Electronic Notes in Theoretical Computer Science 82 No. 1 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 18 pages

State-based Components Made Generic

L. S. Barbosa and J. N. Oliveira [1,2]

Departamento de Informática
Universidade do Minho
Braga, Portugal

Abstract

Genericity is a topic which is not sufficiently developed in state-based systems modelling, mainly due to a myriad of approaches and behaviour models which lack unification. This paper adopts coalgebra theory to propose a generic notion of a state-based *software component*, and an associated calculus, by quantifying over behavioural models specified as strong *monads*. This leads to the *pointfree*, calculational reasoning style which is typical of the so-called Bird-Meertens school.

1 Introduction

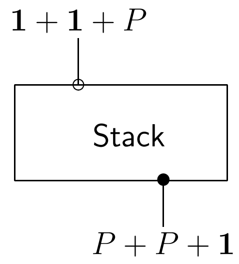
A Mealy machine [19] is an automata in which output symbols are associated to transitions, rather than states, and so depend on both the current state value *and* the supplied input. If such a dependence is relaxed from a strict deterministic discipline, to capture more complex behaviours (as, *e.g.*, partiality or non determinism), a variety of computational structures can be framed as instances of (generalised) Mealy machines. Such is the case, in particular, of *state-based software components* arising in the so-called *model oriented* approach to formal systems design — a widespread paradigm of which VDM [13] and Z [25] are well-known representatives.

A typical example of a state-based component is the ubiquitous *stack*, a computational structure whose specification is captured by a simple signature

[1] Email: lsb@di.uminho.pt
[2] Email: jno@di.uminho.pt

and 'black box' diagram:

$$\begin{cases} \mathsf{pop}: & \mathbf{1} \longrightarrow P \\ \mathsf{top}: & \mathbf{1} \longrightarrow P \\ \mathsf{push}: & P \longrightarrow \mathbf{1} \end{cases}$$



The pop, top and push operations are regarded as 'buttons' or 'ports', whose signatures are grouped together in the diagram (P stands for a particular parameter type, $\mathbf{1}$ for the nullary datatype and $+$ for 'datatype sum').

Component Stack encapsulates a number of services through a public *interface* providing limited access to its internal state space. Furthermore, it *persists* and *evolves* in time, in a way which can only be traced through the observation of what happens at the input/output interface level. One might capture these intuitions by providing an explicit semantic definition in terms of a function $[\![\mathsf{Stack}]\!] : U \times I \longrightarrow (U \times O + 1)$ where U denotes the internal space state and I, O abbreviate $1 + 1 + P$ and $P + P + 1$, respectively. This function — which should describe how Stack reacts to input stimuli, produces output data (if any) and changes state — can also be written as $[\![\mathsf{Stack}]\!] : U \longrightarrow (U \times O + 1)^I$ that is, as a *coalgebra* [24,10] of signature $U \longrightarrow \mathsf{T}\,U$ involving transition 'shape' (functor)

$$\mathsf{T} = ((\mathsf{Id} \times O) + \mathbf{1})^I \tag{1}$$

State-based modelling favours *observational* semantics: after all, the essence of the stack specification above lies in its collection of *observers* and any two internal configurations should be identified wherever indistinguishable by observation. This is nicely captured by coalgebra theory [24].

Coalgebra theory is adopted in this paper to propose a *generic* notion of state-based *software components* as well as some structuring and interfacing mechanisms for compositional development. The qualification *generic* is the key word: we proceed by quantifying over the behavioural dimension in the sense that each specific behaviour model used in a component specification is abstracted into a strong *monad*.

In a sense the research reported here is a particular application of the so-called *functorial* approach to datatypes, originated in the work of the Adj group in the early seventies [9,8], to the area of state-based systems modelling. This approach provides a basis for explaining *polymorphism* [22], and *polytypism* [12] — two steps of the same ladder, that of *generic programming* [1]. This fast evolving discipline raises the level of abstraction of the programming discourse in a way such that seemingly disparate programming techniques and algorithms are unified into idealised, kernel programming schemata. Having recognised that *genericity* is not sufficiently developed on state-based modelling, we would like to frame our contribution in such a broader research initiative. This adds to author's previous work on 'reconstructing' classical process calculi on a coalgebraic basis, leading to the same *pointfree*, calculational style which is typical of the so-called Bird-Meertens approach [4,5].

The paper is organised as follows: the basic steps toward genericity are presented in section 2. Sections 3 and 4 introduce the main contributions: a bicategory of generic components and associated calculus. We conclude with a brief illustration of the combinators presented in the paper (section 5) and some prospect of future work (section 6).

2 Going Generic

2.1 Introducing Generic Components

Software components have been characterised above as dynamic systems with a public interface and a private, encapsulated state. The relevance of state information precludes a 'process-like' (purely behavioural) view of components as inhabitants of a final coalgebra. Components are themselves *concrete* coalgebras. For a given value of the state space — referred to as a *seed* in the sequel — a corresponding 'process', or *behaviour* arises by computing its coinductive extension (or *anamorphic image*, in the terminology of [7]).

We have remarked above that partiality is characteristic to the behaviour of a stack. This is captured by the use of $U \times O + 1$ above, which can be identified as an instance of the popular *maybe* monad. Other components will exhibit different *behaviour models*. For example, one can easily think about systems behaving within a certain degree of non determinism or following a probability distribution. And we may even guess a refinement ordering among such behaviour models. Actually, genericity is achieved by replacing a given behaviour model (such as that captured by the *maybe* monad above) by an arbitrary *strong monad* [3] B, leading to coalgebras for the following composite functor (in Set):

$$\mathsf{T}^{\mathsf{B}} = \mathsf{B}(\mathsf{Id} \times O)^I \tag{2}$$

[3] A *strong monad* is a monad $\langle \mathsf{B}, \eta, \mu \rangle$ where B is a strong functor and both η and μ are strong natural transformations [17]. B being strong means there exist natural transformations $\tau_r^{\mathsf{T}} : \mathsf{T} \times - \Longrightarrow \mathsf{T}(\mathsf{Id} \times -)$ and $\tau_l^{\mathsf{T}} : - \times \mathsf{T} \Longrightarrow \mathsf{T}(- \times \mathsf{Id})$, called the right and left strength, respectively, subject to certain conditions. Their effect is to *distribute* the free variable values in the context "$-$" along functor B. Strength τ_r, followed by τ_l maps $\mathsf{B}I \times \mathsf{B}J$ to $\mathsf{BB}(I \times J)$, which can, then, be flattened to $\mathsf{B}(I \times J)$ via μ. In most cases, however, the *order* of application is relevant for the outcome. The Kleisli composition of the right with the left strength, gives rise to a natural transformation whose component on objects I and J is given by $\delta_r = \tau_{r_{I,J}} \bullet \tau_{l_{\mathsf{B}I,J}}$ Dually, $\delta_l = \tau_{l_{I,J}} \bullet \tau_{r_{I,\mathsf{B}J}}$. Such transformations specify how the monad distributes over product and, therefore, represent a sort of sequential composition of B-computations. Whenever δ_r and δ_l coincide, the monad is said to be *commutative*.

