Generic Process Algebra: A Programming Challenge

Paula R. Ribeiro
Marco Antonio Barbosa
Luís Soares Barbosa
(Dep. Informática, Universidade do Minho,Portugal
{priobom@gmail.pt,lsb}@di.uminho.pt)

Abstract: Emerging interaction paradigms, such as service-oriented computing, and new technological challenges, such as exogenous component coordination, suggest new roles and application areas for process algebras. This, however, entails the need for more generic and adaptable approaches to their design. For example, some applications may require similar programming constructs coexisting with different interaction disciplines. In such a context, this paper pursues a research programme on a coinductive rephrasal of classic process algebra, proposing a clear separation between structural aspects and interaction disciplines. A particular emphasis is put on the study of interruption combinators defined by natural co-recursion. The paper also illustrates the verification of their properties in an equational and pointfree reasoning style as well as their direct encoding in Haskell.

Key Words: Process Algebra, coinduction, coalgebra
Category: D.3.1, F.3.1

1 Introduction

In [Barbosa, 2001] a coinductive approach to the *design* of process algebras was outlined and compared, wrt expressive and calculational power, with the classical frameworks based on the operational semantics. The goal was to apply to this area of computing the same reasoning principles and calculational style one gets used to in the *dual* world of functional programming with algebraic datatypes. Actually, the role of universals constructions, such as *initial* algebras and *final* coalgebras, is fundamental to a whole discipline of algorithm derivation and transformation, which can be traced back to the so-called *Bird-Meertens formalism* [Bird and Meertens, 1987]. Dually, our research programme regards *processes* as inhabitants of coinductive types, *i.e.*, final coalgebras for the powerset functor $\mathcal{P}(Act \times \mathsf{Id})$, where Act denotes a set of *action* identifiers. Finally, process *combinators* are defined as *anamorphisms* [Meijer et al., 1991], *i.e.*, by coinductive extension. Note that, if coalgebras for a functor T can be regarded as generalisations of transition systems of shape T, their behavioural patterns are revealed by the successive observations allowed by the signature of observers recorded in T. Then, just as initial algebras are canonnically defined over the terms generated by successive application of constructors, such 'pure' observed behaviours form the state spaces of final coalgebras. It comes with no surprise

that *bisimulation* coincides with equality in such coalgebras. Therefore our approach has the attraction of replacing proofs by bisimulation, which as in *e.g.*, [Milner, 1989], involves the explicit construction of such a relation, by *equational reasoning*.

Technically, our approach amounts to the systematic use of the universal property which characterizes *anamorphisms*. Recall that, for a functor T and an arbitrary coalgebra $\langle U, p : \mathsf{T}\,U \longleftarrow U \rangle$, an *anamorphism* is the *unique* morphism to the final coalgebra $\mathsf{out}_\mathsf{T} : \mathsf{T}\ \nu_\mathsf{T} \longleftarrow \nu_\mathsf{T}$. Written, in the tradition of [Meijer et al., 1991], as $[\!(p)\!]_\mathsf{T}$ or, simply, $[\!(p)\!]$, an anamorphism satisfies the following universal property:

$$k = [\!(p)\!]_\mathsf{T} \quad \Leftrightarrow \quad \mathsf{out}_\mathsf{T} \cdot k = \mathsf{T}\ k \cdot p \tag{1}$$

from which the following *cancellation*, *reflection* and *fusion* laws are easily derived:

$$\mathsf{out}_\mathsf{T} \cdot [\!(p)\!] \ = \ \mathsf{T}\ [\!(p)\!] \cdot p \tag{2}$$

$$[\!(\mathsf{out}_\mathsf{T})\!] \ = \ \mathsf{id}_{\nu_\mathsf{T}} \tag{3}$$

$$[\!(p)\!] \cdot h \ = \ [\!(q)\!] \quad \text{if} \quad p \cdot h \ = \ \mathsf{T}\ h \cdot q \tag{4}$$

The *existence* assertion underlying (1) (corresponding to the left to right implicants) provides a *definition* principle for (circular) functions to the final coalgebra which amounts to equip their source with a coalgebraic structure specifying the *next-step* dynamics. We call such a coalgebra the *gene* of the definition: it carries the 'genetic inheritance' of the function. Then the anamorphism gives the rest. The *uniqueness* part, underlying right to left implication in (1), on the other hand, offers *coinduction* as a *proof* principle.

Definition and proof by coinduction forms the base of the approach to process calculi design to which this paper aims to contribute. More specifically it reports on two topics:

- The definition of *interruption* combinators resorting to natural corecursion encoded as *apomorphisms* [Vene and Uustalu, 1997]. This coinductive scheme generalises anamorphisms in the sense that the *gene* coalgebra can choose on returning the *next-step* value or else a complete behaviour. In particular we prove, in section 4, a conditional fusion law for apomorphisms.

- The development of a HASKELL library for prototyping process algebras directly based on the coinductive definitions. As a basic design decision, which may justify the qualificative *generic* in our title, structural aspects of process' combinators and interaction disciplines are specified separately.

The paper is organised as follows. Section 2 recalls the basic principles of our approach as reported elsewhere. The definitional and proof style are, however, illustrated with the study of a new combinator whose role is similar to that of *hiding* in CSP [Hoare, 1985]. Then, section 3 reports on functional prototyping of process algebra in HASKELL. Section 4 studies the interruption combinators mentioned above. Finally, section 5 concludes and points out a few issues for future research. Basic functional notation and laws is collected in the Appendix. The reader is referred to [Bird and Moor, 1997] for details.

