A calculus of software components

L. S. Barbosa

Department of Informatics
Minho University
lsb@di.uminho.pt

1 Introducing state-based components

By the end of last century component-based software development [36, 37] emerged as a promising paradigm to deal with the ever increasing need for mastering complexity in software design, evolution and reuse. From object-orientation it retains the basic principle of encapsulation of data and code, but shifts the emphasis from (class) inheritance to (object) composition to avoid interference between the former and encapsulation and, thus, paves 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.

The expression *software component*, however, is so semantically overloaded that its use is often a risk. As put by P. Wadler in a 1999 Seminar suggestively entitled 'Component-based Programming under different paradigms', *just as Eskimos need fifty words for ice, perhaps we need many words for components*. Moreover, as it happened before with object-orientation, and software engineering in the broad sense, component-orientation has grown up to a collection of popular technologies, methods and tools, before consensual definitions and principles (let alone formal foundations) have been put forward.

This report is concerned with a formalization of component-based development, introducing a coalgebraic semantic model for components and a corresponding calculus. Attention is restricted to *state-based*, typically represented by their specifications.

A typical example of such a state-based component is the ubiquitous *stack*. Denoting by U its internal state, a stack of values of type P is handled through the usual

$$\begin{aligned} \mathsf{top} &: U \longrightarrow P \\ \mathsf{pop} &: U \longrightarrow P \times U \\ \mathsf{push} &: U \times P \longrightarrow U \end{aligned}$$

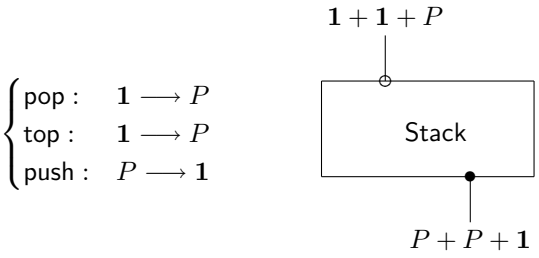
operations. An alternative, 'black box' view hides U from the stack environment and regards each operation as a pair of input/output ports. Such a 'port' signature of, *e.g.*, the top operation is then given by

$$\mathsf{top} : \mathbf{1} \longrightarrow P$$

where $\mathbf{1}$ stands for the nullary (or unit) datatype. The intuition is that top is activated with the simple pushing of a 'button' (its argument being the stack private state space) whose effect is the production of a P value in the corresponding output port. Similarly typing push as

$$\mathsf{push} : P \longrightarrow \mathbf{1}$$

means that an external argument is required on activation but no visible output is produced, but for a trivial indication of successful termination. Such 'port' signatures are grouped together in the diagram below. Note how input (respectively, output) 'ports' are represented by the sum of their parameters. Such sums label the stack input (respectively, output) point represented by an empty (respectively, full) circle in the diagram. Combined input type $\mathbf{1} + \mathbf{1} + P$ models the choice of three functionalities (top, pop and push in this order), of which only one takes input of type P.



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 observations at the interface level. One might capture these intuitions by providing an explicit semantic definition in terms of a function

$$[\![\mathsf{Stack}]\!] : U \times I \longrightarrow (U \times O + \mathbf{1})$$

where I, O abbreviate $\mathbf{1} + \mathbf{1} + P$ and $P + P + \mathbf{1}$, respectively. The presence of $\mathbf{1}$ in its result type indicates that the overall behaviour of this component is *partial*: in a number of state configurations the execution of some operations may fail. This function — which should describe how Stack reacts to input stimuli, produces output data (if any) and changes state — can also be written in a curried form [1] as

$$\overline{[\![\mathsf{Stack}]\!]} : U \longrightarrow (U \times O + \mathbf{1})^I \tag{1}$$

that is, as a *coalgebra* $U \longrightarrow \mathsf{T}\ U$ for functor $\mathsf{T}\ X\ =\ ((X \times O) + \mathbf{1})^I$.

The Stack example illustrates the basic elements of a semantic model for state-based components:

– the presence of an *internal state space* which evolves and persists in time,

[1] In order to emphasize the dependency of the possible observations X from the input, we resort to the standard mathematical notation X^I for functional dependency, instead of the equivalent $I \to X$ more familiar in computing.

– and the possibility of *interaction* with other components through well-defined interfaces and during the overall computation.

This favours adoption of a *behavioural* semantics: components are inherently dynamic, possess an observable behaviour, but their internal configurations remain hidden and should be identified if not distinguishable by observation. The qualificative 'state-based' is used in the sense the word 'state' has in automata theory — the internal memory of the automaton which both constrains and is constrained by the execution of component operations. Such operations are encoded in the specification of a functor which constitutes the (syntax of the)component interface.

But why coalgebras? Our starting point is the conjunction of two key ideas. First, the 'black-box' characterisation of software components favours an *observational* semantics: the essence of the stack specification above lies in the collection of *possible observations* and any two internal configurations should be identified wherever indistinguishable by observation. This is nicely captured by coalgebra theory [33].

Secondly, we aim at *generic* constructions, *i.e.*, independent of any particular notion of component behaviour. Therefore, the other key idea is the application of the so-called *functorial* approach to datatypes, originated in the work of the ADJ group in the early seventies [15], to the area of state-based systems modelling. This approach provides a basis for *generic programming* [3] which raises the level of abstraction of the programming discourse in a way such that seemingly disparate techniques and algorithms are unified into idealised, kernel programming schemata. Moreover, we would like to formalize component calculi in an essentially equational, pointfree way, as one gets used to in functional programming.

References. This report is basically a summary of chapter 5 of [4], to which the reader is referred for all proofs omitted here. Main results appeared in [6, 5] and an alternative model, based in generalised Moore machines, in [13]. A refinement theory for this sort of component models was developed later in [23, 24, 7].

2 Going generic

2.1 Generic components

Software components were characterised in the previous section 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.

We have remarked when introducing component Stack in Section 1, that partiality is a characteristic of its behaviour. This was captured there by resorting to functor $\mathsf{Id} \times O + \mathbf{1}$, *i.e.*, an instance of the popular *maybe* monad. Other components may exhibit different *behaviour models*. For example, one can easily think of components behaving within a certain degree of non determinism or following a probability distribution.

Genericity is achieved by replacing a given behaviour model by an arbitrary *strong monad*[2] B, leading to coalgebras for

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

as a possible general model for state based software components. Therefore 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 effects. Strength, either in its right (τ_r) or left (τ_l) version, cater for context information. Finally, monad commutativity [3] turn up as a welcome (although not crucial) property.

Functor (2) may be regarded as an instance of an even more general shape

$$\mathsf{T}^{\mathsf{B}} = {O'}^{I'} \times \mathsf{B}(\mathsf{Id} \times O)^I$$

which specializes to a variety of interfaces for state-based component strutuctues, namely

- 'Functional' components, as given by (2), which, for $\mathsf{B} = \mathsf{Id}$, correspond to standard *Mealy* machines.
- 'Action' components, with no independent attributes: $\mathsf{T}^{\mathsf{B}} = \mathsf{B}(\mathsf{Id} \times O)$
- 'Silent' components, which evolve invisibly without any sort of external control: $\mathsf{T}^{\mathsf{B}} = O^I \times \mathsf{B}$
- 'Object' components, characterized by an attribute-method pair

  $$\mathsf{T}^{\mathsf{B}} = O \times \mathsf{B}^I$$

  which , for $\mathsf{B} = \mathsf{Id}$, corresponds to *Moore* machines.

