let}\ a = x.get_L; \\
 & \qquad x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; return\ (x', y')\} \\
= & \quad [\![\ (\mathsf{CGetSet}_L)\ (\alpha)\ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (z); \mathbf{let}\ b = x.get_R; y' \leftarrow y.set_L\ b; return\ (x, y')\} \\
= & \quad [\![\ (x, y) = e_{\alpha,\beta}\ (z) \text{ implies } x.get_R = y.get_L \text{ by definition of } e_{\alpha,\beta}\ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (z); \mathbf{let}\ b = y.get_L; y' \leftarrow y.set_L\ b; return\ (x, y')\} \\
= & \quad [\![\ (\mathsf{CGetSet}_L)\ (\beta)\ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (z); return\ (x, y)\} \\
= & \quad [\![\ \text{inline } \mathbf{let}; \mathbf{do}\text{-laws}\ ]\!] \\
 & return\ (e_{\alpha,\beta}\ (z))
\end{array}
$$

$(\mathsf{CSetGet}_L)$ postcomposed with $M\ (e_{\alpha,\beta} \times id)$:

$$
\begin{array}{ll}
 & \mathbf{do}\ \{z' \leftarrow z.set_L\ (a); return\ (e_{\alpha,\beta}\ (z'), z'.get_L)\} \\
= & \quad [\![\ \text{inlining } \mathbf{let}\text{; definition of } z'.get_L\ ]\!] \\
 & \mathbf{do}\ \{z' \leftarrow z.set_L\ (a); \mathbf{let}\ (x', y') = e_{\alpha,\beta}\ (z'); return\ ((x', y'), x'.get_L)\} \\
= & \quad [\![\ (\mathsf{CSetP}_L)(e_{\alpha,\beta})\ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (z); (x', y') \leftarrow (x, y).set_L\ a; return\ ((x', y'), x'.get_L)\} \\
= & \quad [\![\ \text{definition of } (x, y).set_L\ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (z); \\
 & \qquad x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; return\ ((x', y'), x'.get_L)\} \\
= & \quad [\![\ (\mathsf{CSetGet}_L)\ (\alpha)\ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (z); \\
 & \qquad x' \leftarrow x.set_L\ a; \mathbf{let}\ b = x'.get_R; y' \leftarrow y.set_L\ b; return\ ((x', y'), a)\} \\
= & \quad [\![\ \text{definition of } (x, y).set_L\ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (z); (x', y') \leftarrow (x, y).set_L\ a; return\ ((x', y'), a)\} \\
= & \quad [\![\ (\mathsf{CSetP}_L)(e_{\alpha,\beta})\text{; inline } \mathbf{let}\ ]\!] \\
 & \mathbf{do}\ \{z' \leftarrow z.set_L\ a; return\ (e_{\alpha,\beta}\ (z'), a)\}
\end{array}
$$

□

4.2 Well-behavedness of cbx Composition

Having defined a notion of composition for cbx, we must check that it has the properties one would expect, in particular that it is associative and has left and right identities. However, as noted in the Introduction, we cannot expect these properties to hold 'on the nose', but rather only up to some notion of behavioural equivalence. We will now prove that cbx composition is well-behaved up to the equivalence $\equiv$ introduced in Section 3 (Definition 3.6). Recall also the identity cbx $\iota(e) : A \nrightarrow A$ (Example 2.6).

Theorem 4.10. *Coalgebraic bx composition satisfies the following properties (where $\alpha, \alpha' : A \nrightarrow B$, $\beta, \beta' : B \nrightarrow C$, and $\gamma, \gamma' : C \nrightarrow D$, and all compositions are assumed well-defined):*

identities: $\iota(\epsilon_\alpha.get_L) \bullet \alpha \equiv \alpha$ *and* $\alpha \bullet \iota(\epsilon_\alpha.get_R) \equiv \alpha$

congruence: *if* $\alpha \equiv \alpha'$ *and* $\beta \equiv \beta'$ *then* $\alpha \bullet \beta \equiv \alpha' \bullet \beta'$

associativity: $(\alpha \bullet \beta) \bullet \gamma \equiv \alpha \bullet (\beta \bullet \gamma)$

To prove this, we typically need to exhibit a coalgebra morphism from some coalgebra α to a composition $\beta = \psi \bullet \varphi$. As the latter is defined implicitly by the equalizer $e_{\psi,\varphi}$ – which is a *monic* coalgebra morphism from β to $\gamma = \psi \diamond \varphi$ – it is usually easier to reason by instead exhibiting a coalgebra morphism from α into $\gamma = \psi \diamond \varphi$, and then appealing to the following simple lemma:

Lemma 4.11. *Let F be wpp, and let $a : \alpha \to \gamma \leftarrow \beta : b$ be a cospan of pointed F-coalgebra morphisms with b monic. Then any $m : \alpha \to \beta$ with $b \circ m = a$ is also a pointed F-coalgebra morphism. If a is monic, then so is m; and for any q with (a, q) jointly monic, so is (m, q).*

Proof. One may show that $Fb \circ \beta \circ m = Fb \circ Fm \circ \alpha$ by the fact that a, b are coalgebra morphisms and $b \circ m = a$. Then using the fact that F preserves the mono b, we may left-cancel Fb on both sides. Moreover, if $m \circ f = m \circ f'$, then post-composing with b (and applying $b \circ m = a$) we obtain $a \circ f = a \circ f'$; the result then follows. □

Remark 4.12. *In the following proof, we will often apply Lemma 4.11 in the situation where b is given by a equalizer (such as $e_{\psi,\varphi}$, in Remark 4.5) which is also a coalgebra morphism, and where the coalgebra morphism a also equalizes the relevant parallel pairs. Then we obtain the arrow m by universality; and the Lemma ensures it is also a coalgebra morphism, as equalizers are monic.*

Proof. The general strategy is to prove that two compositions $\gamma = \delta \bullet \vartheta$ and $\gamma' = \delta' \bullet \vartheta'$ are $\equiv$-equivalent, by providing a jointly monic pair of pointed coalgebra morphisms p, q from some ζ into $\delta \diamond \vartheta$ and $\delta' \diamond \vartheta'$ respectively, which equalize the relevant parallel pairs. Lemma 4.11 and Remark 4.12 then imply the existence of a jointly monic pair of pointed coalgebra morphisms m, m' into $\delta \bullet \vartheta$ and $\delta' \bullet \vartheta'$, giving the required bisimulation. We indicate the key steps (i), (ii), etc. in each proof below.

Identities: We show that $\iota(\epsilon_\alpha.get_L) \bullet \alpha \equiv \alpha$; the other identity is symmetric. We exhibit the equivalence by taking α itself to be the coalgebra defining a bisimulation between α and $\iota(\epsilon_\alpha.get_L) \bullet \alpha$. To do this, one shows that (i) $h = \langle \alpha.get_L, id \rangle : X \to A \times X$ is a pointed coalgebra morphism from α to the composition $\iota(\epsilon_\alpha.get_L) \diamond \alpha$ defined on pairs, and (ii) h also equalizes the parallel pair $\iota(\epsilon_\alpha.get_L).get_R \circ \pi_1$ and $\alpha.get_L \circ \pi_2$ (characterising the equalizer

and coalgebra morphism e from $\iota(\epsilon_\alpha.get_L) \bullet \alpha$ to $\iota(\epsilon_\alpha.get_L) \diamond \alpha$). By definition of equalizers, this induces a map $p : X \to P_{\alpha,\iota(\epsilon_\alpha.get_L)}$ such that $e \circ p = h$. Remark 4.12 now implies that p is a pointed coalgebra morphism from α to $\iota(\epsilon_\alpha.get_L) \bullet \alpha$, and by Remark 3.7 we obtain the required bisimulation. The only non-trivial step is proving $(\mathsf{CSetP}_L)(h)$, which requires appeal to $(\mathsf{CSetGet}_L)$ (α). As for pointedness, the initial state of $\iota(\epsilon_\alpha.get_L) \diamond \alpha$ is $(\epsilon_\alpha.get_L, \epsilon_\alpha)$, and this is indeed h (ϵ_α) as required.

Congruence: We show how to prove that right-composition $(- \bullet \beta)$ is a congruence: i.e. that $\alpha \equiv \alpha'$ implies $(\alpha \bullet \beta) \equiv (\alpha' \bullet \beta)$. By symmetry, the same argument will show left-composition $(\alpha' \bullet -)$ is a congruence. Then one may use the standard fact that 'bisimulations compose (for wpp functors)': given pointed bisimulations between γ and δ, and between δ and ε, one may obtain a pointed bisimulation between γ and ε – provided the behaviour functor F^M_{AC} is wpp, which follows from our assumption that M is wpp. This allows us to deduce that composition is a congruence in both arguments simultaneously, as required.

So, suppose given a pointed bisimulation between α and α': an F^M_{AB}-coalgebra (R, ϱ) with a jointly monic pair p, p' of pointed coalgebra morphisms from ϱ to α, α' respectively. One exhibits a bisimulation between $\alpha \bullet \beta$ and $\alpha' \bullet \beta$ as follows, by first constructing a suitable coalgebra (S, σ), together with a jointly monic pair (q, q') of coalgebra morphisms from σ to the compositions $\alpha \bullet \beta$, $\alpha' \bullet \beta$. To construct σ, let ζ be the equalizer of the following parallel pair – or equivalently, the pullback of $\varrho.get_R$ and $\beta.get_L$.

$$S \dashrightarrow[\zeta] R \times Y \underset{\beta.get_L \circ \pi_2}{\overset{\varrho.get_R \circ \pi_1}{\rightrightarrows}} B$$

One may then follow the steps (i)–(iii) in Section 4, where ζ and ϱ play the role of the equalizer $e_{\alpha,\beta}$ and β respectively, to construct σ, such that ζ is a coalgebra morphism from σ to $\varrho \diamond \beta$. Even though ϱ is not a coalgebraic bx, it satisfies the following weaker form $(\mathsf{CSetGet}^-_L)(\varrho)$ of $(\mathsf{CSetGet}_L)$, and its R-version:

$$\begin{array}{ll}
(\mathsf{CSetGet}^-_L)(\varrho): & \\
\quad \mathbf{do}\,\{\, r' \leftarrow r.set_L\ a; \mathit{return}\ r'.get_L \,\} & \\
= \quad [\![\ (\mathsf{CGetP}_L)(p)\ ]\!] & \\
\quad \mathbf{do}\,\{\, r' \leftarrow r.set_L\ a; \mathit{return}\ p\ (r').get_L \,\} & \\
= \quad [\![\ (\mathsf{CSetP}_L)(p)\ ]\!] & \\
\quad \mathbf{do}\,\{\, \mathbf{let}\ x = p\ (r); x' \leftarrow x.set_L\ a; \mathit{return}\ x'.get_L \,\} & \\
= \quad [\![\ (\mathsf{CSetGet}_L)\ (\alpha)\ ]\!] & \\
\quad \mathbf{do}\,\{\, \mathbf{let}\ x = p\ (r); x' \leftarrow x.set_L\ a; \mathit{return}\ a \,\} & \\
= \quad [\![\ (\mathsf{CSetP}_L)(p)\ ]\!] & \\
\quad \mathbf{do}\,\{\, r' \leftarrow r.set_L\ a; \mathbf{let}\ x' = p\ (r'); \mathit{return}\ a \,\} & \\
= \quad [\![\ \text{redundant } \mathbf{let}\ ]\!] & \\
\quad \mathbf{do}\,\{\, r' \leftarrow r.set_L\ a; \mathit{return}\ a \,\} &
\end{array}$$

The corresponding continuation version is as follows:

$$(\mathsf{CSetGet}^-_L)(\varrho): \quad \mathbf{do}\,\{\, r' \leftarrow r.set_L\ a; k\ (r'.get_L) \,\} = \{\, r' \leftarrow r.set_L\ a; k\ a \,\}$$

This then justifies the second reasoning step in Lemma 4.7, where the continuation $k\ y'\ b$ is replaced with $k\ b$.