2 Process Calculi Design

2.1 Combinators

In [Barbosa, 2001] processes are regarded as inhabitants of the final coalgebra $\omega : \mathcal{P}(Act \times \nu) \longleftarrow \nu$, where $\mathcal{P}$ is the finite powerset functor. The carrier of ω is the set of possibly infinite labelled trees, finitely branched and quotiented by the greatest bisimulation [Aczel, 1993], on top of which process combinators are defined. For example, *dynamic* combinators, which are 'consumed' on action occurrence, are directly defined as in

$$\begin{array}{rl} \textit{inaction} & \omega \cdot \mathsf{nil} \;=\; \underline{\emptyset} \\ \textit{prefix} & \omega \cdot a. \;=\; \mathsf{sing} \cdot \mathsf{label}_a \\ \textit{choice} & \omega \cdot + \;=\; \cup \cdot (\omega \times \omega) \end{array}$$

where $\mathsf{sing} = \lambda x.\ \ \{x\}$ and $\mathsf{label}_a = \lambda x.\ \ \langle a, x\rangle$. Recursive combinators, on the other hand, are defined as anamorphisms. A typical example is *interleaving* $|||\ :\ \nu \longleftarrow \nu \times \nu$ which represents an interaction-free form of parallel composition. The following definition captures the intuition that the observations over the interleaving of two processes correspond to all possible interleavings of observations of their arguments. Thus, $||| \;=\; [\![(\alpha_{|||})]\!]$, where [1]

$$\alpha_{|||} \;=\; \nu \times \nu \xrightarrow{\ \Delta\ } (\nu \times \nu) \times (\nu \times \nu) \xrightarrow{(\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega)} (\mathcal{P}(Act \times \nu) \times \nu) \times (\nu \times \mathcal{P}(Act \times \nu))$$
$$\xrightarrow{\tau_r \times \tau_l} \mathcal{P}(Act \times (\nu \times \nu)) \times \mathcal{P}(Act \times (\nu \times \nu)) \xrightarrow{\ \cup\ } \mathcal{P}(Act \times (\nu \times \nu))$$

2.2 Interaction

The *synchronous product* models the simultaneous execution of two processes, which, in each step, interact through the actions they realize. Let us, for the

[1] Morphisms $\tau_r : \mathcal{P}(Act \times (X \times C)) \longleftarrow \mathcal{P}(Act \times X) \times C$ and $\tau_l : \mathcal{P}(Act \times (C \times X)) \longleftarrow C \times \mathcal{P}(Act \times X)$ stand for, respectively, the right and left *strength* associated to functor $\mathcal{P}(Act \times \mathsf{Id})$.

moment, represent such interaction by a function $\theta : Act \times Act \longleftarrow Act$. Formally, $\otimes = [\![\alpha_{\otimes}]\!]$ where

$$\alpha_{\otimes} = \nu \times \nu \xrightarrow{(\omega \times \omega)} \mathcal{P}(Act \times \nu) \times \mathcal{P}(Act \times \nu) \xrightarrow{\mathsf{sel} \cdot \delta_r} \mathcal{P}(Act \times (\nu \times \nu))$$

where sel filters out all synchronisation failures. and δ_r is given by

$$\delta_r \langle c_1, c_2 \rangle = \{\langle a' \, \theta \, a, \langle p, p' \rangle \rangle | \, \langle a, p \rangle \in c_1 \wedge \langle a', p' \rangle \in c_2\}$$

But what is θ? This operation defined over Act what we call an *interaction structure*: *i.e.*, an Abelian positive monoid $\langle Act; \theta, 1 \rangle$ with a zero element 0. It is assumed that neither 0 nor 1 belong to the set of elementary actions. The intuition is that θ determines the interaction discipline whereas 0 represents the absence of interaction: for all $a \in Act$,$a\theta 0 = 0$. On the other hand, a positive monoid entails $a\theta a' = 1$ iff $a = a' = 1$. The role of 1, often regarded as an *idle* action, is essentially technical.

As a matter of fact by parameterizing a basic calculus by an interaction structure, one becomes able to design quite a number of different, application-oriented, process combinators. For example, CCS assumes a set L of labels with an involutive operation, represented by an horizontal bar as in $\overline{a}$. Any two actions a and $\overline{a}$ are called complementary and a special action $\tau \notin L$ is introduced to represent the result of a synchronisation between a pair of complementary actions. Therefore, the result of θ is τ whenever applied to a pair of complementary actions and 0 in all other cases, except, obviously, if one of the arguments is 1. In CSP, on the other hand, $a\theta a = a$ for all action $a \in Act$. Yet other examples emerge from recent uses of process algebra, namely in the area of component orchestration. For example, the second author has pointed out, in his forthcoming PhD thesis, situations in which the same process expression has to be read with different underlying interaction structures. Typically a *glass-box view* of a particular architectural configuration (*i.e.*, a 'glued' set of components and software connectors) will call for a *co-occurrence* interaction: θ is defined as $a\theta b = \langle a, b \rangle$, for all $a, b \in Act$ different from 0 and 1. For the *black-box view*, however, actions are taken as sets of labels, and θ defined as set intersection.

Synchronous product depends in a crucial way on the interaction structure adopted. For example its commutativity depends only on the commutativity of the underlying θ. Such is also the case of the standard *parallel composition* which combines the effects of both $|||$ and $\otimes$. Note, however, that such a combination is performed at the *genes* level: $| = [\![\alpha_{|}]\!]$, where

$$\alpha_{|} = \nu \times \nu \xrightarrow{\Delta} (\nu \times \nu) \times (\nu \times \nu) \xrightarrow{(\alpha_{|||} \times \alpha_{\otimes})}$$
$$\mathcal{P}(Act \times (\nu \times \nu)) \times \mathcal{P}(Act \times (\nu \times \nu)) \xrightarrow{\cup} \mathcal{P}(Act \times (\nu \times \nu))$$

2.3 Hiding

We shall now illustrate the rationale underlying our approach by considering the definition of a new combinator $\backslash_k$ whose aim is to make internal all occurrences of a specific action k. Then, $\backslash_k = [\![(\alpha_k)]\!]$, where $\alpha_k = \mathcal{P}(\mathsf{sub}_k \times \mathsf{id}) \cdot \omega$ and $\mathsf{sub}_k \triangleq (=_k) \rightarrow \tau, \mathsf{id}$, τ standing for a representation of an *internal* action. Let us now discuss how this combinator interacts with *interleaving*. This provides a first example of a coinductive proof by calculation, to be opposed to the more classic proof by bisimulation.