In the sequel we assume a collection of sets I, O, ..., acting as component interfaces and the following definition of a component specification:

Definition 1. *A software component is specified by a pointed coalgebra*

$$\langle u_p \in U_p, \overline{a}_p : U_p \longrightarrow \mathsf{B}(U_p \times O)^I \rangle \tag{3}$$

where u_p is the initial state, often referred to as the seed *of the component computation, and the coalgebra dynamics is captured by currying a state-transition function $a_p : U_p \times I \longrightarrow \mathsf{B}\ (U_p \times O)$.*

[2] 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 [21]. 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.

[3] A strong monad is said to be *commutative* whenever δ_r and δ_l coincide.

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. Other 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 a component, some are stipulated to be more likely (cheaper, more secure, *etc*) than others.

All cases correspond to 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. 'Metric' non determinism 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 (elementary) form of probabilistic non determinism.

3 The semantic framework

3.1 A universe of generic components.

Having defined generic components as (pointed) coalgebras, one may wonder how do they get composed and what kind of calculus emerges from this framework. In our framework, interfaces are sets representing the input and output range of a component. Consequently, components are arrows between interfaces and so arrows between components are arrows between arrows. Thus, three notions have to be taken into account: interfaces, components and component morphisms. Formally, this leads to the notion of a *bicategory* [4] to structure our reasoning universe. In brief, we take interfaces (*i.e.*, sets

[4] Basically a *bicategory* [9] is a category in which a notion of arrows between arrows is additionally considered. This means that the the space of morphisms between any given pair of

modelling components' observation universes) as *objects* of a bicategory Cp, whose *arrows* are pointed T^{B}-coalgebras (as defined in (2)) and *2-cells*, the arrows between arrows, the corresponding morphisms. Formally,

Definition 2. *Assume arbitrary sets as* Cp *objects. For each pair* $\langle I, O\rangle$ *of objects, define a category* $\mathsf{Cp}(I, O)$*, whose arrows*

$$h : \langle u_p, \overline{a}_p\rangle \longrightarrow \langle u_q, \overline{a}_q\rangle \qquad \begin{array}{ccc} U_p & \xrightarrow{\overline{a}_p} & \mathsf{T}^{\mathsf{B}}\, U_p \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle \mathsf{T}^{\mathsf{B}}\, h} \\ U_q & \xrightarrow[\overline{a}_q]{} & \mathsf{T}^{\mathsf{B}}\, U_q \end{array}$$

satisfy the following morphism *and* seed preservation *conditions:*

$$\overline{a}_q \cdot h \;=\; \mathsf{T}^{\mathsf{B}}\, h \cdot \overline{a}_p \tag{4}$$

$$h\; u_p \;=\; u_q \tag{5}$$

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*

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

whose action 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* [5]

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

objects, usually referred to as a (hom-)set, acquires itself the structure of a category. Therefore the standard arrow composition and unit laws become functorial, since they transform both objects and arrows of each hom-set in a uniform way. A typical example is Cat itself: the category whose objects are small categories, arrows are functors and arrows between arrows, or 2-cells as they are often called, correspond to natural transformations.

[5] As one would expect, reasoning about generic components entails a number of laws relating monads with common 'housekeeping' morphisms such as product and sum associativity, (a, a_+), commutativity (s, s_+), left and right units (l, l_+ and r, r_+), left and right distributivity (dl, dr) and 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 $\mathsf{m} : (A \times B) \times (C \times D) \longrightarrow (A \times C) \times (B \times D)$. Such laws are thoroughly dealt with in [4]. By convention, binary morphisms always associate to the left.

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 notation, this will be also referred to as* copy_K. *Similarly, the action on morphisms is the constant morphism* $\mathsf{id}_{\mathbf{1}}$.

The fact that, for each strong monad B, components form a bicategory amounts not only to a standard definition of the two basic combinators ; and copy_K of a component calculus, but also to setting up its basic laws. Recall (from *e.g.* [32]) that the graph of a morphism is a bisimulation. Therefore, the existence of a seed preserving morphism between two components makes them T^{B}-bisimilar, leading to the following laws, for appropriately typed components p, q and r:

$$\mathsf{copy}_I \,;\, p \;\sim\; p \;\sim\; p \,;\, \mathsf{copy}_O \tag{6}$$

$$(p \,;\, q) \,;\, r \;\sim\; p \,;\, (q \,;\, r) \tag{7}$$

3.2 Computing behaviour.

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, behaviour $[\![p]\!]$ of a component p is computed by *coinductive extension*, taking the seed-value of p as the starting state. I.e.,

$$[\![p]\!] \;=\; [\![\overline{a}_p]\!] u_p$$

Behaviours organise themselves in a category Bh, whose objects are sets and arrows $b : I \longrightarrow O$ elements 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, composition in Bh can be defined 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 abstract with respect to bisimulation, the bicategorical

structure collapses. Moreover, as discussed below, some tensors in 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 (6) and (7), 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

Lemma 1. Bh *is a category and* $[\![\ ]\!]$ *is a 2-functor from* Cp *to* Bh

Proof. 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 establishes $\mathsf{copy}_I \,;\, b = b$. On the other hand, for suitably typed behaviours b, c and d,

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

For the second part consider

- $[\![{\mathsf{copy}_K}^{\mathsf{Cp}}]\!] \;=\; {\mathsf{copy}_K}^{\mathsf{Bh}}$, which is trivial to check, and
-

$$\begin{aligned}[\![(p \,;^{\mathsf{Cp}}\, q)]\!] \;&= [\![\overline{a}_{p;q}]\!]\langle u_p, u_q\rangle \\ &= [\![(\omega_{I,K} \,;\, \omega_{K,O})]\!] \cdot ([\![\overline{a}_p]\!] \times [\![\overline{a}_q]\!])\langle u_p, u_q\rangle \\ &= \;;^{\mathsf{Bh}} \cdot ([\![\overline{a}_p]\!] \times [\![\overline{a}_q]\!])\langle u_p, u_q\rangle \\ &= \;;^{\mathsf{Bh}} \langle [\![\overline{a}_p]\!] u_p, [\![\overline{a}_q]\!] u_q\rangle \\ &= [\![p]\!] \;;^{\mathsf{Bh}}\; [\![q]\!]\end{aligned}$$

□

4 A component calculus

We shall now look at the structure of Cp by introducing an algebra of T^{B}-components parametric on a behaviour model B. This structure lifts naturally to Bh defining a particular (typed) 'process' algebra.

4.1 Functions as components.

Let us start from the simple observation that functions can be regarded as particular instances of components, whose interfaces are given by their domain and codomain types. Formally,

Definition 3. *A function $f : A \longrightarrow B$ is represented in* Cp *by*

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

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

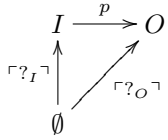
$$a_{\ulcorner f \urcorner} = \mathbf{1} \times A \xrightarrow{\mathsf{id} \times f} \mathbf{1} \times B \xrightarrow{\eta_{(\mathbf{1} \times B)}} \mathsf{B}(\mathbf{1} \times B)$$

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

$$\ulcorner f \cdot g \urcorner \sim \ulcorner g \urcorner ; \ulcorner f \urcorner \tag{8}$$

$$\ulcorner \mathsf{id}_I \urcorner \sim \mathsf{copy}_I \tag{9}$$

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,

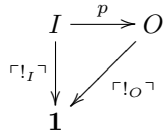


because both $\ulcorner ?_I \urcorner$ and $\ulcorner ?_O \urcorner$ are the *inert* components: the absence of input makes reaction impossible. Formally,

$$\ulcorner ?_I \urcorner ; p \;\sim\; \ulcorner ?_O \urcorner \tag{10}$$

Equation (10) lifts to an equality in Bh, as does any other bisimulation equation in Cp. Therefore, $\emptyset$ is the *initial* object in Bh.

