Coalgebraic Aspects of Bidirectional Computation

Faris Abou-Saleh[1] James McKinna[2] Jeremy Gibbons[1]

1. Department of Computer Science
University of Oxford, UK
{Faris.Abou-Saleh,Jeremy.Gibbons}@cs.ox.ac.uk

2. School of Informatics
University of Edinburgh, UK
James.McKinna@ed.ac.uk

Abstract

We have previously (Bx, 2014; MPC, 2015) 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.

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 [6]. Various formalisms, collectively known as *bidirectional transformations* (bx), have been developed to study them, including (a)symmetric lenses [8, 16], relational bx [30], and triple-graph grammars [27].

In recent years, there has been a drive to understand the similarities and differences between these formalisms [18]; and a few attempts have been made to give a unified treatment. In previous work [5, an extended abstract] and [1, to appear], 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 operations, $get_L : MA$ and $set_L : A \to M\ ()$, and similarly get_R, set_R for B, which allow lookups and updates on both A and B while maintaining R. We defined an *effectful bx* over a monad M in terms of these four operations, written as $t : A \Longleftrightarrow_M B$, subject to several equations in line with the Plotkin–Power equational theory of state [23]. The key difference is that in a bidirectional context, the sources A and B are interdependent, or *entangled*: updates to A (via set_L) should in general affect B, and vice versa. Thus we must abandon some of the Plotkin–Power equations; for instance, one no longer expects set_L to commute with set_R. To distinguish our earlier effectful bx from the coalgebraic treatment to be developed in this paper, we will refer to them as *monadic bx*, or simply *mbx*, from now on.

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. Moreover, we introduced a new class of bx: monadic bx with effects, expressed in terms of 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

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: A. Cunha, E. Kindler (eds.): Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015), L'Aquila, Italy, July 24, 2015, published at http://ceur-ws.org

composition $t_1 \fatsemi t_2$ for those 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 [28]. As with symmetric lenses, composition can only be well-behaved up to some notion of equivalence, due to the different state-spaces involved. We showed that our definition of composition was associative and had identities up to an equivalence defined by monad morphisms – in particular, given by isomorphisms of state-spaces.

Although this equivalence is adequate for analysing such properties, it proves unsuitably fine-grained for comparing other aspects of bx behaviour. For instance, one may be interested in optimisation; one would then like some means of reasoning about different implementations, to check that they indeed have the same behaviour. Such reasoning is not possible based solely on state-space isomorphisms. As most of our work considers state-based bx, it is natural to ask whether one can identify an alternative, less ad-hoc, notion of equivalence.

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 [24] for the theory of mutable state. Previously, we considered structure relative to the categories Set, Hask (Haskell types and functions); in particular, we argued that the composite mbx operations we defined on the underlying simple types respected appropriate set-theoretic invariants. Here, the coalgebraic approach has several benefits: firstly, it gives a more direct categorical treatment of bx composition, in terms of pullbacks. 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 bisimularity [26]; in particular, we relate it to the equivalence on symmetric lenses [16].

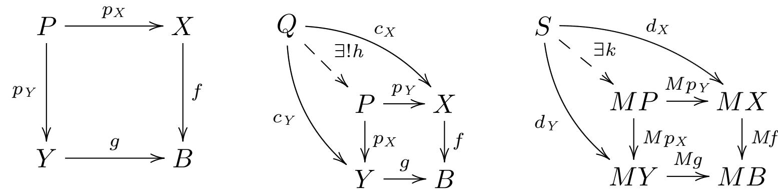
The technical contributions and structure of this paper are as follows. Firstly, in Section 2 we identify a categorical setting, and useful properties, for interpreting bx in terms of coalgebras. In Section 3, we introduce an equivalence on coalgebras, namely pointed coalgebraic bisimulation, and demonstrate how this equivalence relates to that of symmetric lenses, in addition to various effectful bx scenarios. (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 [1], 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 the equivalence we have introduced. Finally, in Section 5 we show that coalgebraic bx composition is coherent with that of [1].

NB We discuss only key ideas and techniques, with some proofs sketched; details are given in Appendix A.

2 Coalgebraic bx

2.1 Categorical Prerequisites

1. We will assume an ambient category $\mathbb{C}$ with finite products $X \times Y$ and exponentials X^Y (essentially, the space of functions from Y to X), on which a strong monad M is defined. (See for example Moggi's original paper [21] for a discussion of the need for such structure.) This allows us to make free use of the equational theory of **do**-notation as the internal language of the Kleisli category $\mathsf{Kl}(M)$ (see [9] for more 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* [11]; 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 'square' 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 'square' in the right-hand diagram commute, there is an arrow k, *not necessarily unique*, making the whole diagram commute.



3. For technical reasons, we assume that the *return* of the monad M is monic: this allows us to observe the value x wrapped in an effect-free computation *return* x. Most computational monads (we are unaware of natural counterexamples) have this property: *e.g.* global state, I/O, non-determinism, and exceptions [22].
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 might be the terminal object 1, but if $\mathbb{C}$ is symmetric monoidal closed (*e.g.* pointed cpos and strict maps), then I could be the monoidal unit.

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 [12, 29], 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 wpp holds for such M [11, Theorem 2.8].*

2.2 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 (see [3] for more general considerations). Corollary 4.14 makes explicit the categorical structure of bx represented by such structures.

Definition 2.2. *For an 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, typically writing α 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:

- **Interactive I/O.** In [1] 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 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, define the functor $F_{AB}^{M}(-) = A \times (M(-))^A \times B \times (M(-))^B$.*

For a given choice of objects A and B, we will use the corresponding functor F_{AB}^{M} to characterise the behaviour of those bx between A and B, as F_{AB}^{M}-coalgebras. By taking projections, we can see the structure map $\alpha : X \to F_{AB}^{M}\ X$ of a pointed F_{AB}^{M}-coalgebra as supplying four pieces of data: a function $X \to A$ which observes the A-value in a state X; a function $X \to (MX)^A$ which, uncurried into the form $X \times A \to MX$, gives us the side-effectful result MX of updating the A-value in the state X; and two similar functions for B. These respectively correspond to the get_L, set_L, get_R, and set_R functions of a monadic bx with respect to the state monad M_X (with state-space X), as outlined above. Note that in this coalgebraic presentation, the behaviour functor makes clear that the *get* operations are pure functions of the coalgebra state – corresponding to 'transparent' bx [1].