Now, to construct $q : S \to P_{\alpha,\beta}$, it is enough to show (i) the composition of the upper and right edges in the following diagram equalizes the given parallel pair:

$$\begin{array}{ccccc} S & \xrightarrow{\quad\zeta\quad} & R \times Y & & \\ {\scriptstyle q}\downarrow & & \downarrow{\scriptstyle p\times id} & & \\ P_{\alpha,\beta} & \xrightarrow{\quad e_{\alpha,\beta}\quad} & X \times Y & \underset{\beta.get_L\circ\pi_2}{\overset{\alpha.get_R\circ\pi_1}{\rightrightarrows}} & B \end{array} \tag{1}$$

By also showing that (ii) $p \times id$ is in fact a coalgebra morphism, we can then appeal to Lemma 4.11 to show that q itself defines a coalgebra morphism from σ into the composition $\alpha \bullet \beta$. One obtains the coalgebra morphism q' from σ to $\alpha' \bullet \beta$ in an analogous way. Finally, from p, p' being jointly monic, and the fact that $e_{\alpha,\beta}$ is an equalizer, we obtain a proof that q, q' are jointly monic.

Associativity: We follow the same strategy as we did for proving the identity laws: we may prove this law by providing a pointed coalgebra morphism p from the left-hand composition to the right-hand one. We will do this in two stages: first, we show how to obtain an arrow $p_0 : P_{(\alpha\bullet\beta),\gamma} \to X \times P_{\beta,\gamma}$ making the square in the following diagram commute; then, by applying Lemma 4.11 and Remark 4.12, we will show it is a pointed coalgebra morphism from $(\alpha \bullet \beta) \bullet \gamma$ to $\alpha \diamond (\beta \bullet \gamma)$.

$$\begin{array}{ccccc} P_{(\alpha\bullet\beta),\gamma} & \xrightarrow{\quad e_{(\alpha\bullet\beta),\gamma}\quad} & P_{\alpha,\beta} \times Z & & \\ {\scriptstyle p_0}\downarrow & & \downarrow{\scriptstyle f} & & \\ X \times P_{\beta,\gamma} & \xrightarrow{\quad id\times e_{\beta,\gamma}\quad} & X \times (Y \times Z) & \underset{id\times(\gamma.get_L\circ\pi_2)}{\overset{id\times(\beta.get_R\circ\pi_1)}{\rightrightarrows}} & X \times C \end{array} \tag{2}$$

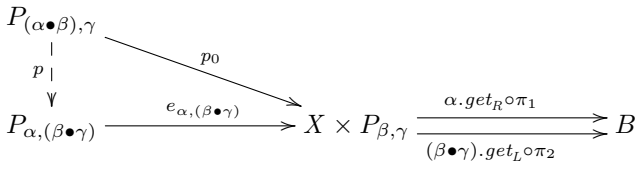
In this diagram, the arrow f is defined by

$$f\ (u, z) = \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (u); \mathit{return}\ (x, (y, z))\}$$

but it may also be expressed as $f = \mathit{assoc} \circ (e_{\alpha,\beta} \times id)$, where we write *assoc* for the associativity isomorphism on products. Note that the functor $X \times (-)$, being a right adjoint, preserves equalizers, and hence $(id \times e_{\beta,\gamma})$ is the equalizer of the given parallel pair.

Following the proof strategy outlined above, to obtain p_0 one must show that: (i) $f \circ e_{(\alpha\bullet\beta),\gamma}$ equalizes the parallel pair in the above diagram, ensuring existence of an arrow p_0 making the square commute; (ii) the equalizer $id \times e_{\beta,\gamma}$ is a pointed coalgebra morphism from $\alpha \diamond (\beta \bullet \gamma)$ to $\alpha \diamond (\beta \diamond \gamma)$; and (iii) f is a pointed coalgebra morphism from $(\alpha \bullet \beta) \diamond \gamma$ to $\alpha \diamond (\beta \diamond \gamma)$. The facts (ii), (iii) allow us to apply Remark 4.12 to deduce that p_0 is a pointed coalgebra morphism.

The final stage of constructing the required arrow $p : P_{(\alpha\bullet\beta),\gamma} \to P_{\alpha,(\beta\bullet\gamma)}$ is to show that: (iv) p_0 equalizes the parallel pair defining $P_{\alpha,(\beta\bullet\gamma)}$ as shown below. Thus we obtain p as the mediating morphism into $P_{\alpha,(\beta\bullet\gamma)}$; by Remark 4.12 it is a coalgebra morphism.



□

This well-behavedness of composition allows us to define a category of cbx as follows. (Note that this is not a typical category of F-coalgebras and morphisms, for some fixed functor F, but rather a category where the *morphisms* $A \to B$ are equivalence classes of F^M_{AB}-coalgebras.)

Corollary 4.13. *There is a category* $\mathsf{Cbx}_\bullet$ *of pointed cbx, whose objects are pairs* (A, a) *of an object* A *from* $\mathbb{C}$ *and an arrow* $a : I \to A$*; and whose arrows* $(A, a) \to (B, b)$ *are* $\equiv$*-equivalence classes* $[\alpha]$ *of cbx* $\alpha : A \not\leftrightarrows B$ *satisfying* $\epsilon_\alpha.get_L = a$ *and* $\epsilon_\alpha.get_R = b$*. (Recall that* $\alpha \equiv \beta$ *iff there is a pointed bisimulation between* α *and* β*.)*

We now describe how this category is related (by specialising $\mathbb{C} = \mathsf{Set}$ and $M = Id$) to the category of symmetric lenses [HPW11]. The point of departure is that cbx encapsulate *additional data*, namely initial values $\epsilon_\alpha.get_L, \epsilon_\alpha.get_R$ for A and B. The difference may be reconciled if one is prepared to extend SLs with such data (and consider distinct initial-values to give distinct SLs, *cf.* the comments beginning Section 2.3):

Corollary 4.14. *Taking* $\mathbb{C} = \mathsf{Set}$ *and* $M = Id$*,* $\mathsf{Cbx}_\bullet$ *is isomorphic to a subcategory of* SL*, the category of SL-equivalence-classes of symmetric lenses; and there is an isomorphism of categories* $\mathsf{Cbx}_\bullet \cong \mathsf{SL}_\bullet$ *where* $\mathsf{SL}_\bullet$ *is the category of (SL-equivalence-classes) of SLs additionally equipped with initial left- and right-values.*

5 Relating Coalgebraic and Monadic bx

Here, we consider the relationship between our coalgebraic notion of bx, which is inherently stateful, and our previous account of transparent monadic bx [ASCG+15], where the *get* and *set* operations are restricted to monads of the form $StateT\ X\ M$, abbreviated to T^M_X, and moreover the *get* operations neither change the state X nor introduce M-effects, i.e. $get_L = \lambda x.return\ (f\ x, x)$ for some $f : X \to A$ (likewise get_R). We identified this restriction in order to permit a smooth definition of mbx composition. We also assume monadic bx have explicit initial states, rather than the more intricate process of initialising by supplying an initial A- or B-value as in our earlier account [ASCG+15].

5.1 Translating a Coalgebraic bx into a Monadic bx

Given a cbx $\alpha : X \to F^M_{AB}\ X$, we can define its *realisation*, or 'monadic interpretation', $[\![\alpha]\!]$ as a transparent mbx with the following operations. (Following conventional Haskell notation, we overload the symbol () to mean the unit type, as well as its unique value.)

$$[\![\alpha]\!].get_L : T^M_X\ A \qquad \stackrel{\text{def}}{=} \mathbf{do}\ \{x \leftarrow get; return\ (x.get_L)\}$$
$$[\![\alpha]\!].get_R : T^M_X\ B \qquad \stackrel{\text{def}}{=} \mathbf{do}\ \{x \leftarrow get; return\ (x.get_R)\}$$

$$[\![\alpha]\!].set_L : A \to T_X^M\ () \stackrel{\text{def}}{=} \lambda a \to \mathbf{do}\,\{x \leftarrow get; x' \leftarrow lift\ (x.set_L\ a); set\ x'\}$$
$$[\![\alpha]\!].set_R : B \to T_X^M\ () \stackrel{\text{def}}{=} \lambda b \to \mathbf{do}\,\{x \leftarrow get; x' \leftarrow lift\ (x.set_R\ b); set\ x'\}$$

Here, the standard polymorphic functions associated with T_X^M, namely $get : \forall\alpha\,.\,T_\alpha^M\ \alpha$, $set : \forall\alpha\,.\,\alpha \to T_\alpha^M\ ()$, and the monad morphism $lift : \forall\alpha\,.\,M\alpha \to T_X^M\ \alpha$ (the curried form of the strength of M) are given as follows, and they satisfy the following laws:

$$\begin{array}{ll} & get = \lambda a \to return\ (a, a) \\ & set = \lambda a'\ a \to return\ ((), a') \\ & lift = \lambda ma\ x \to \mathbf{do}\,\{a \leftarrow ma; return\ (a, x)\} \\ (\mathsf{GetSet}) & \mathbf{do}\,\{x \leftarrow get; set\ x\} = return\ () \\ (\mathsf{SetGet}) & \mathbf{do}\,\{set\ x; x' \leftarrow get; return\ x'\} = \mathbf{do}\,\{set\ x; return\ x\} \end{array}$$

There is also a continuation version of (GetSet), by analogy with (CGetSet):

$$\mathbf{do}\,\{x \leftarrow get; set\ x; k\ x\} = \mathbf{do}\,\{x \leftarrow get; k\ x\}$$

Proposition 5.1. $[\![\alpha]\!]$ *indeed defines a transparent mbx over* T_X^M.