Naturality is lost, however, in the lifting of $!_I : I \longrightarrow \mathbf{1}$, as the following diagram fails to commute for non trivial B



To check this, take B as the finite powerset monad. Clearly, $p ; \ulcorner !_O \urcorner$ deadlocks 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 $\mathbf{1}$ fails to 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 $\mathsf{B} = \mathsf{Id}$).

Clearly, *isomorphisms*, *split monos* and *split epis* lift to Cp as, respectively, isomorphisms, split monos and split epis.

Proof. Let $f : A \longrightarrow B$ be a Set-isomorphism. Then

$$\ulcorner f \urcorner ; \ulcorner f^{\circ} \urcorner \sim \ulcorner f^{\circ} \cdot f \urcorner \sim \ulcorner \mathsf{id}_A \urcorner \sim \mathsf{copy}_A$$

Conversely,

$$\ulcorner f^{\circ} \urcorner ; \ulcorner f \urcorner \sim \ulcorner f \cdot f^{\circ} \urcorner \sim \ulcorner \mathsf{id}_B \urcorner \sim \mathsf{copy}_B$$

For f a split epi (respectively, a split mono) consider the first (respectively, second) part of the proof above, taking f° as a section (respectively, a retraction) of f. (recalling that every epi and every mono with non empty source split in Set)

□

Wires are components over $\mathbf{1}$ defined from identities and structural properties of the underlying category. Typical examples, include the liftings of canonical isomorphisms — a, s, l or r — which leads to bisimilarity up to an isomorphic rearranging of the interface, as well as liftings of embeddings, projections, *codiagonals* and *diagonals*, the latter used to *merge* input and *replicate* output types, as in

$$\ulcorner \triangledown \urcorner ; p ; \ulcorner \triangle \urcorner : I + I \longrightarrow O \times O$$

4.2 Wrapping.

The pre- and post-composition of a component with Cp-lifted functions can be encapsulated into a unique combinator, called *wrapping*, which is reminiscent of the *renaming* connective found in process calculi (*e.g.*, [25]). Let $p : I \longrightarrow O$ be a component and consider functions $f : I' \longrightarrow I$ and $g : O \longrightarrow O'$. Component p wrapped by f and g, denoted by $p[f, g]$ and typed as $I' \longrightarrow O'$, is defined by input pre-composition with f and output post-composition with g. Formally,

Definition 4. *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 component $\langle u_p, \overline{a}_p \rangle$ *into* $\langle u_p, \overline{a}_{p[f,g]} \rangle$*, where*

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

As expected, the following properties hold:

$$p[f, g] \sim \ulcorner f \urcorner ; p ; \ulcorner g \urcorner \tag{11}$$

$$(p[f, g])[f', g'] \sim p[f \cdot f', g' \cdot g] \tag{12}$$

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:

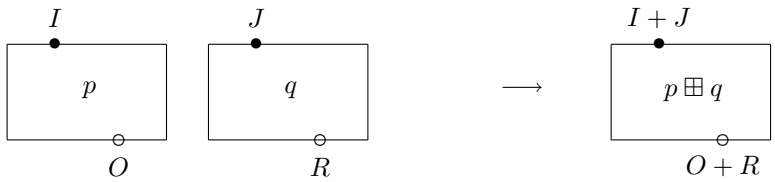
$$\mathsf{inert}_A \;=\; \ulcorner ?_A \urcorner \tag{13}$$

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.*, pushing a button) leading to an (similarly) unstructured reaction (*e.g.*, switching on a led).

4.3 Choice.

Components can be aggregated in a number of different ways, besides the 'pipeline' composition discussed above. Next, we introduce three other generic combinators, corresponding to *choice*, *parallel* and *concurrent* composition.

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. The first composition pattern to be considered is *external choice*, as depicted bellow:



When interacting with $p \boxplus q$, the environment is allowed to choose either to input a value of type I or one of type J, triggering the corresponding component (p or q, respectively) and producing output. Formally,

Definition 5. *The* choice *combinator is defined as a lax functor* $\boxplus : \mathsf{Cp} \times \mathsf{Cp} \longrightarrow \mathsf{Cp}$*, which 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

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

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

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:

$$(p \boxplus p') \,;\, (q \boxplus q') \sim (p \,;\, q) \boxplus (p' \,;\, q') \tag{14}$$

$$\mathsf{copy}_{K \boxplus K'} \sim \mathsf{copy}_K \boxplus \mathsf{copy}_{K'} \tag{15}$$

$$\ulcorner f \urcorner \boxplus \ulcorner g \urcorner \sim \ulcorner f + g \urcorner \tag{16}$$

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

$$(p \boxplus q) \boxplus r \sim (p \boxplus (q \boxplus r))[\mathsf{a}_+, \mathsf{a}_+{}^\circ] \tag{17}$$

$$\mathsf{nil} \boxplus p \;\sim\; p[\mathsf{r}_+, \mathsf{r}_+{}^\circ] \;\text{ and }\; p \boxplus \mathsf{nil} \;\sim\; p[\mathsf{l}_+, \mathsf{l}_+{}^\circ] \tag{18}$$

$$p \boxplus q \sim (q \boxplus p)[\mathsf{s}_+, \mathsf{s}_+] \tag{19}$$

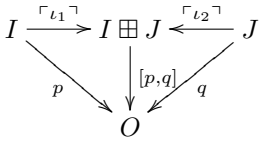
Laws (17) to (19) can be alternatively stated as providing evidence that the canonical Set isomorphisms a_+, r_+, l_+ and s_+, once lifted to Cp, keep their naturality up to bisimulation.

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 the following diagram commutes up to bisimulation,



$$\ulcorner \iota_1 \urcorner \,;\, [p, q] \sim\; p \qquad \ulcorner \iota_2 \urcorner \,;\, [p, q] \sim\; q \tag{20}$$

even though $[p, q]$ is not the unique arrow making the diagram commute. This is formalized in the following lemma whose proof is included to give a flavour of the calculation style adopted here.

Lemma 2. *The choice combinator* $\boxplus$ *lifts to a* weak coproduct *in* Bh.