Convention 2.4. *Given $\alpha : X \to F_{AB}^{M}\ X$, 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.*

Remark 2.5. *Note that F^M_{AB} may be defined in terms of the costore comonad $S \times (-)^S$, used in [24] to describe arrays of independent variables. However, we differ in not allowing simultaneous update of each source (which would take $S = A \times B$), because updates to A may affect B, and vice versa.*

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 2.6. *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. as per Convention 2.4):*

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

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

These laws are the analogues of the (GS), (SG) *laws [1] which generalise those for well-behaved lenses [8, see also Section 5.2 below]. They also correspond to a subset of the laws for coalgebras of the costore comonad $S \times (-)^S$, again for $S = A$ and $S = B$ independently, but excluding the analogue of 'Put-Put' or very-well-behavedness of lenses [10]. We typically refer to a cbx by its structure map, and simply write $\alpha : A \rightleftharpoons_X B$, where we may further omit explicit mention of X.*

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 bx for each *pair* of an object A and initial value $e : A$.

Example 2.7. *Given $(A, e : A)$, there is a trivial cbx structure $\iota(e) : A \rightleftharpoons_A A$ defined by $\epsilon_{\iota(e)} = e$; $a.get_L = a = a.get_R$; $a.set_L\ a' = return\ a' = a.set_R\ a'$.*

Remark 2.8. *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, indeed, maintain consistency wrt 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 bx described in Section 2.2: interactive I/O, failure, and non-determinism.

Definition 3.1. *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 3.2. *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):*

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

We now define a modest generalisation to $\mathbb{C}$ of the standard Set-based definition of (coalgebraic) bisimulation relations [19]. (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 3.3. *A* bisimulation between pointed coalgebras α and β *is a pointed coalgebra ζ and a span (p, q) of pointed coalgebra morphisms $p : \zeta \to \alpha$, $q : \zeta \to \beta$ which is* jointly monic *(in $\mathbb{C}$) – meaning that whenever $p \circ f = p \circ f'$ and $q \circ f = q \circ f'$, we have $f = f'$.*

Definition 3.3 characterises bisimulation in the concrete case $\mathbb{C} = \mathsf{Set}$ and $M = Id$ as follows:

Proposition 3.4. *A pointed F^{Id}_{AB}-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 $x.get_R = y.get_R$;*
- *$(x.set_L\ a, y.set_L\ a) \in R$, and $(x.set_R\ b, y.set_R\ b) \in R$.*

Definition 3.5. *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'.*

3.1 Relationship with Symmetric Lens Equivalence

In this subsection, we describe symmetric lenses (SL) [16] in terms of cbx, and relate pointed bisimilarity between cbx and symmetric lens (SL-)equivalence [*ibid.*, Definition 3.2]. First of all, it is straightforward to describe 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 – as a cbx: 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.* [5, Section 4]). We now define the analogues of the SL-operations for a cbx between A and B:

Definition 3.6. *(Note that this is the opposite L-R convention from [16]!)*

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

Proposition 3.7. *Taking $\mathbb{C} = \mathsf{Set}$ and $M = Id$, a pointed F^{Id}_{AB}-bisimulation R 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 the following:*

- *$x.get_L = y.get_L$ 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 $(x', y') \in R$.*

Expressing cbx equivalence in these terms reveals that it is effectively identical to SL-equivalence. The cosmetic difference is that we are able to observe the 'current' values of A and B in any given state, via the *get* functions. This information is also 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 4.15 below.

3.2 Coalgebraic Bisimilarity with Effects

By introducing effects through M, our coalgebraic definition of behavioural equivalence allows us to characterise a wide class of behaviours in a uniform manner, and we illustrate with the examples anticipated in Section 2.2.

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 $\mathsf{in} : (I \to MX) \to MX$ (c.f. [23]). 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 3.8. *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 equivalence relation such that $m \sim_R n$ whenever:*

- *$m = \mathsf{return}\ x$, $n = \mathsf{return}\ y$, 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 $m' \sim_R n'$; or*
- *$m = \mathsf{in}\ (\lambda i \to m(i))$ and $n = \mathsf{in}\ (\lambda i \to n(i))$, where $m(i) \sim_R n(i)$ 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 $(x, y) \in R$ iff

- $x.get_L = y.get_L$ 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 bx. 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 bx, asserting that sequences of updates to either bx 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 $(x, y) \in R$ iff for all $a : A$ and $b : B$,

- $x.get_L = y.get_L$ and $x.get_R = y.get_R$;
- if $x.set_L\ a = \mathsf{None}$ then $y.set_L\ a = \mathsf{None}$ – and vice versa;
- 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 vice versa;
- two analogous clauses with B and set_R substituted for A and set_L.

3.2.3 Non-determinism

Taking M to be the finitary powerset monad, the resulting behavioural equivalence on bx comes close to the standard notion of strong bisimulation on labelled transition systems – and as we will see, shares its excessive fine-grainedness. 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 $(x, y) \in R$ iff for all $a : A$ and $b : B$,

- $x.get_L = y.get_L$;
- 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 $(x', y') \in R$;
- three analogous clauses with B and get_R/set_R substituted for A and get_L/set_L.

In contrast with the case of user I/O, this equivalence may be too fine for comparing bx 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 $(\mathsf{CGetSet}_L)$). 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 may be more appropriate [14].

4 Coalgebraic bx Composition

A cbx $\alpha : A \rightleftharpoons_X B$ 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 $\beta : B \rightleftharpoons_Y C$, that we may propagate these changes to C (and vice versa), raising the question of whether there exists a composite cbx $\alpha \bullet \beta : A \rightleftharpoons_Z C$ for some Z. Here, we give a more general, coalgebraic definition of cbx composition than our previous account for mbx [1]; this lets us reason about behavioural equivalence of such compositions in Section 4.1.

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:

Lemma 4.1. *A wpp functor preserves monomorphisms, or "monos" (maps f which are 'right cancellable': $g.f = h.f$ implies $g = h$) [25, 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 2.1), 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.*

Finally, we will employ the following in proofs, where $k\ x\ a$ is a block of statements referring to x and a:

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

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

Similarly, there are continuation versions of the coalgebra-morphism laws $(\mathsf{CGetP}_L)(h)$, $(\mathsf{CSetP}_L)(h)$, etc. in Remark 3.2, 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 of coalgebraic bx $\alpha : A \not\rightleftharpoons_X B, \beta : B \not\rightleftharpoons_Y C$ both contain information about B, in addition to A and C respectively. As indicated above, we define the state-space Z of the composite as consisting of those (x, y) pairs 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 pair $(\epsilon_\alpha, \epsilon_\beta)$ of initial states from α and β. To lie in Z, this 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 $\alpha{:}A \not\rightleftharpoons_X B$ and $\beta{:}B \not\rightleftharpoons_Y C$, 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 $\langle p_\alpha, p_\beta \rangle$.*

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

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. *Towards proving properties of the composition $\alpha \bullet \beta$ in Section 3, 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, a fortiori $e_{\alpha,\beta}$ is monic (i.e. left-cancellable), and thus by Lemma 4.1, so too is its image under the wpp functors M and F^M_{AB}.*

This allows us to pick out an initial state for Z, by noting that the arrow $(\epsilon_\alpha, \epsilon_\beta) : I \to X \times Y$ 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 for R):*

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

(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 bx, 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 contrast to the law's requirement that this should not change the state.