Lemma 1

$$\backslash_k \cdot ||| \;=\; ||| \cdot (\backslash_k \times \backslash_k) \tag{5}$$

Proof. Note that equation (5) does not allow a direct application of the fusion law. Since ω is an isomorphism, however, we may rewrite it as

$$\omega \cdot \backslash_k \cdot ||| = \omega \cdot ||| \cdot (\backslash_k \times \backslash_k) \tag{6}$$

which can be further simplified in terms of the corresponding genes, because both $|||$ and $\backslash_k$ were defined by coinduction. Consider first the left hand side of (6).

$$\begin{array}{ll}
& \omega \cdot \backslash_k \cdot ||| \\
= & \quad \{ \text{ definition of } \omega \cdot \backslash_k, \text{ cancellation} \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \alpha_k \cdot ||| \\
= & \quad \{ \text{ definition of } \alpha_k \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \mathcal{P}(\mathsf{sub}_k \times \mathsf{id}) \cdot \omega \cdot ||| \\
= & \quad \{ \; ||| \text{ is a morphism} \; \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \mathcal{P}(\mathsf{sub}_k \times \mathsf{id}) \cdot \mathcal{P}(\mathsf{id} \times |||) \cdot \alpha_{|||} \\
= & \quad \{ \text{ functors and definition of } \alpha_{|||} \; \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \mathcal{P}(\mathsf{id} \times |||) \cdot \mathcal{P}(\mathsf{sub}_k \times (\mathsf{id} \times \mathsf{id})) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot (\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega) \cdot \Delta \\
= & \quad \{ \cup, \tau_r \text{ and } \tau_l \text{ are natural } i.e. \tau_r \cdot (\mathsf{B}f \times g) = \mathsf{B}(f \times g) \cdot \tau_r \text{ e } \tau_l \cdot (f \times \mathsf{B}g) = \mathsf{B}(f \times g) \cdot \tau_l \text{ for } \mathsf{B} = \mathcal{P}(Act \times \mathsf{Id}) \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \cup \cdot (\tau_l \times \tau_r) \cdot (\mathcal{P}(\mathsf{sub}_k \times \mathsf{id}) \cdot \omega \times \mathsf{id}) \times (\mathsf{id} \times \mathcal{P}(\mathsf{sub}_k \times \mathsf{id}) \cdot \omega) \cdot \Delta \\
= & \quad \{ \text{ definition of } \alpha_k \} \\
& \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \cup \cdot (\tau_l \times \tau_r) \cdot ((\alpha_k \times \mathsf{id}) \times (\mathsf{id} \times \alpha_k)) \cdot \Delta
\end{array}$$

Consider, now, the right hand side of the same equation:

$$\begin{array}{ll}
& \omega \cdot ||| \cdot (\backslash_k \times \backslash_k) \\
= & \quad \{ \; ||| \text{ is morphism} \; \} \\
& \mathcal{P}(\mathsf{id} \times |||) \cdot \alpha_{|||} \cdot (\backslash_k \times \backslash_k) \\
= & \quad \{ \text{ defintion of } \alpha_{|||} \} \\
& \mathcal{P}(\mathsf{id} \times |||) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot ((\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega)) \cdot \Delta \cdot (\backslash_k \times \backslash_k) \\
= & \quad \{ \; \Delta \text{ is natural, functors} \; \}
\end{array}$$

$$
\begin{array}{ll}
& \mathcal{P}(\mathsf{id} \times |||) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot ((\omega \cdot \backslash_k \times \backslash_k) \times (\backslash_k \times \omega \cdot \backslash_k)) \cdot \Delta \\
= & \quad \{ \ \backslash_k \text{ is morphism} \ \} \\
& \mathcal{P}(\mathsf{id} \times |||) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot ((\mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \alpha_k \times \backslash_k) \times (\backslash_k \times \mathcal{P}(\mathsf{id} \times \backslash_k) \cdot \alpha_k)) \cdot \Delta \\
= & \quad \{ \text{ functors, } \tau_r \text{ and } \tau_l \text{ are natural} \ \} \\
& \mathcal{P}(\mathsf{id} \times |||) \cdot \cup \cdot \mathcal{P}(\mathsf{id} \times (\backslash_k \times \backslash_k)) \times \mathcal{P}(\mathsf{id} \times (\backslash_k \times \backslash_k)) \cdot (\tau_r \times \tau_l) \\
& \cdot ((\alpha_k \times \mathsf{id}) \times (\mathsf{id} \times \alpha_k)) \cdot \Delta \\
= & \quad \{ \ \cup \text{ is natural} \ \} \\
& \mathcal{P}(\mathsf{id} \times (||| \cdot (\backslash_k \times \backslash_k))) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot ((\alpha_k \times \mathsf{id}) \times (\mathsf{id} \times \alpha_k)) \cdot \Delta
\end{array}
$$

The simplification of both sides of equation (6) did not lead to the same expression. Actually, what we have concluded is that

$$\omega \cdot ||| \cdot (\backslash_k \times \backslash_k) = \mathcal{P}(\mathsf{id} \times (||| \cdot (\backslash_k \times \backslash_k))) \cdot \gamma$$

and

$$\omega \cdot \backslash_k \cdot ||| = \mathcal{P}(\mathsf{id} \times (\backslash_k \cdot |||)) \cdot \gamma$$

for coalgebra

$$\gamma = \cup \cdot (\tau_r \times \tau_l) \cdot ((\alpha_k \times \mathsf{id}) \times (\mathsf{id} \times \alpha_k)) \cdot \Delta$$

This means that both $||| \cdot (\backslash_k \times \backslash_k)$ and $\backslash_k \cdot |||$ are morphisms between γ and the final coalgebra ω. As there can only be one such morphisms we conclude they are equal.

□

We have chosen this example because this sort of proof is quite common in the calculus. The strategy is as follows: once a direct application of fusion is not possible, the aim becomes to show that both forms of composition of the two combinators can be defined as an anamorphism for a common gene coalgebra γ. Clearly, by the universal property, they must coincide. An important issue is the fact that γ was not postulated from the outset, but *inferred* from the calculations process.