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 (20), 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 morphisms from the left to the right. Therefore,

$$
\begin{array}{ll}
 & \mathsf{B}(\pi_1 \times \triangledown) \cdot [\mathsf{B}(\mathsf{id} \times \iota_1), \mathsf{B}(\mathsf{id} \times \iota_2)] \cdot (\mathsf{B}\mathsf{x}\mathsf{r} + \mathsf{B}\mathsf{a}^\circ) \cdot (\tau_r + \tau_l) \cdot (a_p \times \mathsf{id} + \mathsf{id} \times a_q) \\
 & \cdot(\mathsf{x}\mathsf{r} + \mathsf{a}) \cdot \mathsf{d}\mathsf{r} \cdot (\mathsf{id} \times \iota_1) \\
= & \quad \{ \text{ law: } \iota_1 = \mathsf{d}\mathsf{r} \cdot (\mathsf{id} \times \iota_1) \ (\textit{cf.}, [4]) \ \} \\
 & \mathsf{B}(\pi_1 \times \triangledown) \cdot [\mathsf{B}(\mathsf{id} \times \iota_1), \mathsf{B}(\mathsf{id} \times \iota_2)] \cdot (\mathsf{B}\mathsf{x}\mathsf{r} + \mathsf{B}\mathsf{a}^\circ) \cdot (\tau_r + \tau_l) \cdot (a_p \times \mathsf{id} + \mathsf{id} \times a_q) \\
 & \cdot(\mathsf{x}\mathsf{r} + \mathsf{a}) \cdot \iota_1 \\
= & \quad \{ \ + \text{ absorption and cancellation } \} \\
 & \mathsf{B}(\pi_1 \times \triangledown) \cdot \mathsf{B}(\mathsf{id} \times \iota_1) \cdot \mathsf{B}\mathsf{x}\mathsf{r} \cdot \tau_r \cdot a_p \times \mathsf{id} \cdot \mathsf{x}\mathsf{r} \\
= & \quad \{ \text{ routine: } \triangledown \cdot \iota_1 = \mathsf{id} \ \} \\
 & \mathsf{B}(\pi_1 \times \mathsf{id}) \cdot \mathsf{B}\mathsf{x}\mathsf{r} \cdot \tau_r \cdot a_p \times \mathsf{id} \cdot \mathsf{x}\mathsf{r} \\
= & \quad \{ \text{ routine: } (\pi_1 \times \mathsf{id}) \cdot \mathsf{x}\mathsf{r} = \pi_1 \ \} \\
 & \mathsf{B}\pi_1 \cdot \tau_r \cdot a_p \times \mathsf{id} \cdot \mathsf{x}\mathsf{r} \\
= & \quad \{ \text{ law: } \mathsf{B}\pi_1 \cdot \tau_r = \pi_1 \ (\textit{cf.}, [4]) \ \} \\
 & \mathsf{B}\pi_1 \cdot a_p \times \mathsf{id} \cdot \mathsf{x}\mathsf{r} \\
= & \quad \{ \ \times \text{ definition and cancellation } \} \\
 & a_p \cdot \pi_1 \cdot \mathsf{x}\mathsf{r} \\
= & \quad \{ \text{ routine: } (\pi_1 \times \mathsf{id}) \cdot \mathsf{x}\mathsf{r} = \pi_1 \text{ and } \mathsf{x}\mathsf{r} = \mathsf{x}\mathsf{r}^\circ \ \} \\
 & a_p \cdot (\pi_1 \times \mathsf{id})
\end{array}
$$

which establishes the first clause of (20). A similar calculation will prove the second one. Note that in both cases seeds are trivially preserved. It is impossible to turn *either* into a universal construction in Bh. The basic observation is that the codiagonal $\triangledown$ 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 $(p \boxplus p)$; $\ulcorner \triangledown \urcorner$ is $\langle 5, 3, 9, \ldots \rangle$ while $\ulcorner \triangledown \urcorner$; p, resorting only to one copy of p, produces $\langle 5, 8, 12, \ldots \rangle$.

□

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*

$$[\ulcorner \iota_1 \urcorner, \ulcorner \iota_2 \urcorner] \ \sim \ \mathsf{copy}_{I+J} \tag{21}$$

and *absorption*

$$(p \boxplus q) \ ; [p', q'] \ \sim \ [p \ ; p', q \ ; q'] \tag{22}$$

are easy to prove. For example,

$$
\begin{array}{ll}
 & (p \boxplus q)\,;\,[p', q'] \\
\sim & \quad \{ \text{ definition of } \textit{either} \text{ in } \mathsf{Cp} \} \\
 & (p \boxplus q)\,;\,((p' \boxplus q')\,;\ulcorner \triangledown \urcorner) \\
\sim & \quad \{ \text{ ; associative (7) } \} \\
 & ((p \boxplus q)\,;\,(p' \boxplus q'))\,;\ulcorner \triangledown \urcorner \\
\sim & \quad \{ \boxplus \text{ functor (14) } \} \\
 & ((p\,;\,p') \boxplus (q\,;\,q'))\,;\ulcorner \triangledown \urcorner \\
\sim & \quad \{ \text{ definition of } \textit{either} \text{ in } \mathsf{Cp} \} \\
 & [p\,;\,p', q\,;\,q']
\end{array}
$$

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

$$p \boxplus q \;\sim\; [p\,;\ulcorner \iota_1 \urcorner, p\,;\ulcorner \iota_2 \urcorner] \tag{23}$$

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

$$\ulcorner \iota_1 \urcorner\,;\,(p \boxplus q) \;\sim\; p\,;\ulcorner \iota_1 \urcorner \quad \text{and} \quad \ulcorner \iota_2 \urcorner\,;\,(p \boxplus q) \;\sim\; q\,;\ulcorner \iota_2 \urcorner \tag{24}$$

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

$$p\,;\ulcorner \iota_1 \urcorner \;\sim\; \ulcorner \iota_1 \urcorner\,;\,(p \boxplus p) \tag{25}$$

4.5 Parallel.

Parallel composition, denoted by $p \boxtimes q$, 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, product fails as well. Formally,

Definition 6. *The* parallel *combinator $\boxtimes$ is defined by an action $I \boxtimes J \;=\; I \times J$ on objects and a family of functors*

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

which yields

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

where

$$
\begin{array}{rl}
a_{p \boxtimes q} = & U_p \times U_q \times (I \times J) \xrightarrow{\;\mathsf{m}\;} U_p \times I \times (U_q \times J) \\
 & \xrightarrow{\;a_p \times a_q\;} \mathsf{B}\,(U_p \times O) \times \mathsf{B}\,(U_q \times R) \\
 & \xrightarrow{\;\delta_l\;} \mathsf{B}\,(U_p \times O \times (U_q \times R)) \\
 & \xrightarrow{\;\mathsf{B}\,\mathsf{m}\;} \mathsf{B}\,(U_p \times U_q \times (O \times R))
\end{array}
$$

and maps every pair of arrows $\langle h_1, h_2 \rangle$ *into* $h_1 \times h_2$.

The following laws hold for $\boxtimes$:

$$\textit{lax} \quad (p \boxtimes p') \,;\, (q \boxtimes q') \;\sim\; (p \,;\, q) \boxtimes (p' \,;\, q') \tag{26}$$

$$\mathsf{copy}_{K \boxtimes K'} \;\sim\; \mathsf{copy}_K \boxtimes \mathsf{copy}_{K'} \tag{27}$$

$$\textit{functions} \quad \ulcorner f \urcorner \boxtimes \ulcorner g \urcorner \;\sim\; \ulcorner f \times g \urcorner \tag{28}$$

$$\textit{assoc} \quad (p \boxtimes q) \boxtimes r \;\sim\; (p \boxtimes (q \boxtimes r))[\mathsf{a}, \mathsf{a}^\circ] \tag{29}$$

$$\textit{id} \quad \mathsf{idle} \boxtimes p \;\sim\; p[\mathsf{r}, \mathsf{r}^\circ] \tag{30}$$

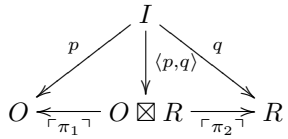
$$\textit{zero} \quad \mathsf{nil} \boxtimes p \;\sim\; \mathsf{nil}[\mathsf{zl}, \mathsf{zl}^\circ] \tag{31}$$

$$\textit{comm} \quad p \boxtimes q \;\sim\; (q \boxtimes p)[\mathsf{s}, \mathsf{s}] \quad \text{if } \mathsf{B} \text{ is } \textit{commutative} \tag{32}$$

Again one may ask whether $\boxtimes$ lifts to a universal product construction at the behavioural level. Dually to the *either* combinator, we start by definning the *split* of two components as

$$\langle p, q \rangle \;=\; \ulcorner \triangle \urcorner \,;\, (p \boxtimes q) \quad \text{where} \quad \triangle = \langle id, id \rangle$$

This definition, however, does not guarantee, in general, the commutativity of



It *does*, however, and a *cancellation* law

$$\langle p, q \rangle \,;\, \ulcorner \pi_1 \urcorner \;\sim\; p \tag{33}$$

holds, for *commutative* monads B which exclude the possibility of *failure* (*e.g.*, the non-empty powerset).