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 $y'.get_L$.)*

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

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

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

□

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

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

although it is equivalent if the monad M preserves jointly monic pairs (Definition 3.3). 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, it 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 $\alpha \diamond \beta : X \times Y \to F^M_{AC}\ (X \times Y)$, restricted to the B-consistent pairs $P_{\alpha,\beta}$, induce well-defined operations $P_{\alpha,\beta} \to F^M_{AC} P_{\alpha,\beta}$ 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 $P_{\alpha,\beta} \to F^M_{AB}\ X$ given by the composition $F^M_{AB}\ \pi_1 \circ (\alpha \diamond \beta) \circ e_{\alpha,\beta}$.

$$\begin{array}{ccc} & & X \times Y \\ & & \downarrow{\scriptstyle \alpha\diamond\beta} \\ & \overset{F^M_{AC}\ \pi_1}{\swarrow} & F^M_{AC}\ (X \times Y) \\ & & \downarrow{\scriptstyle F^M_{AC}\ (\alpha.get_R \times \beta.get_L)} \\ F^M_{AC}\ X & \xrightarrow[F^M_{AC}\langle \alpha.get_R, \alpha.get_R\rangle]{} & F^M_{AC}\ (B \times B) \end{array} \qquad \begin{array}{ccc} P_{\alpha,\beta} & \xrightarrow{e_{\alpha,\beta}} & X \times Y \\ \downarrow{\scriptstyle \delta\ :=} & & \downarrow{\scriptstyle \alpha\diamond\beta} \\ & \overset{F^M_{AC}\ \pi_1}{\swarrow} & F^M_{AC}\ (X \times Y) \\ & & \downarrow{\scriptstyle F^M_{AC}\ (\alpha.get_R \times \beta.get_L)} \\ F^M_{AC}\ X & \xrightarrow[F^M_{AC}\langle \alpha.get_R, \alpha.get_R\rangle]{} & F^M_{AC}\ (B \times B) \end{array}$$

Under the assumption that M is wpp, so is F^M_{AC}. Hence, the image under F^M_{AC} 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 $P_{\alpha,\beta} \to F^M_{AC}\ (P_{\alpha,\beta})$ (not *a priori* unique) as shown below-right. We take this to be the coalgebra structure $\alpha \bullet \beta$ of the composite bx.

$$\begin{array}{ccc} F^M_{AC} P_{\alpha,\beta} & \xrightarrow{F^M_{AC} e_{\alpha,\beta}} & F^M_{AC}\ (X \times Y) \\ \downarrow{\scriptstyle F^M_{AC} p_\alpha} & & \downarrow{\scriptstyle F^M_{AC}\ (\alpha.get_R \times \beta.get_L)} \\ F^M_{AC}\ X & \xrightarrow[F^M_{AC}\langle \alpha.get_R, \alpha.get_R\rangle]{} & F^M_{AC}\ (B \times B) \end{array} \qquad \begin{array}{cccc} P_{\alpha,\beta} & & \xrightarrow{e_{\alpha,\beta}} & X \times Y \\ & \overset{\alpha\bullet\beta}{\searrow} & & \downarrow{\scriptstyle \alpha\diamond\beta} \\ \downarrow{\scriptstyle \delta} & & F^M_{AC} P_{\alpha,\beta} \xrightarrow{F^M_{AC} e_{\alpha,\beta}} & F^M_{AC}\ (X \times Y) \\ & \overset{F^M_{AC} p_\alpha}{\swarrow} & & \downarrow{\scriptstyle F^M_{AC}\ (\alpha.get_R \times \beta.get_L)} \\ F^M_{AC}\ X & & \xrightarrow[F^M_{AC}\langle \alpha.get_R, \alpha.get_R\rangle]{} & F^M_{AC}\ (B \times B) \end{array}$$

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_{AB}, 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 described in (iii) above, 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 law $(\mathsf{CSetP}_L)(e_{\alpha,\beta})$ is given after Definition 3.1 below.) Here is $(\mathsf{CGetSet}_L)\ (\alpha \bullet \beta)$ postcomposed with $Me_{\alpha,\beta}$:

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

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

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

□

4.1 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 3.5). Recall also the identity cbx $\iota\ (a) : A \not\rightleftharpoons A$ introduced in Example 2.7.

Theorem 4.10. *Coalgebraic bx composition satisfies the following properties (where $\alpha, \alpha' : A \not\rightleftharpoons B$, $\beta, \beta' : B \not\rightleftharpoons C$, and $\gamma, \gamma' : C \not\rightleftharpoons D$, 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 $a : \alpha \to \gamma \leftarrow \beta : b$ 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, than 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 a routine diagram-chase. 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 an equalizer (such as $e_{\psi,\varphi}$, after Definition 3.5) 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.*

We also use this simple fact in the first part of the proof (identities for composition):

Remark 4.13. *A* monic *pointed coalgebra morphism h from α to α' trivially yields a bisimulation by taking $(\zeta, p, q) = (\alpha, id, h)$, and hence $\alpha \equiv \alpha'$; but not all bisimulations arise in such a way.*

We are now ready for the proof of Theorem 4.10.

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 sketch the strategy for showing $\iota(\epsilon_\alpha.get_L) \bullet \alpha \equiv \alpha$, as 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 from $\iota(\epsilon_\alpha.get_L) \bullet \alpha$ to $\iota(\epsilon_\alpha.get_L) \diamond \alpha$ – see Remark 4.12). Moreover, h is easily shown to be monic. Hence by Lemma 4.11, h induces a pointed coalgebra morphism from α to $\iota(\epsilon_\alpha.get_L) \bullet \alpha$ which is also monic (in $\mathbb{C}$), and hence by Remark 4.13 we obtain the required bisimulation.
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 $h\ (\epsilon_\alpha)$ 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 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, r) with a jointly monic pair p, p' of pointed coalgebra morphisms from r 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, s), together with a jointly monic pair (q, q') of coalgebra morphisms from s to the compositions $\alpha \bullet \beta$, $\alpha' \bullet \beta$. To construct s, let ζ be the equalizer of the following parallel pair – or equivalently, the pullback of $r.get_R$ and $\beta.get_L$.