In this way, the computation of an action will not simply produce an output and a continuation state, but a B-structure of such pairs. The monadic structure provides tools to handle such computations. Unit (η) and multiplication (μ), provide, respectively, a value embedding and a 'flatten' operation to reduce nested behavioural annotations. Strength, either in its right (τ_r) or left (τ_l) version, will cater for context information. Finally, monad commutativity will turn up as a welcome (although not crucial) property.

As one would expect, reasoning about generic components entails a number of laws relating common 'housekeeping' morphisms to cope with *e.g.* product associativity, commutativity or exchange. Isomorphisms $\mathsf{xl} : A \times (B \times C) \longrightarrow B \times (A \times C)$, $\mathsf{xr} : A \times B \times C \longrightarrow A \times C \times B$ and $\mathsf{m} : (A \times B) \times (C \times D) \longrightarrow (A \times C) \times (B \times D)$ — whose interaction with monad unit, multiplication and strength is thoroughly dealt with in [3] — will be used in the sequel. By convention, binary morphisms always associate to the left.

2.2 Behaviour Models

Several possibilities can be considered for B. The simplest case is, obviously, the *identity* monad, Id, whereby components behave in a totally *deterministic* way. More interesting possibilities, capturing more complex behavioural features, include:

- *Partiality*, *i.e.*, the possibility of deadlock or failure, captured by the maybe monad, $\mathsf{B} = \mathsf{Id} + \mathbf{1}$, as in the stack example above.
- *Non determinism*, introduced by the (finite) powerset monad, $\mathsf{B} = \mathcal{P}$.
- *Ordered non determinism*, based on the (finite) sequence monad, $\mathsf{B} = \mathsf{Id}^*$.
- Monoidal labelling, with $\mathsf{B} = \mathsf{Id} \times M$. Note that, for B to form a monad, parameter M should support a monoidal structure.
- *'Metric' non determinism* capturing situations in which, among the possible future evolutions of the component, some are more likely (or cheaper, more secure, *etc*) than others.

In [3] the latter is based on a general notion of a *bag* monad defined over a structure $\langle M, \oplus, \otimes \rangle$, where both $\oplus$ and $\otimes$ are Abelian monoids and the latter distributes over the former. This gives rise to, *e.g.*,

- *Cost* components: based on Bag_M for $M = \langle \mathbb{N}, +, \times \rangle$, which is just the usual notion of a bag or *multiset*. Components with such a behaviour model assign a cost to each alternative, which may be interpreted as, *e.g.*, a performance measure. Such 'costs' are added when components get composed. This corresponds to the non deterministic generalisation of *monoidal labelling* above.
- *Probabilistic* components: based on $M = \langle [0, 1], \mathsf{min}, \times \rangle$ with the additional requirement that, for each $m \in \mathsf{Bag}_M$, $\sum (\mathcal{P}\pi_2)m = 1$. This assigns probabilities to each possible evolution of a component, introducing a (elemen-

tary) form of probabilistic non determinism.

All of the above situations correspond to known strong monads in Set, which can be composed with each other. The first two and the last one are commutative; the third is not. Commutativity of 'monoidal labelling' depends, of course, on commutativity of the underlying monoid.

3 A (bi)Category of Generic Components

Having defined generic components as (seeded) coalgebras, one may wonder how do they get composed and what kind of calculus emerges from this framework. Coalgebras are arrows and so arrows between coalgebras are arrows between arrows. This motivates the use of *bicategories* [6] which will, following [3], structure our reasoning universe from this point onwards. In brief, we will build a bicategory Cp whose *objects* are the interface (or observation) universes, whose *arrows* are seeded T^{B}-coalgebras and *2-cells*, the arrows between arrows, the corresponding comorphisms.

We assume a collection of sets I, O, ..., acting as component interfaces. By a seeded T^{B}-coalgebra we mean a pair $\langle u_p \in U_p, \overline{a}_p : U_p \longrightarrow \mathsf{B}(U_p \times O)^I \rangle$, where u_p is the seed and the coalgebra dynamics is captured by currying a state-transition function $a_p : U_p \times I \longrightarrow \mathsf{B}\ (U_p \times O)$. Then the construction of bicategory Cp defines, for each pair $\langle I, O\rangle$ of objects, a hom-category $\mathsf{Cp}(I,O)$, whose arrows $h : \langle u_p, \overline{a}_p\rangle \longrightarrow \langle u_q, \overline{a}_q\rangle$ satisfy the following *comorphism* and *seed preservation* conditions:

$$\overline{a}_q \cdot h \;=\; \mathsf{T}^{\mathsf{B}}\, h \cdot \overline{a}_p \quad \text{and} \quad h\, u_p \;=\; u_q \tag{3}$$

Composition is inherited from Set and the identity $1_p : p \longrightarrow p$, on component p, is defined as the identity id_{U_p} on the carrier of p. Next, for each triple of objects $\langle I, K, O\rangle$, a composition law is given by a functor

$$;_{I,K,O} : \mathsf{Cp}(I,K) \times \mathsf{Cp}(K,O) \longrightarrow \mathsf{Cp}(I,O)$$

The action of this on objects p and q is given by

$$p\,;q \;=\; \langle\langle u_p, u_q\rangle \in U_p \times U_q, \overline{a}_{p;q}\rangle$$

where $a_{p;q} : U_p \times U_q \times I \longrightarrow \mathsf{B}(U_p \times U_q \times O)$ is detailed as follows

$$\begin{aligned} a_{p;q} = {} & U_p \times U_q \times I \xrightarrow{\mathsf{xr}} U_p \times I \times U_q \xrightarrow{a_p \times \mathsf{id}} \mathsf{B}(U_p \times K) \times U_q \\ & \xrightarrow{\tau_r} \mathsf{B}(U_p \times K \times U_q) \xrightarrow{\mathsf{B}(\mathsf{a}\cdot\mathsf{xr})} \mathsf{B}(U_p \times (U_q \times K)) \\ & \xrightarrow{\mathsf{B}(\mathsf{id}\times a_q)} \mathsf{B}(U_p \times \mathsf{B}(U_q \times O)) \xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times (U_q \times O)) \\ & \xrightarrow{\mathsf{BBa}^\circ} \mathsf{BB}(U_p \times U_q \times O) \xrightarrow{\mu} \mathsf{B}(U_p \times U_q \times O) \end{aligned}$$