Proof. To establish (33) it is enough to check whether $\pi_1 : U_p \times U_q \longrightarrow U_p$ is a morphism. In fact,

$$\begin{aligned}
& \mathsf{B}(\pi_1 \times \mathsf{id}) \cdot a_{\langle p,q \rangle ; \ulcorner \pi_1 \urcorner} \\
= \quad & \{ \text{ definitions } \} \\
& \mathsf{B}(\pi_1 \times \pi_1) \cdot \mathsf{Bm} \cdot \delta_l \cdot (a_p \times a_q) \cdot \mathsf{m} \cdot (\mathsf{id} \times \triangle) \\
= \quad & \{ \text{ routine: } \pi_1 \times \pi_1 = \pi_1 \cdot \mathsf{m}, \mathsf{m}^\circ = \mathsf{m} \} \\
& \mathsf{B}\pi_1 \cdot \delta_l \cdot (a_p \times a_q) \cdot \mathsf{m} \cdot (\mathsf{id} \times \triangle) \\
= \quad & \{ \star \} \\
& \pi_1 \cdot (a_p \times a_q) \cdot \mathsf{m} \cdot (\mathsf{id} \times \triangle) \\
= \quad & \{ \times \text{ cancellation } \} \\
& a_p \cdot \pi_1 \cdot \mathsf{m} \cdot (\mathsf{id} \times \triangle) \\
= \quad & \{ \text{ routine: } \pi_1 \times \pi_1 = \pi_1 \cdot \mathsf{m} \text{ and } \pi_1 \cdot \triangle = \mathsf{id} \} \\
& a_p \cdot (\pi_1 \times \mathsf{id})
\end{aligned}$$

□

On the other hand, diagonal $\triangle$ keeps its naturality when lifted to Cp, for B expressing derministic behaviour (*e.g.*, the identity or the *maybe* monad), entailing a *fusion* law:$r \,;\, \langle p, q\rangle \sim \langle r \,;\, p, r \,;\, q\rangle$.

Proof.

$$
\begin{array}{ll}
 & a_{(p \boxtimes p)} \cdot (\triangle \times \triangle) \\
= & \quad \{ \ \boxtimes \text{ definition, } \mathsf{m} \cdot (\triangle \times \triangle) = \triangle \ \} \\
 & \mathsf{Bm} \cdot \delta_l \cdot (a_p \times a_p) \cdot \triangle \\
= & \quad \{ \ \triangle \text{ natural } \} \\
 & \mathsf{Bm} \cdot \delta_l \cdot \triangle \cdot a_p \\
= & \quad \{ \ \star \ \} \\
 & \mathsf{Bm} \cdot \mathsf{B} \triangle \cdot a_p \\
= & \quad \{ \ \mathsf{m} \cdot \triangle = \triangle \times \triangle \text{ and definition } \} \\
 & \mathsf{B}(\triangle \times \mathsf{id}) \cdot a_{p;\ulcorner \triangle \urcorner}
\end{array}
$$

However, $\delta_l \cdot \triangle = \mathsf{B} \triangle$ does not hold for any monad involving the notion of a collection. As a counterexample consider

$$
\begin{aligned}
(\delta_l \cdot \triangle)\{5,2\} &= \{\{5,5\},\{5,2\},\{2,5\},\{2,2\}\} \\
(\mathcal{P}\triangle)\{5,2\} &= \{\{5,5\},\{2,2\}\}
\end{aligned}
$$

On the other hand, the *fusion* law follows from

$$
\begin{array}{ll}
 & r \,;\, \langle p, q\rangle \\
\sim & \quad \{ \ \text{definition } \} \\
 & r \,;\, (\ulcorner \triangle \urcorner \,;\, (p \boxtimes q)) \\
\sim & \quad \{ \ \text{assumption and assoc } \} \\
 & \ulcorner \triangle \urcorner \,;\, ((r \boxtimes r) \,;\, (p \boxtimes q)) \\
\sim & \quad \{ \ ; \text{lax functor} \} \\
 & \ulcorner \triangle \urcorner \,;\, ((r \,;\, p) \boxtimes (r \,;\, q)) \\
\sim & \quad \{ \ \text{definition } \} \\
 & \langle r \,;\, p, r \,;\, q\rangle
\end{array}
$$

□

Combining these two results, one concludes that $\boxtimes$ is a *product* in Bh, but only for behaviour models excluding both failure and non determinism, which narrows the applicability scope of this fact to the category of total deterministic components. However,

reflection, *absorption* and *definition* laws hold for any B:

$$\textit{reflection} \quad \langle \ulcorner \pi_1 \urcorner, \ulcorner \pi_2 \urcorner \rangle \;\sim\; \mathsf{copy}_{O \times R} \tag{34}$$

$$\textit{absorption} \quad \langle p, q \rangle \,;\, (p' \boxtimes q') \;\sim\; \langle p \,;\, p', q \,;\, q' \rangle \quad \text{for } \mathsf{B} \text{ commutative} \tag{35}$$

$$\textit{definition} \quad p \boxtimes q \;\sim\; \langle \ulcorner \pi_1 \urcorner \,;\, p, \ulcorner \pi_2 \urcorner \,;\, q \rangle \tag{36}$$

Product projections, on the other hand, keep naturality only when cancellation holds. Always, however, one has

$$(\ulcorner f \urcorner \boxtimes q) \,;\, \ulcorner \pi_2 \urcorner \;\sim\; \ulcorner \pi_2 \urcorner \,;\, q \tag{37}$$

$$(p \boxtimes \ulcorner f \urcorner) \,;\, \ulcorner \pi_1 \urcorner \;\sim\; \ulcorner \pi_1 \urcorner \,;\, p \tag{38}$$

4.6 Concurrent.

Finally, *concurrent* composition, denoted by $\circledast$, combines choice and parallel, in the sense that p and q can be executed independently or jointly, depending on the input supplied. Formally,

Definition 7. *The* concurrent *combinator is defined by an action* $I \circledast J \;=\; I + J + I \times J$ *on objects and a family of functors*

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

yielding

$$p \circledast q \;=\; \langle \langle u_0, v_0 \rangle \in U_p \times U_q, \overline{a}_{p \circledast q} \rangle$$

where

$$a_{p \circledast q} = \qquad U_p \times U_q \times (I \circledast J) \;\Big\downarrow{\scriptstyle [\mathsf{B}(\mathsf{id} \times \iota_1), \mathsf{B}(\mathsf{id} \times \iota_2)] \cdot (a_{p \boxplus q} + a_{p \boxtimes q}) \cdot \mathsf{dr}}\; \mathsf{B}\,(U_p \times U_q \times (O \circledast R))$$