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

One may then follow the steps (i)-(iii) in Section 4, where ζ and r play the role of the equalizer $e_{\alpha,\beta}$ and β respectively, to construct s, such that ζ is a coalgebra morphism from s to $r \diamond \beta$. Even though r is not a coalgebraic bx, it satisfies the following weaker form of $(\mathsf{CSetGet}_L)$ (and its R-version), sufficient for Lemma 4.7 to hold. (Here, we take $j : R$ and $a : A$; there is a continuation version as in Lemma 4.3.)

$$(\mathsf{CSetGet}^-_L)(r) : \mathbf{do}\ \{j' \leftarrow j.set_L\ a; return\ j'.get_L\} = \{j' \leftarrow j.set_L\ a; return\ a\}$$

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:

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

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 s into the composition $\alpha \bullet \beta$. One obtains the coalgebra morphism q' from s 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 LHS composition to the RHS, which is moreover monic. 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 $\alpha \diamond (\beta \bullet \gamma)$, which is also monic.

$$\begin{array}{ccccc} P_{(\alpha\bullet\beta),\gamma} & \xrightarrow{e_{(\alpha\bullet\beta),\gamma}} & P_{\alpha,\beta} \times Z & & \\ \downarrow{\scriptstyle p_0} & & \downarrow{\scriptstyle f} & & \\ X \times P_{\beta,\gamma} & \xrightarrow{id \times e_{\beta,\gamma}} & X \times (Y \times Z) & \underset{id \times (\gamma.get_L \circ \pi_2)}{\overset{id \times (\beta.get_R \circ \pi_1)}{\rightrightarrows}} & X \times C \end{array} \tag{1}$$

In this diagram, the arrow f is defined by $f\ (u, z) = \mathbf{do}\ \{\mathbf{let}\ (x, y) = e_{\alpha,\beta}\ (u); return\ (x, (y, z))\}$, but it may also be expressed as $f = assoc \circ (e_{\alpha,\beta} \times id)$, where we write *assoc* for the associativity isomorphism on products; the isomorphism, and the pairing with id, make it apparent that f is monic. 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 (and moreover monic).

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 Lemma 4.11 and Remark 4.12, to deduce that p_0 is a pointed coalgebra morphism, and moreover monic (using the fact that f and $e_{(\alpha\bullet\beta),\gamma}$ are, and hence their composition too).

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 the equalizer $P_{\alpha,(\beta\bullet\gamma)}$; by Remark 4.12 it is a coalgebra morphism, and it is monic because p_0 is monic.

$$\begin{array}{ccccc} P_{(\alpha\bullet\beta),\gamma} & & & & \\ \downarrow{\scriptstyle p} & \searrow^{p_0} & & & \\ P_{\alpha,(\beta\bullet\gamma)} & \xrightarrow{e_{\alpha,(\beta\bullet\gamma)}} & X \times P_{\beta,\gamma} & \underset{(\beta\bullet\gamma).get_L \circ \pi_2}{\overset{\alpha.get_R \circ \pi_1}{\rightrightarrows}} & B \end{array}$$

□

This well-behavedness of composition allows us to define a category of cbx:

Corollary 4.14. *There is a category* $\mathsf{Cbx}_\bullet$ *of pointed cbx, whose objects are* pointed sets $(A, a : A)$ – *sets with distinguished, "initial" values* a – *and whose arrows* $(A, a) \to (B, b)$ *are* $\equiv$*-equivalence classes* $[\alpha]$ *of coalgebraic bx* $\alpha : A \rightleftharpoons B$ *satisfying* $\epsilon_\alpha.get_L = a$ *and* $\epsilon_\alpha.get_R = b$.

We now describe how this category is related (by taking $\mathbb{C} = \mathsf{Set}$ and $M = Id$) to the category of symmetric lenses defined in [16]. 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 2.2):

Corollary 4.15. *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 and our previous monadic account [1]. The latter characterised bx in terms of monadic *get* and *set* operations, where the monad was not assumed *a priori* to refer to any notion of state – such as the monad $StateT\ X\ M$, abbreviated to T_X^M throughout this paper. The monad was only restricted to this stateful form for defining bx composition; such monadic bx were

called *StateTBX*. By contrast, our coalgebraic bx are explicitly stateful from the outset. Therefore, to compare cbx and mbx in a meaningful way, throughout this Section we restrict mbx to *StateTBX*, i.e. with respect to the monad T_X^M (as in most of our leading examples of monadic bx). Moreover, as was necessary for defining composition for such mbx, we also assume them to be *transparent*: the *get* operations neither change the state nor introduce M-effects, i.e. $get_L = \lambda x.return\ (f\ x, x)$ for some $f :: X \to A$ (likewise get_R).

Finally, we also assume monadic bx $A \Longleftrightarrow B$ have explicit initial states, rather than the more intricate process of initialising by supplying an initial A- or B-value as in [1].

5.1 Translating a Coalgebraic bx into a Monadic bx

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

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

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 by:

$$
\begin{aligned}
&get = \lambda a \to return\ (a, a) \qquad\qquad set = \lambda a'\ a \to return\ ((), a') \\
&lift = \lambda ma\ x \to \mathbf{do}\ \{a \leftarrow ma; return\ (a, x)\}
\end{aligned}
$$

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

Lemma 5.2. *The translation* $[\![\cdot]\!]$ *– from cbx for monad* M *with carrier* X*, to transparent StateTBX 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. 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 given in [1] for composing *StateTBX*, 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)*.*

$$
\text{(VU)}\quad u\ (a, (v\ a)) = a \qquad \text{(UV)}\ v\ (u\ (a, b')) = b' \qquad \text{(UU)}\ u\ (u\ (a, b'), b'') = u\ (a, b'')
$$

Lenses have a very well-developed literature [6, 8, 16, 18, among others], which we do not attempt to recap here. For more discussion of lenses, see Section 2.5 of [1].

We will apply a mild adaptation of a result in Shkaravska's early work [28].

