State-based Components Made Generic

L. S. Barbosa and J. N. Oliveira

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

©2003 Published by Elsevier Science B. V.
and ‘black box’ diagram:

\[
\begin{cases}
\text{pop} : & 1 \rightarrow P \\
\text{top} : & 1 \rightarrow P \\
\text{push} : & P \rightarrow 1
\end{cases}
\]

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, 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 \( [\text{Stack}] : U \times I \rightarrow (U \times O + 1) \) 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 \( [\text{Stack}] : U \rightarrow (U \times O + 1)^I \) that is, as a coalgebra [24,10] of signature \( U \rightarrow T U \) involving transition ‘shape’ (functor)

\[
T = ((\text{Id} \times O) + 1)^I
\]  

(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 polymorphism [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 $B$, leading to coalgebras for the following composite functor (in $\text{Set}$):

\[ T^B = B(\text{Id} \times O)^I \] (2)

3 A strong monad is a monad $(B, \eta, \mu)$ where $B$ is a strong functor and both $\eta$ and $\mu$ are strong natural transformations [17]. $B$ being strong means there exist natural transformations $\tau^T_r : T \times - \rightarrow T(\text{Id} \times -)$ and $\tau^T_l : - \times T \rightarrow T(- \times \text{Id})$, 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 $\tau_r$, followed by $\tau_l$ maps $BI \times BJ$ to $B(I \times J)$, which can, then, be flattened to $B(I \times J)$ via $\mu$. 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 $\delta_r = \tau_{r,I,J} \circ \tau_{l,J}$. Dually, $\delta_l = \tau_{l,I,J} \circ \tau_{r,J}$. Such transformations specify how the monad distributes over product and, therefore, represent a sort of sequential composition of $B$-computations. Whenever $\delta_r$ and $\delta_l$ 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 (τr) or left (τl) 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 xl : A × (B × C) → B × (A × C), xr : A × B × C → A × C × B and m : (A × B) × (C × D) → (A × C) × (B × 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, B = Id + 1, as in the stack example above.
- Non determinism, introduced by the (finite) powerset monad, B = P.
- Ordered non determinism, based on the (finite) sequence monad, B = Id*. 
- Monoidal labelling, with B = Id × 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 ⟨M, ⊕, ⊗⟩, where both ⊕ and ⊗ are Abelian monoids and the latter distributes over the former. This gives rise to, e.g.,

- Cost components: based on BagM for M = ⟨N, +, ×⟩, 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 = ⟨[0, 1], min, ×⟩ with the additional requirement that, for each m ∈ BagM, ∑(Pπ2)m = 1. This assigns probabilities to each possible evolution of a component, introducing a (elemen-
Barbosa and Oliveira

tary) form of probabilistic non determinism.

All of the above situations correspond to known strong monads in \textbf{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 $\mathbf{Cp}$ whose objects are the interface (or observation) universes, whose arrows are seeded $\mathbb{T}^B$-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 $\mathbb{T}^B$-coalgebra we mean a pair $\langle u_p \in U_p, \pi_p : U_p \rightarrow \mathcal{B}(U_p \times O) \rangle$, where $u_p$ is the seed and the coalgebra dynamics is captured by currying a state-transition function $\alpha_p : U_p \times I \rightarrow \mathcal{B}(U_p \times O)$. Then the construction of bicategory $\mathbf{Cp}$ defines, foreach pair $\langle I, O \rangle$ of objects, a hom-category $\mathbf{Cp}(I, O)$, whose arrows $h : \langle u_p, \pi_p \rangle \rightarrow \langle u_q, \pi_q \rangle$ satisfy the following comorphism and seed preservation conditions:

$$\pi_q \cdot h = \mathbb{T}^B h \cdot \pi_p \quad \text{and} \quad h u_p = u_q \quad \text{(3)}$$

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

$$;_{I,K,O} : \mathbf{Cp}(I, K) \times \mathbf{Cp}(K, O) \rightarrow \mathbf{Cp}(I, O)$$