The action of ; on 2-cells reduces to $h\,;k \;=\; h \times k$. Finally, for each object K, an identity law is given by a functor

$$\mathsf{copy}_K : \mathbf{1} \longrightarrow \mathsf{Cp}(K,K)$$

whose action on objects is the constant component $\langle * \in \mathbf{1}, \overline{a}_{\mathsf{copy}_K}\rangle$, where $a_{\mathsf{copy}_K} \;=\; \eta_{\mathbf{1}\times K}$. Slightly abusing on notation, this will be also referred to as copy_K. Similarly, the action on morphisms is the constant comorphism $\mathsf{id}_{\mathbf{1}}$.

All in all, the fact that, for each strong monad B, components form a bicategory [4] amounts not only to a standard definition of the two basic combinators ; and copy_K of the component calculus, but also to setting up its basic laws. Recall (from *e.g.* [23]) that the graph of a comorphism is a bisimulation. Therefore, the existence of a seed preserving comorphism between two components makes them T^{B}-bisimilar, leading to the following laws, for appropriately typed components p, q and r:

$$\mathsf{copy}_I\,; p \;\sim\; p \;\sim\; p\,; \mathsf{copy}_O \tag{4}$$

$$(p\,;q)\,;r \;\sim\; p\,;(q\,;r) \tag{5}$$

The dynamics of a component specification is essentially 'one step': it describes immediate reactions to possible state/input configurations. Its temporal extension becomes the component's *behaviour*. Formally, the behaviour $[\![(p)]\!]$ of a component p is computed by applying the induced *anamorphism* to the seed-value of p. I.e., $[\![(p)]\!] \;=\; [\![(\overline{a}_p)]\!]u_p$

Behaviours organise themselves in a category Bh_{B}, or, simply, Bh, whose objects are sets and each arrow $b : I \longrightarrow O$ is an element of $\nu_{I,O}$, the carrier of the final coalgebra $\omega_{I,O}$ for functor $\mathsf{B}(\mathsf{Id} \times O)^I$. To define composition in Bh, first note that the definition of $\overline{a}_{p;q}$ above actually introduces an operator — **;** — between coalgebras: $\overline{a}_{p;q}$ could actually have been written as $\overline{a}_p\,;\overline{a}_q$. Thus, we may define composition in Bh by a family of combinators, for each I, K and O, ${;_{I,K,O}}^{\mathsf{Bh}} :\; \mathsf{Bh}(I,K) \times \mathsf{Bh}(K,O) \longrightarrow \mathsf{Bh}(I,O)$, such that

$${;_{I,K,O}}^{\mathsf{Bh}} =\; [\![(\omega_{I,K}\,;\omega_{K,O})]\!]$$

On the other hand, identities are given by

$${\mathsf{copy}_K}^{\mathsf{Bh}} :\; \mathbf{1} \longrightarrow \mathsf{Bh}(K,K) \quad \text{and} \quad {\mathsf{copy}_K}^{\mathsf{Bh}} =\; [\![(\overline{a}_{\mathsf{copy}_K})]\!]\; *$$

i.e., the behaviour of component copy_K, for each K.

It should be observed that the structure of Bh mirrors whatever structure Cp possesses. In fact, the former is isomorphic to a sub-(bi)category of the latter whose arrows are components defined over the corresponding final coalgebra. Alternatively, we may think of Bh as constructed by quotienting Cp by the greatest T^{B}-bisimulation. However, as final coalgebras are fully

[4] The reader is referred to [3] for all omitted proofs.

abstract with respect to bisimulation, the bicategorical structure collapses: the hom-categories become simply hom-*sets*. Moreover, as discussed below, some tensors in $\mathsf{Cp_B}$ become universal constructions in Bh, for some particular instances of B. This also explains why properties holding in Cp up to bisimulation, do hold 'on the nose' in the behaviour category. For example, we may rephrase laws (4) and (5), for suitably typed behaviours b, c and d, in Bh, as

$$\mathsf{copy}_I \,;\, b \;=\; b \;=\; b \,;\, \mathsf{copy}_O \quad \text{and} \quad (b\,;\,c)\,;\,d \;=\; b\,;\,(c\,;\,d)$$

First, however, we have to check that Bh is indeed a category. Let $b : I \longrightarrow O$ be a behaviour. Then,

$$b\,;\,\mathsf{copy}_O \;=\; [\![\omega_{I,O}\,;\,\mathsf{copy}_O]\!]\langle b, *\rangle \;=\; [\![\omega_{I,O}]\!] b \;=\; b$$

A similar calculation will establish $\mathsf{copy}_I \,;\, b = b$. On the other hand, for suitably typed behaviours b, c and d,

$$(b;c);d \;=\; [\![(\omega_{I,K};\omega_{K,L});\omega_{L,O}]\!]\langle\langle b,c\rangle,d\rangle \;=\; [\![\omega_{I,K};(\omega_{K,L};\omega_{L,O})]\!]\langle b,\langle c,d\rangle\rangle \;=\; b;(c;d)$$

So Bh is a category. Note the genericity and simplicity of the required proofs. For space economy, we omit the proof that construction $[\![\;]\!]$ is a 2-functor [16] from Cp to Bh, which follows the same calculational style (see [3]).

4 A Glimpse at the Component Calculus

This section investigates the structure of $\mathsf{Cp_B}$ by introducing an algebra of $\mathsf{T^B}$-components which is parametric on the behaviour model. This structure lifts naturally to $\mathsf{Bh_B}$ defining a particular (typed) 'process' algebra.