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

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

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 $X \times Y$:

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

This gives us a cospan of monad morphisms $\mathit{left} = \vartheta\ (l_1) : T^M_X \dot{\to} T^M_{(X\times Y)} \dot{\leftarrow} T^M_Y : \vartheta\ (l_2) = \mathit{right}$, 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 *StateTBX* $t_1 : A \Longleftrightarrow_{T^M_X} B$, $t_2 : B \Longleftrightarrow_{T^M_Y} C$ with

$$\begin{array}{llll} t_1.\mathit{get}_L :: T^M_X\ A & t_1.\mathit{get}_R :: T^M_X\ B & t_2.\mathit{get}_L :: T^M_Y\ B & t_2.\mathit{get}_R :: T^M_Y\ C \\ t_1.\mathit{set}_L :: A \to T^M_X\ () & t_1.\mathit{set}_R :: B \to T^M_X\ () & t_2.\mathit{set}_L :: B \to T^M_Y\ () & t_2.\mathit{set}_R :: C \to T^M_Y\ () \end{array}$$

In [1], we used *left*, *right* to define $t_1 \fatsemi t_2$, a composite *StateTBX* with state-space $X \times Y$, as follows:

$$\begin{array}{l} (t_1 \fatsemi t_2).\mathit{get}_L \quad = \mathit{left}\ (t_1.\mathit{get}_L) \qquad\qquad (t_1 \fatsemi t_2).\mathit{get}_R = \mathit{right}\ (t_2.\mathit{get}_R) \\ (t_1 \fatsemi t_2).\mathit{set}_L\ a = \mathbf{do}\ \{\mathit{left}\ (t_1.\mathit{set}_L\ a); b \leftarrow \mathit{left}\ (t_1.\mathit{get}_R); \mathit{right}\ (t_2.\mathit{set}_L\ b)\} \\ (t_1 \fatsemi t_2).\mathit{set}_R\ c = \mathbf{do}\ \{\mathit{right}\ (t_2.\mathit{set}_R\ c); b \leftarrow \mathit{right}\ (t_2.\mathit{get}_L); \mathit{left}\ (t_1.\mathit{set}_R\ b)\} \end{array}$$

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 *StateTBX*.

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 1-1 correspondence between transparent *StateTBX* 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 *StateTBX* – may be reconciled by showing the pairwise definitions coherent:

Theorem 5.5. *Coherence of composition: The definitions of StateTBX 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:

$$\begin{array}{l} [\![\alpha]\!].\mathit{get}_L : T^M_X\ A \qquad [\![\alpha]\!].\mathit{set}_L : A \to T^M_X\ () \qquad \vartheta\ (v, u)\ ma : \forall \alpha\,.\ T^M_Z\ \alpha \\ [\![\alpha]\!].\mathit{get}_L = \lambda x \to \mathit{return}\ (x.\mathit{get}_L) \qquad [\![\alpha]\!].\mathit{set}_L\ a = \lambda x \to \mathbf{do}\ \{x' \leftarrow x.\mathit{set}_L\ a; \mathit{return}\ ((), x')\} \\ \vartheta\ (v, u)\ ma = \lambda z \to \mathbf{do}\ \{(a', s') \leftarrow ma\ (v\ z); \mathbf{let}\ z' = u\ (z, x'); \mathit{return}\ (a', z')\} \end{array}$$

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

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

This allows us to unpack the operations of the composite $[\![\alpha]\!] \fatsemi [\![\beta]\!]$ to show they are the same as $[\![\alpha \diamond \beta]\!]$. □

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 *StateTBX*. In [1] (Theorem 26), we defined an equivalence on *StateTBX* ($\sim$, say) given by operation-respecting state-space isomorphisms, and showed that $\bullet$-composition is associative up to $\sim$. In line with Corollary 4.14, one obtains a category $\mathsf{Mbx}_\bullet$ of $\sim$-equivalence-classes of transparent *StateTBX*, 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 *StateTBX* 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 coalgebraic bx given by restricting* $\equiv$ *to bisimulations whose pointed coalgebra morphisms are isomorphisms; and let* $\mathsf{Cbx}^!_\bullet$ *be the category of equivalence-classes of cbx up to* $\equiv_!$. *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 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. Moreover, these structures, being defined categorically, may be interpreted in a wide variety of settings under very modest assumptions about the underlying category $\mathbb{C}$.

Initial states were handled by introducing pointed coalgebras and bisimulations. A more elegant approach would be to work in the category $I / \mathbb{C}$ of *pointed objects* from the outset – i.e. pairs $(A, a : I \to A)$ of an object of $\mathbb{C}$ and an initial value a. Coalgebra morphisms and bisimulations are then automatically pointed, and the assumption of initial B-consistency used in Section 4 ((i)) is enforced at a type level. However, lifting exponentials (and strong monads) from $\mathbb{C}$ into $I / \mathbb{C}$ is rather delicate; we leave the details for future work.

Our definition allows a conceptually more direct, if technically slightly subtle, treatment of composition – in which 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 bx, and could be introduced via alternative methods such as [14]; but we do not have space to explore this further here.

It is also worth investigating the relationship between our coalgebraic formulation of bx, and the spans of lenses considered by Johnson and Rosebrugh [20]. A coalgebraic bx may be seen as a generalisation to a span of 'monadic lenses' or 'M-lenses', allowing effectful updates. However, in defining equivalence for spans of lenses, Johnson and Rosebrugh 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. It is unclear to us the relationship between this condition, and pointed coalgebraic bisimulation. We leave such investigations 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 in [1] – of which bx 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 [4]. 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 [33] can help here.

We set great store by the fact that our definitions are interpretable in categories other than Set; we hope thereby to incorporate richer, more intensional structures, such as delta lenses [7], edit lenses [17] and ordered updates [15] in our framework, and not merely the state-based extensional bx we have considered so far. We also hope to explore the connections between our work and existing coalgebraic notions of software components [4], and techniques for composing coalgebras [13].

Acknowledgements An extended, unpublished abstract [2] 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* [32] (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

[1] F. Abou-Saleh, J. Cheney, J. Gibbons, J. McKinna, and P. Stevens. Notions of bidirectional computation and entangled state monads. In *MPC*, June 2015. To appear. Extended version available at `http://groups.inf.ed.ac.uk/bx/bx-effects-tr.pdf`.

[2] F. Abou-Saleh and J. McKinna. A coalgebraic approach to bidirectional transformations. In *CMCS*. ETAPS, 2014. Talk abstract.

[3] J. Adámek, S. Milius, L. S. Moss, and L. Sousa. Well-pointed coalgebras. *Logical Methods in Computer Science*, 9(3), 2013.

[4] L. S. Barbosa. Towards a calculus of state-based software components. *J. UCS*, 9(8):891–909, 2003.

[5] J. Cheney, J. McKinna, P. Stevens, J. Gibbons, and F. Abou-Saleh. Entangled state monads (extended abstract). In Terwilliger and Hidaka [31].

[6] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

[7] Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. *JOT*, 10:6: 1–25, 2011.

[8] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM TOPLAS*, 29(3):17, 2007.

[9] J. Gibbons and R. Hinze. Just **do** it: Simple monadic equational reasoning. In *ICFP*, pages 2–14, 2011.

[10] J. Gibbons and M. Johnson. Relating algebraic and coalgebraic descriptions of lenses. *ECEASST*, 49, 2012.

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

[12] H. P. Gumm. Copower functors. *TCS*, 410(12):1129–1142, 2009.

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

[14] I. Hasuo, B. Jacobs, and A. Sokolova. Generic trace semantics via coinduction. *CoRR*, abs/0710.2505, 2007.