3 Functional Prototyping

One advantage of this approach to process algebra design is the fact that it allows an almost direct translation for a functional programming language like Haskell. This section highlights a few issues in the construction of a Haskell library for process algebra prototyping. Our starting point is the definition of the powerset functor `Pr` (assuming an implementation of sets as lists) and the definition of the semantic universe of processes as the coinductive type `Proc a`, as follows,

```
type Proc a = Nu (Pr a)
data Pr a x =  C [(a, x)]  deriving Show
instance Functor (Pr a)
    where fmap f (C s) = C (map (id >< f) s)

obsProc :: Pr a x -> [(a, x)]
obsProc p = f    where  (C f) = p

newtype Nu f = Fin (f (Nu f))
unFin :: Nu f -> f (Nu f)
unFin (Fin x) = x
```

The second step is the definition of the *interaction structure* as an *inductive* type, parametric on an arbitrary set of actions, over which one defines operator θ, denoted here as `prodAct`. To compare actions one must include in the class requirements a notion of action equality `eqAct`, expressed as the closure of an order relation `leAct`. For example, the CCS interaction structure requires the following definition of *actions*:

```
data Act l = A l | AC l | Nop | Tau | Id  deriving Show
```

Dynamic combinators have a direct translation as functions over the final universe, as exemplified in the encoding of *prefix* and *choice*:

```
preP :: Act a -> Proc (Act a) -> Proc (Act a)
preP act p = Fin (C [(act,p)])

sumP ::  Proc (Act a) -> Proc (Act a) -> Proc (Act a)
sumP p q  = Fin (C (pp ++ qq)) where
            (C pp) = (unFin p)
            (C qq) = (unFin q)
```

On the other hand the definitons of static combinators are directly translated to HASKELL, provided that first one defines anamorphisms as a (generic) combinator. The following definition is standard:

```
ana :: Functor f => (c -> f c) -> c -> Nu f
ana phi = Fin . fmap (ana phi) . phi
```

Note, for example, how parallel composition | is defined in terms of the genes of $|||$ (`alphai`) and $\otimes$ (`alphap`):

```
par :: (Eq a) => (Proc (Act a), Proc (Act a)) -> Proc (Act a)
par (p, q) = ana alpha (p, q)
     where alpha (p, q) =
       C ((obsProc (alphai (p,q))) ++ (obsProc (alphap (p,q))))
```

4 Interruption and Recovery

4.1 Apomorphisms

This section introduces two *interruption* combinators, defined by *natural corecursion*, and encoded as *apomorphisms* [Vene and Uustalu, 1997]. In this pattern the final result can be either generated in successive steps or 'all at once'

without recursion. Therefore, the codomain of the source 'coalgebra' becomes the sum of its carrier with the coinductive type itself. The universal property is

$$h = \mathsf{apo}\, p \iff \mathsf{out}_{\mathsf{T}} \cdot h = \mathsf{T}\,[h, \mathsf{id}] \cdot p \tag{7}$$

from which one can easily deduce the following *cancellation*, *reflection* and *fusion* laws.

$$\mathsf{out}_{\mathsf{T}} \cdot \mathsf{apo}\, \varphi = \mathsf{T}[\mathsf{apo}\, \varphi, \mathsf{id}] \cdot \varphi \tag{8}$$

$$\mathsf{id} = \mathsf{apo}\, \mathsf{T}(\iota_1) \cdot \mathsf{out}_{\mathsf{T}} \tag{9}$$

$$\psi \cdot f = \mathsf{T}(f + \mathsf{id}) \cdot \varphi \Rightarrow \mathsf{apo}\, \psi \cdot f = \mathsf{apo}\, \varphi \tag{10}$$

4.2 Parallel Composition with Interruption

Our first combinator is a form of parallel composition which may terminate if some undesirable situation results from the interaction of the two processes. Such undesirable situation is abstractly represented by a particular form of interaction denoted by $*$. Therefore, combinator $\ddagger$ terminates execution as a result of an $*$-valued interaction. Formally, it is defined by an apomorphism $\ddagger = \mathsf{apo}\, \alpha_{\ddagger}$, according to the following diagram

$$\begin{array}{ccc} \nu \times \nu & \xrightarrow{\alpha_{\ddagger}} & \mathcal{P}(Act \times ((\nu \times \nu) + \nu)) \\ {\scriptstyle \ddagger}\downarrow & & \downarrow{\scriptstyle \mathcal{P}(\mathsf{id} \times [\ddagger, \mathsf{id}])} \\ \nu & \xrightarrow{\omega} & \mathcal{P}(Act \times \nu) \end{array}$$

where

$$\begin{aligned} \alpha_{\ddagger} = \nu \times \nu &\xrightarrow{\omega \times \omega} \mathcal{P}(Act \times \nu) \times \mathcal{P}(Act \times \nu) \\ &\xrightarrow{\mathcal{P}\tau_l \cdot \tau_r} \mathcal{P}\mathcal{P}((Act \times \nu) \times (Act \times \nu)) \\ &\xrightarrow{\mathcal{P}\mathsf{m} \cdot \cup} \mathcal{P}((Act \times Act) \times (\nu \times \nu)) \\ &\xrightarrow{\mathcal{P}(\theta \times \mathsf{id})} \mathcal{P}(Act \times (\nu \times \nu)) \\ &\xrightarrow{\mathcal{P}\mathsf{test}} \mathcal{P}(Act \times ((\nu \times \nu) + \nu)) \end{aligned}$$

where $\mathsf{test} = \langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle$. and $m : (A \times C) \times (B \times D) \longleftarrow (A \times B) \times (C \times D)$ is a natural isomorphism which exchanges the relative positions of factors in a product. Let us now illustrate how to compute with apomorphisms, by discussing the comutativity of this combinator, *i.e.*, the validity of the following equation, where s is the comutativity isomorphism:

$$\ddagger \cdot \mathsf{s} = \ddagger \tag{11}$$

As a first step we derive

$$
\begin{array}{ll}
& \ddagger \cdot \mathsf{s} = \ddagger \\
\equiv & \quad \{ \ \ddagger \text{ definition} \} \\
& \mathsf{apo}\,\alpha_\ddagger \cdot \mathsf{s} \ = \ \mathsf{apo}\,\alpha_\ddagger \\
\Leftarrow & \quad \{ \ \text{apomorphism fusion law} \ \} \\
& \alpha_\ddagger \cdot \mathsf{s} \ = \ \mathcal{P}(\mathsf{id} \times (\mathsf{s} + \mathsf{id})) \cdot \alpha_\ddagger
\end{array}
$$

Now, let us unfold the left hand side of this last equality.

