Coalgebraic Structures in Program Construction

Luís S. Barbosa

[1] Departamento de Informática
Universidade do Minho
Braga, Portugal

lsb@di.uminho.pt

Abstract. *Both* initial *algebras and* final *coalgebras are devices which provide abstract descriptions of a variety of phenomena in programming, in particular of* data *and* behavioural *structures, respectively. The former are specified by a set of* constructors *and well known in the programming practice. The latter resorts to a collection of* observers *and have been recently recognised as a promising framework to model and reason about dynamical, state-based systems. Both initiality and finality, as universal properties, entail definitional and proof principles,* i.e., *a basis for the development of program calculi directly based on (actually driven by) type specifications. Moreover, such properties can be turned into programming* combinators *and used, not only to calculate programs, but also to program with. This tutorial provides a first introduction to* coalgebraic *structures, from a programming point of view, to foster further applications in the area of program construction.*

1. Introduction

Program Construction. Program construction is the systematic derivation of computational structures in such a way that correctness is guaranteed by construction. The development of program calculi directly based on, actually driven by, type specifications has had a fundamental impact on pursuing this goal, mainly in the area of *functional programming*. In fact, since John McCarthy original papers [McCarthy, 1960, McCarthy, 1963], and, later, Backus FP [Backus, 1978] — a functional language based on combinators related by algebraic laws — functional programming and *program calculi* have developed in an intertwined way. Moreover, program properties have been turned into programming *combinators* and used, not only to calculate programs, but also to program with. This discipline of algorithm derivation and transformation can be traced back to the so-called *Bird-Meertens formalism* [Bird, 1987, Bird and Meertens, 1987] and the foundational work of T. Hagino [Hagino, 1987] [1].

Can this calculation method, and the underlying modelling principles, be extended to the world of *dynamical*, *state-based*, *communicating* systems? To answer this question we shall first attempt to formalise our understanding of such systems.

[1]Since then, the area has known a remarkable progress, as witnessed by the vast bibliography published both on theory and applications — see [Malcolm, 1990, Meijer et al., 1991, Bird and Moor, 1997], among many others references.

State-based Systems. One of the most elementary models for a computational process is that of a *function*

$$f : I \longrightarrow O \tag{1}$$

which specifies a transformation rule between two structures I and O. The behaviour of a function is captured by the output it produces, which is completely determined by the supplied input. In a (metaphorical) sense, this may be dubbed as the 'engineer's view' of computation: *here is a recipe (a tool, a technology) to build gnus from gnats*.

Often, however, reality is not so simple. For example, one may know how to produce 'gnus' from 'gnats' but not in all cases. This is expressed by observing the output of f in a more refined context: O is replaced by $O + \mathbf{1}$ and f is said to be a *partial* function, returning either a valid output or an exception value. Or else one may be uncertain of the outcome of f, in the sense that the evolution of the system being observed (the 'gnu population') may be non deterministic. The output of f becomes typed as $\mathcal{P}O$, where $\mathcal{P}$ denotes finite powerset. In other situations one may recognise that there is some environmental (or context) information about 'gnats' that, for some reason, should be hidden from input. It may be the case that such information is too extensive to be supplied to f by its user, or that it is shared by other functions as well. It might also be the case that building gnus will eventually modify the environment, thus influencing latter production of more 'gnus'. For U a denotation of such context information, the signature of f becomes [2]

$$f : I \longrightarrow (O \times U)^U \tag{2}$$

A function computed within a context is often referred to as 'state-based', in the sense the word 'state' has in automaton theory — the internal memory of the automaton which both constraints and is constrained by the execution of actions. In fact, the 'nature' of $f : I \longrightarrow (O \times U)^U$ as a 'state-based function' is made more explicit by rewriting its signature as

$$f : U \longrightarrow (O \times U)^I \tag{3}$$

This, in turn, may suggest an alternative model for computations, which (again in a metaphorical sense) one may dub as the 'natural scientist's view'. Instead of a recipe to build 'gnus' from 'gnats', she just recognises that *there exist gnus and gnats and that their evolution can be observed.* That *observation* may entail some form of *interference* is well known, even from Physics, and thus the underlying notion of computation is not necessarily a passive one.

The able 'natural scientist' will equip herself with the right 'lens' — that is, a tool to observe with, which necessarily entails a particular shape for observation. The emphasis is now placed on states: input and output parameters may or may not become relevant, depending on the particular kind of observation one may want to perform. In other words, one's focus becomes the 'nature', or the 'universe' or, more pragmatically, the *state space*. That we can observe 'gnus' being produced out of 'gnats' is just one, among other possible observations. The basic ingredients required to support an 'observa-

[2]Notation B^A represents the space of functions from A to B. In the sequel $\overline{f} : A \times B \longrightarrow C$ stands for the uncurry of function $f : A \longrightarrow C^B$ (see appendix A).

tional', or 'state-based', view of computational processes may be summarised as follows:

a *lens*:	$\bigcirc\frown\bigcirc$	
an *observation structure*:	universe	$\xrightarrow{f}$ $\bigcirc\frown\bigcirc$ universe

Informally, the *lens* can be thought of as providing a *shape* which f is able to deal with in each of the above mentioned situations. From a technical point of view, as this tutorial aims to make clear, the pair $\langle$universe, $f\rangle$, as above, constitutes a *coalgebra* with interface $\bigcirc\frown\bigcirc$. Coalgebras provide simple models for state-based (dynamical) systems.

Differently from familiar, inductive, data types (like, *e.g.*, finite sequences) which are completely defined by a set of constructors (*e.g.*, the empty list and appending), the sort of computational structures we have in mind are characterised by the fact that their behaviour is determined, not only by an external (input) stimuli, but also by some internal 'memory' to which there is, in general, no direct access. Typical examples of such structures are *processes*, *transition systems*, *objects*, *stream-like structures* used in lazy programming languages, 'infinite' or *non well-founded objects* arising in semantics, and so on. Such 'systems' are inherently dynamic, possess an observable behaviour, but their internal configurations remain hidden and have therefore to be identified if not distinguishable by observation. Summing up, this tutorial aims to develop techniques to model and calculate with systems characterised by:

- the presence of an *internal state space* which evolves and persists in time,
- the possibility of *interaction* with other systems during the overall computation,
- their typical use as a 'building blocks' of larger, often concurrent, systems.

The relevance of this program is emphasised by recognising that such systems are 'everywhere', from sophisticated management systems, like plant control or administrative applications, to formal automata or domestic appliances.

Plan. A number of basic concepts (*e.g.*, *system*, *behaviour* and *morphism*) are introduced through an example in section 2. and later generalized in section 3.. Section 4. discusses their application to reason about systems and behaviours. Equational reasoning and proofs by bisimulation are compared and shown to be essentially the two sides of a same coin. Finally, section 5. reports on a case study: the application of coalgebraic techniques to the design of process calculi. Some of the constructions discussed are *animated* in CHARITY [Cockett and Fukushima, 1992], a *total* functional language which offers explicit support for both *inductive* and *coinductive* types and enforces a discipline for their use. In appendix some more details are given on the underlying semantic universe and the CHARITY programming language Most of the material presented here is rather standard (even if quite recent). Some of the applications, however, namely on *process calculi* design, corresponds to author's recent research documented in [Barbosa, 2001, Barbosa and Oliveira, 2002].

2. Systems and Behaviours

The basic insight in coalgebraic modelling is the representation of state-based systems by functions typed as

$$p : U \longrightarrow \bigcirc\!\frown\!\bigcirc U \tag{4}$$

where U is a set of states (usually called the *carrier* of p). For every state $u \in U$, function p (said the system's *dynamics*) describes the observable effects of an elementary step in the evolution of the system (*i.e.*, of a state transition). The possible outcomes of taking such a step are captured by notation $\bigcirc\!\frown\!\bigcirc\ U$, where $\bigcirc\!\frown\!\bigcirc$ is correctly understood as a *shape* for the observation structure or, as a programmer would say, an *interface* or a system's *type*.

This section introduces some basic ideas on coalgebraic structures by focusing on a particularly simple sort of systems: those reacting to an *attribute* $\mathsf{at} : U \longrightarrow O$ and whose (deterministic) evolution is conducted by a *method* (or *action*) $\mathsf{m} : U \longrightarrow U$ with no external influence (but for, say, pushing a button). Those two functions can be 'glued' together by a *split* combinator [3], leading to

$$p \;=\; \langle \mathsf{at}, \mathsf{m} \rangle : U \longrightarrow O \times U \tag{5}$$

Successive observations of (or experiments with) a system p reveals its allowed behavioural patterns. For each state value $u \in U$, the *behaviour* of p at u (more precisely, *from* u onwards) — represented by $[\!(p)\!]\ u$ — is an *infinite* sequence of values of type O computed by observing the successive state configurations, *i.e.*,

$$[\!(p)\!]\ u \;=\; <\mathsf{at}\ u,\ \mathsf{at}\ (\mathsf{m}\ u),\ \mathsf{at}\ (\mathsf{m}\ (\mathsf{m}\ u)), ...> \tag{6}$$

Thus, the space of *all* behaviours, for this sort of systems, is the set of *streams* (infinite sequences) of O, *i.e.*, O^ω.