[15] S. J. Hegner. An order-based theory of updates for closed database views. *Ann. Math. Art. Int.*, 40:63–125, 2004.

[16] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *POPL*, pages 371–384. ACM, 2011.

[17] M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *POPL*, pages 495–508. ACM, 2012.

[18] Z. Hu, A. Schurr, P. Stevens, and J. F. Terwilliger. Dagstuhl seminar 11031: Bidirectional transformations (BX). *SIGMOD Record*, 40(1):35–39, 2011. DOI: `10.4230/DagRep.1.1.42`.

[19] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin EATCS*, 62:222–259, 1997.

[20] M. Johnson and R. Rosebrugh. Spans of lenses. In Terwilliger and Hidaka [31].

[21] E. Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989.

[22] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[23] G. D. Plotkin and J. Power. Notions of computation determine monads. In *FOSSACS*, volume 2303 of *LNCS*, pages 342–356. Springer, 2002.

[24] J. Power and O. Shkaravska. From comodels to coalgebras: State and arrays. *ENTCS*, 106, 2004.

[25] J. Power and H. Watanabe. An axiomatics for categories of coalgebras. *ENTCS*, 11:158–175, 1998.

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

[27] A. Schürr and F. Klar. 15 years of triple graph grammars. In *ICGT*, volume 5214 of *LNCS*, pages 411–425. Springer, 2008.

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

[29] A. Sokolova. Probabilistic systems coalgebraically: A survey. *TCS*, 412(38):5095–5110, 2011.

[30] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *SoSyM*, 9(1):7–20, 2010.

[31] J. Terwilliger and S. Hidaka, editors. *BX Workshop*. `http://ceur-ws.org/Vol-1133/#bx`, 2014.

[32] 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.

[33] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *LICS*, pages 280–291. IEEE, Computer Society Press, 1997.

A Appendix: Proofs

We begin with some technical observations which we use in what follows.

Firstly, we show that F^M_{AB} of Definition 2.3 does indeed define a functor:

Lemma A.1. *For $h : X \to Y$, and any $Z, s : (M\ X)^Z$, we define*

$$\overline{h}_Z\ (s) = \lambda\ (z : Z).\mathbf{do}\ \{x \leftarrow s\ z; return\ (h\ x)\} : (M\ Y)^Z$$

Then the action of F^M_{AB} on h given by $F^M_{AB}\ h\ (a, f, b, g) = (a, \overline{h}_A\ (f), b, \overline{h}_B\ (g))$ is functorial.

Proof. Taking projections, it suffices to show for any $Z, s{:}(M\ X)^Z$ that $\overline{id}_Z\ (s) = s$ and $\overline{(h \circ k)}_Z\ (s) = \overline{h}_Z\ (\overline{k}_Z\ (s))$. But each of these follows from the laws of **do**-notation. □

Secondly, to show that cbx are well-defined, it is convenient to prove the forms of the laws ($\mathsf{CGetSet}_A$) (α) and ($\mathsf{CSetGet}_A$) (α) in Definition 2.6, in terms of *return* statements; however, to *apply* these laws in proofs it is much more convenient to appeal to Lemma 4.3.

Finally, we re-iterate our proof strategy, stated in Section 4 ((iii)) (and applied in Lemma 4.11): 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. We then use the fact that these monos are coalgebra morphisms; a common use (e.g. Theorem 4.10) is to reason about the operations of a cbx $\alpha \bullet \beta$ – defined implicitly on the pullback in Section 4 – in terms of the operations of $\alpha \diamond \beta$, defined explicitly on pairs $(x, y) : X \times Y$.

A.1 Proof of Theorem 4.10

Proof. We typically dispense with justification for sequences of steps involving little more than substituting definitions and basic product manipulations. For the various arrows defined in the main text, what largely remains is to show that they lift to pointed coalgebra morphisms satisfying the cbx laws. Category-theoretically, this amounts to (a lot of) diagram chasing; our background in FP inclines us to equational reasoning in **do**-notation.

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 2.7):

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

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

- (CGetP_L)(h):

$$\begin{aligned}
&(\iota(\epsilon_\alpha.get_L) \diamond \alpha).get_L \circ h \\
=\ &\iota(\epsilon_\alpha.get_L).get_L \circ \pi_1 \circ h \\
=\ &id \circ \pi_1 \circ h \\
=\ &\alpha.get_L
\end{aligned}$$

- (CGetP_R)(h):

$$\begin{aligned}
&(\iota(\epsilon_\alpha.get_L) \diamond \alpha).get_R \circ h \\
=\ &\alpha.get_R \circ \pi_2 \circ h \\
=\ &\alpha.get_R
\end{aligned}$$

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

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

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

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

Congruence

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

We first prove that r satisfies the law $(\mathsf{CSetGet}^-_L)(\cdot)$ as described in the main text:

$$
\begin{array}{ll}
& \textbf{do}\,\{j' \leftarrow j.set_L\ a; return\ j'.get_L\} \\
= & [\![\ (\mathsf{CGetP}_L)(p)\ ]\!] \\
& \textbf{do}\,\{j' \leftarrow j.set_L\ a; return\ p\ (j').get_L\} \\
= & [\![\ (\mathsf{CSetP}_L)(p)\ ]\!] \\
& \textbf{do}\,\{\textbf{let}\ x = p\ (j); x' \leftarrow x.set_L\ a; return\ x'.get_L\} \\
= & [\![\ (\mathsf{CSetGet}_L)\ (\alpha)\ ]\!] \\
& \textbf{do}\,\{\textbf{let}\ x = p\ (j); x' \leftarrow x.set_L\ a; return\ a\} \\
= & [\![\ (\mathsf{CSetP}_L)(p)\ ]\!] \\
& \textbf{do}\,\{j' \leftarrow j.set_L\ a; \textbf{let}\ x' = p\ (j'); return\ a\} \\
= & [\![\ \text{redundant } \textbf{let}\ ]\!] \\
& \textbf{do}\,\{j' \leftarrow j.set_L\ a; return\ a\}
\end{array}
$$

Following the logic of Section 4 ((iii)), Lemma 4.7 then implies the existence of a coalgebra structure on S, which we call s, such that ζ is a coalgebra morphism from s to $r \diamond \beta$.