Proof. We need to establish the (MGetSet) and (MSetGet) laws for the defined operations on the mbx $[\![\alpha]\!]$ at L and R:

$$\begin{array}{ll} (\mathsf{MGetSet}) & \mathbf{do}\,\{a \leftarrow [\![\alpha]\!].get_L; [\![\alpha]\!].set_L\ a\} = return\ () \\ (\mathsf{MSetGet}) & \mathbf{do}\,\{[\![\alpha]\!].set_L\ a; [\![\alpha]\!].get_L\} = \mathbf{do}\,\{[\![\alpha]\!].set_L\ a; return\ a\} \end{array}$$

We give the argument for L; that for R is entirely analogous.

$$\begin{array}{ll} & \mathbf{do}\,\{a \leftarrow [\![\alpha]\!].get_L; [\![\alpha]\!].set_L\ a\} \\ = & \quad [\![\text{ definition of } [\![\alpha]\!] \ ]\!] \\ & \mathbf{do}\,\{a \leftarrow \mathbf{do}\,\{x \leftarrow get; return\ \alpha.get_L\ (x)\}; \\ & \qquad \mathbf{do}\,\{x \leftarrow get; x' \leftarrow lift\ (x.set_L\ a); set\ x'\}\} \\ = & \quad [\![\ \mathbf{do}\text{-laws} \ ]\!] \\ & \mathbf{do}\,\{x \leftarrow get; a \leftarrow return\ (\alpha.get_L\ x); x' \leftarrow lift\ (x.set_L\ x\ a); set\ x'\} \\ = & \quad [\![\text{ replace } a \leftarrow return\ldots \text{ with } \mathbf{let}\ a = \ldots \text{ and inline} \ ]\!] \\ & \mathbf{do}\,\{x \leftarrow get; x' \leftarrow lift\ (x.set_L\ x\ (\alpha.get_L\ x)); set\ x'\} \\ = & \quad [\![\ (\mathsf{CGetSet})(\alpha) \ ]\!] \\ & \mathbf{do}\,\{x \leftarrow get; x' \leftarrow lift\ (return\ x); set\ x'\} \\ = & \quad [\![\ lift \text{ is a monad morphism, so } lift\ (return\ x) = return\ x \ ]\!] \\ & \mathbf{do}\,\{x \leftarrow get; x' \leftarrow return\ x; set\ x'\} \\ = & \quad [\![\text{ replace } x' \leftarrow return\ldots \text{ with } \mathbf{let}\ x' = \ldots \text{ and inline} \ ]\!] \\ & \mathbf{do}\,\{x \leftarrow get; set\ x\} \\ = & \quad [\![\ (\mathsf{GetSet}) \ ]\!] \\ & return\ () \end{array}$$

$$\begin{array}{ll} & \mathbf{do}\,\{[\![\alpha]\!].set_L\ a; [\![\alpha]\!].get_L\} \\ = & \quad [\![\text{ definition of } [\![\alpha]\!] \ ]\!] \\ & \mathbf{do}\,\{\mathbf{do}\,\{x \leftarrow get; x' \leftarrow lift\ (x.set_L\ a); set\ x'\}; \\ & \qquad \mathbf{do}\,\{x \leftarrow get; return\ \alpha.get_L\ (x)\}\} \\ = & \quad [\![\ \mathbf{do}\text{-laws, alpha-converting } x \to x'' \text{ in second } \mathbf{do} \ ]\!] \\ & \mathbf{do}\,\{x \leftarrow get; x' \leftarrow lift\ (x.set_L\ a); set\ x'; x'' \leftarrow get; return\ \alpha.get_L\ (x'')\} \end{array}$$

$$
\begin{array}{ll}
= & [\![\ (\mathsf{SetGet});\ \text{inline resulting } \mathbf{let}\ x'' = x'\ ]\!] \\
& \mathbf{do}\ \{x \leftarrow get; x' \leftarrow lift\ (x.set_L\ a); set\ x'; return\ \alpha.get_L\ (x')\} \\
= & [\![\ \text{introduce } \mathbf{let}\ ]\!] \\
& \mathbf{do}\ \{x \leftarrow get; x' \leftarrow lift\ (x.set_L\ a); set\ x'; \mathbf{let}\ a' = \alpha.get_L\ (x'); return\ a'\} \\
= & [\![\ \text{pure } \mathbf{let}\text{s commute with monadic computations}\ ]\!] \\
& \mathbf{do}\ \{x \leftarrow get; x' \leftarrow lift\ (x.set_L\ a); \mathbf{let}\ a' = \alpha.get_L\ (x'); set\ x'; return\ a'\} \\
= & [\![\ \text{replace } \mathbf{let}\ a' = ..\ \text{with } a' \leftarrow return\ ..;\ \text{then } lift\ \text{respects } return\ ]\!] \\
& \mathbf{do}\ \{x \leftarrow get; x' \leftarrow lift\ (x.set_L\ a); a' \leftarrow lift\ (return\ x'.get_L); set\ x'; return\ a'\} \\
= & [\![\ lift\ \text{is a monad morphism}\ ]\!] \\
& \mathbf{do}\ \{x \leftarrow get; (x', a') \leftarrow lift\ (\mathbf{do}\ \{x' \leftarrow x.set_L\ a; return\ (x', x'.get_L)\}); \\
& \quad set\ x'; return\ a'\} \\
= & [\![\ (\mathsf{CSetGet})(\alpha)\ ]\!] \\
& \mathbf{do}\ \{x \leftarrow get; (x', a') \leftarrow lift\ (\mathbf{do}\ \{x' \leftarrow x.set_L\ a; return\ (x', a)\}); \\
& \quad set\ x'; return\ a'\} \\
= & [\![\ \mathbf{do}\text{-laws}\ ]\!] \\
& \mathbf{do}\ \{x \leftarrow get; x' \leftarrow lift\ (x.set_L\ a); set\ x'; return\ a\} \\
= & [\![\ \text{definition of } [\![\alpha]\!]\ ]\!] \\
& \mathbf{do}\ \{[\![\alpha]\!].set_L\ a; return\ a\}
\end{array}
$$

□

Lemma 5.2. *The translation $[\![\cdot]\!]$ – from cbx for monad M with carrier X, to transparent mbx with an initial state – is surjective; it is also injective, using our initial assumption that return is monic.*

This fully identifies the subset of monadic bx which correspond to coalgebraic bx – namely, where the monad is $StateT\ X\ M$ for some state-space X, there is a given initial state in X, and the *get* operations are transparent. Note that the translation $[\![\cdot]\!]$ is defined on an *individual* coalgebraic bx, not an equivalence class; we will say a little more about the respective categories at the end of the next section.

5.2 Composing Stateful Monadic bxs

We will review the method previously given [ASCG+15] for composing transparent mbx, using monad morphisms induced by lenses on the state-spaces. We show that this essentially simplifies to step (ii) of our definition in Section 4 above; thus, our definition may be seen as a more categorical treatment of the set-based monadic bx definition.

Definition 5.3. *An* asymmetric lens *from* source A *to* view B, *written* $l : A \Rightarrow B$, *is given by a pair of maps* $l = (v, u)$, *where* $v : A \to B$ *(the* view *or* get *mapping) and* $u : A \times B \to A$ *(the* update *or* put *mapping). It is* well-behaved *if it satisfies the first two laws* (VU), (UV) *below, and* very well-behaved *if it also satisfies* (UU).

$$
\begin{array}{lll}
(\mathsf{VU}) & u\ (a, (v\ a)) & = a \\
(\mathsf{UV}) & v\ (u\ (a, b')) & = b' \\
(\mathsf{UU}) & u\ (u\ (a, b'), b'') & = u\ (a, b'')
\end{array}
$$

Lenses have a very well-developed literature [FGM+07, HPW11, CFH+09, HSST11, among others], which we do not attempt to recap here; see our earlier work [ASCG+15, Section 2.5] for further discussion of lenses.

We will apply a mild adaptation of a result in Shkaravska's early work [Shk05].

Proposition 5.4. *Let $l = (v, u) : Z \Rightarrow X$ be a lens, and define $\vartheta\ (l)$ to be the following natural transformation:*

$$\begin{aligned}
&\vartheta\ (l) : \forall \alpha\,.\ T_X^M\ \alpha \to T_Z^M\ \alpha \\
&\vartheta\ (l)\ ma \stackrel{\text{def}}{=} \mathbf{do}\ \{ z \leftarrow get; \\
&\qquad\qquad (a', x') \leftarrow lift\ (ma\ (v\ z)); z' \leftarrow lift\ (u\ (z, x')); set\ z'; return\ a' \}
\end{aligned}$$

If l is very well-behaved, then $\vartheta\ (l)$ is a monad morphism.

We apply Prop. 5.4 to the following two lenses, allowing views and updates of the projections from $X \times Y$:

$$\begin{array}{ll}
l_1 = (v_1, u_1) : (X \times Y) \Rightarrow X & l_2 = (v_2, u_2) : (X \times Y) \Rightarrow Y \\
\quad v_1\ (x, y) \quad = x & \quad v_2\ (x, y) \quad = y \\
\quad u_1\ ((x, y), x') = (x', y) & \quad u_2\ ((x, y), y') = (x, y')
\end{array}$$

This gives us a cospan, $left = \vartheta\ (l_1) : T_X^M \dot{\to} T_{(X \times Y)}^M \dot{\leftarrow} T_Y^M : \vartheta\ (l_2) = right$, of monad morphisms allowing us to embed computations involving state-space X or Y into computations on the combined state-space $X \times Y$.