$$
\begin{array}{ll}
& \alpha_\ddagger \cdot \mathsf{s} \\
= & \quad \{ \ \alpha_\ddagger \text{ definition} \ \} \\
& \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega \cdot \mathsf{s} \\
= & \quad \{ \ \mathsf{s} \text{ natural} \ \} \\
& \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \mathsf{s} \cdot \omega \times \omega \\
= & \quad \{ \ \tau_r \cdot \mathsf{s} = \mathcal{P}\mathsf{s} \cdot \tau_l \ \} \\
& \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \mathcal{P}\mathsf{s} \cdot \tau_l \cdot \omega \times \omega \\
= & \quad \{ \ \tau_l \cdot \mathsf{s} = \mathcal{P}\mathsf{s} \cdot \tau_r, \text{ functors} \ \} \\
& \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\mathcal{P}\mathsf{s} \cdot \mathcal{P}\tau_r \cdot \tau_l \cdot \omega \times \omega \\
= & \quad \{ \ \cup \text{ and } \mathsf{m} \text{ natural: } \mathsf{m} \cdot \mathsf{s} = (\mathsf{s} \times \mathsf{s}) \cdot \mathsf{m} \ \} \\
& \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}(\mathsf{s} \times \mathsf{s}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_r \cdot \tau_l \cdot \omega \times \omega \\
= & \quad \{ \ \mathcal{P}\tau_r \cdot \tau_l = \mathcal{P}\tau_l \cdot \tau_r, \text{ because } \mathcal{P} \text{ is a commutative monad [Kock, 1972]; functors} \ \} \\
& \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}((\theta \cdot \mathsf{s}) \times \mathsf{s}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega \\
= & \quad \{ \ \times\text{-fusion} \ \} \\
& \mathcal{P}(\langle \pi_1 \cdot ((\theta \cdot \mathsf{s}) \times \mathsf{s}) \,,\, (=_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2) \cdot ((\theta \cdot \mathsf{s}) \times \mathsf{s}) \rangle) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \\
& \cdot \tau_r \cdot \omega \times \omega \\
= & \quad \{ \ \text{conditional fusion, } \times\text{-cancellation, constant function} \ \} \\
& \mathcal{P}(\langle \theta \cdot \mathsf{s} \cdot \pi_1 \,,\, (=_* \cdot \theta \cdot \mathsf{s} \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \mathsf{s} \cdot \pi_2) \rangle) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega
\end{array}
$$

Unfolding the right hand side we arrive at

$$
\begin{array}{ll}
& \mathcal{P}(\mathsf{id} \times (\mathsf{s} + \mathsf{id})) \cdot \alpha_\ddagger \\
= & \quad \{ \ \alpha_\ddagger \text{ definition} \ \} \\
& \mathcal{P}(\mathsf{id} \times (\mathsf{s} + \mathsf{id})) \cdot \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \\
& \cdot \tau_r \cdot \omega \times \omega \\
= & \quad \{ \ \text{functors, } \times\text{-absorption} \ \} \\
& \mathcal{P}(\langle \pi_1, (\mathsf{s} + \mathsf{id}) \cdot (=_* \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \pi_2) \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega \\
= & \quad \{ \ \text{conditional fusion} \ \} \\
& \mathcal{P}(\langle \pi_1, =_* \cdot \pi_1 \rightarrow (\mathsf{s} + \mathsf{id}) \cdot \iota_2 \cdot \mathsf{nil}, (\mathsf{s} + \mathsf{id}) \cdot \iota_1 \cdot \pi_2 \rangle) \cdot \mathcal{P}(\theta \times \mathsf{id}) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \\
& \cdot \tau_r \cdot \omega \times \omega \\
= & \quad \{ \ +\text{-cancellation, conditional fusion law, functors, } \times\text{-fusion} \ \} \\
& \mathcal{P}(\langle \theta \cdot \pi_1 \,,\, =_* \cdot \theta \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \mathsf{s} \cdot \pi_2 \rangle) \cdot \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega
\end{array}
$$

These two unfolding processes did not lead to the same expression; equation (11) is, therefore, in general false. Note, however, that the difference between the

two expressions is only in the order in which the same arguments are supplied to θ. We may thus suppose the existence of a result weaker than (11), but still relevant and useful, may result from this calculation. This requires a more general discussion which follows.

4.3 Conditional Fusion

The aim of the previous calculation was to prove equation (11) which, by fusion, reduced to

$$\alpha_{\ddagger} \cdot \mathsf{s} \;=\; \mathcal{P}(\mathsf{id} \times (\mathsf{s} + \mathsf{id})) \cdot \alpha_{\ddagger} \tag{12}$$

Note the advantage of using a fusion law is to get rid of direct manipulation of recursion: all computation is done in terms of the recursion *genes*. In this way we succeeded in reducing (12) to

$$\begin{array}{l} \quad \mathcal{P}(\langle \theta \cdot \mathsf{s} \cdot \pi_1 \,,\, (=_* \cdot \theta \cdot \mathsf{s} \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \mathsf{s} \cdot \pi_2) \rangle) \cdot \gamma \\ = \\ \quad \mathcal{P}(\langle \theta \cdot \pi_1 \,,\, (=_* \cdot \theta \cdot \pi_1 \rightarrow \iota_2 \cdot \mathsf{nil}, \iota_1 \cdot \mathsf{s} \cdot \pi_2) \rangle) \cdot \gamma \end{array}$$

where $\gamma \;=\; \mathcal{P}\mathsf{m} \cdot \cup \cdot \mathcal{P}\tau_l \cdot \tau_r \cdot \omega \times \omega$. Now note that this equation is only valid if one postulates an additional condition expressing the commutativity of θ, *i.e.*,

$$\theta \cdot \mathsf{s} \;=\; \theta \tag{13}$$

The interesting question is then: what does such a conditional validity at the *genes* level imply with respect to the validity of the original equation (11)? In general, suppose that in a proof, one concludes that the validity of the antecedent of the fusion law

$$\alpha \cdot f = \mathsf{T}(f + \mathsf{id}) \cdot \beta \;\;\Rightarrow\;\; \mathsf{apo}\,\alpha \cdot f \;=\; \mathsf{apo}\,\beta \tag{14}$$

depends on an additional condition Φ, *i.e.*,

$$\Phi \;\;\Rightarrow\;\; \alpha \cdot f \;=\; \mathsf{T}(f + \mathsf{id}) \cdot \beta \tag{15}$$

What happens is that Φ is stated as a *local* condition on the genes of the apomorphisms, *i.e.*, on the imediate derivatives of the proocesses involved. Such a condition needs to be made stronger enforcing validity over *all* derivatives. Technically, Φ should be transformed into an *invariant*: *i.e.*, a predicate which is preserved by the coalgebra dynamics, ω, in the present case. To state such a result we need a modal language interpreted over coalgebras. The following notions are relatively standard in the literature (see, *e.g.*, [Moss, 1999] or [Jacobs, 1999]).