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

The laws of concurrent composition combine corresponding results about $\boxplus$ and $\boxtimes$. In particular we get again permutation with sequential composition and the structure of a tensor product, which is symmetric for commutative behaviour monads. Moreover, the following reduction laws relate $\circledast$ to the other two tensors:

$$\ulcorner \iota_1 \urcorner \,;\, (p \circledast q) \;\sim\; (p \boxplus q) \,;\, \ulcorner \iota_1 \urcorner \tag{39}$$

$$\ulcorner \iota_2 \urcorner \,;\, (p \circledast q) \;\sim\; (p \boxtimes q) \,;\, \ulcorner \iota_2 \urcorner \tag{40}$$

Proof.

$$
\begin{array}{ll}
& a_{(p \circledast q)[\iota_1, \mathsf{id}]} \\
= & \quad \{ \ \circledast \text{ and } \textit{wrapping} \text{ definitions } \} \\
& [\mathsf{B}(\mathsf{id} \times \iota_1), \mathsf{B}(\mathsf{id} \times \iota_2)] \cdot (a_{(p \boxplus q)} + a_{(p \boxtimes q)}) \cdot \mathsf{dr} \cdot (\mathsf{id} \times \iota_1) \\
= & \quad \{ \ \text{routine } \} \\
& [\mathsf{B}(\mathsf{id} \times \iota_1), \mathsf{B}(\mathsf{id} \times \iota_2)] \cdot (a_{(p \boxplus q)} + a_{(p \boxtimes q)}) \cdot \iota_1 \\
= & \quad \{ \ + \text{ absorption } \} \\
& [\mathsf{B}(\mathsf{id} \times \iota_1) \cdot a_{(p \boxplus q)}, \mathsf{B}(\mathsf{id} \times \iota_1) \cdot a_{(p \boxtimes q)}] \cdot \iota_1 \\
= & \quad \{ \ + \text{ cancellation, } \circledast \text{ and } \textit{wrapping} \text{ definition } \} \\
& \mathsf{B}(\mathsf{id} \times \iota_1) \cdot a_{(p \boxplus q)} \ = \ a_{(p \boxplus q)}[\mathsf{id}, \iota_1]
\end{array}
$$

□

4.7 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 forces *part* of the output of a component to be fed back as input. Formally,

Definition 8. *The* hook *combinator $-\,{}^{\curvearrowleft}_{Z}$ is defined, for each tuple of objects $\langle I, O, Z \rangle$, as a functor between the (categories underlying) hom-sets $\mathsf{Cp}(I+Z, O+Z)$ and $\mathsf{Cp}(I+Z, O+Z)$ which is is the identity on arrows and maps each component $p : I + Z \longrightarrow O + Z$ to $p\,{}^{\curvearrowleft}_{Z} \colon I + Z \longrightarrow O + Z$ given by*

$$p\,{}^{\curvearrowleft}_{Z} = \ \langle u_p \in U_p, \overline{a}_{p\,{}^{\curvearrowleft}_{Z}} \rangle$$

where

$$
\begin{array}{rl}
a_{p\,{}^{\curvearrowleft}_{Z}} \ = & U_p \times (I + Z) \xrightarrow{\quad a_p \quad} \mathsf{B}(U_p \times (O + Z)) \\
& \xrightarrow{\mathsf{B}((\mathsf{id} \times \iota_1 + \mathsf{id} \times \iota_2) \cdot \mathsf{dr})} \mathsf{B}(U_p \times (O + Z) + U_p \times (I + Z)) \\
& \xrightarrow{\mathsf{B}(\eta + a_p)} \mathsf{B}(\mathsf{B}(U_p \times (O + Z)) + \mathsf{B}(U_p \times (O + Z))) \\
& \xrightarrow{\quad \mu \cdot \mathsf{B} \nabla \quad} \mathsf{B}(U_p \times (O + Z))
\end{array}
$$

i.e., $a_{p\,{}^{\curvearrowleft}_{Z}} = (\nabla \cdot (\eta + a_p) \cdot (\mathsf{id} \times \iota_1 + \mathsf{id} \times \iota_2) \cdot \mathsf{dr}) \bullet a_p$.

For components with the same input/output type, the *hook* combinator has a particularly simple definition as the Kleisli composition of the original dynamics. It is then called a *feedback* and denoted by

$$p\,{}^{\curvearrowleft} \colon Z \longrightarrow Z \ = \langle u_p \in U_p, \overline{a}_{p\,{}^{\curvearrowleft}} \rangle$$

where

$$a_{p^{\curvearrowleft}} = \quad U_p \times Z \xrightarrow{a_p} \mathsf{B}(U_p \times Z) \xrightarrow{\mathsf{B}a_p} \mathsf{BB}(U_p \times Z) \xrightarrow{\mu} \mathsf{B}(U_p \times Z)$$

i.e., $a_{p^{\curvearrowleft}} = a_p \bullet a_p$,

Both *hook* and *feedback* specialise to components representing functions according to the following laws,

$$\ulcorner f \urcorner \, \curvearrowleft \sim \ulcorner f \cdot f \urcorner \tag{41}$$

$$\ulcorner g \urcorner \curvearrowleft_Z \sim \ulcorner [\iota_1, g \cdot \iota_2] \cdot g \urcorner \tag{42}$$

for $f : Z \longrightarrow Z$ and $g : I + Z \longrightarrow O + Z$.
Moreover, for components $p : Z \longrightarrow Z$ and $q : I \longrightarrow O$, one has

$$p \, \curvearrowleft \sim p[\mathsf{r}_+, \mathsf{r}_+{}^\circ] \curvearrowleft_Z [\mathsf{r}_+{}^\circ, \mathsf{r}_+] \tag{43}$$

$$q \boxplus p \, \curvearrowleft \sim (q \boxplus p) \curvearrowleft_Z \tag{44}$$

$$p \curvearrowleft \boxplus q \sim (q \boxplus p) \curvearrowleft_Z [\mathsf{s}_+, \mathsf{s}_+] \tag{45}$$

All laws above, with the exception of (45) are actually strict Cp arrow equalities, and not just bisimulations. Also notice that equation (44) generalises to

$$(q \boxplus p)[\mathsf{a}_+, \mathsf{a}_+{}^\circ] \curvearrowleft_Z \sim (q \boxplus p \curvearrowleft_Z)[\mathsf{a}_+, \mathsf{a}_+{}^\circ] \tag{46}$$

for $p : J + Z \longrightarrow R + Z$.

The last set of laws relate *hook* and *feedback* with the other combinators in the calculus. Let $p : Z \longrightarrow Z$ and $q : R \longrightarrow R$ be components and $i : W \longrightarrow Z$ be a Set isomorphism. Then

$$p \, \curvearrowleft [i, i^\circ] \sim p[i, i^\circ] \, \curvearrowleft \tag{47}$$

$$(p \square q) \, \curvearrowleft \sim p \, \curvearrowleft \square \, q \, \curvearrowleft \tag{48}$$

for $\square = \boxplus, \boxtimes$ or $\circledast$. Finally, let $p : I + K \longrightarrow O + K$ and $q : J \longrightarrow R$ be components, $f : I' \longrightarrow I$, $g : O \longrightarrow O'$ functions and $i : W \longrightarrow K$ a Set isomorphism. Then

$$p \curvearrowleft_K [f + i, g + i^\circ] \sim p[f + i, g + i^\circ] \curvearrowleft_W \tag{49}$$

$$(p \boxplus q)[\mathsf{xr}_+, \mathsf{xr}_+] \curvearrowleft_K \sim (p \curvearrowleft_K \boxplus q)[\mathsf{xr}_+, \mathsf{xr}_+] \tag{50}$$

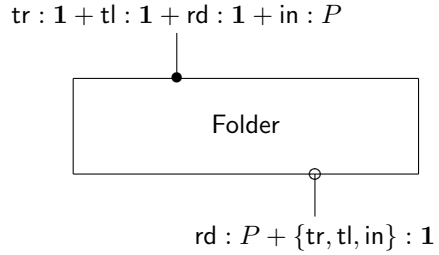
Note that all equations are strict Cp arrow equalities. However, validity of (48), for $\square = \boxtimes, \circledast$, depends on the commutativity of the behaviour monad B. The reader is refered to [4] for proofs of all laws mentioned in this section.