We show step (i), that $(p \times id) \circ \zeta$ indeed equalizes the parallel pair of the previous diagram:

$$
\begin{array}{ll}
& \alpha.get_R \circ \pi_1 \circ (p \times id) \circ \zeta \\
= & [\![\ \pi_1 \text{ is a natural transformation }]\!] \\
& \alpha.get_R \circ p \circ \pi_1 \circ \zeta \\
= & [\![\ (\mathsf{CGetP}_R)(p)\ ]\!] \\
& r.get_R \circ \pi_1 \circ \zeta
\end{array}
$$

$$
\begin{array}{l}
= \quad [\![\text{ definition of } \zeta \text{ as an equalizer }]\!] \\
\quad \beta.get_L \circ \pi_2 \circ \zeta \\
= \quad [\![\text{ properties of products }]\!] \\
\quad \beta.get_L \circ \pi_2 \circ (p \times id) \circ \zeta
\end{array}
$$

Now for step (ii), that $p \times id$ is a coalgebra morphism from $r \diamond \beta$ to $\alpha \diamond \beta$. Here, we take some $k : S$.

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

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

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

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

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

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

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

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

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.

Thus, Lemma 4.11 applies, and we have a pointed coalgebra morphism q from s to $\alpha \bullet \beta$; one obtains a similar arrow q' from s to $\alpha' \bullet \beta$ in exactly the same way, giving us the pointed coalgebra morphisms q, q' from s into $\alpha \bullet \beta, \alpha' \bullet \beta$ defining a bisimulation. It only remains to check that q, q' are jointly monic; this follows from the fact that p, p' are jointly monic, and hence so are $p \times id, p' \times id$; finally, the pre-composition of such a pair with a mono, such as ζ, preserves this property; and then the second half of Lemma 4.11 applies.

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 $P_{\alpha, \beta}$)

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

(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 $u : P_{\beta,\gamma}$).

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

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

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

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

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

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

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

do $\{$**let** $(x', (y, z)) = (id \times e_{\beta,\gamma})\ (x, u);$
$\quad (x'', (y', z')) \leftarrow (x', (y, z)).set_R\ d_0; return\ (x'', (y', z'))\}$

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

(iii): Firstly, $f\ (u,z) = \textbf{do}\ \{\textbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); return\ (x,(y,z))\}$ 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 $u : P_{\beta,\gamma}$)

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

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

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

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

$$
\begin{aligned}
&\quad \textbf{do}\,\{\textbf{let}\ (x,(y,z')) = f\ (u,z); \\
&\qquad (x',(y',z'')) \leftarrow (x,(y,z')).set_L\ a_0; return\ (x',(y',z''))\} \\
&= \quad [\![\text{ definition of } f \ ]\!] \\
&\quad \textbf{do}\,\{\textbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); \\
&\qquad (x',(y',z')) \leftarrow (x,(y,z)).set_L\ a_0; return\ (x',(y',z'))\} \\
&= \quad [\![\text{ definition of } (\alpha \diamond (\beta \diamond \gamma)).set_L \ ]\!] \\
&\quad \textbf{do}\,\{\textbf{let}\ (x,y) = e_{\alpha,\beta}\ (u); x' \leftarrow x.set_L\ a_0; \textbf{let}\ b = x'.get_R; \\
&\qquad (y',z') \leftarrow (y,z).set_L\ b; return\ (x',(y',z'))\}
\end{aligned}
$$

$= \quad [\![\text{ definition of } (\beta \diamond \gamma).set_L \quad]\!]$
$\quad \textbf{do}\,\{\textbf{let}\ (x, y) = e_{\alpha,\beta}\ (u); x' \leftarrow x.set_L\ a_0; \textbf{let}\ b = x'.get_R;$
$\quad\quad y' \leftarrow y.set_L\ b; \textbf{let}\ c = y'.get_R;$
$\quad\quad z' \leftarrow z.set_L\ c; return\ (x', (y', z'))\}$
$= \quad [\![\text{ definitions of } (\alpha \diamond \beta).set_L \text{ and } get_R \quad]\!]$
$\quad \textbf{do}\,\{\textbf{let}\ (x, y) = e_{\alpha,\beta}\ (u); (x', y') \leftarrow (x, y).set_L\ a_0; \textbf{let}\ c = (x', y').get_R;$
$\quad\quad z' \leftarrow z.set_L\ c; return\ (x', (y', z'))\}$
$= \quad [\![\text{ inserting redundant } \textbf{let} \text{ (to simplify dependencies on } (x', y'), \text{ for the next step)} \quad]\!]$
$\quad \textbf{do}\,\{\textbf{let}\ (x, y) = e_{\alpha,\beta}\ (u); (x', y') \leftarrow (x, y).set_L\ a_0; \textbf{let}\ (x'', y'') = (x', y');$
$\quad\quad \textbf{let}\ c = (x'', y'').get_R; z' \leftarrow z.set_L\ c; return\ (x'', (y'', z'))\}$
$= \quad [\![\ (\mathsf{CSetP}_L)(e_{\alpha,\beta}), \text{ continuation version } \quad]\!]$
$\quad \textbf{do}\,\{u' \leftarrow u.set_L\ a_0; \textbf{let}\ (x'', y'') = e_{\alpha,\beta}\ (u'); \textbf{let}\ c = (x'', y'').get_R;$
$\quad\quad z' \leftarrow z.set_L\ c; return\ (x'', (y'', z'))\}$
$= \quad [\![\text{ inline } \textbf{let} \text{ and apply } (\mathsf{CGetP}_L)(e_{\alpha,\beta}) \quad]\!]$
$\quad \textbf{do}\,\{u' \leftarrow u.set_L\ a_0; \textbf{let}\ (x'', y'') = e_{\alpha,\beta}\ (u'); \textbf{let}\ c = u'.get_R;$
$\quad\quad z' \leftarrow z.set_L\ c; return\ (x'', (y'', z'))\}$
$= \quad [\![\text{ move } \textbf{let}\ (x'', y'') = \ldots; \text{ definition of } ((\alpha \bullet \beta) \diamond \gamma).set_L \quad]\!]$
$\quad \textbf{do}\,\{(u', z') \leftarrow (u, z).set_L\ a_0; \textbf{let}\ (x'', y'') = e_{\alpha,\beta}\ (u'); return\ (x'', (y'', z'))\}$
$= \quad [\![\text{ definition of } f \quad]\!]$
$\quad \textbf{do}\,\{(u', z') \leftarrow (u, z).set_L\ a_0; return\ f\ (u', z')\}$

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

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

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

$\quad \alpha.get_R \circ \pi_1 \circ p_0\ (u)$
$= \quad [\![\text{ properties of products } \quad]\!]$