Now suppose we are given two transparent mbx, from A to B and B to C, with state-spaces X and Y respectively. Previously [ASCG+15], we used *left*, *right* to define $t_1 \fatsemi t_2$, a composite mbx with state-space $X \times Y$, as follows:

$$\begin{aligned}
&(t_1 \fatsemi t_2).get_L \quad = left\ (t_1.get_L) \\
&(t_1 \fatsemi t_2).get_R \quad = right\ (t_2.get_R) \\
&(t_1 \fatsemi t_2).set_L\ a = \mathbf{do}\ \{ left\ (t_1.set_L\ a); b \leftarrow left\ (t_1.get_R); right\ (t_2.set_L\ b) \} \\
&(t_1 \fatsemi t_2).set_R\ c = \mathbf{do}\ \{ right\ (t_2.set_R\ c); b \leftarrow right\ (t_2.get_L); left\ (t_1.set_R\ b) \}
\end{aligned}$$

We then defined the subset, $X \bowtie Y$, of $X \times Y$ given by B-consistent pairs, and argued that $t_1 \fatsemi t_2$ preserved B-consistency – hence its state-space could be restricted from $X \times Y$ to $X \bowtie Y$. We will use the notation $t_1 \bullet t_2$ for the resulting composite mbx.

In the context of coalgebraic bx, we made this part of the argument categorical by defining $X \bowtie Y$ to be a pullback, and formalising the move from the pairwise definition $\alpha \diamond \beta$ to the full composition $\alpha \bullet \beta$ by step (iii) of Section 4. Given the one-to-one correspondence between transparent mbx and cbx given by Lemma 5.2, this may be considered to be the formalisation of the monadic move from the composite $t_1 \fatsemi t_2$ on the product state-space to the composite $t_1 \bullet t_2$ on the pullback.

This allows us to state our second principal result, namely that the two notions of composition – coalgebraic bx (as in Definition 4.4) and mbx – may be reconciled by showing the pairwise definitions coherent:

Theorem 5.5. *Coherence of composition: The definitions of monadic and coalgebraic bx composition on product state-spaces are coherent: $[\![\alpha]\!] \fatsemi [\![\beta]\!] = [\![\alpha \diamond \beta]\!]$. Hence, the full definitions (on B-consistent pairs) are also coherent: $[\![\alpha]\!] \bullet [\![\beta]\!] = [\![\alpha \bullet \beta]\!]$.*

Proof. The operations of the monadic bx $[\![\alpha]\!]$ at the beginning of Section 5.1, and the computation $\vartheta\ (v, u)\ ma$, may be re-written in **do** notation for the monad M rather than in $StateT\ X\ M$, as follows:

$$\begin{aligned}
&[\![\alpha]\!].get_L = \lambda x \to return\ (x.get_L) \\
&[\![\alpha]\!].set_L\ a = \lambda x \to \mathbf{do}\ \{ x' \leftarrow x.set_L\ a; return\ ((), x') \} \\
&\vartheta\ (v, u)\ ma = \lambda z \to \mathbf{do}\ \{ (a', s') \leftarrow ma\ (v\ z); \mathbf{let}\ z' = u\ (z, x'); return\ (a', z') \}
\end{aligned}$$

Applying ϑ to the lenses l_1 and l_2 gives the monad morphisms *left* and *right* as follows:

$$
\begin{array}{l}
\mathit{left} \;= \vartheta\,(l_1) : \forall \alpha\,.\; T^M_X\;\alpha \to T^M_{(X\times Y)}\;\alpha \\
\mathit{right} = \vartheta\,(l_2) : \forall \alpha\,.\; T^M_Y\;\alpha \to T^M_{(X\times Y)}\;\alpha \\
\mathit{left} \;= \lambda \mathit{mxa}\;(x,y) \to \mathbf{do}\,\{(a',x') \leftarrow \mathit{mxa}\;x; \mathit{return}\;(a',(x',y))\} \\
\mathit{right} = \lambda \mathit{mya}\;(x,y) \to \mathbf{do}\,\{(a',y') \leftarrow \mathit{mya}\;y; \mathit{return}\;(a',(x,y'))\}
\end{array}
$$

This allows us to unpack the operations of the composite $[\![\alpha]\!] \fatsemi [\![\beta]\!]$ to show they are the same as $[\![\alpha \diamond \beta]\!]$, as follows (we omit the entirely analogous R-operations):

$$
\begin{array}{l}
([\![\alpha]\!] \fatsemi [\![\beta]\!]).\mathit{get}_L\;(x,y) \\
\quad = \quad [\![\text{ definition of } \fatsemi \;\;]\!] \\
\mathit{left}\;([\![\alpha]\!].\mathit{get}_L)\;(x,y) \\
\quad = \quad [\![\text{ unpacking } \mathit{left} \text{ as above } \;\;]\!] \\
\mathbf{do}\,\{(a',x') \leftarrow [\![\alpha]\!].\mathit{get}_L\;x; \mathit{return}\;(a',(x',y))\} \\
\quad = \quad [\![\text{ definition of } [\![\alpha]\!].\mathit{get}_L \text{ in terms of monad } M \;\;]\!] \\
\mathbf{do}\,\{(a',x') \leftarrow \mathit{return}\;x.\mathit{get}_L; \mathit{return}\;(a',(x',y))\} \\
\quad = \quad [\![\text{ definition of } [\![\alpha \diamond \beta]\!].\mathit{get}_L \;\;]\!] \\
\mathbf{do}\,\{(a',x') \leftarrow [\![\alpha \diamond \beta]\!].\mathit{get}_L\;x; \mathit{return}\;(a',(x',y))\} \\
\quad = \quad [\![\text{ repacking } \mathit{left} \;\;]\!] \\
\mathit{left}\;([\![\alpha \diamond \beta]\!].\mathit{get}_L)\;(x,y) \\
\quad = \quad [\![\text{ definition of } \fatsemi \;\;]\!] \\
(\alpha \diamond \beta).\mathit{get}_L\;(x,y)
\end{array}
$$

$$
\begin{array}{l}
([\![\alpha]\!] \fatsemi [\![\beta]\!]).\mathit{set}_L\;a'\;(x,y) \\
\quad = \quad [\![\text{ definition of } \fatsemi, \text{ in monad } M \text{ rather than } \mathit{StateT}\;(X\times Y)\;M \;\;]\!] \\
\mathbf{do}\,\{(\_,(x',y')) \leftarrow \mathit{left}\;([\![\alpha]\!].\mathit{set}_L\;a')\;(x,y); \\
\qquad (b',(x'',y'')) \leftarrow \mathit{left}\;([\![\alpha]\!].\mathit{get}_R)\;(x',y'); \mathit{right}\;([\![\beta]\!].\mathit{set}_L\;b)\;(x'',y'')\} \\
\quad = \quad [\![\text{ unpacking } \mathit{left} \text{ and } \mathit{right}, \text{ inlining } y' = y \;\;]\!] \\
\mathbf{do}\,\{(\_,x') \leftarrow [\![\alpha]\!].\mathit{set}_L\;a'\;x; (b',x'') \leftarrow [\![\alpha]\!].\mathit{get}_R\;x'; \\
\qquad (\_,y'') \leftarrow [\![\beta]\!].\mathit{set}_L\;b'\;y; 1\;\mathit{return}\;((),(x'',y''))\} \\
\quad = \quad [\![\text{ definition of } \mathit{get} \text{ and } \mathit{set} \text{ for } [\![\cdot]\!] \text{ as given above } \;\;]\!] \\
\mathbf{do}\,\{(\_,x') \leftarrow x.\mathit{set}_L\;a'; (b',x'') \leftarrow x'.\mathit{get}_R; \\
\qquad (\_,y'') \leftarrow y.\mathit{set}_L\;b'; \mathit{return}\;((),(x'',y''))\} \\
\quad = \quad [\![\text{ definition of } \alpha \diamond \beta \text{ in Section 4 ((ii))} \;\;]\!] \\
\mathbf{do}\,\{(x'',y'') \leftarrow (\alpha \diamond \beta).\mathit{set}_L\;a'\;(x,y); \mathit{return}\;((),(x'',y''))\} \\
\quad = \quad [\![\text{ definition of } [\![\cdot]\!] \text{ on } (\alpha \diamond \beta) \;\;]\!] \\
[\![\alpha \diamond \beta]\!].\mathit{set}_L\;a'\;(x,y)
\end{array}
$$

□

Again, this result concerns individual cbx, and not the $\equiv$-equivalence classes used to define $\mathsf{Cbx}_\bullet$. We now comment on how one may embed the latter into a category of transparent mbx. Previously [ASCG+15, Theorem 26], we defined an equivalence on such mbx ($\sim$, say) given by operation-respecting state-space isomorphisms, and showed that $\bullet$-composition is associative up to $\sim$. In line with Corollary 4.13, one obtains a category $\mathsf{Mbx}_\bullet$ of $\sim$-equivalence-classes of transparent mbx, and initial states as at the start of Section 5.

Translating this into our setting (by reversing $[\![\cdot]\!]$ from Lemma 5.2), one finds that two transparent mbx are $\sim$-equivalent iff there is an *isomorphism* of pointed coalgebra

morphisms between their carriers – which is generally finer than $\equiv$. Therefore, we cannot hope for an equivalence-respecting embedding from $\mathsf{Cbx}_\bullet$ to $\mathsf{Mbx}_\bullet$. However, we may restrict $\equiv$ to make such an embedding possible:

Corollary 5.6. *Let $\equiv_!$ be the equivalence relation on cbx given by restricting $\equiv$ to bisimulations whose pointed coalgebra morphisms are isomorphisms; and let $\mathsf{Cbx}^!_\bullet$ be the category of $\equiv_!$-equivalence classes of cbx. Then there is an isomorphism $\mathsf{Cbx}^!_\bullet \cong \mathsf{Mbx}_\bullet$.*

6 Conclusions and Further Work

In our search for unifying foundations of the field of bidirectional transformations (bx), we have investigated a number of approaches, including monadic bx, building on those of (symmetric) lenses and relational bx.

We have given a coalgebraic account of state-based bx in terms of intuitively simple building blocks: two data sources A and B; a state space X; operations on X to observe (get_L, get_R) and update (set_L, set_R) each data source; an ambient monad M of additional computational effects; and a collection of laws, entirely analogous to those in existing bx formalisms. Our definition allows a conceptually more direct, if technically slightly subtle, treatment of composition than our previous work – the state space, defined by a pullback, captures the idea of communication across a shared data source, via the idea of B-consistent pairs. Our proof techniques involved reasoning about composition by considering operations defined on such pairs.

We defined an equivalence on cbx based on coalgebraic bisimulation, and showed that composition does indeed define a category, up to equivalence. The notion of bisimulation improves on existing, more ad-hoc definitions, such as that of symmetric lens equivalence, and we take this as further evidence for the suitability of our framework and definitions. We described several concrete instantiations of the general definition of bisimulation, given by varying the effect monad M. Coarser equivalences may be suitable for modelling non-deterministic cbx, and could be introduced via alternative methods such as coinductive trace semantics [HJS07]; but we do not explore this further here.