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

$$p \cdot q = \langle \langle u_p, u_q \rangle \in U_p \times U_q, \pi_{pq} \rangle$$

where $\alpha_{pq} : U_p \times U_q \times I \rightarrow \mathcal{B}(U_p \times U_q \times O)$ is detailed as follows

$$\alpha_{pq} = U_p \times U_q \times I \xrightarrow{x_r} U_p \times I \times U_q \xrightarrow{\alpha_p \times \text{id}} \mathcal{B}(U_p \times K) \times U_q \xrightarrow{\tau_r} \mathcal{B}(U_p \times K) \times U_q \xrightarrow{\mathbb{B}(\alpha_p \times \text{id})} \mathbb{B} \mathcal{B}(U_p \times U_q \times O) \xrightarrow{\mu} \mathbb{B} \mathcal{B}(U_p \times U_q \times O)$$
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
\[
\text{copy}_K : 1 \longrightarrow \text{Cp}(K, K)
\]
whose action on objects is the constant component \( * \in 1, \pi_{\text{copy}_K} \), where
\( a_{\text{copy}_K} = \eta_{1 \times K} \). Slightly abusing on notation, this will be also referred to as
\( \text{copy}_K \). Similarly, the action on morphisms is the constant comorphism \( \text{id}_1 \).

All in all, the fact that, for each strong monad \( \mathcal{B} \), components form a
bicategory \(^4 \) amounts not only to a standard definition of the two basic combi-

\[\begin{align*}
\text{copy}_I ; p & \sim p \sim p; \text{copy}_O \\
(p; q) ; r & \sim p; (q; r)
\end{align*}\]

The dynamics of a component specification is essentially ‘one step’: it
describes immediate reactions to possible state/input configurations. Its tem-
poral extension becomes the component’s \textit{behaviour}. Formally, the behaviour
\( \llbracket p \rrbracket \) of a component \( p \) is computed by applying the induced \textit{anamorphism} to
the seed-value of \( p \). I.e., \( \llbracket p \rrbracket = \llbracket \pi_p \rrbracket u_p \)

Behaviours organise themselves in a category \( \mathcal{Bh} \), or, simply, \( \mathcal{Bh} \), 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 \( \mathcal{B}((\text{id} \times O) \rangle)^4 \). To define composition in
\( \mathcal{Bh} \), first note that the definition of \( \pi_{pq} \) above actually introduces an operator
\( ; \) between coalgebras: \( \pi_{pq} \) could actually have been written as \( \pi_p ; \pi_q \).
Thus, we may define composition in \( \mathcal{Bh} \) by a family of combinators, for each
\( I, K \) and \( O \), \( ;_{I,K,O}^{\mathcal{Bh}} : \mathcal{Bh}(I, K) \times \mathcal{Bh}(K, O) \longrightarrow \mathcal{Bh}(I, O) \), such that
\[
;_{I,K,O}^{\mathcal{Bh}} \llbracket \omega_{I,K} ; \omega_{K,O} \rrbracket
\]
On the other hand, identities are given by
\[
\text{copy}_K^{\mathcal{Bh}} : 1 \longrightarrow \mathcal{Bh}(K, K) \quad \text{and} \quad \text{copy}_K^{\mathcal{Bh}} = \llbracket \pi_{\text{copy}_K} \rrbracket * 
\]
\( i.e., \) the behaviour of component \( \text{copy}_K \), for each \( K \).