4.1 Functions as Components

Let us start from the simple observation that functions can be regarded as a particular case of components, whose interfaces are given by their domain and codomain types. Formally, a function $f : A \longrightarrow B$ is represented in Cp as

$$\ulcorner f \urcorner \;=\; \langle * \in \mathbf{1}, \overline{a}_{\ulcorner f \urcorner}\rangle$$

i.e., as a coalgebra over $\mathbf{1}$ whose action is given by the currying of

$$a_{\ulcorner f \urcorner} = \mathbf{1} \times A \xrightarrow{\mathsf{id}\times f} \mathbf{1} \times B \xrightarrow{\eta_{(\mathbf{1}\times B)}} \mathsf{B}(\mathbf{1} \times B)$$

Note that, up to bisimulation, function lifting is functorial, that is, for $g : I \longrightarrow K$ and $f : K \longrightarrow O$ functions, one has

$$\ulcorner f \cdot g \urcorner \sim \ulcorner g \urcorner \,;\, \ulcorner f \urcorner \tag{6}$$

$$\ulcorner \mathsf{id}_I \urcorner \sim \mathsf{copy}_I \tag{7}$$

Moreover, isomorphisms, split monos and split epis lift to Cp as, respectively, isomorphisms, split monos and split epis. Actually, lifting canonical Set arrows to Cp is a simple way to explore the structure of Cp itself. For instance, consider the lifting of $?_I : \emptyset \longrightarrow I$. Clearly, $?_I$ keeps its naturality as, for any $p : I \longrightarrow O$, the following diagram commutes up to bisimulation,

$$\begin{tikzcd} I \arrow[r, "p"] & O \\ \emptyset \arrow[u, "\ulcorner ?_I \urcorner"] \arrow[ur, "\ulcorner ?_O \urcorner"'] & \end{tikzcd}$$

because both $\ulcorner ?_I \urcorner$ and $\ulcorner ?_O \urcorner$ are the *inert* components: the absence of input makes reaction impossible. Formally:

$$\ulcorner ?_I \urcorner \, ; p \sim \ulcorner ?_O \urcorner \tag{8}$$

Equation (8) lifts to an equality in Bh, as does any other bisimulation equation in Cp. Therefore, $\emptyset$ is the *initial* object in Bh.

A different situation emerges in lifting $!_I : I \longrightarrow \mathbf{1}$ because naturality is lost. In fact, the following diagram fails to commute for non trivial B

$$\begin{tikzcd} I \arrow[r, "p"] \arrow[d, "\ulcorner !_I \urcorner"'] & O \arrow[dl, "\ulcorner !_O \urcorner"] \\ \mathbf{1} & \end{tikzcd}$$

To check this, take B as the finite powerset monad. Clearly, $p \, ; \ulcorner !_O \urcorner$ will deadlock whenever p does. By 'deadlocking' we mean the empty set of responses is produced. On the other hand, $\ulcorner !_I \urcorner$ never deadlocks as this is prevented by the definition of function lifting above. Therefore, the two components are not bisimilar and so $\mathbf{1}$ does not become the final object in Bh_B, for non trivial monads. It is, however, the final object in the behaviours category of deterministic components (*i.e.*, for $\mathsf{B} = \mathsf{Id}$).

4.2 Wrapping

The pre- and post-composition of a component with Cp-lifted functions can be encapsulated into an unique combinator, called *wrapping*, which may be thought of as an extension of the *renaming* connective found in process calculi (*e.g.*, [20]). Let $p : I \longrightarrow O$ be a component and consider functions $f : I' \longrightarrow I$ and $g : O \longrightarrow O'$. By $p[f, g]$ we will denote component p wrapped by f and g. This has type $I' \longrightarrow O'$ and is defined by input pre-composition with f and output post-composition with g. Formally, the wrapping combinator is a functor

$$-[f, g] : \mathsf{Cp}(I, O) \longrightarrow \mathsf{Cp}(I', O')$$

which is the identity on morphisms and maps a component $\langle u_p, \overline{a}_p \rangle$ into

$\langle u_p, \overline{a}_{p[f,g]} \rangle$, where

$$a_{p[f,g]} = U_p \times I' \xrightarrow{\mathsf{id} \times f} U_p \times I \xrightarrow{a_p} \mathsf{B}(U_p \times O) \xrightarrow{\mathsf{B}(\mathsf{id} \times g)} \mathsf{B}(U_p \times O')$$

The following properties about wrapping hold:

$$p[f,g] \sim \ulcorner f \urcorner \,;\, p \,;\, \ulcorner g \urcorner \tag{9}$$

$$(p[f,g])[f',g'] \sim p[f \cdot f', g' \cdot g] \tag{10}$$

Some simple components arise by lifting elementary functions to Cp. We have already remarked that the lifting of the canonical arrow associated to the initial Set object plays the role of an *inert* component, unable to react to the outside world. Let us give this component a name:

$$\mathsf{inert}_A \;=\; \ulcorner ?_A \urcorner \tag{11}$$

In particular, we define the nil component $\mathsf{nil} = \mathsf{inert}_\emptyset = \ulcorner ?_\emptyset \urcorner = \ulcorner \mathsf{id}_\emptyset \urcorner$ typed as $\mathsf{nil} : \emptyset \longrightarrow \emptyset$. Note that any component $p : I \longrightarrow O$ can be made inert by wrapping. For example, $p[?_I, !_O] \sim \mathsf{inert}_{\mathbf{1}}$. A somewhat dual role is played by component $\mathsf{idle} = \ulcorner \mathsf{id}_{\mathbf{1}} \urcorner$. Note that $\mathsf{idle} : \mathbf{1} \longrightarrow \mathbf{1}$ will propagate an unstructured stimulus (*e.g.*, the push of a button) leading to a (similarly) unstructured reaction (*e.g.*, exciting a led).

4.3 Tensors

Components can be aggregated in several different ways, besides the 'pipeline' composition discussed above. Next, we introduce three other generic combinators and characterise them as lax functors in Cp.

The first composition pattern to be considered is *external choice*. Let $p : I \longrightarrow O$ and $q : J \longrightarrow R$ be two components defined by $\langle u_p, \overline{a}_p \rangle$ and $\langle u_q, \overline{a}_q \rangle$, respectively. When interacting with $p \boxplus q$, the environment will be allowed to choose either to input a value of type I or one of type J, which will trigger the corresponding component (p or q, respectively), producing the relevant output.

The other two tensors in the calculus are *parallel* and *concurrent* composition, denoted by $p \boxtimes q$ and p ¸ q, respectively. The first one corresponds to a synchronous product: both components are executed simultaneously when triggered by a pair of legal input values. Note, however, that the behaviour effect, captured by monad B, propagates. For example, if B can express component failure and one of the arguments fails, the product will fail as well. Finally, *concurrent* composition, denoted by ¸, combines choice and parallel, in the sense that p and q can be executed independently or jointly, depending on the input supplied.