A predicate $\phi : \mathbb{B} \longleftarrow U$ over the carrier of a T-coalgebra $\langle U, \gamma : \mathsf{T}\ U \longleftarrow U\rangle$ is called a γ-*invariant* if closed under γ. Formally, one defines a predicate combinator $\circ_\gamma$ [2]:

$$(\circ_\gamma\ \phi)\ u\ \equiv\ \forall_{u' \in_{\mathsf{T}} \gamma\ u}\ .\ \phi\ u'$$

whose meaning reads: $\circ_\gamma\ \phi$ is valid in all states whose immediate γ-derivatives verify ϕ. Then, ϕ is an invariant iif $\phi\ \Rightarrow\ \circ_\gamma\ \phi$, that is $\phi \subseteq \circ_\gamma\ \phi$

The closure of $\circ_\gamma$ defines the coalgebraic equivalent to the *always in the future* modal operator (just as $\circ_\gamma$ corresponds to the *next* operator in modal logic). Thus, $\Box_\gamma\ \phi$ is introduced in [Jacobs, 1999] as the *greatest* fixpoint of function $\lambda_x\ .\ \phi \cap \circ_\gamma\ x$. Intuitively, $\Box_\gamma\ \phi$ reads 'ϕ holds in the current state and all the other states under γ. In such a definition contains the key to answer our previous question, as stated in the following lemma.

Lemma 2 *Let α and β stand for two T-coalgebras and Φ a predicate over the carrier of β. Then,*

$$(\Phi \Rightarrow \alpha \cdot h = \mathsf{T}(h + \mathsf{id}) \cdot \beta)\ \ \Rightarrow\ \ (\Box_\beta\ \Phi \Rightarrow (\mathsf{apo}_\alpha \cdot h = \mathsf{apo}_\beta \mathsf{T})) \qquad (16)$$

Proof. Let X be the carrier of β and i_Φ the inclusion in X of the subset classified by predicate Φ, *i.e.*, $\Phi \cdot i_\Phi = \underline{true}\cdot!$. Any β-invariant induces a subcoalgebra β' which makes $i_{\Box_\beta\ \Phi}$ a coalgebra morphism from β' to β. Then,

$$\begin{array}{ll}
& \Phi \Rightarrow \alpha \cdot h = \mathsf{T}(h + \mathsf{id}) \cdot \beta \\
\equiv & \quad \{ \text{ definition of inclusion } i_\Phi \} \\
& \alpha \cdot h \cdot i_\Phi = \mathsf{T}(h + \mathsf{id}) \cdot \beta \cdot i_\Phi \\
\Rightarrow & \quad \{ \Box_\beta \Phi \subseteq \Phi \} \\
& \alpha \cdot h \cdot i_{\Box_\beta \Phi} = \mathsf{T}(h + \mathsf{id}) \cdot \beta \cdot i_{\Box_\beta \Phi} \\
\equiv & \quad \{ i_{\Box_\beta \Phi} \text{ is a morphism from } \beta' \text{ to } \beta \} \\
& \alpha \cdot h \cdot i_{\Box_\beta \Phi} = \mathsf{T}(h + \mathsf{id}) \cdot \mathsf{T}(i_{\Box_\beta \Phi}) \cdot \beta' \\
\equiv & \quad \{ \text{ functors, apomorfism fusion law } \} \\
& \mathsf{apo}\,\alpha \cdot h \cdot i_{\Box_\beta \Phi} = \mathsf{apo}\,\beta' \\
\equiv & \quad \{ i_{\Box_\beta \Phi} \text{ is a coalgebra morphism } \} \\
& \mathsf{apo}\,\alpha \cdot h \cdot i_{\Box_\beta \Phi} = \mathsf{apo}\,\beta \cdot i_{\Box_\beta \Phi} \\
\equiv & \quad \{ \text{ inclusion } i_{\Box_\beta\ \Phi} \} \\
& \Box_\beta \Phi \Rightarrow (\mathsf{apo}\,\alpha \cdot h = \mathsf{apo}\,\beta)
\end{array}$$

□

[2] Notation $\in_{\mathsf{T}}$ refers to the extension of the membership relation to regular functors [Meng and Barbosa, 2004].

We call formula (16) the *conditional fusion* law for apomorphisms. A similar result, but restricted to anamorphisms was proved in [Barbosa, 2001]. Let's come back to our example. Note that in this case the relevant predicate, given by equation (13), does not involve states, but just *actions*. Therefore

$$\Box_\omega\,(\theta \cdot \mathsf{s} = \theta) \;=\; (\theta \cdot \mathsf{s} = \theta)$$

which, according to lemma 2, is the predicate to be used as the antecedent of (12). We may now conclude this example stating the following general law concerning the interruption operator:

$$(\theta \cdot \mathsf{s} = \theta) \;\;\Rightarrow\;\; \ddagger \cdot \mathsf{s} = \ddagger \tag{17}$$

4.4 A Recovery Operator

We now discuss a combinator which models fault recovery[3]. Intuitively, the combinator allows the execution of its first argument until an *error* state is reached. By convention, an error occurrence is signalled by the execution of a special action x. When this is detected, execution control is passed to the second process. This process, which is the combinator second argument, is an abstraction for the system's recovery code. The combinator is defined as $\rhd \;=\; \mathsf{apo}\,\alpha_\rhd$, where

$$\begin{aligned}
\alpha_\rhd \;=\; \nu \times \nu &\xrightarrow{\omega \times \mathsf{id}} \mathcal{P}(Act \times \nu) \times \nu \\
&\xrightarrow{\mathsf{t}_x \cdot \pi_1 \to \iota_1, \iota_2 \cdot \pi_2} \mathcal{P}(Act \times \nu) \times \nu + \nu \\
&\xrightarrow{\tau_r + \omega} \mathcal{P}(Act \times (\nu \times \nu)) + \mathcal{P}(Act \times \nu) \\
&\xrightarrow{[\mathcal{P}(\mathsf{id} \times \iota_1), \mathcal{P}(\mathsf{id} \times \iota_2)]} \mathcal{P}(Act \times (\nu \times \nu + \nu))
\end{aligned}$$

where $\mathsf{t}_x : \mathbb{B} \longleftarrow \mathcal{P}(Act \times \nu)$ is given by $\mathsf{t}_x \;= \notin_x \cdot \mathcal{P}\pi_1$.