It should be observed that the structure of \( \mathcal{Bh} \) mirrors whatever structure \( \mathcal{Cp} \) 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 \( \mathcal{Bh} \) as constructed by quotienting \( \mathcal{Cp} \) by the greatest \( \mathcal{T}^{\mathcal{B}} \)-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 \( \mathcal{C}_B \) become universal constructions in \( \mathcal{B}_h \), for some particular instances of \( B \). This also explains why properties holding in \( \mathcal{C}_B \) 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 \( \mathcal{B}_h \), as

\[
\text{copy}_I \cdot b = b = b ; \text{copy}_O \quad \text{and} \quad (b ; c) ; d = b ; (c ; d)
\]

First, however, we have to check that \( \mathcal{B}_h \) is indeed a category. Let \( b : I \rightarrow O \) be a behaviour. Then,

\[
b ; \text{copy}_O = \langle [\omega_{I,O} ; \text{copy}_O](b,*), 1, \text{copy}_O \rangle = \langle [\omega_{I,O}]b, b \rangle
\]

A similar calculation will establish \( \text{copy}_I \cdot b = b \). On the other hand, for suitably typed behaviours \( b, c \) and \( d \),

\[
(b;c)d = \langle [\omega_{I,K} \cdot \omega_{K,L} \cdot \omega_{L,O}](b,c,d), 1, \text{copy}_O \rangle = \langle [\omega_{I,K} ; (\omega_{K,L} ; \omega_{L,O})](b,c,d), 1, \text{copy}_O \rangle = b ; (c ; d)
\]

So \( \mathcal{B}_h \) 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 \( \mathcal{C}_B \) to \( \mathcal{B}_h \), which follows the same calculational style (see [3]).

## 4 A Glimpse at the Component Calculus

This section investigates the structure of \( \mathcal{C}_B \) by introducing an algebra of \( T^B \)-components which is parametric on the behaviour model. This structure lifts naturally to \( \mathcal{B}_h_B \) 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 \( f : A \rightarrow B \) is represented in \( \mathcal{C}_B \) as

\[
\langle \ast \in 1, \text{id} \times f \rangle
\]

i.e., as a coalgebra over \( 1 \) whose action is given by the currying of

\[
\text{ar}_f = 1 \times A \xrightarrow{\text{id} \times f} 1 \times B \xrightarrow{\eta(1 \times B)} B(1 \times B)
\]

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

\[
\langle \ast \in 1, \text{id} \times f \rangle \sim \langle \ast \in 1, \text{id} \times g \rangle \quad \text{and} \quad \langle \ast \in 1, \text{id} \times f \rangle \sim \langle \ast \in 1, \text{id} \times g \rangle
\]

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

\[
\begin{array}{ccc}
I & \xrightarrow{p} & O \\
\downarrow{?_I} & & \downarrow{?_O} \\
\emptyset & \xleftarrow{\gamma} & \emptyset
\end{array}
\]

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

\[
\gamma ?_I \gamma ; p \sim \gamma ?_O \gamma \tag{8}
\]

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

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

\[
\begin{array}{ccc}
I & \xrightarrow{p} & O \\
\downarrow{?_I} & & \downarrow{?_O} \\
1 & \xleftarrow{} & 1
\end{array}
\]

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

4.2 Wrapping

The pre- and post-composition of a component with $\mathbf{Cp}$-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 $p : I \to O$ be a component and consider functions $f : I' \to I$ and $g : O \to O'$. By $p[f, g]$ we will denote component $p$ wrapped by $f$ and $g$. This has type $I' \to O'$ and is defined by input pre-composition with $f$ and output post-composition with $g$. Formally, the wrapping combinator is a functor

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

which is the identity on morphisms and maps a component $\langle u_p, \overline{a}_p \rangle$ into
\[(u_p, \overline{a_p(f,g)}),\] where
\[a_p(f,g) = U_p \times I' \xrightarrow{\text{id} \times f} U_p \times I \xrightarrow{a_p} B(U_p \times O) \xrightarrow{B(\text{id} \times g)} B(U_p \times O')\]

The following properties about wrapping hold:

\[p[f,g] \sim \overline{f} \; p \; \overline{g} (9)\]
\[(p[f,g])[f',g'] \sim p[f \cdot f', g' \cdot g] (10)\]

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

\[\text{inert}_A = \overline{?}_A\]

In particular, we define the nil component \(\text{nil} = \text{inert}_\emptyset = \overline{?}_\emptyset = \overline{\text{id} \emptyset}\) typed as \(\text{nil} : \emptyset \rightarrow \emptyset\). Note that any component \(p : I \rightarrow O\) can be made inert by wrapping. For example, \(p[?_I, !_O] \sim \text{inert}_1\). A somewhat dual role is played by component \(\text{idle} = \overline{\text{id} 1}\). Note that \(\text{idle} : 1 \rightarrow 1\) 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 \(C_p\).