Bringing *input* information into the scene leads to a mild sophistication of this model. The result is a *Moore* or a *Mealy* machine depending on the way input affects the computation of attributes [4]. Represented as state-based functions they are described as

$$\langle \mathsf{at}, \mathsf{m} \rangle : U \longrightarrow O \times U^I \quad \text{and} \quad \overline{\langle \mathsf{at}, \mathsf{m} \rangle} : U \longrightarrow (O \times U)^I \tag{7}$$

respectively.

Let us concentrate on Mealy machines for a while. Their behaviours organise themselves into *tree*-like structures, because they depend on the sequences of input processed. Such trees, whose arcs are labelled with I values and nodes with O values, can be represented by functions from non empty sequences of I (as no input means no observation!) to the attribute type O. In other words, the space of behaviours of Mealy machines (on I and O) is the set O^{I^+}. The behaviour of p at a particular state u is computed by [5]:

[3] Functional combinators, like *split*, are reviewed in appendix A.

[4] In classical automaton theory a distinction is traced between *Moore* machines [Moore, 1966], where each state is associated to an output symbol, whereas such symbols are associated to transitions, rather than states, in a *Mealy* machine [Mealy, 1955].

[5] In the sequel $<>$ stands for the empty sequence. We also adopt the rather liberal notation $<i : s : j>$ to access elements of a sequence (indicated by letters i, j, k, ...) and sub-sequences (s, t, ...).

$$[\![p]\!]\ u \;=\; \lambda\, {<}s : i{>}\ .\ \mathsf{at}\ \langle (\mathsf{next}\ u)\ s, i\rangle \tag{8}$$

where

$$(\mathsf{next}\ u)\ {<>} = u \quad \text{and} \quad (\mathsf{next}\ u)\ {<}s : i{>} = \mathsf{m}\ \langle (\mathsf{next}\ u)\ s, i\rangle \tag{9}$$

Now note that set O^{I^+} itself can be equipped with the structure of a Mealy machine as well. Actually, define

$$\omega \;=\; \overline{\langle \mathsf{root}, \mathsf{branches}\rangle} : O^{I^+} \longrightarrow (O \times O^{I^+})^I \tag{10}$$

where,

$$\mathsf{root}\ \langle \phi, i\rangle = \phi\ {<}i{>} \quad \text{and} \quad \mathsf{branches}\ \langle \phi, i\rangle = \lambda\, s\ .\ \phi\ {<}i : s{>}$$

Note that a state in ω is a function ϕ. Therefore, the attribute is computed by function application, whereas the method gives a new function which reacts to a sequence s of inputs exactly as ϕ would react to the appending of i to s. As we shall see soon, this system has a very peculiar role among the collection of all Mealy machines. Before that, however, we should look for a way of *relating* this sort of systems.

As usual, in Mathematics, the introduction of a new structure is not complete unless accompanied by the definition of a notion of a morphism which preserves it. In our case, a *Mealy morphism* $h : \langle U, p\rangle \longrightarrow \langle V, q\rangle$, is simply a function $h : U \longrightarrow V$ relating the underlying state spaces in such a way that the *dynamics* of the source system is preserved in the target one. This can be asserted by requiring the commutativity of the following diagram:

$$\begin{tikzcd} U \times I \arrow[r, "\langle \mathsf{at}, \mathsf{m}\rangle"] \arrow[d, "h \times \mathsf{id}"'] & O \times U \arrow[d, "\mathsf{id} \times h"] \\ V \times I \arrow[r, "\langle \mathsf{at}', \mathsf{m}'\rangle"] & O \times V \end{tikzcd} \tag{11}$$

Simplifying the underlying equation (resorting to $\times$ *fusion* and *absorption* — laws (38) and (39), respectively), yields the following pair of equations:

$$\mathsf{at} = \mathsf{at}' \cdot (h \times \mathsf{id}) \tag{12}$$

and

$$h \cdot \mathsf{m} = \mathsf{m}' \cdot (h \times \mathsf{id}) \tag{13}$$

Now, if the set of observations O^{I^+} can be turned into a system itself, it is not surprising that, for each system p, states can be mapped into their behaviours in a 'well-behaved' way. In other words, $[\![p]\!]$ can be singled out as a *morphism* from p to ω, because:

Proof. We check conditions (12) and (13), instantiating h with $[\![p]\!] : p \longrightarrow \omega$. Thus,

$$\begin{aligned}
& (\mathsf{root} \cdot ([\![p]\!] \times \mathsf{id}))\ \langle u, i\rangle \\
= & \qquad \{\ \mathsf{root}\ \text{definition}\ \} \\
& ([\![p]\!]\ u)\ {<}i{>} \\
= & \qquad \{\ [\![p]\!]\ \text{definition}\ \} \\
& \mathsf{at}\ \langle (\mathsf{next}\ u)\ {<>}, i\rangle \\
= & \qquad \{\ \mathsf{next}\ \text{definition}\ \} \\
& \mathsf{at}\ \langle u, i\rangle
\end{aligned}$$

And, similarly,

$$\begin{array}{ll}
& (\mathsf{branches} \cdot ([\![p]\!] \times \mathsf{id}))\ \langle u, i\rangle \\
= & \quad \{\ \mathsf{branches}\ \text{definition}\ \} \\
& \lambda <s : j> .\ \ ([\![p]\!]\ u)\ <i : s : j> \\
= & \quad \{\ [\![p]\!]\ \text{definition}\ \} \\
& \lambda <s : j> .\ \ \mathsf{at}\ \langle(\mathsf{next}\ u)\ <i : s>, j\rangle \\
= & \quad \{\ \mathsf{next}\ \text{definition}\ \} \\
& \lambda <s : j> .\ \ \mathsf{at}\ \langle(\mathsf{next}\ \mathsf{m}\ \langle u, i\rangle)s, j\rangle \\
= & \quad \{\ [\![p]\!]\ \text{definition}\ \} \\
& [\![p]\!]\ (\mathsf{m}\ \langle u, i\rangle)
\end{array}$$

□

Another basic fact on behaviours concerns their *preservation by morphisms*. Formally, given two Mealy machines p and q and a morphism $h : p \longrightarrow q$ between them, one has $[\![p]\!]\ u = [\![q]\!]\ h\ u$.

Proof. We first show that $h \cdot \mathsf{next} = \mathsf{next} \cdot h$ by induction on the sequence of inputs. Let $u \in U$ and $l \in I^*$. Clearly, $(h \cdot \mathsf{next}\ u)\ <> = h\ u$, by definition of next. For the inductive step, consider

$$\begin{array}{ll}
& (h\ (\mathsf{next}\ u))\ <s : i> \\
= & \quad \{\ \mathsf{next}\ \text{definition}\ \} \\
& h(\mathsf{m}\ \langle(\mathsf{next}\ u)\ s, i\rangle) \\
= & \quad \{\ h\ \text{is a morphism}\ \} \\
& \mathsf{m}'\ \langle h\ ((\mathsf{next}\ u)\ s), i\rangle \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
& \mathsf{m}'\ \langle(\mathsf{next}\ h\ u)\ s, i\rangle \\
= & \quad \{\ \mathsf{next}\ \text{definition}\ \} \\
& (\mathsf{next}\ h\ u)\ <s : i>
\end{array}$$

Using this result, the behaviour preservation proof is straightforward:

$$\begin{array}{ll}
& ([\![p]\!]\ u)\ <s : i> \\
= & \quad \{\ \text{behaviour definition}\ \} \\
& \mathsf{at}\ \langle(\mathsf{next}\ u)\ s, i\rangle \\
= & \quad \{\ h\ \text{morphism}\ \} \\
& \mathsf{at}'\ \langle(h\ \mathsf{next}\, u)\ \ s, i\rangle \\
= & \quad \{\ \text{just proved}\ \} \\
& \mathsf{at}'\ \langle(\mathsf{next}\ h\ u)\ s, i\rangle \\
= & \quad \{\ \text{behaviour definition}\ \} \\
& ([\![p']\!]\ h\ u)\ <s : i>
\end{array}$$

$\square$

If there is always a morphism $[(p)]$ from any Mealy machine p to ω, and, as we have just shown, morphisms preserve behaviour, such a morphism is *unique*. This makes ω a very special Mealy machine: it is the only such system to which, from any other one, there is one and only one morphism. We say ω is the *final* Mealy machine. *Finality* is an example of an *universal* property [6] which, up to isomorphism, provides a complete characterisation of ω. In fact, suppose finality is shared by two Mealy machines ω and ω'. The *existence* component of the property gives rise to two morphisms h and h' connecting both machines in reverse directions. On the other hand, *uniqueness* implies $h \cdot h' = \mathsf{id}$ and $h' \cdot h = \mathsf{id}$, thus establishing h and h' as isomorphisms.