We have also investigated the relationship between our coalgebraic formulation of bx, and the spans of lenses considered by Johnson and Rosebrugh [JR14]. A coalgebraic bx may be seen as a span of "monadic lenses" or "M-lenses", generalising the notion of lens to allow the update operation to have side effects. In recent work elsewhere [ASCG+16], we have developed a fully-fledged theory of such M-lenses, and recast our notion of coalgebraic bisimulation in terms of an equivalence $\equiv_b$ on spans of M-lenses. We have further shown that this relation is coarser than (an analogue of) Johnson–Rosebrugh span equivalence. However, in defining equivalence for spans of lenses, they consider an additional condition, namely that the mediating arrow between two spans, witnessing equivalence, is a lens in which the get arrow is a split epi; our definition is subtly different, and accounts for the initialisation of M-lenses via a *create* function. We conjectured that this inclusion of relations between our notion of span equivalence and bisimulation equivalence is strict, but have left investigation of this to future work. Similarly, given that our notion of span equivalence does not admit a direct comparison with that of Johnson and Rosebrugh, we leave the precise connection with their notion also to future work.

Another area to explore is combinatory structure for building larger cbx out of smaller pieces, in the style of existing work in the lens community. Some examples

were given previously [ASCG+15] – of which mbx composition was by far the most challenging to formulate – and we expect the other combinators to apply routinely for cbx as well, along the same lines as that of Barbosa [Bar03]. One would expect coalgebraic bisimulation to be a congruence for these combinators; it would be interesting to investigate whether the work of Turi and Plotkin [TP97] can help here. It would also be interesting to explore whether our logical reasoning can be expressed in categorical logic, perhaps with reference to effectus theory [CJWW15]. For instance, B-consistency, in Definition 4.4 and Lemma 4.7 could be expressed in terms of effectful predicates – suitable arrows $X \to M\ (1 + 1)$ – rather than in terms of pairs $(B \times B)$.

By being explicit about the categorical structure used (such as Cartesian closure), we pave the way for exploration of other categorical settings for reasoning about the behaviour of bx – such as the category of small categories (Cat), or categories of partial orders (Cpo); *strict* order-preserving maps ($\mathsf{Cpo}_\perp$) would introduce an interplay of *monoidal* closure and Cartesian products. In settings without closed structure, one might consider categories of *components* rather than coalgebras [HJ11].

Our next challenge is to adapt our approach to model richer, more intensional structures, such as delta lenses [DXC11], edit lenses [HPW12] and ordered updates [Heg04], as opposed to the state-based, extensional bx we have considered, here and in [ASCG+15]. This would require a more detailed model of coalgebraic behaviour than the functor F^M_{AB}. It is not enough simply to equip the category $\mathbb{C}$ with such delta-structure, e.g. by taking $\mathbb{C} = \mathsf{Cat}$ (where objects are themselves small categories); that functor then 'over-specifies' the bx operations, e.g. set_L must then specify not just the propagation of deltas on A into deltas on the state-space X (as one would expect), but also how to *simultaneously propagate* pairs of A-deltas and X-deltas – which is unlikely to be desirable. We hope to explore the correct way to model delta-based bx in future work. We also hope to explore the connections between our work and existing coalgebraic notions of software components [Bar03], and techniques for composing coalgebras [Has11].

Acknowledgements We thank the JOT reviewers for their detailed feedback and suggestions. An extended, unpublished abstract [ASM14] of some of the results here was presented at CMCS 2014; we thank the participants there for useful feedback. Our work is supported by the UK EPSRC-funded project *A Theory of Least Change for Bidirectional Transformations* [TLC16] (EP/K020218/1, EP/K020919/1); we are grateful to our colleagues for their support, encouragement and feedback during the writing of the present paper.

References

[AMMS13] Jirí Adámek, Stefan Milius, Lawrence S. Moss, and Lurdes Sousa. Well-pointed coalgebras. *Logical Methods in Computer Science*, 9(3), 2013.

[ASCG+15] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Notions of bidirectional computation and entangled state monads. In *Mathematics of Program Construction*, volume 9129 of *Lecture Notes in Computer Science*, pages 187–214. Springer, June 2015. Extended version available at `http://groups.inf.ed.ac.uk/bx/bx-effects-tr.pdf`.

[ASCG+16] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Reflections on monadic lenses. In *A List of*

Successes that can Change the World, volume 9600 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 04 2016.

[ASM14] Faris Abou-Saleh and James McKinna. A coalgebraic approach to bidirectional transformations. In *Coalgebraic Methods in Computer Science*. ETAPS, 2014. Talk abstract.

[Bar03] Luís Soares Barbosa. Towards a calculus of state-based software components. *Journal of Universal Computer Science*, 9(8):891–909, 2003.

[CFH+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer-Verlag, 2009.

[CJWW15] Kenta Cho, Bart Jacobs, Bas Westerbaan, and Abraham Westerbaan. An introduction to effectus theory. *arXiv:1512.05813*, 2015.

[CMS+14] James Cheney, James McKinna, Perdita Stevens, Jeremy Gibbons, and Faris Abou-Saleh. Entangled state monads (extended abstract). In Terwilliger and Hidaka [TH14].

[DXC11] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From state- to delta-based bidirectional model transformations: The asymmetric case. *Journal of Object Technology*, 10:6: 1–25, 2011.

[FGM+07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007. doi:http://portal.acm.org/citation.cfm?doid=1232420.1232424.

[GH11] Jeremy Gibbons and Ralf Hinze. Just **do** it: Simple monadic equational reasoning. In *International Conference on Functional Programming*, pages 2–14. ACM Press, 2011.

[GJ12] Jeremy Gibbons and Michael Johnson. Relating algebraic and coalgebraic descriptions of lenses. *Proceedings of the First BX Workshop, Electronic Communication of the EASST*, 49, 2012. doi:http://dx.doi.org/10.14279/tuj.eceasst.49.726.

[Gum01] H. Peter Gumm. Functors for coalgebras. *Algebra Universalis*, 45(2-3):135–147, 2001.

[Gum09] H. Peter Gumm. Copower functors. *Theoretical Computer Science*, 410(12):1129–1142, 2009.

[Has11] Ichiro Hasuo. The microcosm principle and compositionality of GSOS-based component calculi. In *Algebra and Coalgebra in Computer Science*, pages 222–236. Springer, 2011.

[Heg04] Stephen J. Hegner. An order-based theory of updates for closed database views. *Annals of Mathematics and Artificial Intelligence*, 40:63–125, 2004.

[HJ11] Ichiro Hasuo and Bart Jacobs. Traces for coalgebraic components. *Mathematical Structures in Computer Science*, 21(02):267–320, 2011.

[HJS07] Ichiro Hasuo, Bart Jacobs, and Ana Sokolova. Generic trace semantics via coinduction. *Logical Methods in Computer Science*, 3(4), 2007. URL: `http://dx.doi.org/10.2168/LMCS-3(4:11)2007`, `doi:10.2168/LMCS-3(4:11)2007`.

[HPW11] Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In *Principles of Programming Languages*, pages 371–384. ACM Press, 2011.

[HPW12] Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Edit lenses. In *Principles of Programming Languages*, pages 495–508. ACM Press, 2012.

[HSST11] Zhenjiang Hu, Andy Schurr, Perdita Stevens, and James F. Terwilliger. Dagstuhl seminar 11031: Bidirectional transformations (BX). *SIGMOD Record*, 40(1):35–39, 2011. DOI: `10.4230/DagRep.1.1.42`. `doi:10.4230/DagRep.1.1.42`.

[JR13] Michael Johnson and Robert D. Rosebrugh. Delta lenses and opfibrations. *Electronic Communication of the EASST*, 57, 2013. `doi:10.14279/tuj.eceasst.57.875`.

[JR14] Michael Johnson and Robert Rosebrugh. Spans of lenses. In Terwilliger and Hidaka [TH14].

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[PP02] Gordon D. Plotkin and John Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, 2002.

[PS04] John Power and Olha Shkaravska. From comodels to coalgebras: State and arrays. *Proceedings of the Workshop on Coalgebraic Methods in Computer Science, Electronic Notes in Theoretical Computer Science*, 106, 2004.

[PW98] John Power and Hiroshi Watanabe. An axiomatics for categories of coalgebras. *Proceedings of the First Workshop on Coalgebraic Methods in Computer Science, Electronic Notes in Theoretical Computer Science*, 11:158–175, 1998.

[Rut00] Jan Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.

[Shk05] Olha Shkaravska. Side-effect monad, its equational theory and applications. Seminar slides available at: `http://www.ioc.ee/~tarmo/tsem05/shkaravska1512-slides.pdf`, 2005.

[SK08] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In *Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2008.

[Sok11] Ana Sokolova. Probabilistic systems coalgebraically: A survey. *Theoretical Computer Science*, 412(38):5095–5110, 2011.

[Ste10] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *Software and Systems Modelling*, 9(1):7–20, 2010.

[TH14] James Terwilliger and Soichiro Hidaka, editors. *BX Workshop*. http://ceur-ws.org/Vol-1133/#bx, 2014.

[TLC16] TLCBX Project. A theory of least change for bidirectional transformations. http://www.cs.ox.ac.uk/projects/tlcbx/, http://groups.inf.ed.ac.uk/bx/, 2013–2016.

[TP97] Daniele Turi and Gordon Plotkin. Towards a mathematical operational semantics. In *Logic in Computer Science*, pages 280–291. IEEE, Computer Society Press, 1997.

A Appendix: Omitted Details in Proof of Theorem 4.10

Proof. Note that when applying the laws ($\mathsf{CGetSet}_A$) (α) and ($\mathsf{CSetGet}_A$) (α), it is often very convenient to appeal to Lemma 4.3 and use the continuation versions of the laws. Below, we typically dispense with justification for sequences of steps involving little more than substituting definitions and basic product manipulations.