The first composition pattern to be considered is external choice. Let \(p : I \rightarrow O\) and \(q : J \rightarrow R\) 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 \(\boxplus : \mathbf{C}_p \times\)
\[ \Box \colon \mathbb{C}_p(I, O) \times \mathbb{C}_p(J, R) \to \mathbb{C}_p(I + J, O + R) \]

yielding

\[ p \Box q = \langle \langle u_p, u_q \rangle \in U_p \times U_q, \pi_p \Box \pi_q \rangle \]

\[
\begin{align*}
\alpha_{p \Box q} &= U_p \times U_q \times (I + J) \xrightarrow{\text{dr}} U_p \times I \times U_q + U_p \times (U_q \times J) \\
& \xrightarrow{\tau_r + \tau_l} B (U_p \times O) \times U_q + U_p \times B (U_q \times R) \\
& \xrightarrow{\Box r + Br} B (U_p \times U_q \times O) + B (U_p \times U_q \times R) \\
& \xrightarrow{[B (\text{id} \times \text{id})], B (\text{id} \times \text{id})} B (U_p \times U_q \times (O + R))
\end{align*}
\]

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 \( \Box \) is a lax functor in \( \mathbb{C}_p \), for components \( p, q, p' \) and \( q' \), and functions \( f, g \):

\[ (p \Box p') ; (q ; q') \sim (p ; q) \Box (p' ; q') \quad (12) \]

\[ \text{copy}_K \Box \text{copy}_{K'} \sim \text{copy}_K \Box \text{copy}_{K'} \quad (13) \]

\[ \Gamma f \Box \Gamma g \sim \Gamma f + \Gamma g \quad (14) \]

Moreover, up to isomorphic wiring, \( \Box \) is a symmetric tensor product in each hom-category, with \( \text{nil} \) as unit, \( i.e. \),

\[ (p \Box q) \Box r \sim (p \Box (q \Box r))[a_+, a_+] \quad (15) \]

\[ \text{nil} \Box p \sim p[r_+, r_+] \quad \text{and} \quad p \Box \text{nil} \sim p[l_+, l_+] \quad (16) \]

\[ p \Box q \sim (q \Box p)[s_+, s_+] \quad (17) \]

Laws (15) to (17) can be alternatively stated as providing evidence that the canonical \( \text{Set} \) isomorphisms \( a_+, r_+, l_+ \) and \( s_+ \), once lifted to \( \mathbb{C}_p \), 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 \( \mathbb{C}_p \) to the \textit{either} construction in \( \text{Set} \). The answer is partly positive. Let \( p : I \to O \) and \( q : J \to O \) be two components sharing a common output type \( O \), and define

\[ [p, q] = (p \Box q) ; \Gamma \nabla \]

where \( \nabla = [\text{id}, \text{id}] \). It can be shown that that the following diagram commutes up to bisimulation,
even though \([p, q]\) is not the unique arrow making the diagram commute. This means that the choice combinator, \(⊞\), 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 \(⊞\), 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

\[
\begin{align*}
[\Gamma_{t_1}, \Gamma_{t_2}] & \sim \text{copy}_{I+J} \\
(p \boxplus q) ; [p', q'] & \sim [p ; p', q ; q']
\end{align*}
\]

are easy to prove. For example,

\[
\begin{align*}
(p \boxplus q) ; [p', q'] & \sim \{ \text{definition of either in } Cp \} \\
(p \boxplus q) ; ((p' \boxplus q') ; \Gamma_{\nabla}) & \sim \{ ; \text{associative (5)} \} \\
((p \boxplus q) ; (p' \boxplus q')) ; \Gamma_{\nabla} & \sim \{ \boxplus \text{functor (12)} \} \\
((p ; p') \boxplus (q ; q')) ; \Gamma_{\nabla} & \sim \{ \text{definition of either in } Cp \} \\
[p ; p', q ; q'] &
\end{align*}
\]