These three tensors are presented in detail in [3]. In this paper we restrict ourselves to the *choice* combinator, which, defined as a lax functor $\boxplus : \mathsf{Cp} \times$

$\mathsf{Cp} \longrightarrow \mathsf{Cp}$, consists of an action on objects given by $I \boxplus J = I + J$ and a family of functors

$$\boxplus_{I,O,J,R} : \mathsf{Cp}(I,O) \times \mathsf{Cp}(J,R) \longrightarrow \mathsf{Cp}(I+J, O+R)$$

yielding

$$p \boxplus q \;=\; \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p \boxplus q} \rangle$$

$$\begin{aligned} a_{p\boxplus q} \;=\; & U_p \times U_q \times (I+J) \xrightarrow{(\mathsf{xr}+\mathsf{a})\cdot\mathsf{dr}} U_p \times I \times U_q + U_p \times (U_q \times J) \\ & \xrightarrow{a_p \times \mathsf{id} + \mathsf{id} \times a_q} \mathsf{B}\,(U_p \times O) \times U_q + U_p \times \mathsf{B}\,(U_q \times R) \\ & \xrightarrow{\tau_r + \tau_l} \mathsf{B}\,(U_p \times O \times U_q) + \mathsf{B}\,(U_p \times (U_q \times R)) \\ & \xrightarrow{\mathsf{Bxr}+\mathsf{Ba}^\circ} \mathsf{B}\,(U_p \times U_q \times O) + \mathsf{B}\,(U_p \times U_q \times R) \\ & \xrightarrow{[\mathsf{B}\,(\mathsf{id}\times\iota_1), \mathsf{B}\,(\mathsf{id}\times\iota_2)]} \mathsf{B}\,(U_p \times U_q \times (O+R)) \end{aligned}$$

and mapping pairs of arrows $\langle h_1, h_2 \rangle$ into $h_1 \times h_2$.

The following laws arise from the fact that $\boxplus$ is a lax functor in Cp, for components p, q, p' and q', and functions f, g:

$$(p \boxplus p') \,;\, (q \boxplus q') \sim (p \,;\, q) \boxplus (p' \,;\, q') \tag{12}$$

$$\mathsf{copy}_{K \boxplus K'} \sim \mathsf{copy}_K \boxplus \mathsf{copy}_{K'} \tag{13}$$

$$\ulcorner f \urcorner \boxplus \ulcorner g \urcorner \sim \ulcorner f + g \urcorner \tag{14}$$

Moreover, up to isomorphic wiring, $\boxplus$ is a symmetric tensor product in each hom-category, with nil as unit, *i.e.*,

$$(p \boxplus q) \boxplus r \sim (p \boxplus (q \boxplus r))[\mathsf{a}_+, \mathsf{a}_+{}^\circ] \tag{15}$$

$$\mathsf{nil} \boxplus p \;\sim\; p[\mathsf{r}_+, \mathsf{r}_+{}^\circ] \;\text{ and }\; p \boxplus \mathsf{nil} \;\sim\; p[\mathsf{l}_+, \mathsf{l}_+{}^\circ] \tag{16}$$

$$p \boxplus q \sim (q \boxplus p)[\mathsf{s}_+, \mathsf{s}_+] \tag{17}$$

Laws (15) to (17) can be alternatively stated as providing evidence that the canonical Set isomorphisms a_+, r_+, l_+ and s_+, once lifted to Cp, keep their naturality up to bisimulation.

4.4 An Either Construction

The definition of a choice combinator raises the question whether there is a counterpart in Cp to the *either* construction in Set. The answer is partly positive. Let $p : I \longrightarrow O$ and $q : J \longrightarrow O$ be two components sharing a common output type O, and define

$$[p, q] \;=\; (p \boxplus q) \,;\, \ulcorner \nabla \urcorner$$

where $\nabla = [id, id]$. It can be shown that that the following diagram commutes up to bisimulation,

$$
\begin{tikzcd}
I \arrow[r, "\ulcorner \iota_1 \urcorner"] \arrow[dr, "p"'] & I \boxplus J \arrow[d, "{[p,q]}"] & J \arrow[l, "\ulcorner \iota_2 \urcorner"'] \arrow[dl, "q"] \\
 & O &
\end{tikzcd}
\qquad
\begin{aligned}
\ulcorner \iota_1 \urcorner \,;\, [p,q] &\sim p \\
\ulcorner \iota_2 \urcorner \,;\, [p,q] &\sim q
\end{aligned}
\tag{18}
$$

even though $[p,q]$ is not the unique arrow making the diagram commute. This means that the choice combinator, $\boxplus$, lifts to a *weak coproduct* in Bh. A proof is given in appendix A as an illustration of the adopted calculation style.

Failing universality means there is not a *fusion* law for $\boxplus$, even in the deterministic case. However, *cancellation*, *reflection* and *absorption* laws do hold strictly in Bh and, up to bisimulation, in Cp. Cancellation has just been dealt with. The other two — reflection

$$[\ulcorner \iota_1 \urcorner, \ulcorner \iota_2 \urcorner] \;\sim\; \mathsf{copy}_{I+J} \tag{19}$$

and absorption

$$(p \boxplus q) \,;\, [p', q'] \;\sim\; [p \,;\, p', q \,;\, q'] \tag{20}$$

are easy to prove. For example,

$$
\begin{aligned}
& (p \boxplus q) \,;\, [p', q'] \\
\sim \quad & \{ \text{ definition of } \textit{either} \text{ in } \mathsf{Cp} \,\} \\
& (p \boxplus q) \,;\, ((p' \boxplus q') \,;\, \ulcorner \nabla \urcorner) \\
\sim \quad & \{ \;;\, \text{associative (5)} \,\} \\
& ((p \boxplus q) \,;\, (p' \boxplus q')) \,;\, \ulcorner \nabla \urcorner \\
\sim \quad & \{ \boxplus \text{ functor (12)} \,\} \\
& ((p \,;\, p') \boxplus (q \,;\, q')) \,;\, \ulcorner \nabla \urcorner \\
\sim \quad & \{ \text{ definition of } \textit{either} \text{ in } \mathsf{Cp} \,\} \\
& [p \,;\, p', q \,;\, q']
\end{aligned}
$$