These two aspects of *finality* provide both a *definition* scheme and a *proof* principle upon which coalgebraic reasoning is based, as we shall illustrate in section 4.. Before that, however, let us see how these basic facts about Mealy machines lift to a more general setting.

3. Systems as Coalgebras

Interfaces. In the previous section Mealy machines (on I and O) were introduced as systems observed through the following interface (or 'lens'):

$$\bigcirc\!\frown\!\bigcirc \;=\; (O \times -)^I \tag{14}$$

Interface $\bigcirc\!\frown\!\bigcirc$ can be regarded as a mapping to decompose the 'observable-universe' U into an 'observation context' $(O \times U)^I$. Of course different interfaces give rise to different classes of systems. Let us consider some possibilities.

An extreme case is the 'opaque' interface $\bigcirc\!\frown\!\bigcirc = \mathbf{1}$: no matter what we try to observe through it, the outcome is always the same. A slightly more interesting lens is $\bigcirc\!\frown\!\bigcirc = \mathbf{2}$ which has the ability to classify states into two different classes (say, 'black' or 'white') and, therefore, to identify *subsets* of U, as a system for this interface will map elements of U to one or another element of $\mathbf{2}$. Should an arbitrary set O be chosen to colour the 'lens', the possible observations become more discriminating. A system with interface $\bigcirc\!\frown\!\bigcirc = O$ is a 'colouring' device in the sense that elements of the universe are classified (*i.e.*, regarded as distinct) by being assigned to different elements of O. A 'colour set' as $\mathbf{1}$, $\mathbf{2}$ or O above, acts as a *classifier* of the state space. Corresponding systems are *pure* observation structures. Naturally, the same 'universe' can be observed through different attributes and, furthermore, such observations can be carried out on parallel. Thus, the shape of a 'multi-attribute' lens is

$$\bigcirc\!\frown\!\bigcirc \;=\; \prod_{k \in K} O_k$$

where K is a finite set of attribute names and $\prod$ is the K-ary cartesian product.

[6]Because, roughly speaking, it singles out an entity (ω) among a family of 'similar' entities to which every other member of the family can be *reduced* or *traced back*. The study of *universal* properties is the 'essence' of category theory [Mac Lane, 1971].

Another interface which is not particularly useful is provided by the 'transparent' lens for which $\bigcirc\!\frown\!\bigcirc\ U = U$ holds. Any function $p : U \longrightarrow U$ is a system for this interface. This means that, by using p, the state space U can indeed be modified. But, on the other hand, the absence of attributes makes any meaningful observation impossible. Coming back to our 'lens' metaphor, the best we can say is just that *things happen*.

A better alternative consists of combining attributes with state modifiers, or update operations, to model the 'universe' evolution. The latter are called *actions* or *methods*. Such a combination leads to a richer stock of lens. We might consider, for example, that

1. *things happen and disappear* or *stop, i.e.*

$$\bigcirc\!\frown\!\bigcirc\ U \;=\; U + \mathbf{1}$$

   Systems for this interface are the *partial* functions.
2. *things happen and, in doing so, some of their attributes become visible, i.e.,* a (non trivial) output is produced:

$$\bigcirc\!\frown\!\bigcirc\ U \;=\; O \times U$$

3. *the evolution of things is triggered by some external stimulus, i.e.,* additional *input* is accepted:

$$\bigcirc\!\frown\!\bigcirc\ U \;=\; U^I$$

4. *we are not completely sure about what has happened,* in the sense that the evolution of the system being observed may be non deterministic. In this case, the lens above can be combined with

$$\bigcirc\!\frown\!\bigcirc\ U \;=\; \mathcal{P}U$$

The interfaces for Mealy or Moore machines, discussed above, arise exactly in this way, by gluing elementary 'lens' with set-theoretic constructions like cartesian product, disjoint union or exponentiation.

Coalgebras. For a particular universe U and observation structure $p : U \longrightarrow \bigcirc\!\frown\!\bigcirc\ U$, the pair $\langle U, p\rangle$ is called a $\bigcirc\!\frown\!\bigcirc$-*coalgebra*. A *morphism* connecting two such coalgebras is a function between their carriers which preserves the dynamics, *i.e.*, such that the following diagram commutes:

$$\begin{tikzcd} U \arrow[r, "p"] \arrow[d, "h"'] & \bigcirc\!\frown\!\bigcirc\ U \arrow[d, "\bigcirc\!\frown\!\bigcirc\ h"] \\ V \arrow[r, "p'"'] & \bigcirc\!\frown\!\bigcirc\ U' \end{tikzcd} \tag{15}$$

Note that diagram (11) is an instance of this general case. As another example, let $\bigcirc\!\frown\!\bigcirc \;=\; O$. In this case a morphism from p to q is just a 'colour-preserving' function, *i.e.*, $q \cdot h = p$ holds. This means that if two elements of the universe are grouped together by p, their images under h remain together when compared by q.

Functors. This diagram raises an important point: $\bigcirc\!\frown\!\bigcirc$ should be applicable not only to *sets*, but also to *functions* in order to give meaning to, *e.g.*, $\bigcirc\!\frown\!\bigcirc\ h$. The idea of an uniform transformation of both sets and functions is captured by the notion of a *functor* [7]. Recall (*cf.*, appendix A) that our models 'live' in a space of typed functions, something one could model as a graph with sets as nodes and set-theoretic functions as arrows. As functions (with the right types) can be composed and, for each set S, one may single out a function id_S (the identity on S) acting as the neutral element for composition, this working universe has the structure of a (partial) monoid. In this setting, a *functor* is simply a function T over this universe which preserves the graph and monoidal structure, *i.e.*, for each function $f : A \longrightarrow B$, $\mathsf{T}f$ is typed as $\mathsf{T}A \longrightarrow \mathsf{T}B$ and verifies:

$$\mathsf{T}\mathsf{id}_X \;= \mathsf{id}_{\mathsf{T}X} \tag{16}$$

$$\mathsf{T}(f \cdot g) \;= \mathsf{T}f \cdot \mathsf{T}g \tag{17}$$

Interfaces are thus modelled by functors. The 'transparent' lens corresponds to the identity functor Id defined by $\mathsf{Id}\ U = U$ and $\mathsf{Id}\ \mathsf{id}_U = \mathsf{id}_{\mathsf{Id}\ U}$. 'Coloured' lenses are instances of *constant* functors: for example, functor O maps every other set to O and every function $f : A \longrightarrow B$ to identity on O. More complex interfaces can be built combining such elementary functors with some form of 'addition' and 'multiplication' [8] giving rise to what is called the class of *polynomial* functors. Examples presented in this text, also resort to the direct exponential ($\mathsf{T}X = X^A$) and finite powerset ($\mathsf{T}X = \mathcal{P}X$) functors [9].

Systems and Behaviours Revisited. From the discussion above, a state-based system whose interface is captured by a functor T is modelled by a T-*coalgebra*, *i.e.*, a pair $\langle U, p : U \longrightarrow \mathsf{T}\ U\rangle$. The results we have analysed for the particular case of Mealy machines can now be formulated in this more general setting. In particular,

- Any morphism between two T-coalgebras preserve behaviour;
- Whenever the *space of behaviours* of a class of T-coalgebras can be turned on a T-coalgebra itself (written as $\omega_{\mathsf{T}} : \nu_{\mathsf{T}} \longrightarrow \mathsf{T}\nu_{\mathsf{T}}$), this is the *final* coalgebra: from any other T-coalgebra p there is a unique morphism $[\![(p)]\!]$ making the following diagram to commute:

$$\begin{tikzcd}
\nu_{\mathsf{T}} \arrow[r, "\omega_{\mathsf{T}}"] & \mathsf{T}\nu_{\mathsf{T}} \\
U \arrow[u, "{[\![(p)]\!]}"] \arrow[r, "p"'] & \mathsf{T}U \arrow[u, "{\mathsf{T}[\![(p)]\!]}"']
\end{tikzcd}$$

The universal property is, equivalently, captured by the following law:

$$k = [\![(p)]\!] \;\Leftrightarrow\; \omega_{\mathsf{T}} \cdot k = \mathsf{T}\ k \cdot p \tag{18}$$

[7] As most conceptual structures used in mathematics and computer science, this notion is borrowed from category theory, where it can be appreciated in its full genericity.