As expected, the \(⊞\) combinator can be written in terms of an either construction on components. In fact, for \(p : I \rightarrow O\) and \(q : J \rightarrow R\), we obtain

\[
\begin{align*}
p \boxplus q & \sim [p ; \Gamma_{t_1} ; p ; \Gamma_{t_2}]
\end{align*}
\]

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

\[
\begin{align*}
\Gamma_{t_1} \& (p \boxplus q) & \sim p ; \Gamma_{t_1} \quad \text{and} \quad \Gamma_{t_2} ; (p \boxplus q) & \sim q ; \Gamma_{t_2}
\end{align*}
\]

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

\[
\begin{align*}
p ; \Gamma_{t_1} & \sim \Gamma_{t_1} \& (p \boxplus p)
\end{align*}
\]
The dual situation, involving parallel aggregation \(\boxdot\) and a *split* construction, is studied in [3], but the results are a bit different. It turns out that a *cancellation* result — \(\langle p,q \rangle; \pi_1 \sim p\) — is only valid for a monad \(B\) which excludes the possibility of *failure* (e.g., the non-empty powerset). On the other hand, diagonal \(\triangle\) 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 \(\boxdot\) 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 \(\text{Set}\). 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 \(\gamma_Z\): \(Cp(I+Z,O+Z) \rightarrow Cp(I+Z,O+Z)\). This combinator is the identity on arrows and maps each component \(p: I+Z \rightarrow O+Z\) to \(p^{\gamma_Z}: I+Z \rightarrow O+Z\) given by

\[
p^{\gamma_Z} = \langle u_p \in U_p, \overline{a_p} \rangle
\]

where

\[
a_p^{\gamma_Z} = \begin{array}{c}
U_p \times (I+Z) \\
\xrightarrow{a_p} B(U_p \times (O+Z)) \\
\xrightarrow{B((id \times \iota_1 + id \times \iota_2) \cdot \delta \eta)} B(U_p \times (O+Z) + U_p \times (I+Z)) \\
\xrightarrow{B(\eta + a_p)} B(B(U_p \times (O+Z)) + B(U_p \times (O+Z))) \\
\xrightarrow{\mu \cdot B \nu} B(U_p \times (O+Z))
\end{array}
\]

### 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, \(\text{Vdm}\)-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:

$$
\begin{align*}
\text{Folder} & : \mathbf{1} + \text{rd} : \mathbf{1} + \text{tl} : \mathbf{1} + \text{in} : P \\
\text{rd} & : P + \{\text{tr}, \text{tl}, \text{in}\} : \mathbf{1}
\end{align*}
$$

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 $\nabla_P$ morphism. On the other hand, access to the top button on the left stack is removed by $\iota_2$. 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 $!_{P+1}$. So we define:

$$
\begin{align*}
\text{RightS} & = \text{Stack}[\text{id} + \nabla, \text{id}] : \mathbf{1} + \mathbf{1} + (P + P) \rightarrow P + P + 1 \\
\text{LeftS} & = \text{Stack}[\iota_2 + \text{id}, (\text{id} + !_{P+1}) \cdot \text{a}_+] : \mathbf{1} + P \rightarrow P + 1
\end{align*}
$$

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

$$
\text{LeftS} \boxplus \text{RightS} : \mathbf{1} + P + (\mathbf{1} + \mathbf{1} + (P + P)) \rightarrow P + \mathbf{1} + (P + P + 1)
$$

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

$$
\text{AlmostFolder} = ((\text{LeftS} \boxplus \text{RightS})[\text{wi}, \text{wo}])^{\eta_{P+P}}
$$