5 Example: A 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 [29]. 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 'port' 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 that two stacks are used to 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 observes 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 $\triangledown_P$ function. 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:

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

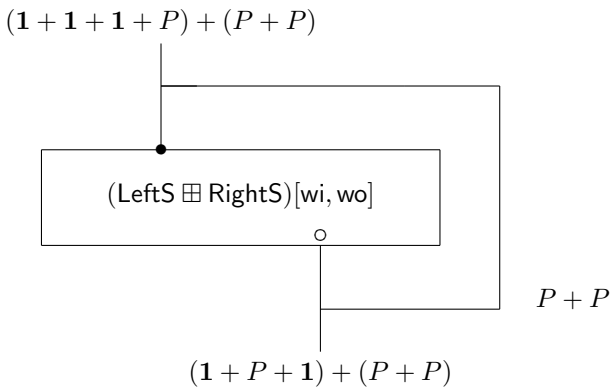
The next step builds the desirable connections using *hook* over this composite, which requires a previous wrapping by a pair of suitable isomorphisms:

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

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

$$\mathsf{wi} \ = \ \Big[\big[[\iota_{11}, \iota_{211}], \iota_{212}\big], \iota_{222}\Big], [\iota_{221}, \iota_{12}]\Big] \quad \mathsf{wo} \ = \ \Big[[\iota_{21}, \iota_{111}], \big[[\iota_{22}, \iota_{112}], \iota_{12}\big]\Big]$$

In a diagram:



Finally, to conform 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 $\mathsf{fo} = \big[[[\iota_2, \iota_1], \iota_2], \iota_2 \cdot !_{P+P}\big]$. 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 Example: The game of life

The following example illustrates the use of some component combinators to connect elementary state-based specifications. The component to be built is known as the *game of life*, a simple model of cellular behaviour which has been popularised as a common screen locker for computers.

The game is based on a grid of cells each of which sends and receives elementary stimulus to and from its four adjacent neighbours. A stimulus is a Boolean value indicating whether the cell is either 'alive' or 'dead'. The following few rules govern the survival, death and birth of cell generations:

- Each living cell with less than two or more than three living neighbours dies in the next generation.
- Each dead cell with exactly three living neighbours becomes alive.
- Each living cell with less than two or three living neighbours survives until the next generation.

Each cell will be specified as a component Cell whose input is a tuple of four Boolean values, each one to be supplied by one of the four adjacent cells. The cell reacts to such a stimulus by computing its new state — 'dead' or 'alive' — and by making it available as an output to its neighbours, used to compute the next cell generation. Formally, we define

$$\mathsf{Cell} : \mathbf{2} \times \mathbf{2} \times \mathbf{2} \times \mathbf{2} \longrightarrow \mathbf{2} \quad = \quad \langle \mathsf{true} \in \mathbf{2}, \overline{a}_{\mathsf{Cell}} \rangle$$

where

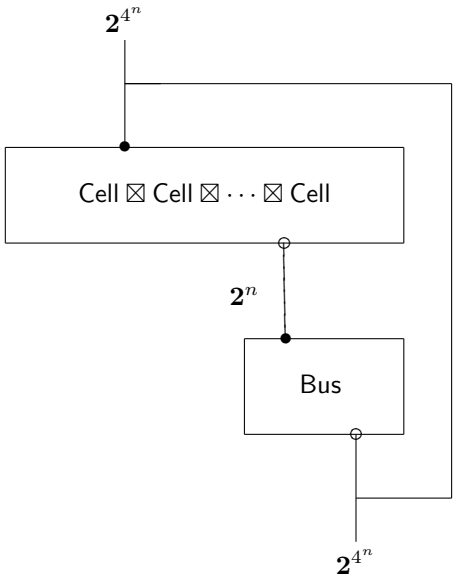
$$a_{\mathsf{Cell}}\ \langle u, t\rangle\ =\ \text{let}\ n = \mathsf{living}\ t \quad \text{in} \begin{cases} \langle \mathsf{false}, \mathsf{false}\rangle & \text{if}\ u = \mathsf{true} \wedge (n < 1 \vee n > 3) \\ \langle \mathsf{true}, \mathsf{true}\rangle & \text{if}\ u = \mathsf{false} \wedge n = 3) \\ \langle u, u\rangle & \text{otherwise} \end{cases}$$

Function living above, counts the number of living stimuli (*i.e.*, the number of true values) in a four Boolean tuple. So, $U_{\mathsf{Cell}} = \mathbf{2}$ and $\mathsf{B} = \mathsf{Id}$. The game's behaviour is, of course, deterministic and all cells in the grid react simultaneously to produce the new generation. To form a grid of n cells we simply connect them using the *parallel* combinator $\boxtimes$. The crucial point is to devise a wiring scheme to guarantee that the joint output of the n connected cells is appropriately fed back. The composed system is pictured below, where component

$$\mathsf{Bus} : \mathbf{2}^n \longrightarrow \mathbf{2}^{4^n}$$

concentrates and correctly distributes the output.

The n cells are organised as a fully connected matrix of k rows and l columns ($n = k \times l$), so that the neighbours of cell $\langle i, j\rangle$ are $\langle i-1, j\rangle$, $\langle i+1, j\rangle$, $\langle i, j-1\rangle$ and $\langle i, j+1\rangle$ (in the 'west', 'east', 'north' and 'south' directions, respectively) computed in the k and l rings (*i.e.*, $1 - 1 = k$, $k + 1 = 1$ and $1 - 1 = l$, $l + 1 = 1$).



To specify Bus we adopt the following convention: the first cell in the $\boxtimes$-expression has coordinates $\langle 1, 1\rangle$, second is $\langle 1, 2\rangle$ and so on until column n is reached; the next cell is then $\langle 2, 1\rangle$. Under this convention the output produced by cell $\langle i, j\rangle$ is selected from the

global output tuple as the $j + (n \times (i-1))$-projection, *i.e.*

$$\begin{aligned} \mathsf{out}_{\langle i,j\rangle} &: \mathbf{2}^n \longrightarrow \mathbf{2} \\ \mathsf{out}_{\langle i,j\rangle} &= \pi_{j+(n\times(i-1))} \end{aligned}$$

Now, the input to cell $\langle i, j\rangle$ is simply the *split* of the outputs of its neighbours, *i.e.*,

$$\begin{aligned} \mathsf{in}_{\langle i,j\rangle} &: \mathbf{2}^n \longrightarrow \mathbf{2}^4 \\ \mathsf{in}_{\langle i,j\rangle} &= \langle \mathsf{out}_{\langle i,\mathsf{dec}_n j\rangle}, \mathsf{out}_{\langle \mathsf{dec}_n i,j\rangle}, \mathsf{out}_{\langle i,\mathsf{inc}_n j\rangle}, \mathsf{out}_{\langle \mathsf{inc}_n i,j\rangle}\rangle \end{aligned}$$

where $\mathsf{dec}_n x = (x = 1 \rightarrow n,\ x-1)$ and $\mathsf{inc}_n x = (x = n \rightarrow 1,\ x+1)$. Finally, Bus is defined as the lifting of the *split*

$$\mathsf{w} = \langle \mathsf{in}_{\langle i,j\rangle} \mid i, j \in 1..n\rangle$$

The *game of life* component is then written as

$$\mathsf{GameLife} = ((\mathsf{Cell} \boxtimes \mathsf{Cell} \boxtimes \cdots \boxtimes \mathsf{Cell}) \,;\, \mathsf{Bus})\ {}^{\circlearrowleft}$$

where

$$\mathsf{Bus} = {}^{\ulcorner}\mathsf{w}^{\urcorner}$$

Note how the *hook* combinator is responsible for extending the game's behaviour to the infinite, once the component has been stimulated with an initial input.