[8] *i.e.*, the product and sum functors on variables x and y, respectively. As expected, a pair $\langle x, y\rangle$ of sets or functions is mapped to $x \times y$ in the first case, and to $x + y$ in the second.

[9] These functors act on functions as follows: $f^A\ h = f \cdot h$ and $\mathcal{P}f\ X = \{f\ x |\ x \in X\}$.

Morphism $[\!(p)\!]$ applied to a state value u gives, of course, the (observable) behaviour of a sequence of p transitions starting at u. It is called the *coinductive extension* of p [Turi and Rutten, 1998] or the *anamorphism* generated by p [Meijer et al., 1991]. Being completely determined by p, p is also referred to as its *gene*.

As we have already remarked, the *existence* part of this universal property provides a *definition* principle for functions to spaces of behaviours (technically, carriers of final coalgebras). This is called definition by *co-recursion* and boils down to equipping the source of the function to be defined with a coalgebra to capture the 'one-step' dynamics in the behaviour generation process. Then the corresponding anamorphism gives the rest. The *uniqueness* part, on the other hand, entails a powerful *proof* principle — *coinduction*, also illustrated below.

For each interface T, inhabitants of the carrier ν_{T} of the corresponding final coalgebra are T-*behaviours*: all possible observations of systems with this interface are collected in ν_{T}. In this context, final coalgebras are called *coinductive* or *left datatypes* in [Hagino, 1987] or [Cockett and Spencer, 1992], *codata* and *codatatypes* in [Kieburtz, 1998], *final systems* in [Rutten, 2000] or *object types* in [Jacobs, 1996b]. Reasoning coalgebraically, which is the topic of the following section, is This explains why coalgebraic reasoning about state-based systems is often referred to as *coinductive* reasoning. Irrespective of the preferred designation, such is the topic of the following section.

4. Reasoning Coalgebraically

Basic Techniques. Law (18) is the basic tool for calculating with behaviours. Being an *universal* property, it asserts, for each *gene* coalgebra p, the *existence* and *uniqueness* of anamorphism $[\!(p)\!]$. Because $[\!(p)\!]$ is itself a morphism between coalgebras, it verifies the preservation condition expressed by the commutativity of diagram (15), now suitably instantiated to

$$\omega_{\mathsf{T}} \cdot [\!(p)\!] \;=\; \mathsf{T}[\!(p)\!] \cdot p \tag{19}$$

Moreover the following results, which are usually referred to as the *cancellation*, *reflection* and *fusion* law, respectively, are a direct consequence of (18).

$$\omega_{\mathsf{T}} \cdot [\!(p)\!] \;=\; \mathsf{T}\;[\!(p)\!] \cdot p \tag{20}$$

$$[\!(\omega_{\mathsf{T}})\!] \;=\; \mathsf{id}_{\nu_{\mathsf{T}}} \tag{21}$$

$$[\!(p)\!] \cdot h \;=\; [\!(q)\!] \quad \text{if} \quad p \cdot h \;=\; \mathsf{T}\; h \cdot q \tag{22}$$

Let us prove the last of this results which asserts the possibility of 'fusing' an arbitrary coalgebra morphism into a anamorphism. Typical applications of this law deal with the elimination of a pre-processing stage (with h) by incorporating it in the co-recursion pattern represented by $[\!(p)\!]$.

Proof.

$$\begin{aligned} & [\!(p)\!] \cdot h = [\!(q)\!] \\ = & \qquad \{ \text{ anamorphism universal characterisation — law (18) } \} \\ & \omega_{\mathsf{T}} \cdot [\!(p)\!] \cdot h = \mathsf{T}([\!(p)\!] \cdot h) \cdot q \end{aligned}$$

$$
\begin{array}{ll}
= & \{\ [\![p]\!] \text{ morphism (19)}\ \} \\
& \mathsf{T}[\![p]\!] \cdot p \cdot h = \mathsf{T}([\![p]\!] \cdot h) \cdot q \\
= & \{\ \text{as a functor } \mathsf{T} \text{ preserves composition — law (17)}\ \} \\
& \mathsf{T}[\![p]\!] \cdot p \cdot h = \mathsf{T}[\![p]\!] \cdot \mathsf{T}h \cdot q \\
\Leftarrow & \{\ \text{cancel } \mathsf{T}[\![p]\!] \text{ from both sides}\ \} \\
& p \cdot h = \mathsf{T}h \cdot q
\end{array}
$$

□

An Instructive Calculation. A nice illustration of the use of coinduction not only to prove systems' properties, but also to derive the required constructions, is provided by the following attempt to establish (the coalgebra version of) Lambek's Lemma. This is a 'folklore' result in categorical algebra [10], still very useful in practical calculations. It states that the dynamics of the final coalgebra is an isomorphism, *i.e.*, there is always an inverse to $\omega_\mathsf{T} : \nu_\mathsf{T} \longrightarrow \mathsf{T}\nu_\mathsf{T}$.

The proof below gives some intuition on the kind of calculational skills we are interested in. Moreover, as the result is valid for any functor T possessing a final coalgebra, the calculation is *generic* in the sense that all steps are parametrized by observation interface T.

Proof. We start by assuming the existence of such an inverse, written as α_T, which is expected to verify identities: $\alpha_\mathsf{T} \cdot \omega_\mathsf{T} = \mathsf{id}_{\nu_\mathsf{T}}$ and $\omega_\mathsf{T} \cdot \alpha_\mathsf{T} = \mathsf{id}_{\mathsf{T}\nu_\mathsf{T}}$. The proof idea is to take one of this requirements and use it to *conjecture* a definition for α_T (in programming terms, one would say an *implementation*). Then we check the validity of this conjecture by verifying with it the other requirement. Thus,

$$
\begin{array}{ll}
& \alpha_\mathsf{T} \cdot \omega_\mathsf{T} = \mathsf{id}_{\nu_\mathsf{T}} \\
= & \{\ \text{reflection — law (21)}\ \} \\
& \alpha_\mathsf{T} \cdot \omega_\mathsf{T} = [\![\omega_\mathsf{T}]\!] \\
= & \{\ \text{anamorphism universal characterisation — law (18)}\ \} \\
& \omega_\mathsf{T} \cdot \alpha_\mathsf{T} \cdot \omega_\mathsf{T} = \mathsf{T}(\alpha_\mathsf{T} \cdot \omega_\mathsf{T}) \cdot \omega_\mathsf{T} \\
= & \{\ \text{as a functor } \mathsf{T} \text{ preserves composition — law (17)}\ \} \\
& \omega_\mathsf{T} \cdot \alpha_\mathsf{T} \cdot \omega_\mathsf{T} = \mathsf{T}\alpha_\mathsf{T} \cdot \mathsf{T}\omega_\mathsf{T} \cdot \omega_\mathsf{T} \\
= & \{\ \text{cancel } \omega_\mathsf{T} \text{ from both sides}\ \} \\
& \omega_\mathsf{T} \cdot \alpha_\mathsf{T} = \mathsf{T}\alpha_\mathsf{T} \cdot \mathsf{T}\omega_\mathsf{T} \\
= & \{\ \text{anamorphism universal characterisation — law (18)}\ \} \\
& \alpha_\mathsf{T} = [\![\mathsf{T}\omega_\mathsf{T}]\!]
\end{array}
$$

We are half done! Notice, in particular, how the reflection law (21) was used to introduce an anamorphism in the calculation, whereas its 'normal' role seems to be exactly the opposite: getting

[10]The original statement of this lemma, formulated for *initial* algebras, appeared in [Barr, 1970], where it was credited to J. Lambek.

rid of anamorphisms by reducing them to the identity. This proof is then a nice example of the often neglected fact that calculations also require *insight* and that there are no proofs without ideas.

Now the verification step:

$$\begin{array}{ll} & \omega_{\mathsf{T}} \cdot \alpha_{\mathsf{T}} \\ = & \quad \{ \text{ replace } \alpha_{\mathsf{T}} \text{ by the derived conjecture } \} \\ & \omega_{\mathsf{T}} \cdot [\![(\mathsf{T}\omega_{\mathsf{T}})]\!] \\ = & \quad \{ \; [\![(\mathsf{T}\omega_{\mathsf{T}})]\!] \text{ is a morphism — law (19) } \} \\ & \mathsf{T}[\![(\mathsf{T}\omega_{\mathsf{T}})]\!] \cdot \mathsf{T}\omega_{\mathsf{T}} \\ = & \quad \{ \text{ as a functor } \mathsf{T} \text{ preserves composition — law (17) } \} \\ & \mathsf{T}([\![(\mathsf{T}\omega_{\mathsf{T}})]\!] \cdot \omega_{\mathsf{T}}) \\ = & \quad \{ \text{ just proved } \} \\ & \mathsf{T}\ \mathsf{id}_{\nu_{\mathsf{T}}} \\ = & \quad \{ \text{ as a functor } \mathsf{T} \text{ preserves identities — law (16) } \} \\ & \mathsf{id}_{(\mathsf{T}\mathsf{id}_{\nu_{\mathsf{T}}})} \end{array}$$

$\square$

In the next paragraph this kind of reasoning is illustrated with the simple object model specified by definition (5). Later, in section 5., more sophisticated systems will be considered — the reasoning principles and (conceptual) tools will, however, remain the same. We shall also take the opportunity to 'bring to life' some of our constructions by encoding them in a programming language.