$\pi_2 \circ (id \times (\alpha.get_R \circ \pi_1)) \circ (id \times e_{\beta,\gamma}) \circ p_0\ (u)$
$= \quad [\![$ commutativity of square in Diagram 1 (beginning of Associativity proof!) $]\!]$
$\alpha.get_R \circ \pi_1 \circ f \circ e_{(\alpha\bullet\beta),\gamma}\ (u)$
$= \quad [\![$ definition of f; properties of products $]\!]$
$\mathbf{do}\ \{\mathbf{let}\ (v,z) = e_{(\alpha\bullet\beta),\gamma}\ (u); \mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (v); return\ x.get_R\}$
$= \quad [\![\ x.get_R = y.get_L$ by Remark 4.5 $]\!]$
$\mathbf{do}\ \{\mathbf{let}\ (v,z) = e_{(\alpha\bullet\beta),\gamma}\ (u); \mathbf{let}\ (x,y) = e_{\alpha,\beta}\ (v); return\ y.get_L\}$
$= \quad [\![$ definition of f $]\!]$
$\mathbf{do}\ \{\mathbf{let}\ (v,z) = e_{(\alpha\bullet\beta),\gamma}\ (u); \mathbf{let}\ (x,(y,z')) = f\ (v,z); return\ y.get_L\}$
$= \quad [\![$ definition of $\beta \diamond \gamma.get_L$ $]\!]$
$\mathbf{do}\ \{\mathbf{let}\ (v,z) = e_{(\alpha\bullet\beta),\gamma}\ (u); \mathbf{let}\ (x,(y,z')) = f\ (v,z); return\ (y,z').get_L\}$
$= (\beta \diamond \gamma).get_L \circ \pi_2 \circ f \circ e_{(\alpha\bullet\beta),\gamma}\ (u)$
$= \quad [\![$ commutativity of square in Diagram 1 again $]\!]$
$(\beta \diamond \gamma).get_L \circ \pi_2 \circ (id \times e_{\beta,\gamma}) \circ p_0\ (u)$
$= \quad [\![$ properties of products $]\!]$
$(\beta \diamond \gamma).get_L \circ e_{\beta,\gamma} \circ p_0\ (u)$
$= \quad [\![\ (\mathsf{CGetP}_L)(e_{\beta,\gamma})\ ]\!]$
$(\beta \bullet \gamma).get_L \circ p_0\ (u)$

□

A.2 Proof of Proposition 5.1

Proof. We need to establish the (GS) and (SG) laws for the defined operations on $[\![\alpha]\!]$ at L and R. We give the argument for L; that for R is entirely analogous.

We will appeal to the standard laws governing $get :: T^M_X\ X$ and $set :: X \to T^M_{()}$ as defined in Section 5.1:

(MGetSet) $\quad \mathbf{do}\ \{x \leftarrow get; set\ x\} = return\ ()$
(MSetGet) $\quad \mathbf{do}\ \{set\ x; x' \leftarrow get; return\ x'\} = \mathbf{do}\ \{set\ x; return\ x\}$

and there is also a continuation version of (MGetSet):

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

- (GS):

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

- (SG):

$\mathbf{do}\ \{[\![\alpha]\!].set_L\ a; [\![\alpha]\!].get_L\}$
$= \quad [\![$ definition of $[\![\alpha]\!]$ $]\!]$

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

□

A.3 Proof of Theorem 5.5

First, we note that the operations of the monadic bx $[\![\alpha]\!]$ defined at the beginning of Section 5.1 may be re-written in **do** notation for the monad M rather than in $StateT\ X\ M$, as follows:

$$
\begin{array}{l}
[\![\alpha]\!].get_L : T^M_X\ A\\
[\![\alpha]\!].get_L = \lambda x \to return\ (x.get_L)\\
[\![\alpha]\!].set_L : A \to T^M_X\ ()\\
[\![\alpha]\!].set_L\ a = \lambda x \to \mathbf{do}\,\{x' \leftarrow x.set_L\ a; return\ ((), x')\}
\end{array}
$$

In the same way, we may rewrite ϑ in the monad M:

$$
\begin{array}{l}
\vartheta\ (v, u)\ ma = \lambda z \to \mathbf{do}\,\{(a', s') \leftarrow ma\ (v\ z);\\
\quad \mathbf{let}\ z' = u\ (z, x'); return\ (a', z')\}
\end{array}
$$

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

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

This allows us to unpack the 'L'-operations of the composite $[\![\alpha]\!] \fatsemi [\![\beta]\!]$ into the following. (As usual, we omit the treatment of the 'R' operations as it is entirely analogous.)

get_L:

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

$=\quad [\![$ definition of $[\![\alpha \diamond \beta]\!].get_L \quad]\!]$

$\mathbf{do}\ \{(a', x') \leftarrow [\![\alpha \diamond \beta]\!].get_L\ x; return\ (a', (x', y))\}$

$=\quad [\![$ repacking $left \quad]\!]$

$left\ ([\![\alpha \diamond \beta]\!].get_L)\ (x, y)$

$=\quad [\![$ definition of $\fatsemi \quad]\!]$

$(\alpha \diamond \beta).get_L\ (x, y)$

set_L:

$([\![\alpha]\!] \fatsemi [\![\beta]\!]).set_L\ a'\ (x, y)$

$=\quad [\![$ definition of $\fatsemi$, in monad M rather than $StateT\ (X \times Y)\ M \quad]\!]$

$\mathbf{do}\ \{(_, (x', y')) \leftarrow left\ ([\![\alpha]\!].set_L\ a')\ (x, y);$
$\quad (b', (x'', y'')) \leftarrow left\ ([\![\alpha]\!].get_R)\ (x', y');$
$\quad right\ ([\![\beta]\!].set_L\ b)\ (x'', y'')\}$

$=\quad [\![$ unpacking $left$ and $right$, inlining $y' = y \quad]\!]$

$\mathbf{do}\ \{(_, x') \leftarrow [\![\alpha]\!].set_L\ a'\ x;$
$\quad (b', x'') \leftarrow [\![\alpha]\!].get_R\ x';$
$\quad (_, y'') \leftarrow [\![\beta]\!].set_L\ b'\ y;$
$\quad return\ ((), (x'', y''))\}$

$=\quad [\![$ definition of get and set for $[\![\cdot]\!]$ as given above $\quad]\!]$

$\mathbf{do}\ \{(_, x') \leftarrow x.set_L\ a'; (b', x'') \leftarrow x'.get_R; (_, y'') \leftarrow y.set_L\ b'; return\ ((), (x'', y''))\}$

$=\quad [\![$ definition of $\alpha \diamond \beta$ in Section 4 ((ii)) $\quad]\!]$

$\mathbf{do}\ \{(x'', y'') \leftarrow (\alpha \diamond \beta).set_L\ a'\ (x, y); return\ ((), (x'', y''))\}$

$=\quad [\![$ definition of $[\![\cdot]\!]$ on $(\alpha \diamond \beta) \quad]\!]$

$[\![\alpha \diamond \beta]\!].set_L\ a'\ (x, y)$