As expected, the $\boxplus$ combinator can be written in terms of an *either* construction on components. In fact, for $p : I \longrightarrow O$ and $q : J \longrightarrow R$, we obtain

$$p \boxplus q \;\sim\; [p \,;\, \ulcorner \iota_1 \urcorner, p \,;\, \ulcorner \iota_2 \urcorner] \tag{21}$$

That is to say, Set coproduct embeddings — once lifted to Cp, — keep their naturality:

$$\ulcorner \iota_1 \urcorner \,;\, (p \boxplus q) \;\sim\; p \,;\, \ulcorner \iota_1 \urcorner \quad \text{and} \quad \ulcorner \iota_2 \urcorner \,;\, (p \boxplus q) \;\sim\; q \,;\, \ulcorner \iota_2 \urcorner \tag{22}$$

A direct corollary of this fact is the following ‘idempotency’ result:

$$p \,;\, \ulcorner \iota_1 \urcorner \sim \ulcorner \iota_1 \urcorner \,;\, (p \boxplus p) \tag{23}$$

The dual situation, involving parallel aggregation $\boxtimes$ and a *split* construction, is studied in [3], but the results are a bit different. It turns out that a *cancellation* result — $\langle p, q\rangle \,;\ulcorner \pi_1 \urcorner \sim p$ — is only valid for a monad B which excludes the possibility of *failure* (*e.g.*, the non-empty powerset). On the other hand, diagonal $\vartriangle$ keeps its naturality when lifted to Cp, for B expressing deterministic behaviour (*e.g.*, the identity or the *maybe* monad), entailing a *fusion* law: $r\,;\langle p, q\rangle \sim \langle r\,;p, r\,;q\rangle$ Combining these two results, one concludes that $\boxtimes$ is a *product* in Bh, but only for behaviour models excluding failure and non determinism, which narrows the applicability scope of this fact to the category of total deterministic components. However, *reflection* and *absorption* laws hold for any B.

4.5 Interaction

So far component interaction was centred upon sequential composition, which is the Cp counterpart to functional composition in Set. This can be generalised to a new combinator, called *hook*, which connects some input to some output wires and, consequently, forces *part* of the output of a component to be fed back as input. Being defined in terms of functors among some families of Cp hom-categories, *hook* is a 'partial' combinator, whose rich set of laws is omitted here for lack of space. Formally, for each tuple of objects I, O and Z, we define $-\,\Lsh_Z \colon \mathsf{Cp}(I+Z, O+Z) \longrightarrow \mathsf{Cp}(I+Z, O+Z)$. This combinator is the identity on arrows and maps each component $p : I+Z \longrightarrow O+Z$ to $p\,\Lsh_Z \colon I+Z \longrightarrow O+Z$ given by

$$p\,\Lsh_Z = \langle u_p \in U_p, \overline{a}_{p\Lsh_Z}\rangle$$

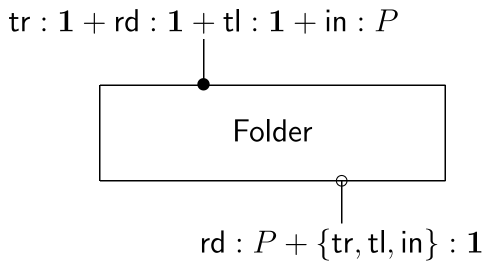
where

$$\begin{aligned} a_{p\Lsh_Z} \;=\; & U_p \times (I+Z) \xrightarrow{\quad a_p \quad} \mathsf{B}(U_p \times (O+Z)) \\ & \xrightarrow{\mathsf{B}((\mathsf{id}\times\iota_1 + \mathsf{id}\times\iota_2)\cdot\mathsf{dr})} \mathsf{B}(U_p \times (O+Z) + U_p \times (I+Z)) \\ & \xrightarrow{\mathsf{B}(\eta + a_p)} \mathsf{B}(\mathsf{B}(U_p \times (O+Z)) + \mathsf{B}(U_p \times (O+Z))) \\ & \xrightarrow{\quad \mu\cdot\mathsf{B}\triangledown \quad} \mathsf{B}(U_p \times (O+Z)) \end{aligned}$$

5 A (Generic) Folder from Two Stacks

The purpose of this section is to illustrate how new components can be built from old ones, relying solely on the functionality available. The example is the construction of a *folder* out of two *stacks*. Although these components are parametric on the type of stacked objects, we will refer to these as 'pages', by analogy with a folder in which new 'pages' are inserted on and retrieved ('read') from the righthandside pile.

A static, VDM-like specification of the component we have in mind can be found in [21]. According to this specification, the Folder component should provide operations to *read*, *insert* a new page, *turn a page right* and *turn a page left*. Reading returns the page which is immediately accessible once the folder is open at some position. Insertion takes as argument the page to be inserted. The other two operations are simply state updates. Let P be the type of a *page*. The Folder signature may be represented as follows, where input and output types are decorated with the corresponding action names:



Our exercise consists in building Folder assuming two stacks model the *left* and *right* piles of pages, respectively. The intuition is that the push action of the right stack will be used to model page insertion into the folder, *i.e.*, action in. On the other hand, it should also be connected to the pop of the left one to model tr, the 'turn page right' action. A symmetric connection will be used to model tl. The rd operation consumes the 'front' page — the one which can be accessed by top on the right stack.

According to this plan, the assembly of Folder starts by defining RightS as a Stack component suitably wrapped to meet the above mentioned constraints. At the input level we need to replicate the input to push by wrapping p with the codiagonal $\triangledown_P$ morphism. On the other hand, access to the top button on the left stack is removed by ι_2. At the output level, because of the additive interface structure, we cannot get rid of the top result. It is possible, however, to associate it to the push output and collapse both into $\mathbf{1}$, via $!_{P+\mathbf{1}}$. So we define:

$$\begin{aligned}\mathsf{RightS} &= \mathsf{Stack}[\mathsf{id}+\triangledown, \mathsf{id}] \;:\; \mathbf{1}+\mathbf{1}+(P+P) \longrightarrow P+P+\mathbf{1}\\ \mathsf{LeftS} &= \mathsf{Stack}[\iota_2+\mathsf{id}, (\mathsf{id}+!_{P+\mathbf{1}})\cdot \mathsf{a}_+] \;:\; \mathbf{1}+P \longrightarrow P+\mathbf{1}\end{aligned}$$