Programming with Streams. Let us sum up, from a coalgebraic point of view, what we know about simple, one attribute, systems, modelled as coalgebras for functor [11] $\mathsf{T}_O = O \times -$ (recall discussion in section 2.). Its behaviours are O streams, *i.e.*, elements of O^ω. Moreover O^ω can be equipped with a coalgebra structure whose dynamics consists of two observers, accessing, respectively, the $head$ and $tail$ of the stream. Formally,

$$\langle O^\omega, \langle \mathsf{hd}, \mathsf{tl} \rangle : O^\omega \longrightarrow O \times O^\omega \rangle \tag{23}$$

which is final among such systems.

Coinductive types — *i.e., behaviours* — are directly supported by the CHARITY programming language [Cockett and Fukushima, 1992]. Final coalgebra (23) is declared as follows [12]:

```
data  S -> stream(O) =   hd: S -> O  |  tl: S -> S.
```

By finality, to define a function over a coinductive type it is enough to give its *gene*, which specifies the 'one-step' dynamics. As a first example consider how an infinite

[11]Notice how notation T_O makes explicit the output parameter which adds to the genericity of the constructions to follow.

[12]The notation is, hopefully, almost self-explanatory; a brief overview is provided in appendix B.

sequence of the same value (say, a) is produced. The typing of the envisaged function is $\mathsf{gen} : O \longrightarrow O^*$. If we want to explain how stream a^ω is generated, we shall probably say something like *in each step, given a value a, join it to the result and keep another copy of a to repeat this process*. Two remarks are now in order. First notice there is no *until* in this loop: the result, being an infinite behaviour, cannot to be shown 'once and for all', but *unfolded* step by step (that is where CHARITY lazy evaluation mechanism comes into scene). Second, and more fundamental, note there is a basic instruction in the description of the generation process: *in each step produce a pair $\langle a, a\rangle$* to be consumed along the way (the first a is to be joined to the stream, the second to go on). This basic instruction corresponds to a simple function $\vartriangle$: $O \longrightarrow O \times O$, known as the *diagonal* function and defined as $\vartriangle = \langle \mathsf{id}, \mathsf{id}\rangle$. Clearly, $\vartriangle$ is a coalgebra for T_O. Therefore it uniquely determines an anamorphism $[\!(\vartriangle)\!]$ which corresponds to the envisaged function, *i.e.*

$$\mathsf{gen} \;=\; [\!(\vartriangle)\!] \tag{24}$$

Function $\vartriangle$ carries the 'genetic inheritance' of the generating process, as represented in the diagram below. From a programming point of view recognising this fact is the *eureka* step — the one requiring *insight*; all the rest is for free.

$$\begin{tikzcd}
O^\omega \arrow[r, "\langle \mathsf{hd},\mathsf{tl}\rangle"] & O \times O^\omega \\
O \arrow[u, "\mathsf{gen}"] \arrow[r, "\vartriangle"] & O \times O \arrow[u, "\mathsf{id}\times\mathsf{gen}"']
\end{tikzcd}$$

A simple calculation provides a definition of gen making explicit the implicit self-reference:

$$\begin{array}{ll}
 & (\mathsf{id} \times \mathsf{gen})\cdot \vartriangle \;=\; \langle \mathsf{hd}, \mathsf{tl}\rangle \cdot \mathsf{gen} \\
= & \quad \{ \;\vartriangle \text{ definition} \} \\
 & (\mathsf{id} \times \mathsf{gen}) \cdot \langle \mathsf{id}, \mathsf{id}\rangle \;=\; \langle \mathsf{hd}, \mathsf{tl}\rangle \cdot \mathsf{gen} \\
= & \quad \{ \;\times \text{ absorption and fusion — laws (39) and (38)} \} \\
 & \langle \mathsf{id}, \mathsf{gen}\rangle \;=\; \langle \mathsf{hd} \cdot \mathsf{gen}, \mathsf{tl} \cdot \mathsf{gen}\rangle \\
= & \quad \{ \text{ structural equality — law (40)} \} \\
 & \mathsf{hd} \cdot \mathsf{gen} = \mathsf{id} \;\;\wedge\;\; \mathsf{tl} \cdot \mathsf{gen} = \mathsf{gen} \\
= & \quad \{ \text{ going pointwise} \} \\
 & \mathsf{hd}\ (\mathsf{gen}\ a) = a \;\;\wedge\;\; \mathsf{tl}\ (\mathsf{gen}\ a) = \mathsf{gen}\ a
\end{array}$$

There is a general 'lesson' to be learnt from the specification of gen in this format, which is true for every function defined as an anamorphism: the function is given by specifying its behaviour under all the observers [13]. It should be noted, however, that this transformation is unnecessary: once the 'gene' has been identified the problem is solved. In CHARITY, for example, function gen is defined exactly as in (24), as the language forbids self-reference:

[13]Recall that a function over an inductive type is specified in a dual way by describing its effect on the constructors.

```
def gen: O  -> stream(O)
        = n => (| n => head: n
               |      tail: n
               |) n.
```

The same definitional principle can be applied to generate functions whose both source and target are behaviours. For example, a combinator for merging two streams is defined by

$$\begin{tikzcd}
O^\omega \arrow[r, "\langle \mathsf{hd},\mathsf{tl} \rangle"] & O \times O^\omega \\
O^\omega \times O^\omega \arrow[u, "\mathsf{merge} = [\![g]\!]"] \arrow[r, "g"] & O \times (O^\omega \times O^\omega) \arrow[u, "\mathsf{id} \times \mathsf{merge}"']
\end{tikzcd}$$

where $g = \langle \mathsf{hd} \cdot \pi_1, \mathsf{s} \cdot (\mathsf{tl} \times \mathsf{id}) \rangle$. This means that in each step the head of the first stream is taken as the head of the result, and the generating process is prepared to the following iteration by taking the second stream, the tail of the first and swapping them (s is $\times$ commutativity isomorphism). In CHARITY,

```
def merge: stream(O) * stream(O)  -> stream(O)
        = x => (| (s,t) => head: head s
               |          tail: (t, tail s)
               |) x.
```

Suppose, now, we want to obtain a stream of as and bs strictly interleaved, *i.e.*$(ab)^\omega$. Two possibilities emerge. One of them resorts to a particular generating function twist specified as

$$\begin{tikzcd}
O^\omega \arrow[r, "\langle \mathsf{hd},\mathsf{tl} \rangle"] & O \times O^\omega \\
O \times O \arrow[u, "\mathsf{twist} = [\![\langle \pi_1, \mathsf{s} \rangle]\!]"] \arrow[r, "\langle \pi_1, \mathsf{s} \rangle"] & O \times (O \times O) \arrow[u, "\mathsf{id} \times \mathsf{twist}"']
\end{tikzcd}$$

and encoded in CHARITY as

```
def twist: O * O  -> stream(O)
        = x => (| (m,n) => head: m
               |          tail: (n,m)
               |) x.
```

Alternatively one may combine a^ω and b^ω with merge. As a further illustration of coalgebraic reasoning, let us show that both approaches are equivalent, *i.e.*,

$$\mathsf{merge} \cdot (\mathsf{gen} \times \mathsf{gen}) = \mathsf{twist} \tag{25}$$

Proof.