Identities We begin by showing the easier step (ii): $h = \langle \alpha.get_L, id \rangle$ equalizes the parallel pair $\alpha.get_L \circ \pi_2$ and $\iota(\epsilon_\alpha.get_L).get_R \circ \pi_1$, where the final step uses the fact that $\iota(\epsilon_\alpha.get_L).get_R = id : A \to A$ by definition of the identity bx (Example 2.6):

$$\begin{aligned}
&\alpha.get_L \circ \pi_2 \circ h \\
&= \alpha.get_L \circ \pi_2 \circ \langle \alpha.get_L, id \rangle \\
&= \alpha.get_L \\
&= id \circ \pi_1 \circ h \\
&= \iota(\epsilon_\alpha.get_L).get_R \circ \pi_1 \circ h
\end{aligned}$$

Now show step (i): h is a coalgebra morphism, as it satisfies

- $(\mathsf{CGetP}_L)(h)$:

$$(\iota(\epsilon_\alpha.get_L) \diamond \alpha).get_L \circ h = \iota(\epsilon_\alpha.get_L).get_L \circ \pi_1 \circ h = id \circ \pi_1 \circ h = \alpha.get_L$$

- $(\mathsf{CGetP}_R)(h)$:

$$(\iota(\epsilon_\alpha.get_L) \diamond \alpha).get_R \circ h = \alpha.get_R \circ \pi_2 \circ h = \alpha.get_R$$

- $(\mathsf{CSetP}_L)(h)$:

$$\begin{aligned}
&\mathbf{do}\,\{\mathbf{let}\ (i, x') = h\ x; (i', x'') \leftarrow (i, x').set_L\ a_0; return\ (i', x'')\} \\
=\ &[\![\ \text{defn. of } h, \text{ inlining } x' = x, \text{ writing } x.get_L \text{ for } \alpha.get_L\ x,\ x'' \text{ becomes } x'\ ]\!] \\
&\mathbf{do}\,\{\mathbf{let}\ i = x.get_L; (i', x') \leftarrow (i, x).set_L\ a_0; return\ (i', x')\} \\
=\ &[\![\ \text{definition of } (\iota(\epsilon_\alpha.get_L) \diamond \alpha).set_L\ ]\!] \\
&\mathbf{do}\,\{\mathbf{let}\ i = x.get_L; i' \leftarrow i.set_L\ a_0; \mathbf{let}\ a = i'.get_R; x' \leftarrow x.set_L\ a; return\ (i', x')\} \\
=\ &[\![\ i.set_L = return \text{ and } i'.get_R = i' \text{ by definition}\ ]\!] \\
&\mathbf{do}\,\{\mathbf{let}\ i = x.get_L; \mathbf{let}\ i' = a_0; \mathbf{let}\ a = i'; x' \leftarrow x.set_L\ a; return\ (i', x')\} \\
=\ &[\![\ \text{inline } \mathbf{let}\text{s}\ ]\!] \\
&\mathbf{do}\,\{x' \leftarrow x.set_L\ a_0; return\ (a_0, x')\} \\
=\ &[\![\ (\mathsf{CSetGet}_L)\ (\alpha)\ ]\!] \\
&\mathbf{do}\,\{x' \leftarrow x.set_L\ a_0; return\ (x'.get_L, x')\}
\end{aligned}$$

- $(\mathsf{CSetP}_R)(h)$:

$$\begin{aligned}
&\mathbf{do}\,\{\mathbf{let}\ (i, x') = h\ x; (i', x'') \leftarrow (i, x').set_R\ b_0; return\ (i', x'')\} \\
=\ &[\![\ \text{defn. of } h, \text{ inlining } x' = x, \text{ writing } x.get_L \text{ for } \alpha.get_L\ x,\ x'' \text{ becomes } x'\ ]\!] \\
&\mathbf{do}\,\{\mathbf{let}\ i = x.get_L; (i', x') \leftarrow (i, x).set_R\ b_0; return\ (i', x')\} \\
=\ &[\![\ \text{definition of } (\iota(\epsilon_\alpha.get_L) \diamond \alpha).set_R\ ]\!] \\
&\mathbf{do}\,\{\mathbf{let}\ i = x.get_L; x' \leftarrow x.set_R\ b_0; \mathbf{let}\ a' = x'.get_L; i' \leftarrow i.set_R\ a'; return\ (i', x')\} \\
=\ &[\![\ i.set_L = return \text{ and } i'.get_R = i' \text{ by definition}\ ]\!] \\
&\mathbf{do}\,\{\mathbf{let}\ i = x.get_L; x' \leftarrow x.set_R\ b_0; \mathbf{let}\ a' = x'.get_L; \mathbf{let}\ i' = a'; return\ (i', x')\} \\
=\ &[\![\ \text{inline } \mathbf{let}\text{s}\ ]\!] \\
&\mathbf{do}\,\{x' \leftarrow x.set_R\ b_0; return\ (x'.get_L, x')\}
\end{aligned}$$

Congruence To obtain a coalgebra structure on S and make ζ into a coalgebra morphism, it is useful to define a F_{AC}^{M}-coalgebra structure on $R \times Y$ as we did in Section 4(ii) – even though we have not assumed that (R, ϱ) is a coalgebraic bx – which we may as well call $\varrho \diamond \beta$, in exactly the same way as Definition 4.6.

We now show step (i) of the proof, that $(p \times id) \circ \zeta$ indeed equalizes the parallel pair of the diagram 1 on page 18:

$$
\begin{array}{ll}
 & \alpha.get_R \circ \pi_1 \circ (p \times id) \circ \zeta \\
= & \quad [\![\ \pi_1 \text{ is a natural transformation} \ ]\!] \\
 & \alpha.get_R \circ p \circ \pi_1 \circ \zeta \\
= & \quad [\![\ (\mathsf{CGetP}_R)(p) \ ]\!] \\
 & r.get_R \circ \pi_1 \circ \zeta \\
= & \quad [\![\ \text{definition of } \zeta \text{ as an equalizer} \ ]\!] \\
 & \beta.get_L \circ \pi_2 \circ \zeta \\
= & \quad [\![\ \text{properties of products} \ ]\!] \\
 & \beta.get_L \circ \pi_2 \circ (p \times id) \circ \zeta
\end{array}
$$

Now for step (ii), that $p \times id$ is a coalgebra morphism from $\varrho \diamond \beta$ to $\alpha \diamond \beta$.

- $(\mathsf{CGetP}_L)(p \times id)$:

$$
\begin{array}{ll}
 & ((p \times id)\ (r, y)).get_L \\
= & (p\ (r), y).get_L \\
= & \quad [\![\ \text{definition of } \varrho \diamond \beta \ ]\!] \\
 & p\ (r).get_L \\
= & \quad [\![\ (\mathsf{CGetSet}_L)\ (p) \ ]\!] \\
 & r.get_L \\
= & (r, y).get_L
\end{array}
$$

- $(\mathsf{CGetP}_R)(p \times id)$:

  As above, $((p \times id)\ (r, y)).get_R = y.get_R = (r, y).get_R$.

- $(\mathsf{CSetP}_L)(p \times id)$:

$$
\begin{array}{ll}
 & \mathbf{do}\ \{(r', y') \leftarrow (r, y).set_L\ a; return\ (p\ (r'), y')\} \\
= & \quad [\![\ \text{definition of } (\varrho \diamond \beta).set_L \ ]\!] \\
 & \mathbf{do}\ \{r' \leftarrow r.set_L\ a; \mathbf{let}\ b' = r'.get_R; y' \leftarrow y.set_L\ b'; return\ (p\ (r'), y')\} \\
= & \quad [\![\ (\mathsf{CSetP}_L)(\varrho) \ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ x = p\ (r); x' \leftarrow x.set_L\ a; \mathbf{let}\ b' = x'.get_R; y' \leftarrow y.set_L\ b'; return\ (x', y')\} \\
= & \quad [\![\ \text{definition of } (\alpha \diamond \beta).set_L \ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ x = p\ (r); (x', y') \leftarrow (x, y).set_L\ a; return\ (x', y')\} \\
= & \quad [\![\ \text{properties of } \mathbf{let} \text{ binding} \ ]\!] \\
 & \mathbf{do}\ \{\mathbf{let}\ (x, y') = (p \times id)\ (r, y); (x', y'') \leftarrow (x, y').set_L\ a; return\ (x', y'')\}
\end{array}
$$

- $(\mathsf{CSetP}_R)(p \times id)$:

$$
\begin{array}{ll}
 & \mathbf{do}\ \{(r', y') \leftarrow (r, y).set_R\ c; return\ (p\ (r'), y')\} \\
= & \quad [\![\ \text{definition of } (\varrho \diamond \beta).set_R \ ]\!] \\
 & \mathbf{do}\ \{y' \leftarrow y.set_R\ c; \mathbf{let}\ b' = y'.get_L; r' \leftarrow r.set_R\ b'; return\ (p\ (r'), y')\} \\
= & \quad [\![\ (\mathsf{CSetP}_R)(\varrho) \ ]\!]
\end{array}
$$

$$
\begin{array}{ll}
& \mathbf{do}\,\{\,y' \leftarrow y.set_R\ c; \mathbf{let}\ b' = y'.get_L; \mathbf{let}\ x = p\ (r); x' \leftarrow x.set_R\ b'; return\ (x', y')\,\} \\
= & \quad [\![\ \text{move } \mathbf{let}\ x = p\ (r) \text{ to front; definition of } (\alpha \diamond \beta).set_R\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ x = p\ (r); (x', y') \leftarrow (x, y).set_R\ a; return\ (x', y')\,\} \\
= & \quad [\![\ \text{properties of } \mathbf{let} \text{ binding}\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x, y') = (p \times id)\ (r, y); (x', y'') \leftarrow (x, y').set_R\ a; return\ (x', y'')\,\}
\end{array}
$$

We check that the coalgebra morphisms are pointed: by construction (see Section 4(i)), the initial state ϵ_s of S satisfies $\zeta\ (\epsilon_s) = (\epsilon_r, \epsilon_\beta)$, and the image of this under $(p \times id)$ is $(\epsilon_\alpha, \epsilon_\beta)$ as required, as p is a pointed coalgebra morphism.

Associativity We conclude the proof of associativity by completing the proofs of (i)–(iv) for the coalgebra morphism p.