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

$$
\begin{align*}
\text{wi} & = \left[\left[\left[\iota_{11}, \iota_{112}, \iota_{212}, \iota_{222}\right], \iota_{21}, \iota_{122}\right]\right] \\
\text{wo} & = \left[\left[\iota_{12}, \iota_{111}, \left[\iota_{211}, \iota_{22}, \iota_{21}\right]\right]\right]
\end{align*}
$$
In a diagram:

\[
(1 + 1 + 1 + P) + (P + P) \]

\[
(1 + P + 1) + (P + P) \]

Finally, to conform AlmostFolder to the Folder interface, we restrict the feedback input — by pre-composing with \( fi = \iota_1 \) — and collapse both the trivial output and the feedback one to 1, by post-composing with \( fo = \left[ [t_2, t_1], t_2, t_2 \cdot P + P \right] \).

Therefore, we complete the exercise by defining

\[
\text{Folder} = \text{AlmostFolder}[fi, fo] \]

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 preprocessor similar to POLYP [11] for the behaviour monads considered in [3] is currently being considered.

References


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 □ 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

\[ [p,q]_{[t_1,\nabla]} \sim p \text{ and } [p,q]_{[t_2,\nabla]} \sim q \]

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,

\[ \text{B}(\pi_1 \times \nabla) \cdot [\text{B}(\text{id} \times t_1), \text{B}(\text{id} \times t_2)] \cdot (\text{B}x + \text{Ba}^0) \cdot (\tau_r + \tau_l) \cdot (a_p \times \text{id} + \text{id} \times a_q) \]

\[ \cdot (\text{Dr} + a) \cdot (\text{id} \times t_1) \]

= \{ \text{law: } t_1 = \text{Dr} \cdot (\text{id} \times t_1) \text{ (cf., [3])} \}

\[ \text{B}(\pi_1 \times \nabla) \cdot (\tau_r + \tau_l) \cdot (a_p \times \text{id} + \text{id} \times a_q) \]

\[ \cdot (\text{Dr} + a) \cdot t_1 \]

= \{ \text{ + absorption and cancellation } \}

\[ \text{B}(\pi_1 \times \nabla) \cdot \text{B}(\text{id} \times t_1) \cdot \text{B}x \cdot \tau_r \cdot a_p \times \text{id} \cdot x_r \]

= \{ \text{ routine: } \nabla \cdot t_1 = \text{id} \}

\[ \text{B}(\pi_1 \times \nabla) \cdot \text{B}x \cdot \tau_r \cdot a_p \times \text{id} \cdot x_r \]

= \{ \text{ routine: } (\pi_1 \times \text{id}) \cdot x_r = \pi_1 \}

\[ \text{B}\pi_1 \cdot \tau_r \cdot a_p \times \text{id} \cdot x_r \]

= \{ \text{ law: } \text{B}\pi_1 \cdot \tau_r = \pi_1 \text{ (cf., [3])} \}

\[ \text{B}\pi_1 \cdot a_p \times \text{id} \cdot x_r \]

= \{ \text{ + definition and cancellation } \}

\[ a_p \cdot \tau_1 \cdot x_r \]

= \{ \text{ routine: } (\pi_1 \times \text{id}) \cdot x_r = \pi_1 \text{ and } x_r = x_r^0 \}

\[ a_p \cdot (\pi_1 \times \text{id}) \]

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 \( \nabla \) does not keep its naturality when lifted to \( \text{Cp} \). In fact, a counterexample can be found even in the simple setting of deterministic components (i.e., with \( \text{B} = \text{id} \)). Let \( p = \langle 0 \in \mathbb{N}, t_q \rangle : \mathbb{N} \rightarrow \mathbb{N} \) 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 \( \mathbb{N} + \mathbb{N} \)): \( s = \langle t_5, t_3, t_4, \ldots \rangle \). The reaction to \( s \) of \( \nabla \); \( p \) is \( \langle 5, 3, 9, \ldots \rangle \) while \( p \); \( \nabla \), resorting only to one copy of \( p \), produces \( \langle 5, 8, 12, \ldots \rangle \). \( \Box \)