$$
\begin{array}{ll}
& \mathsf{merge} \cdot (\mathsf{gen} \times \mathsf{gen}) \;=\; \mathsf{twist} \\
= & \qquad \{ \text{ merge definition } \} \\
& [\![(\langle \mathsf{hd} \cdot \pi_1, \mathsf{s} \cdot (\mathsf{tl} \times \mathsf{id}) \rangle)]\!] \cdot (\mathsf{gen} \times \mathsf{gen}) \;=\; \langle \pi_1, \mathsf{s} \rangle \\
\Leftarrow & \qquad \{ \text{ anamorphism fusion — law (22) } \} \\
& \langle \mathsf{hd} \cdot \pi_1, \mathsf{s} \cdot (\mathsf{tl} \times \mathsf{id}) \rangle \cdot (\mathsf{gen} \times \mathsf{gen}) \;=\; \mathsf{id} \times (\mathsf{gen} \times \mathsf{gen}) \cdot \langle \pi_1, \mathsf{s} \rangle \\
= & \qquad \{ \times \text{ absorption and reflection — laws (39) and (36) } \} \\
& \langle \mathsf{hd} \cdot \mathsf{gen} \cdot \pi_1, \mathsf{s} \cdot ((\mathsf{tl} \cdot \mathsf{gen}) \times \mathsf{gen}) \rangle \;=\; \mathsf{id} \times (\mathsf{gen} \times \mathsf{gen}) \cdot \langle \pi_1, \mathsf{s} \rangle \\
= & \qquad \{ \mathsf{tl} \cdot \mathsf{gen} = \mathsf{gen} \text{ and } \mathsf{hd} \cdot \mathsf{gen} = \mathsf{id} \} \\
& \langle \pi_1, \mathsf{s} \cdot (\mathsf{gen} \times \mathsf{gen}) \rangle \;=\; \mathsf{id} \times (\mathsf{gen} \times \mathsf{gen}) \cdot \langle \pi_1, \mathsf{s} \rangle \\
= & \qquad \{ \times \text{ absorption — law (39) } \} \\
& \langle \pi_1, \mathsf{s} \cdot (\mathsf{gen} \times \mathsf{gen}) \rangle \;=\; \langle \pi_1, (\mathsf{gen} \times \mathsf{gen}) \cdot \mathsf{s} \rangle \\
= & \qquad \{ \mathsf{s} \text{ is natural, } \textit{i.e.,} \ (f \times g) \cdot \mathsf{s} = \mathsf{s} \cdot (g \times f) \} \\
& \langle \pi_1, \mathsf{s} \cdot (\mathsf{gen} \times \mathsf{gen}) \rangle \;=\; \langle \pi_1, \mathsf{s} \cdot (\mathsf{gen} \times \mathsf{gen}) \rangle
\end{array}
$$

□

Observational Equivalence and Bisimulation. The state space of a system (or, in technical terms, the carrier of a coalgebra) is accessed through the observers encoded in its interface. Therefore, states that exhibit the same behaviour should always be identified. Such an indistinguishability relation on states is called *observational equivalence* and defined by

$$u \sim v \iff [\![(p)]\!]\, u = [\![(q)]\!]\, v \tag{26}$$

for u and v in the carriers of coalgebras $\langle U, p\rangle$ and $\langle V, q\rangle$, respectively. This definition agrees with our intuition that the final coalgebra is the coalgebra of *all* behaviours.

A somewhat simpler way of establishing *observational equivalence*, which moreover has the advantage of not depending on the existence of final coalgebras, is to look for a morphism h such that one of the states is the h-image of the other. Once conjectured, h determines a relation $R \subseteq U \times V$ such that

$$\langle u, v\rangle \in R \Rightarrow u \sim v \tag{27}$$

Such a relation is, of course, the *graph* of h, *i.e.*, $\{\langle x, h\ x\rangle \mid x \in U\}$.

Can this idea be generalised? More precisely, what properties must a relation R have so that one may conclude $u \sim v$ simply by checking whether $\langle u, v\rangle$ is in R?

Let us consider again systems whose behaviours are streams. Given two states u and v and coalgebras $p = \langle U, \langle \mathsf{at}_p, \mathsf{m}_p\rangle\rangle$ and $q = \langle V, \langle \mathsf{at}_q, \mathsf{m}_q\rangle\rangle$, we want check if u under p and v under q behave similarly, *i.e.*, generate the same streams. A constructive way of carrying out such a verification consists of

- first checking the attributes: $\mathsf{at}_p\ u = \mathsf{at}_q\ v$,

- then, assuring that the continuation states produced by $\mathsf{m}_p\ u$ and $\mathsf{m}_q\ v$, respectively, support identical verification.

This equivales to find out a relation $R \subseteq U \times V$ such that

$$\langle u, v\rangle \in R \ \Rightarrow\ \mathsf{at}_p\ u = \mathsf{at}_q\ v\ \wedge\ \langle \mathsf{m}_p\ u, \mathsf{m}_q\ v\rangle \in R \tag{28}$$

We may then conclude that, to serve as a *witness* for observational equivalence, a relation R has to be *closed* under the coalgebra dynamics. This sort of relations are well-known in the literature on the semantics of concurrency under the generic designation of *bisimulations*. The name conveys the intuition that two systems running from equivalent states have the ability to simulate each other. Can we arrive at a *generic* notion of bisimulation, *i.e.*, independent of the system's interface? Let us see if we can rewrite (28) into a form easier to generalise. This entails the need for both eliminating variables and making explicit the structure involved. We thus proceed by conjecturing a coalgebra structure in R:

$$\rho = \langle \mathsf{at}_\rho, \mathsf{m}_\rho\rangle : R \longrightarrow O \times R \tag{29}$$

such that

$$\mathsf{at}_\rho = \mathsf{at}_p \cdot \pi_1 = \mathsf{at}_q \cdot \pi_2 \quad \wedge \quad \mathsf{m}_\rho = \mathsf{m}_p \times \mathsf{m}_q \tag{30}$$

Now, a simple calculation yields:

$$\begin{array}{cl}
 & \mathsf{at}_\rho = \mathsf{at}_p \cdot \pi_1 = \mathsf{at}_q \cdot \pi_2\ \wedge\ \mathsf{m}_\rho = \mathsf{m}_p \times \mathsf{m}_q \\
\equiv & \quad \{\ \times \text{ reflection — law (36)}\} \\
 & \mathsf{at}_\rho = \mathsf{at}_p \cdot \pi_1 = \mathsf{at}_q \cdot \pi_2\ \wedge\ \mathsf{m}_\rho = \mathsf{m}_p \times \mathsf{m}_q \\
\equiv & \quad \{\ \times \text{ fusion and absorption — laws (38) and (39)}\} \\
 & \mathsf{at}_\rho = \mathsf{at}_p \cdot \pi_1 = \mathsf{at}_q \cdot \pi_2\ \wedge\ \langle \pi_1 \cdot \mathsf{m}_\rho, \pi_2 \cdot \mathsf{m}_\rho\rangle = \langle \mathsf{m}_p \cdot \pi_1, \mathsf{m}_q \cdot \pi_2\rangle \\
\equiv & \quad \{\ \text{structural equality — law (40)}\} \\
 & \langle \mathsf{at}_\rho, \pi_1 \cdot \mathsf{m}_\rho\rangle = \langle \mathsf{at}_p \cdot \pi_1, \mathsf{m}_p \cdot \pi_1\rangle\ \wedge\ \langle \mathsf{at}_\rho, \pi_2 \cdot \mathsf{m}_\rho\rangle = \langle \mathsf{at}_q \cdot \pi_2, \mathsf{m}_q \cdot \pi_2\rangle \\
\equiv & \quad \{\ \times \text{ fusion and absorption — laws (38) and (39)}\} \\
 & (\mathsf{id} \times \pi_1) \cdot \langle \mathsf{at}_\rho, \mathsf{m}_\rho\rangle = \langle \mathsf{at}_p, \mathsf{m}_p\rangle \cdot \pi_1\ \wedge\ (\mathsf{id} \times \pi_2) \cdot \langle \mathsf{at}_\rho, \mathsf{m}_\rho\rangle = \langle \mathsf{at}_q, \mathsf{m}_q\rangle \cdot \pi_2
\end{array}$$

The last pair of equations asserts the commutativity of the following diagram:

$$\begin{tikzcd}
U \arrow[d, "p"'] & R \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "\rho"'] & V \arrow[d, "q"'] \\
O \times U & O \times R \arrow[l, "\mathsf{id}\times\pi_1"'] \arrow[r, "\mathsf{id}\times\pi_2"] & O \times V
\end{tikzcd}$$

This fact can be shown to hold in a generic context: a relation on the carriers of T-coalgebras p and q is a (T)-*bisimulation* if it can be equipped with a coalgebra structure ρ such that projections π_1 and π_2 lift to T-morphisms, as expressed by the commutativity of the following diagram:

$$\begin{tikzcd}
U \arrow[d, "p"'] & R \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "\rho"'] & V \arrow[d, "q"'] \\
\mathsf{T}\,U & \mathsf{T}\,R \arrow[l, "\mathsf{T}\ \pi_1"'] \arrow[r, "\mathsf{T}\ \pi_2"] & \mathsf{T}\,V
\end{tikzcd}$$

i.e.,

$$\mathsf{T}\,\pi_1 \cdot \rho = p \cdot \pi_1 \;\wedge\; \mathsf{T}\,\pi_2 \cdot \rho = q \cdot \pi_2 \tag{31}$$

As discussed in the next section, the behaviours of non-deterministic systems specified as coalgebras for $\mathsf{T} = \mathcal{P}(O \times -)$ correspond to what is usually called a *process* in process calculi such CCS [Milner, 1980, Milner, 1989] or CSP [Hoare, 1985]. Originally the notion of bisimulation was introduced in the context of such calculi to relate processes upon which observation produces equal results and this is maintained along all possible transitions — *cf.*, Park's landmark paper [Park, 1981]. Later [Aczel and Mendler, 1988] gave a categorical definition [14] of bisimulation which applies, not only to the kind of transition systems underlying the operational semantics of process calculi, but also to arbitrary coalgebras. Bisimulation *acquired a shape*: the shape of the chosen observation interface T.