(i): (where $u : P_{(\alpha \bullet \beta),\gamma}$, and variable v is of type $P_{\alpha,\beta}$)

$$
\begin{array}{ll}
& ((id \times (\beta.get_R \circ \pi_1)) \circ f \circ e_{(\alpha \bullet \beta),\gamma})\ u \\
= & \quad [\![\ \text{definition of } f, \text{ inlining } \mathbf{let}\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (v, z) = e_{(\alpha \bullet \beta),\gamma}\ u; \\
& \qquad \mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (v); return\ (id \times (\beta.get_R \circ \pi_1))\ (x, (y, z))\,\} \\
= & \quad [\![\ \text{simplifying product}\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (v, z) = e_{(\alpha \bullet \beta),\gamma}\ u; \mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (v); return\ (x, y.get_R)\,\} \\
= & \quad [\![\ (\mathsf{CGetP}_R)(e_{\alpha,\beta})\ (\text{replace } y.get_R = ((x, y).get_R) \text{ with } v.get_R)\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (v, z) = e_{(\alpha \bullet \beta),\gamma}\ u; \mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (v); return\ (x, v.get_R)\,\} \\
= & \quad [\![\ v.get_R = z.get_L \text{ By definition of the equalizer } e_{(\alpha \bullet \beta),\gamma}\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (v, z) = e_{(\alpha \bullet \beta),\gamma}\ u; \mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (v); return\ (x, z.get_L)\,\} \\
= & \quad [\![\ \text{un-simplifying product}\ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (v, z) = e_{(\alpha \bullet \beta),\gamma}\ u; \\
& \qquad \mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (v); return\ (id \times (\gamma.get_L \circ \pi_2))\ (x, (y, z))\,\} \\
= & \quad [\![\ \text{definition of } f\ ]\!] \\
& ((id \times (\gamma.get_L \circ \pi_2)) \circ f \circ e_{(\alpha \bullet \beta),\gamma})\ u
\end{array}
$$

(ii): The initial state $\epsilon_{\alpha \diamond (\beta \bullet \gamma)} = (\epsilon_\alpha, \epsilon_{\beta \bullet \gamma})$ is mapped by $(id \times e_{\beta,\gamma})$ to $(\epsilon_\alpha, (\epsilon_\beta, \epsilon_\gamma))$ by definition of the equalizer $e_{\beta,\gamma}$, and the latter is precisely the initial state of $\alpha \diamond (\beta \diamond \gamma)$. We now check the coalgebra morphism laws (where $u : P_{\beta,\gamma}$).

- $(\mathsf{CGetP}_L)(id \times e_{\beta,\gamma})$:

$$
\begin{array}{ll}
& (\alpha \diamond (\beta \diamond \gamma)).get_L \circ (id \times e_{\beta,\gamma})\ (x, u) \\
= & \alpha.get_L \circ \pi_1 \circ (id \times e_{\beta,\gamma})\ (x, u) \\
= & \alpha.get_L\ x \\
= & \alpha.get_L \circ \pi_1\ (x, u) \\
= & (\alpha \diamond (\beta \bullet \gamma)).get_L\ (x, u)
\end{array}
$$

- $(\mathsf{CGetP}_R)(id \times e_{\beta,\gamma})$:

$$
\begin{array}{ll}
& (\alpha \diamond (\beta \diamond \gamma)).get_R \circ (id \times e_{\beta,\gamma})\ (x, u) \\
= & (\beta \diamond \gamma).get_R \circ \pi_2 \circ (id \times e_{\beta,\gamma})\ (x, u) \\
= & (\beta \diamond \gamma).get_R \circ e_{\beta,\gamma}\ (u) \\
= & \quad [\![\ (\mathsf{CGetP}_R)(e_{\beta,\gamma})\ ]\!] \\
& (\beta \bullet \gamma).get_R\ (u) \\
= & (\beta \bullet \gamma).get_R \circ \pi_2\ (x, u) \\
= & (\alpha \diamond (\beta \bullet \gamma)).get_R\ (x, u)
\end{array}
$$

- $(\mathsf{CSetP}_L)(id \times e_{\beta,\gamma})$:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\mathbf{let}\ (x',(y,z)) = (id \times e_{\beta,\gamma})\ (x,u);\\
& \qquad (x'',(y',z')) \leftarrow (x',(y,z)).set_L\ a_0; return\ (x'',(y',z'))\}\\
= & \quad [\![\ \text{properties of products; inlining } \mathbf{let}\text{s, alpha-converting } x'' \to x'\ ]\!]\\
& \mathbf{do}\,\{\mathbf{let}\ (y,z) = e_{\beta,\gamma}\ (u);\\
& \qquad (x',(y',z')) \leftarrow (x,(y,z)).set_L\ a_0; return\ (x',(y',z'))\}\\
= & \quad [\![\ \text{definition of } (\alpha \diamond (\beta \diamond \gamma)).set_L\ ]\!]\\
& \mathbf{do}\,\{\mathbf{let}\ (y,z) = e_{\beta,\gamma}\ (u); x' \leftarrow x.set_L\ a_0;\\
& \qquad \mathbf{let}\ b = x'.get_R; (y',z') \leftarrow (y,z).set_L\ b; return\ (x',(y',z'))\}\\
= & \quad [\![\ (\mathsf{CSetP}_L)(e_{\beta,\gamma})\ ]\!]\\
& \mathbf{do}\,\{x' \leftarrow x.set_L\ a_0; \mathbf{let}\ b = x'.get_R; u' \leftarrow u.set_L\ b; return\ (x', e_{\beta,\gamma}\ (u'))\}\\
= & \quad [\![\ \text{definition of } (\alpha \diamond (\beta \bullet \gamma)).set_L\ ]\!]\\
& \mathbf{do}\,\{(x',u') \leftarrow (x,u).set_L\ a_0; return\ (x', e_{\beta,\gamma}\ (u'))\}\\
= & \quad [\![\ \text{properties of products}\ ]\!]\\
& \mathbf{do}\,\{(x',u') \leftarrow (x,u).set_L\ a_0; return\ (id \times e_{\beta,\gamma})\ (x',u')\}
\end{array}
$$

- $(\mathsf{CSetP}_R)(id \times e_{\beta,\gamma})$: (This is very similar.)

$$
\begin{array}{ll}
& \mathbf{do}\,\{\mathbf{let}\ (x',(y,z)) = (id \times e_{\beta,\gamma})\ (x,u);\\
& \qquad (x'',(y',z')) \leftarrow (x',(y,z)).set_R\ d_0; return\ (x'',(y',z'))\}\\
= & \quad [\![\ \text{properties of products; inlining } \mathbf{let}\text{s, alpha-converting } x'' \to x'\ ]\!]\\
& \mathbf{do}\,\{\mathbf{let}\ (y,z) = e_{\beta,\gamma}\ (u);\\
& \qquad (x',(y',z')) \leftarrow (x,(y,z)).set_R\ d_0; return\ (x',(y',z'))\}\\
= & \quad [\![\ \text{definition of } (\alpha \diamond (\beta \diamond \gamma)).set_L\ ]\!]\\
& \mathbf{do}\,\{\mathbf{let}\ (y,z) = e_{\beta,\gamma}\ (u); (y',z') \leftarrow (y,z).set_R\ d_0;\\
& \qquad \mathbf{let}\ b = (y',z').get_L; x' \leftarrow x.set_R\ b; return\ (x',(y',z'))\}\\
= & \quad [\![\ (\mathsf{CSetP}_R)(e_{\beta,\gamma})\ \text{(continuation version)}\ ]\!]\\
& \mathbf{do}\,\{u' \leftarrow u.set_R\ d_0;\\
& \qquad \mathbf{let}\ b = (e_{\beta,\gamma}\ (u')).get_L; x' \leftarrow x.set_R\ b; return\ (x', e_{\beta,\gamma}\ (u'))\}\\
= & \quad [\![\ (\mathsf{CGetP}_L)(e_{\beta,\gamma})\ ]\!]\\
& \mathbf{do}\,\{u' \leftarrow u.set_R\ d_0; \mathbf{let}\ b = u'.get_L; x' \leftarrow x.set_R\ b; return\ (x', e_{\beta,\gamma}\ (u'))\}\\
= & \quad [\![\ \text{definition of } (\alpha \diamond (\beta \bullet \gamma)).set_R\ ]\!]\\
& \mathbf{do}\,\{(x',u') \leftarrow (x,u).set_R\ d_0; return\ (x', e_{\beta,\gamma}\ (u'))\}\\
= & \quad [\![\ \text{properties of products}\ ]\!]\\
& \mathbf{do}\,\{(x',u') \leftarrow (x,u).set_R\ d_0; return\ (id \times e_{\beta,\gamma})\ (x',u')\}
\end{array}
$$

(iii): Firstly, $f\ (u,z) = \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); return\ (x,(y,z))\}$ maps the initial state $(\epsilon_{\alpha \bullet \beta}, \epsilon_\gamma)$ to the initial state $(\epsilon_\alpha, (\epsilon_\beta, \epsilon_\gamma))$ as required.

- $(\mathsf{CGetP}_L)(f)$: (where $u : P_{\beta,\gamma}$)

$$
\begin{array}{ll}
& (\alpha \diamond (\beta \diamond \gamma)).get_L \circ f\ (u,z)\\
= & \alpha.get_L \circ \pi_1 \circ f\ (u,z)\\
= & \mathbf{do}\,\{\mathbf{let}\ (x,(y,z)) = f\ (u,z); return\ x.get_L\}\\
= & \quad [\![\ \text{definition of } f, \text{ inlining } \mathbf{let}\ ]\!]\\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); return\ x.get_L\}\\
= & \quad [\![\ \text{definition of } (\alpha \diamond \beta).get_L\ ]\!]\\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); return\ (x,y).get_L\}\\
= & \quad [\![\ (\mathsf{CGetP}_L)(e_{\alpha,\beta})\ ]\!]
\end{array}
$$

$$
\begin{array}{ll}
& \mathbf{do}\,\{\mathit{return}\ u.\mathit{get}_L\} \\
= & (\alpha \bullet \beta).\mathit{get}_L\ u = (\alpha \bullet \beta).\mathit{get}_L \circ \pi_1 \circ (u,z) = ((\alpha \bullet \beta) \diamond \gamma).\mathit{get}_L\ (u,z)
\end{array}
$$

- $(\mathsf{CGetP}_R)(f)$:

$$
\begin{array}{ll}
& (\alpha \diamond (\beta \diamond \gamma)).\mathit{get}_R \circ f\ (u,z) \\
= & (\beta \diamond \gamma).\mathit{get}_R \circ \pi_2 \circ f\ (u,z) \\
= & \mathbf{do}\,\{\mathbf{let}\ (x,(y,z')) = f\ (u,z); \mathit{return}\ (y,z').\mathit{get}_R\} \\
= & \quad [\![\text{ definition of } (\beta \diamond \gamma).\mathit{get}_R \ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,(y,z')) = f\ (u,z); \mathit{return}\ z'.\mathit{get}_R\} \\
= & \quad [\![\text{ definition of } f\text{; inlining } \mathbf{let}\text{s} \ ]\!] \\
& \mathbf{do}\,\{\mathit{return}\ z.\mathit{get}_R\} \\
= & \gamma.\mathit{get}_R\ z = \gamma.\mathit{get}_R \circ \pi_2\ (u,z) = ((\alpha \bullet \beta) \diamond \gamma).\mathit{get}_R\ (u,z)
\end{array}
$$