We shall go on exploring the calculational power of this approach to process algebra through the discussion of a new conditional property. The intuition says that should no faults be detected in the first process, the recovery process will not be initiated. In other words, in the absence of faults, a process running in a fault tolerant environment behaves just as it would do if executed autonomously. Formally,

Lemma 3

$$\Box_\omega\,(\notin_x \cdot \mathcal{P}\pi_1) \;\;\Rightarrow\;\; \rhd \;=\; \pi_1 \tag{18}$$

[3] Although the very abstract level in which it is approached here, it should be underlined that *fault tolerance* is a fundamental issue in software engineering.

Proof. Note that predicate $\notin_x \cdot \mathcal{P}\pi_1$ only states fault absence in the immediate successors of each state. It is, therefore, sufficient to establish the antecedent of the fusion law, as shown below.

$$
\begin{array}{ll}
 & \mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot \alpha_{\triangleright} \\
= & \quad \{ \text{ definition of } \mathrm{t}_x \text{, assuming hypothesis } \notin_x \cdot \mathcal{P}\pi_1 \ \} \\
 & \mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot [\mathcal{P}(\mathsf{id} \times \iota_1), \mathcal{P}(\mathsf{id} \times \iota_2)] \cdot (\tau_r + \omega) \cdot \iota_1 \cdot \omega \times \mathsf{id} \\
= & \quad \{ \ \tau_r + \omega = [\iota_1 \cdot \tau_r, \iota_2 \cdot \omega] \} \\
 & \mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot [\mathcal{P}(\mathsf{id} \times \iota_1), \mathcal{P}(\mathsf{id} \times \iota_2)] \cdot [\iota_1 \cdot \tau_r, \iota_2 \cdot \omega] \cdot \iota_1 \cdot \omega \times \mathsf{id} \\
= & \quad \{ \ +\text{-cancellation} \ \} \\
 & \mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot \mathcal{P}(\mathsf{id} \times \iota_1) \cdot \tau_r \cdot \omega \times \mathsf{id} \\
= & \quad \{ \ \mathcal{P} \text{ is a functor, } +\text{-cancellation, functors} \ \} \\
 & \mathcal{P}(\mathsf{id} \times \iota_1) \cdot \mathcal{P}(\mathsf{id} \times \pi_1) \cdot \tau_r \cdot \omega \times \mathsf{id} \\
= & \quad \{ \ \mathcal{P}(\mathsf{id} \times \pi_1) \cdot \tau_r \ = \ \pi_1 \ \} \\
 & \mathcal{P}(\mathsf{id} \times \iota_1) \cdot \pi_1 \cdot \omega \times \mathsf{id} \\
= & \quad \{ \ f \times g = \langle f, g \rangle, \ \times\text{-cancellation} \ \} \\
 & \mathcal{P}(\mathsf{id} \times \iota_1) \cdot \omega \cdot \pi_1
\end{array}
$$

Note that what we would expect to have proven was

$$\mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot \alpha_{\triangleright} \ = \ \omega \cdot \pi_1$$

but, actually, all that was shown was that

$$\mathcal{P}(\mathsf{id} \times (\pi_1 + \mathsf{id})) \cdot \alpha_{\triangleright} \ = \ \mathcal{P}(\mathsf{id} \times \iota_1) \cdot \omega \cdot \pi_1$$

This comes to no surprise: the role of the additional factor $\mathcal{P}(\mathsf{id} \times \iota_1)$ in the right hand side is to ensure type compatibility between both sides of the equation. The important point, however, is the fact that the whole proof was carried under the assumption, recorded in the very first step, that $\notin_x \cdot \mathcal{P}\pi_1$. Thus, by lema 2, we conclude as expected.

□

5 Conclusions and Future Work

Final semantics for processes is an active research area, namely after Aczel's landmark paper [Aczel, 1993]. Related work on coalgebraic modelling of process algebras can be found in *e.g.*, [Schamschurko, 1998, Wolter, 1999]. Our emphasis, however, is placed on the design side: we intend to show how process calculi can be developed and their laws proved along the lines one gets used to in (data-oriented) program calculi.

The most interesting laws in process algebras are formulated in terms of some form of *weak* bisimulation, which abstracts away from, *e.g.*, internal computation. Dealing with such weak process equivalences in a coalgebraic setting is not trivial (but see, for example, [Rothe and Masulovic, 2002, Sokolova et al., 2005]). A concrete approach, based on transposition to a category of binary relations, is proposed in the first author MSc thesis [Ribeiro, 2005], still in accordance with the calculational style favoured in this paper. On the other hand, whether this work scales up to process algebras with *mobility* remains an open question.

From a software engineering point of view, our interest in generic process algebras arise from on-going work on the semantics of component orchestration languages [Barbosa and Barbosa, 2004]. As briefly discussed in section 2, in such a context one often needs to tailor process algebras to quite specific interaction disciplines, later giving rise to programming constructs for component glueing. The generic framework outlined in this paper, and the associated HASKELL prototyper, proves to be quite effective in that task. This is why we consider the design of *generic* process algebras a programming challenge.

Acknowledgements. This research was carried on in the context of the PURE Project supported by FCT under contract `POSI/ICHS/44304/2002`.

References

[Aczel, 1993] Aczel, P. (1993). Final universes of processes. In *Proc. Math. Foundations of Programming Semantics*. Springer Lect. Notes Comp. Sci. (802).

[Barbosa, 2001] Barbosa, L. S. (2001). Process calculi *à la* Bird-Meertens. In *CMCS'01*, volume 44.4, pages 47–66, Genova. Elect. Notes in Theor. Comp. Sci., Elsevier.

[Barbosa and Barbosa, 2004] Barbosa, M. A. and Barbosa, L. S. (2004). Specifying software connectors. In Araki, K. and Liu, Z., editors, *1st International Colloquium on Theorectical Aspects of Computing (ICTAC'04)*, pages 53–68, Guiyang, China. Springer Lect. Notes Comp. Sci. (3407).

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

[Bird and Meertens, 1987] Bird, R. S. and Meertens, L. (1987). Two exercises found in a book on algorithmics. In Meertens, L., editor, *Program Specification and Transformation*, pages 451–458. North-Holland.

[Gibbons, 1997] Gibbons, J. (1997). Conditionals in distributive categories. CMS-TR-97-01, School of Computing and Mathematical Sciences, Oxford Brookes University.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International.