7 Discussion

This section introduced a semantic model for software components, regarded as concrete pointed coalgebras for some Set endofunctors, and a calculus to reason about (and transform) component-based designs. Both the model and the calculus are parametric on a strong monad capturing the intended behaviour model. The approach focuses on *state-based* components with a form of *synchronous* interaction. Such assumptions, which underly popular technologies like, *e.g.*, CORBA [35], DCOM [16] or JAVABEANS [22], reflects what could be called the *object orientation* legacy. A component, in this sense, is essentially a collection of objects and, therefore, component interaction is achieved by mechanisms implementing the usual *method call* semantics.

The bicategorical setting adopted in this section seems appropriate to capture a 'two-level structure' in the component models. This is clearly in debt to previous work by R. Walters and his collaborators on models for deterministic input-driven systems [19, 18, 20]. Two other influences should be acknowledged. The first is the recent area of coalgebraic specification of object-oriented systems (see *e.g.*, [31, 17]), which has been developed with a similar motivation, although in a property-oriented, or axiomatic, framework. The other is the 'dataflow paradigm' [28] to which some of the aggregation patterns and the general idea of structured wiring can eventually be traced back.

An alternative approach to componentware is inspired by research on coordination languages [14, 30] and favors strict component decoupling in order to support a looser

inter-component dependency. Here computation and coordination are clearly separated, communication becomes *anonymous* and component interconnection is externally controlled. This model is (partially) implemented in JAVASPACES on top of JINI [27] and fundamental to a number of approaches to componentware which identify communication by generic channels as the basic interaction mechanism — see, *e.g.*, REO [1], PICCOLA [34, 26], [12, 10] or [8].

Finally a pointfree, essentially equational calculational proof style, has been used. In particular, equational proofs replace the more traditional use of coinduction (in terms of explicit construction of bisimulations). Generic proofs performed in this style are often long, even if easy to follow. In most cases their length results from the systematic recording of almost all elementary steps. On the other hand, this style has become familiar to the functional programming community, where it has been popularised under the 'Bird-Meertens formalism' heading (see *e.g.*, [2, 11] or [3]).

References

1. F. Arbab. Abstract behaviour types: a foundation model for components and their composition. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 33–70. Springer Lect. Notes Comp. Sci. (2852), 2003.
2. R. Backhouse. An exploration of the Bird-Meertens formalism. CS 8810, Groningen University, 1988.
3. 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.
4. L. S. Barbosa. *Components as Coalgebras*. PhD thesis, DI, Universidade do Minho, 2001.
5. L. S. Barbosa. Towards a Calculus of State-based Software Components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.
6. L. S. Barbosa and J. N. Oliveira. State-based components made generic. In H. Peter Gumm, editor, *CMCS'03, Elect. Notes in Theor. Comp. Sci.*, volume 82.1. Elsevier, 2003.
7. L. S. Barbosa and J. N. Oliveira. Transposing partial components: an exercise on coalgebraic refinement. *Theor. Comp. Sci.*, 365(1-2):2–22, 2006.
8. M. A. Barbosa and L. S. Barbosa. Specifying software connectors. In K. Araki and Z. Liu, editors, *1st International Colloquium on Theorectical Aspects of Computing (ICTAC'04)*, pages 53–68, Guiyang, China, September 2004. Springer Lect. Notes Comp. Sci. (3407).
9. J. Benabou. Introduction to bicategories. *Springer Lect. Notes Maths. (47)*, pages 1–77, 1967.
10. K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A Formal Model for Componentware. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press, 2000.
11. R. Bird and O. Moor. *The Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997.
12. M. Broy. Semantics of finite and infinite networks of communicating agents. *Distributed Computing*, (2), 1987.
13. A. Cruz, L. Barbosa, and J. Oliveira. From algebras to objects: Generation and composition. *Journal of Universal Computer Science*, 11(10):1580–1612, 2005.

14. D. Gelernter and N. Carrier. Coordination languages and their significance. *Communication of the ACM*, 2(35):97–107, February 1992.
15. 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.
16. R. Grimes. *Profissional DCOM Programming*. Wrox Press, 1997.
17. B. Jacobs. Objects and classes, co-algebraically. In C. Lengauer B. Freitag, C.B. Jones and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, 1996.
18. P. Katis. *Categories and Bicategories of Processes*. PhD thesis, University of Sydney, 1996.
19. P. Katis, N. Sabadini, and R. F. C. Walters. Bicategories of processes. *Journal of Pure and Applied Algebra*, 115(2):141–178, 1997.
20. 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.
21. A. Kock. Strong functors and monoidal monads. *Archiv für Mathematik*, 23:113–120, 1972.
22. V. Matena and B Stearns. *Applying Entreprise JavaBeans: Component-Based Development for the J2EE Platform*. Addison-Wesley, 2000.
23. Sun Meng and L. S. Barbosa. On refinement of generic software components. In C. Rettray, S. Maharaj, and C. Shankland, editors, *10th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 506–520, Stirling, 2004. Springer Lect. Notes Comp. Sci. (3116). Best Student Co-authored Paper Award.
24. Sun Meng and L. S. Barbosa. Components as coalgebras: The refinement dimension. *Theor. Comp. Sci.*, 351:276–294, 2005.
25. R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.
26. O. Nierstrasz and F. Achermann. A calculus for modeling software components. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 339–360. Springer Lect. Notes Comp. Sci. (2852), 2003.
27. S. Oaks and H. Wong. *Jini in a Nutshell*. O'Reilly and Associates, 2000.
28. J. N. Oliveira. *The Formal Semantics of Deterministic Dataflow Programs*. PhD thesis, Department of Computer Science, University of Manchester, February 1984.
29. J. N. Oliveira. Formal Software Development. Lecture Notes for the MSc in Computer Science, Minho University, 1992.
30. G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers — The Engineering of Large Systems*, volume 46, pages 329–400. 1998.
31. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.
32. J. Rutten. Universal coalgebra: A theory of systems. Technical report, CWI, Amsterdam, 1996.
33. 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).
34. J.-G. Schneider and O. Nierstrasz. Components, scripts, glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures - Advances and Applications*, pages 13–25. Springer-Verlag, 1999.
35. R. Siegel. *CORBA: Fundamentals and Programming*. John Wiley & Sons Inc, 1997.
36. C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
37. P. Wadler and K. Weihe. Component-based programming under different paradigms. Technical report, Dagstuhl Seminar 99081, February 1999.