As one would expect, instantiating (31) with $\mathsf{T} = \mathcal{P}(O \times -)$ leads, after a few steps in a simple calculation, to

$$\langle u, v\rangle \in R \;\Rightarrow\; \forall_{\langle o,x\rangle \in p\,u} \,.\, \exists_{\langle o,y\rangle \in q\,v} \,.\, \langle x, y\rangle \in R \;\wedge\; \forall_{\langle o,y\rangle \in q\,v} \,.\, \exists_{\langle o,x\rangle \in p\,u} \,.\, \langle x, y\rangle \in R$$

which corresponds to what is called *strict bisimulation* in the CCS literature. On the other hand, instantiating (31) with the identity functor (the 'transparent lens') leads to

$$\begin{array}{cl}
 & \pi_1 \cdot \rho = p \cdot \pi_1 \;\wedge\; \pi_2 \cdot \rho = q \cdot \pi_2 \\
\equiv & \qquad \{ \text{ structural equality — law (40)} \} \\
 & \langle \pi_1 \cdot \rho, \pi_2 \cdot \rho\rangle \;=\; \langle p \cdot \pi_1, q \cdot \pi_2\rangle \\
\equiv & \qquad \{ \;\times \text{ fusion and absorption — laws (38) and (39)} \} \\
 & \langle \pi_1, \pi_2\rangle \cdot \rho \;=\; p \times q \cdot \langle \pi_1, \pi_2\rangle \\
\equiv & \qquad \{ \;\times \text{ reflection — law (36)} \} \\
 & \rho \;=\; p \times q \\
\equiv & \qquad \{ \text{ going pointwise } \} \\
 & \langle u, v\rangle \in R \;\;\mathsf{iff}\;\; u \in U \wedge v \in V
\end{array}$$

which means that any relation preserving p and q dynamics is a bisimulation on those coalgebras.

Remark. Bisimulations provide a 'relational' view of coalgebra morphisms. In fact, as discussed above, the graph of a T-coalgebra morphism is a T-bisimulation. This and several other results on bisimulations are proved in [Rutten, 2000]. In particular, it is shown that the *converse* of a bisimulation and, under some conditions on T, the *composition* of two bisimulations are still bisimulations. As a corollary, the (relational) *direct* and *inverse* images of a bisimulation, as well as the *kernel* of a coalgebra morphism, are bisimulations as well.

Bisimulation entails, thus, a *local* proof theory for *observational equivalence*, which is widely used in practice and in a variety of contexts. Usually in the coalgebra literature what is understood by a *coinductive proof* is the explicit construction of a

[14] The definition used in this text is an instance of it assuming a set-theoretic semantic universe.

bisimulation containing the pair of states one want to prove equivalent (see, for example, [Rutten, 2000] or [Jacobs, 2002]). In general, however, we favour the calculational approach exemplified above which resorts explicitly to the universal property, because it is closer to the calculation of functional programs (as in, *e.g.*, [Bird and Moor, 1997]). To a large extent this choice is, however, a matter of taste: the two methods (both rooted in *uniqueness* part of the universal property) are the two sides of the same coin — the theorem which (for a large class of functors, which include the ones considered in this text) establishes final coalgebras as *full abstract* with respect to bisimulation (for a proof see, *e.g.*, [Turi and Rutten, 1998]).

5. Application: Process Calculi

Motivation. This section outlines a medium-size case study in the use of coalgebraic structures to model and reason about dynamical systems which corresponds to some recent research of the author documented in [Barbosa, 2001] and [Barbosa and Oliveira, 2002]. The object of study are classical *process calculi* in the tradition of CCS [Milner, 1989] for specification of concurrent computation. Our goal is to apply the concepts and techniques discussed above to the design of process combinators and the 'discovery' of their laws.

Our approach relies on the representation of processes as inhabitants of coinductive types, *i.e.*, final coalgebras for suitable functors. Final semantics for processes is an active research area, namely after Aczel's landmark paper [Aczel, 1993]. Our emphasis is, however, actually 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. We believe that the proposed approach has a number of advantages:

- First of all it provides a *uniform* treatment of processes and other computational structures, *e.g.*, data structures, both represented as categorical types for functors capturing interfaces of, respectively, observers and constructors (see section 6.). Placing data and behaviour at a similar level conveys the idea that process models can be chosen and specified according to a given application area, in the same way that a suitable data structure is defined to meet a particular engineering problem. As a prototyping platform, CHARITY is a sober tool in which different process combinators can be defined, interpreters for the associated languages quickly developed and the expressiveness of different calculi compared with respect to the intended applications.
- The approach is independent of any particular process calculus and makes explicit the different ingredients present in the design of any such calculi. In particular, structural aspects of the underlying *behaviour model* (*e.g.*, the dichotomies such as active *vs* reactive, deterministic *vs* non-deterministic) become clearly separated from the *interaction* structure which defines the synchronisation discipline.
- Proofs are carried out in a purely calculational (basically *equational* and *pointfree*) style, therefore circumventing the explicit construction of bisimulations used in most of the literature on process calculi.

Processes. The operational semantics of a process calculus is usually given in terms of a transition relation $\xrightarrow{a}$ over processes, indexed by a set Act of actions, in which a process gets committed, and the resulting 'continuations', *i.e.*, the behaviours subsequently exhibited. A first, basic design decision concerns the definition of what should be understood by such a collection. As a rule, it is defined as a *set*, in order to express non-determinism. Other, more restrictive, possibilities consider a sequence or even just a single continuation, modelling, respectively, 'ordered' non-determinism or determinism. In general, this underlying behaviour model can be represented by a functor B.

An orthogonal decision concerns the intended interpretation of the transition relation, which is usually left implicit or underspecified in process calculi. We may, however, distinguish between

- An 'active' interpretation, in which a transition $p \xrightarrow{a} q$ is informally characterised as 'p evolves to q by performing an action a', both q and a being solely determined by p.
- A 'reactive' interpretation, informally reading 'p reacts to an external stimulus a by evolving to q'.

Processes will then be taken as inhabitants of the carrier of the final coalgebra $\omega : \nu \longrightarrow \mathsf{T}\ \nu$, with T defined as $\mathsf{B}\ (Act \times \mathsf{Id})$, in the first case, and $(\mathsf{B}\ \mathsf{Id})^{Act}$, in the second. To illustrate our approach, we shall focus on the particular case where B is the finite powerset functor and the 'active' interpretation is adopted. The transition relation, for this case, is given by $p \xrightarrow{a} q \quad \mathsf{iff} \quad \langle a, q\rangle \in \omega\ p$.

The restriction to the finite powerset avoids cardinality problems and assures the existence of a final coalgebra for T [15]. This restricts us to *image-finite* processes, a not too severe restriction in practice which may be partially circumvented by a suitable definition of the structure of Act. For instance, by taking Act as *channel* names through which data flows. This corresponds closely to 'CCS with value passing' [Milner, 1989]. Therefore, only the set of channels, and not the messages (seen as pairs channel/data), must remain finite. In fact, as detailed below, an algebraic structure should be imposed upon the set Act of actions in order to capture different interaction disciplines. This will be called an *interaction structure* in the sequel.

Down to the programming level, we shall start by declaring a process space as the coinductive type `Pr(A)`, parametrized by a specification A of the interaction structure:

```
data C -> Pr(A) = bh: C -> set(A * C).
```

where `set` stands for a suitable implementation of (finite) sets.