Then, we form the $\boxplus$ composition of both components:

$$\mathsf{LeftS} \boxplus \mathsf{RightS} : \; \mathbf{1}+P+(\mathbf{1}+\mathbf{1}+(P+P)) \longrightarrow P+\mathbf{1}+(P+P+\mathbf{1})$$

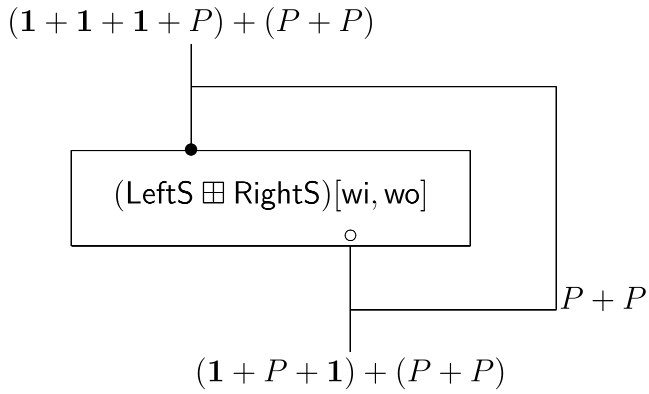
The next step builds the desirable connections using *hook* over this composite, which requires a previous wrapping by a pair of suitable isomorphisms:

$$\mathsf{AlmostFolder} \;=\; ((\mathsf{LeftS} \boxplus \mathsf{RightS})[\mathsf{wi}, \mathsf{wo}]) \Lsh_{P+P}$$

where, denoting by ι_{ij} the composite $\iota_i \cdot \iota_j$,

$$\mathsf{wi} \;=\; \Big[[[[\iota_{11}, \iota_{112}], \iota_{212}], \iota_{222}], [\iota_{21}, \iota_{122}]\Big] \qquad \mathsf{wo} \;=\; \Big[[\iota_{12}, \iota_{111}], [[\iota_{211}, \iota_{22}], \iota_{21}]\Big]$$

In a diagram:



Finally, to conform $\mathsf{AlmostFolder}$ to the Folder interface, we restrict the feed back input — by pre-composing with $\mathsf{fi} = \iota_1$ — and collapse both the trivial output and the feed back one to $\mathbf{1}$, by post-composing with $\mathsf{fo} = \left[[[\iota_2, \iota_1], \iota_2], \iota_2 \cdot !_{P+P}\right]$. Therefore, we complete the exercise by defining

$$\mathsf{Folder} \;=\; \mathsf{AlmostFolder}[\mathsf{fi}, \mathsf{fo}]$$

which respects the intended interface. Note this design retains the *architecture* of the 'folder' component without any commitment to a particular behaviour model.

6 Conclusions and Future Work

This paper introduces a semantic model for state-based software components, regarded as concrete coalgebras for some Set endofunctors with specified initial conditions and *parametric* on a model of behaviour. It also discusses the development of associated component calculi to reason about (and transform) component-based designs. Initial steps in this direction, although based in a different model which leads to a less expressive calculus, are described in our previous paper [2].

The bicategorical setting adopted is in debt to previous work by R. Walters and his collaborators on models for deterministic input-driven systems [14,15]. However, whereas R. Walters' work deals essentially with deterministic systems, our monadic parametrization allows to focus on the relevant structure of components, factoring out details about the specific behavioural effects that may be produced. The hook combinator and tensors are also new. Also close to our modelling approach is [18] which proposes an axiomatization of what is called a 'notion of a process' in a monoidal category. This work, however, does not cover neither the definition of generic combinators nor the development of an associated calculus.

Our initial motivation for studying state-based components arose in the context of *model-oriented* specification methods. Later, it has evolved toward a more general approach which we believe may be useful in starting a coalgebraic study of software components in the broader sense of the emerging *component-oriented* programming paradigm [26,27]. This retains from object-orientation the basic principle of encapsulation of data and code, but shifts the emphasis from (class) inheritance to (object) composition, paving the way to a development methodology based on *third-party assembly* of components. The paradigm is often illustrated by the visual metaphor of a *palette* of computational units, treated as black boxes, and a *canvas* into which they can be dropped. Connections are established by drawing *wires*, corresponding to some sort of interfacing code.

Actually, our present work is framed in such a broader context. In particular, we have been working on a theory of *component refinement* and *customising*, the latter being concerned with tuning software components to particular *use cases*. On the practical side, the prospect of building a Charity pre processor similar to PolyP [11] for the behaviour monads considered in [3] is currently being considered.

References

[1] R. C. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Third International Summer School on Advanced Functional Programming, Braga*, pages 28–115. Springer Lect. Notes Comp. Sci. (1608), September 1998.

[2] L. S. Barbosa. Components as processes: An exercise in coalgebraic modeling. In S. F. Smith and C. L. Talcott, editors, *FMOODS'2000 - Formal Methods for Open Object-Oriented Distributed Systems*, pages 397–417. Kluwer Academic Publishers, September 2000.

[3] L. S. Barbosa. *Components as Coalgebras.* PhD thesis, DI, Universidade do Minho, 2001.

[4] L. S. Barbosa. Process calculi *à la* Bird-Meertens. In *CMCS'01 - Workshop on Coalgebraic Methods in Computer Science*, pages 47–66, Genova, April 2001. ENTCS, volume 44.4, Elsevier.

[5] L. S. Barbosa and J. N. Oliveira. Coinductive interpreters for process calculi. In *Proc. of FLOPS'02*, pages 183–197, Aizu, Japan, September 2002. Springer Lect. Notes Comp. Sci. (2441).

[6] J. Benabou. Introduction to bicategories. *Springer Lect. Notes Maths. (47)*, pages 1–77, 1967.

[7] R. Bird and O. Moor. *The Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997.

[8] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall International, 1978.

[9] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial algebra semantics and continuous algebras. *Jour. of the ACM*, 24(1):68–95, January 1977.

[10] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–159, 1997.

[11] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[12] J. Jeuring and P. Jansson. Polytypic programming. In T. Launchbury, E. Meijer, and T. Sheard, editors, *International Summer School on Advanced Functional Programming*, pages 68–114. Springer Lect. Notes Comp. Sci. (1129), 1996.

[13] Cliff B. Jones. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice-Hall International, 1986.

[14] P. Katis, N. Sabadini, and R. F. C. Walters. Bicategories of processes. *Journal of Pure and Applied Algebra*, 115(2):141–178, 1997.

[15] P. Katis, N. Sabadini, and R. F. C. Walters. On the algebra of systems with feedback and boundary. *Rendiconti del Circolo Matematico di Palermo*, II(63):123–156, 2000.