[Jacobs, 1999] Jacobs, B. (1999). The temporal logic of coalgebras via Galois algebras. Techn. rep. CSI-R9906, Comp. Sci. Inst., University of Nijmegen.

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

[Meijer et al., 1991] Meijer, E., Fokkinga, M., and Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. In Hughes, J., editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Lect. Notes Comp. Sci. (523).

[Meng and Barbosa, 2004] Meng, S. and Barbosa, L. S. (2004). On refinement of generic software components. In Rettray, C., Maharaj, S., and Shankland, C., editors, *10th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 506–520, Stirling. Springer Lect. Notes Comp. Sci. (3116). Best Student Co-authored Paper Award.

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

[Moss, 1999] Moss, L. (1999). Coalgebraic logic. *Ann. Pure & Appl. Logic*.

[Ribeiro, 2005] Ribeiro, P. R. (2005). Coinductive programming: Calculi and applications. MSc Thesis (in portuguese), DI, Universidade do Minho.

[Rothe and Masulovic, 2002] Rothe, J. and Masulovic, D. (2002). Towards weak bisimulation for coalgebras. In Kurz, A., editor, *Proc. Int. Workshop on Categorical Methods for Concurrency, Interaction, and Mobility (CMCIM'02)*, volume 68. Elect. Notes in Theor. Comp. Sci., Elsevier.

[Schamschurko, 1998] Schamschurko, D. (1998). Modeling process calculi with PVS. In *CMCS'98, Elect. Notes in Theor. Comp. Sci.*, volume 11. Elsevier.

[Sokolova et al., 2005] Sokolova, A., Vink, E. d., and Woracek, H. (2005). Weak bisimulation for action-type coalgebras. In Birkedal, L., editor, *Proc. Int. Conf. on Category Theory and Computer Science (CTCS'04)*, volume 122, pages 211–228. Elect. Notes in Theor. Comp. Sci., Elsevier.

[Vene and Uustalu, 1997] Vene, V. and Uustalu, T. (1997). Functional programming with apomorphisms (corecursion). In *Proc. 9th Nordic Workshop on Programming Theory*.

[Wolter, 1999] Wolter, U. (1999). A coalgebraic introduction to CSP. In *Proc. of CMCS'99*, volume 19. Elect. Notes in Theor. Comp. Sci., Elsevier.

A Product and Sum Laws

Functions with a common domain can be glued through a *split* $\langle f, g\rangle$ defined by the following universal property:

$$k = \langle f, g\rangle \quad \equiv \quad \pi_1 \cdot k = f \ \wedge \ \pi_2 \cdot k = g \tag{19}$$

from which the following properties can be derived:

$$\langle \pi_1, \pi_2\rangle \ = \ \mathsf{id}_{A\times B} \tag{20}$$

$$\pi_1 \cdot \langle f, g\rangle = f \ , \ \pi_2 \cdot \langle f, g\rangle = g \tag{21}$$

$$\langle g, h\rangle \cdot f \ = \ \langle g \cdot f, h \cdot f\rangle \tag{22}$$

$$(i \times j) \cdot \langle g, h\rangle \ = \ \langle i \cdot g, j \cdot h\rangle \tag{23}$$

known respectively as $\times$ *reflection*, *cancelation*, *fusion* and *absorption* laws. Similarly arises *structural equality*:

$$\langle f, g\rangle = \langle k, h\rangle \ \equiv \ f = k \ \wedge \ g = h \tag{24}$$

Finally note that the product construction is *functorial*: $f \times g = \lambda \langle a, b\rangle \ . \ \langle f \ a, g \ b\rangle$.

Dually, functions sharing the same codomain may be glued together through an *either* combinator, expressing alternative behaviours, and introduced as the universal arrow in a datatype sum construction. $A + B$ is defined as the target of two arrows $\iota_1 : A + B \longleftarrow A$ and $\iota_2 : A + B \longleftarrow B$, called the *injections*, which satisfy the following universal property:

$$k = [f, g] \quad \equiv \quad k \cdot \iota_1 = f \ \wedge \ k \cdot \iota_2 = g \tag{25}$$

from which one infers correspondent *cancelation*, *reflection* and *fusion* results:

$$[f,g]\cdot\iota_1 = f\ ,\ [f,g]\cdot\iota_2 = g \tag{26}$$

$$[\iota_1,\iota_2] \ = \ \mathsf{id}_{X+Y} \tag{27}$$

$$f\cdot[g,h] \ = \ [f\cdot g, f\cdot h] \tag{28}$$

Products and sums interact through the following *exchange* law

$$[\langle f,g\rangle,\langle f',g'\rangle] \ = \ \langle [f,f'],[g,g']\rangle \tag{29}$$

provable by either product (19) or sum (25) universality. The *sum* combinator also applies to functions yielding $f+g : A'+B' \longleftarrow A+B$ defined as $[\iota_1\cdot f, \iota_2\cdot g]$.

Conditional expressions are modelled by coproducts. In this paper we adopt the McCarthy conditional constructor written as $(p\ \rightarrow\ f,\ g)$, where $p : \mathbb{B} \longleftarrow A$ is a predicate. Intuitively, $(p\ \rightarrow\ f,\ g)$ reduces to f if p evaluates to true and to g otherwise. The conditional construct is defined as

$$(p\ \rightarrow\ f,\ g) \ = \ [f,g]\cdot p?$$

where $p? : A + A \longleftarrow A$ is determined by predicate p as follows

$$p? = \qquad A \xrightarrow{\langle \mathsf{id},p\rangle} A\times(\mathbf{1}+\mathbf{1}) \xrightarrow{\mathsf{dl}} A\times\mathbf{1}+A\times\mathbf{1} \xrightarrow{\pi_1+\pi_1} A+A$$

where dl is the distributivity isomorphism. The following laws are usefull to calculate with conditionals [Gibbons, 1997].

$$h\cdot(p\ \rightarrow\ f,\ g) \ = \ (p\ \rightarrow\ h\cdot f,\ h\cdot g) \tag{30}$$

$$(p\ \rightarrow\ f,\ g)\cdot h \ = \ (p\cdot h\ \rightarrow\ f\cdot h,\ g\cdot h) \tag{31}$$

$$(p\ \rightarrow\ f,\ g) \ = \ (p\ \rightarrow\ (p\ \rightarrow\ f,\ g),\ (p\ \rightarrow\ f,\ g)) \tag{32}$$