- $(\mathsf{CSetP}_L)(f)$:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\mathbf{let}\ (x,(y,z')) = f\ (u,z); \\
& \qquad (x',(y',z'')) \leftarrow (x,(y,z')).\mathit{set}_L\ a_0; \mathit{return}\ (x',(y',z''))\} \\
= & \quad [\![\text{ definition of } f \ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); \\
& \qquad (x',(y',z')) \leftarrow (x,(y,z)).\mathit{set}_L\ a_0; \mathit{return}\ (x',(y',z'))\} \\
= & \quad [\![\text{ definition of } (\alpha \diamond (\beta \diamond \gamma)).\mathit{set}_L \ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); x' \leftarrow x.\mathit{set}_L\ a_0; \mathbf{let}\ b = x'.\mathit{get}_R; \\
& \qquad (y',z') \leftarrow (y,z).\mathit{set}_L\ b; \mathit{return}\ (x',(y',z'))\} \\
= & \quad [\![\text{ definition of } (\beta \diamond \gamma).\mathit{set}_L \ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); x' \leftarrow x.\mathit{set}_L\ a_0; \mathbf{let}\ b = x'.\mathit{get}_R; \\
& \qquad y' \leftarrow y.\mathit{set}_L\ b; \mathbf{let}\ c = y'.\mathit{get}_R; \\
& \qquad z' \leftarrow z.\mathit{set}_L\ c; \mathit{return}\ (x',(y',z'))\} \\
= & \quad [\![\text{ definitions of } (\alpha \diamond \beta).\mathit{set}_L \text{ and } \mathit{get}_R \ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); (x',y') \leftarrow (x,y).\mathit{set}_L\ a_0; \mathbf{let}\ c = (x',y').\mathit{get}_R; \\
& \qquad z' \leftarrow z.\mathit{set}_L\ c; \mathit{return}\ (x',(y',z'))\} \\
= & \quad [\![\text{ inserting redundant } \mathbf{let} \text{ (to simplify dependencies on } (x',y')) \ ]\!] \\
& \mathbf{do}\,\{\mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); (x',y') \leftarrow (x,y).\mathit{set}_L\ a_0; \mathbf{let}\ (x'',y'') = (x',y'); \\
& \qquad \mathbf{let}\ c = (x'',y'').\mathit{get}_R; z' \leftarrow z.\mathit{set}_L\ c; \mathit{return}\ (x'',(y'',z'))\} \\
= & \quad [\![\ (\mathsf{CSetP}_L)(e_{\alpha,\beta}) \text{, continuation version} \ ]\!] \\
& \mathbf{do}\,\{u' \leftarrow u.\mathit{set}_L\ a_0; \mathbf{let}\ (x'',y'') = e_{\alpha,\beta}\ (u'); \mathbf{let}\ c = (x'',y'').\mathit{get}_R; \\
& \qquad z' \leftarrow z.\mathit{set}_L\ c; \mathit{return}\ (x'',(y'',z'))\} \\
= & \quad [\![\text{ inline } \mathbf{let} \text{ and apply } (\mathsf{CGetP}_L)(e_{\alpha,\beta}) \ ]\!] \\
& \mathbf{do}\,\{u' \leftarrow u.\mathit{set}_L\ a_0; \mathbf{let}\ (x'',y'') = e_{\alpha,\beta}\ (u'); \mathbf{let}\ c = u'.\mathit{get}_R; \\
& \qquad z' \leftarrow z.\mathit{set}_L\ c; \mathit{return}\ (x'',(y'',z'))\} \\
= & \quad [\![\text{ move } \mathbf{let}\ (x'',y'')\text{= ...; definition of } ((\alpha \bullet \beta) \diamond \gamma).\mathit{set}_L \ ]\!] \\
& \mathbf{do}\,\{(u',z') \leftarrow (u,z).\mathit{set}_L\ a_0; \mathbf{let}\ (x'',y'') = e_{\alpha,\beta}\ (u'); \mathit{return}\ (x'',(y'',z'))\} \\
= & \quad [\![\text{ definition of } f \ ]\!] \\
& \mathbf{do}\,\{(u',z') \leftarrow (u,z).\mathit{set}_L\ a_0; \mathit{return}\ f\ (u',z')\}
\end{array}
$$

- $(\mathsf{CSetP}_R)(f)$:

$$
\begin{array}{l}
\mathbf{do}\,\{\mathbf{let}\ (x,(y,z')) = f\ (u,z); \\
\qquad (x',(y',z'')) \leftarrow (x,(y,z')).\mathit{set}_R\ d_0; \mathit{return}\ (x',(y',z''))\}
\end{array}
$$

$$
\begin{array}{ll}
= & [\![\text{ definition of } f \ ]\!] \\
& \textbf{do}\,\{\textbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); \\
& \quad (x',(y',z')) \leftarrow (x,(y,z)).set_R\ d_0; \mathit{return}\ (x',(y',z'))\} \\
= & [\![\text{ definition of } (\alpha \diamond (\beta \diamond \gamma)).set_R \ ]\!] \\
& \textbf{do}\,\{\textbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); \\
& \quad (y',z') \leftarrow (y,z).set_R\ d_0; \\
& \quad \textbf{let}\ b = (y',z').get_L; x' \leftarrow x.set_R\ b; \\
& \quad \mathit{return}\ (x',(y',z'))\} \\
= & [\![\text{ definition of } (\beta \diamond \gamma).set_L \ ]\!] \\
& \textbf{do}\,\{\textbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); \\
& \quad z' \leftarrow z.set_R\ d_0; \textbf{let}\ c = z'.get_L; \\
& \quad y' \leftarrow y.set_R\ c; \textbf{let}\ b = y'.get_L; x' \leftarrow x.set_R\ b; \mathit{return}\ (x',(y',z'))\} \\
= & [\![\text{ definition of } (\alpha \diamond \beta).set_R \ ]\!] \\
& \textbf{do}\,\{\textbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); \\
& \quad z' \leftarrow z.set_R\ d_0; \textbf{let}\ c = z'.get_L; \\
& \quad (x',y') \leftarrow (x,y).set_R\ c; \mathit{return}\ (x',(y',z'))\} \\
= & [\![\text{ moving } \textbf{let}\text{; inserting a redundant } \textbf{let} \text{ as above }]\!] \\
& \textbf{do}\,\{z' \leftarrow z.set_R\ d_0; \textbf{let}\ c = z'.get_L; \textbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); \\
& \quad (x',y') \leftarrow (x,y).set_R\ c; \textbf{let}\ (x'',y'') = (x',y'); \mathit{return}\ (x'',(y'',z'))\} \\
= & [\![\ (\mathsf{CSetP}_R)(e_{\alpha,\beta}) \ ]\!] \\
& \textbf{do}\,\{z' \leftarrow z.set_R\ d_0; \textbf{let}\ c = z'.get_L; \\
& \quad u' \leftarrow u.set_R\ c; \textbf{let}\ (x'',y'') = e_{\alpha,\beta}\ (u'); \mathit{return}\ (x'',(y'',z'))\} \\
= & [\![\text{ definition of } ((\alpha \bullet \beta) \diamond \gamma).set_R \ ]\!] \\
& \textbf{do}\,\{(u',z') \leftarrow (u,z).set_R\ d_0; \textbf{let}\ (x'',y'') = e_{\alpha,\beta}\ (u'); \mathit{return}\ (x'',(y'',z'))\} \\
= & [\![\text{ definition of } f \ ]\!] \\
& \textbf{do}\,\{(u',z') \leftarrow (u,z).set_L\ a_0; \mathit{return}\ f\ (u',z')\}
\end{array}
$$

(iv): (where $u : P_{(\alpha\bullet\beta),\gamma}$ and $v : P_{\alpha,\beta}$)

$$
\begin{array}{ll}
& \alpha.get_R \circ \pi_1 \circ p_0\ (u) \\
= & [\![\text{ properties of products }]\!] \\
& \pi_2 \circ (id \times (\alpha.get_R \circ \pi_1)) \circ (id \times e_{\beta,\gamma}) \circ p_0\ (u) \\
= & [\![\text{ commutativity of square in Diagram 2, page 18 }]\!] \\
& \alpha.get_R \circ \pi_1 \circ f \circ e_{(\alpha\bullet\beta),\gamma}\ (u) \\
= & [\![\text{ definition of } f\text{; properties of products }]\!] \\
& \textbf{do}\,\{\textbf{let}\ (v,z) = e_{(\alpha\bullet\beta),\gamma}\ (u); \textbf{let}\ (x,y) = e_{\alpha,\beta}\ (v); \mathit{return}\ x.get_R\} \\
= & [\![\ x.get_R = y.get_L \text{ by Remark 4.5 }]\!] \\
& \textbf{do}\,\{\textbf{let}\ (v,z) = e_{(\alpha\bullet\beta),\gamma}\ (u); \textbf{let}\ (x,y) = e_{\alpha,\beta}\ (v); \mathit{return}\ y.get_L\} \\
= & [\![\text{ definition of } f \ ]\!] \\
& \textbf{do}\,\{\textbf{let}\ (v,z) = e_{(\alpha\bullet\beta),\gamma}\ (u); \textbf{let}\ (x,(y,z')) = f\ (v,z); \mathit{return}\ y.get_L\} \\
= & [\![\text{ definition of } \beta \diamond \gamma.get_L \ ]\!] \\
& \textbf{do}\,\{\textbf{let}\ (v,z) = e_{(\alpha\bullet\beta),\gamma}\ (u); \textbf{let}\ (x,(y,z')) = f\ (v,z); \mathit{return}\ (y,z').get_L\} \\
= & (\beta \diamond \gamma).get_L \circ \pi_2 \circ f \circ e_{(\alpha\bullet\beta),\gamma}\ (u) \\
= & [\![\text{ commutativity of square in Diagram 2 again }]\!] \\
& (\beta \diamond \gamma).get_L \circ \pi_2 \circ (id \times e_{\beta,\gamma}) \circ p_0\ (u) \\
= & [\![\text{ properties of products }]\!] \\
& (\beta \diamond \gamma).get_L \circ e_{\beta,\gamma} \circ \pi_2 \circ p_0\ (u) \\
= & [\![\ (\mathsf{CGetP}_L)(e_{\beta,\gamma}) \ ]\!] \\
& (\beta \bullet \gamma).get_L \circ \pi_2 \circ p_0\ (u)
\end{array}
$$

□