[16] G. M. Kelly. *Basic Concepts of Enriched Category Theory*, volume 64 of *London Mathematical Society Lecture Notes Series*. Cambridge University Press, 1982.

[17] A. Kock. Strong functors and monoidal monads. *Archiv für Mathematik*, 23:113–120, 1972.

[18] S. Krstic, J. Launchbury, and D. Pavlovic. Categories of processes enriched in final coalgebras. In *Proceedings of* Fossacs, pages 303–317. Springer Lect. Notes Comp. Sci. (2030), 2001.

[19] G. H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Techn. Jour.*, 34(5):1045–1079, 1955.

[20] R. Milner. *Communication and Concurrency.* Series in Computer Science. Prentice-Hall International, 1989.

[21] J. N. Oliveira. Formal Software Development. Lecture Notes for the MSc in Computer Science, Minho University, 1992.

[22] J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing 83*, pages 513–523, 1983.

[23] J. Rutten. Universal coalgebra: A theory of systems. Technical report, CWI, Amsterdam, 1996.

[24] J. Rutten. Universal coalgebra: A theory of systems. *Theor. Comp. Sci.*, 249(1):3–80, 2000. (Revised version of CWI Techn. Rep. CS-R9652, 1996).

[25] J. M. Spivey. *The* Z *Notation: A Reference Manual (2nd ed)*. Series in Computer Science. Prentice-Hall International, 1992.

[26] C. Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[27] P. Wadler and K. Weihe. Component-based programming under different paradigms. Technical report, Report on the Dagstuhl Seminar 99081, February 1999.

A A Sample Proof

Reference [3] contains a comprehensive account of the calculus sketched here, with all proofs carried out in the pointfree style. As an illustration consider the proof that $\boxplus$ lifts to a *weak coproduct* in Bh, required in section 4.4:

Proof. A weak coproduct is defined like a coproduct but for the uniqueness of the mediating arrow (the *either* construction). Existence, *i.e.*, the validity of (18), is proved considering the equivalent formulation

$$[p, q][\iota_1, \nabla] \sim p \quad \text{and} \quad [p, q][\iota_2, \nabla] \sim q$$

replacing composition with lifted functions by wrapping. We show that both the first and the second projection are comorphisms from the left to the right. Therefore,

$$
\begin{array}{cl}
& \mathsf{B}(\pi_1 \times \nabla) \cdot [\mathsf{B}(\mathsf{id} \times \iota_1), \mathsf{B}(\mathsf{id} \times \iota_2)] \cdot (\mathsf{Bxr} + \mathsf{Ba}^\circ) \cdot (\tau_r + \tau_l) \cdot (a_p \times \mathsf{id} + \mathsf{id} \times a_q) \\
& \cdot(\mathsf{xr} + \mathsf{a}) \cdot \mathsf{dr} \cdot (\mathsf{id} \times \iota_1) \\
= & \quad \{ \text{ law: } \iota_1 = \mathsf{dr} \cdot (\mathsf{id} \times \iota_1) \ (\textit{cf.}, [3]) \ \} \\
& \mathsf{B}(\pi_1 \times \nabla) \cdot [\mathsf{B}(\mathsf{id} \times \iota_1), \mathsf{B}(\mathsf{id} \times \iota_2)] \cdot (\mathsf{Bxr} + \mathsf{Ba}^\circ) \cdot (\tau_r + \tau_l) \cdot (a_p \times \mathsf{id} + \mathsf{id} \times a_q) \\
& \cdot(\mathsf{xr} + \mathsf{a}) \cdot \iota_1 \\
= & \quad \{ \ + \text{ absorption and cancellation } \} \\
& \mathsf{B}(\pi_1 \times \nabla) \cdot \mathsf{B}(\mathsf{id} \times \iota_1) \cdot \mathsf{Bxr} \cdot \tau_r \cdot a_p \times \mathsf{id} \cdot \mathsf{xr} \\
= & \quad \{ \text{ routine: } \nabla \cdot \iota_1 = \mathsf{id} \ \} \\
& \mathsf{B}(\pi_1 \times \mathsf{id}) \cdot \mathsf{Bxr} \cdot \tau_r \cdot a_p \times \mathsf{id} \cdot \mathsf{xr} \\
= & \quad \{ \text{ routine: } (\pi_1 \times \mathsf{id}) \cdot \mathsf{xr} = \pi_1 \ \} \\
& \mathsf{B}\pi_1 \cdot \tau_r \cdot a_p \times \mathsf{id} \cdot \mathsf{xr} \\
= & \quad \{ \text{ law: } \mathsf{B}\pi_1 \cdot \tau_r = \pi_1 \ (\textit{cf.}, [3]) \ \} \\
& \mathsf{B}\pi_1 \cdot a_p \times \mathsf{id} \cdot \mathsf{xr} \\
= & \quad \{ \ \times \text{ definition and cancellation } \} \\
& a_p \cdot \pi_1 \cdot \mathsf{xr} \\
= & \quad \{ \text{ routine: } (\pi_1 \times \mathsf{id}) \cdot \mathsf{xr} = \pi_1 \text{ and } \mathsf{xr} = \mathsf{xr}^\circ \ \} \\
& a_p \cdot (\pi_1 \times \mathsf{id})
\end{array}
$$

which establishes the first clause of (18). A similar calculation will prove the second one. Note that in both cases seeds are trivially preserved.

Note the impossibility of turning *either* into an universal construction in Bh. The basic observation is that the codiagonal ∇ does not keep its naturality when lifted to Cp. In fact, a counterexample can be found even in the simple setting of deterministic components (*i.e.*, with $\mathsf{B} = \mathsf{Id}$). Let $p = \langle 0 \in \mathbb{N}, \overline{a}_p \rangle : \mathbb{N} \longrightarrow \mathbb{N}$ be such that, upon receiving an input i, i is added to the current state value and the result sent to the output. Consider the following sequence of inputs (of type $\mathbb{N}+\mathbb{N}$): $s = \langle \iota_1 5, \iota_2 3, \iota_1 4, ... \rangle$. The reaction to s of $\ulcorner\nabla\urcorner \,;\, (p \boxplus p)$ is $\langle 5, 3, 9, \ldots \rangle$ while $p \,;\, \ulcorner\nabla\urcorner$, resorting only to one copy of p, produces $\langle 5, 8, 12, \ldots \rangle$.

□