Dynamic Combinators. The cornerstone in the design of a process calculi is the judicious selection of a (hopefully small) set of process combinators. In [Milner, 1989], R. Milner classifies them into two distinct groups. The first group consists of all combinators which persist through action, *i.e.*, which are present before and after a transition occurs. They are called *static* and used to set up process' architectures, specifying

[15] Lambek lemma, proved in the previous section, implies that a final coalgebra for the unrestricted powerset functor in a universe of sets cannot exist, as it would violate Cantor's theorem. In fact there are several constructions and results of coalgebra theory which depend on properties of functor T. Even if we have restrict ourselves to a particularly well-behaved class of functors, the reader is referred to specialised references to get the 'big picture' ([Rutten, 2000] is a good starting point).

how their components are linked and which parts of their interface are public or private. *Dynamic* combinators, on the other hand, are 'consumed' on action occurrence, disappearing from the expression representing the process continuation. In this paragraph the usual CCS dynamic combinators — *i.e., inaction, prefix* and non-deterministic *choice* — are defined as operators on the final universe of processes considered above. Notice that, being non recursive, they have a direct (coinductive) definition which depends solely on the chosen process structure. Therefore, the inactive process is represented as a constant $\mathsf{nil} : \mathbf{1} \longrightarrow \nu$ upon which no relevant observation can be made. Prefix gives rise to an Act-indexed family of operators $a. : \nu \longrightarrow \nu$, with $a \in Act$. Finally, the possible actions of the non deterministic choice of two processes p and q corresponds to the collection of all actions allowed for p and q. Therefore, the operator $+ : \nu \times \nu \longrightarrow \nu$ can only be defined over a process structure in which observations form a collection. Formally,

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

where $\mathsf{sing} = \lambda x\,.\ \{x\}$ and $\mathsf{label}_a = \lambda x\,.\ \langle a, x\rangle$. These definitions are directly translated to CHARITY as functions `bnil`, `bpre` and `bcho`, respectively:

```
def bnil: 1 -> Pr(A)
    = () => (bh: empty).

def bpre: A * Pr(A) -> Pr(A)
    = (a, t)  => (bh: sing(a,t)).

def bcho: Pr(A) * Pr(A)  -> Pr(A)
    = (t1, t2) => (bh: union(bh t1, bh t2)).
```

Dynamic combinators satisfy a number of laws. For example, structure $\langle \nu; +, \mathsf{nil}\rangle$ forms an Abelian idempotent monoid, a fact that is proved in the literature (*e.g.,* [Milner, 1989]) by exhibiting a suitable bisimulation. All proofs of this sort can be made, however, by simple equational reasoning. Moreover, finality turns ω into an isomorphism and therefore, to prove $e = e'$ it is enough to show that $\omega \cdot e = \omega \cdot e'$. To illustrate the proposed proof style, consider the proof of two simple results: $+$ commutativity and associativity [16], *i.e.,*

$$+ \cdot \mathsf{s} = + \tag{32}$$

and

$$+ \cdot (+ \times \mathsf{id}) = + \cdot (\mathsf{id} \times +) \cdot \mathsf{a} \tag{33}$$

[16]In the sequel, process properties are stated pointfree, for which we shall resort to standard natural isomorphisms in Set. In particular, associativity, commutativity and product left and right units, will be denoted by $\mathsf{a} : (A \times B) \times C \longrightarrow A \times (B \times C)$, $\mathsf{s} : A \times B \longrightarrow B \times A$, $\mathsf{r} : \mathbf{1} \times A \longrightarrow A$ and $\mathsf{l} : A \times \mathbf{1} \longrightarrow A$, respectively. The converse of an isomorphism i is written as i°.

Proof. The following calculation establishes $+$ commutativity:

$$\begin{array}{cl}
 & \omega \cdot + \cdot \mathsf{s} \\
\equiv & \quad \{ + \text{ definition} \} \\
 & \cup \cdot (\omega \times \omega) \cdot \mathsf{s} \\
\equiv & \quad \{ \mathsf{s} \text{ natural} \} \\
 & \cup \cdot \mathsf{s} \cdot (\omega \times \omega) \\
\equiv & \quad \{ \cup \text{ commutative} \} \\
 & \cup \cdot (\omega \times \omega) \\
\equiv & \quad \{ + \text{ definition} \} \\
 & \omega \cdot +
\end{array}$$

The proof of associativity also relies on the properties of set union, but is a bit longer. Notice, however, that all elementary steps are shown.

$$\begin{array}{cl}
 & \omega \cdot + \cdot (+ \times \mathsf{id}) \\
\equiv & \quad \{ + \text{ definition} \} \\
 & \cup \cdot (\omega \times \omega) \cdot (+ \times \mathsf{id}) \\
\equiv & \quad \{ \times \text{ as a functor respects composition} \} \\
 & \cup \cdot (\omega \cdot + \times \omega) \\
\equiv & \quad \{ + \text{ definition} \} \\
 & \cup \cdot (\cup \cdot (\omega \times \omega)) \times \omega \\
\equiv & \quad \{ \times \text{ as a functor respects composition} \} \\
 & \cup \cdot (\cup \times \mathsf{id}) \cdot ((\omega \times \omega) \cdot \omega) \\
\equiv & \quad \{ \mathsf{a} \text{ is an isomorphism} \} \\
 & \cup \cdot (\cup \times \mathsf{id}) \cdot ((\omega \times \omega) \cdot \omega) \cdot \mathsf{a}^{\circ} \cdot \mathsf{a} \\
\equiv & \quad \{ \mathsf{a}^{\circ} \text{ is natural, } \textit{i.e.} \ ((f \times g) \cdot h) \cdot \mathsf{a}^{\circ} = \mathsf{a}^{\circ} \cdot (f \times (g \times h)) \} \\
 & \cup \cdot (\cup \times \mathsf{id}) \cdot \mathsf{a}^{\circ} \cdot (\omega \times (\omega \times \omega)) \cdot \mathsf{a} \\
\equiv & \quad \{ \cup \text{ associative} \} \\
 & \cup \cdot (\mathsf{id} \times \cup) \cdot (\omega \times (\omega \times \omega)) \cdot \mathsf{a} \\
\equiv & \quad \{ \times \text{ as a functor respects composition} \} \\
 & \cup \cdot (\omega \times (\cup \cdot (\omega \times \omega)) \cdot \mathsf{a} \\
\equiv & \quad \{ + \text{ definition} \} \\
 & \cup \cdot (\omega \times \omega \cdot +) \cdot \mathsf{a} \\
\equiv & \quad \{ \times \text{ as a functor respects composition} \} \\
 & \cup \cdot (\omega \times \omega) \cdot (\mathsf{id} \times +) \cdot \mathsf{a} \\
\equiv & \quad \{ + \text{ definition} \} \\
 & \omega \cdot + \cdot (\mathsf{id} \times +) \cdot \mathsf{a}
\end{array}$$

□

Static Combinators. Persistence through action occurrence leads to the *co-recursive* definition of *static* combinators. This means they arise as anamorphisms generated by suitable 'gene' coalgebras. *Interleaving*, *restriction* and *renaming* are examples of *static* combinators, which, moreover, depend only on the process structure. On the other hand, *synchronous product* and *parallel composition* also rely on the interaction structure underlying the calculus. In each case, we give both a mathematical definition of the combinator and the corresponding CHARITY code. There is a direct correspondence between these two levels. Some 'housekeeping' morphisms, like the *diagonal* $\triangle$, used in the former, are more conveniently handled by the CHARITY term logic.

Interleaving. Although *interleaving*, a binary operator $||| : \nu \times \nu \longrightarrow \nu$, is not considered as a combinator in most process calculi, it is the simplest form of 'parallel' aggregation in the sense that it is independent of any particular interaction discipline. The definition below captures the intuition that the observations over the interleaving of two processes correspond to all possible interleavings of the observations of its arguments. Thus, one defines $||| \ = \ [\![\alpha_{|||}]\!]$, where

$$\begin{aligned}
\alpha_{|||} \ = \ \nu \times \nu & \xrightarrow{\triangle} (\nu \times \nu) \times (\nu \times \nu) \\
& \xrightarrow{(\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega)} (\mathcal{P}(Act \times \nu) \times \nu) \times (\nu \times \mathcal{P}(Act \times \nu)) \\
& \xrightarrow{\tau_r \times \tau_l} \mathcal{P}(Act \times (\nu \times \nu)) \times \mathcal{P}(Act \times (\nu \times \nu)) \\
& \xrightarrow{\cup} \mathcal{P}(Act \times (\nu \times \nu))
\end{aligned}$$

Notice that, technically, morphisms τ_r and τ_l above are, respectively, the right and left strength associated to functor $\mathcal{P}(Act \times -)$. For our purposes their definition in elementary terms is enough (plus the observation that both verify a *natural* property). Thus, define

$$\tau_r \ \langle X, c \rangle \ = \ \{ \langle a, \langle x, c \rangle \rangle | \ \langle a, x \rangle \in X \} \tag{34}$$

which is straightforwardly encoded in CHARITY

```
def taur  =  (s,t) => set{(a,x) => (a, (x,t))} s.
```

The definition of τ_l is similar.

The CHARITY code for this combinator is

```
def bint: Pr(Ac(L)) * Pr(Ac(L))  -> Pr(Ac(L))
 = (t1, t2) =>
   (| (r1,r2) => bh: union(taur(bh r1, r2), taul(bh r2, r1))
    |) (t1,t2).
```

As one could expect, proofs of properties of static combinators often resort to the anamorphism fusion law (22). The proof of commutativity is carried out below, as an example. Commutativity states that $p|||q = q|||p$, *i.e.*, going pointfree, that $||| \cdot \mathsf{s} = |||$. Thus

Proof.

$$
\begin{array}{cl}
& ||| \cdot \mathsf{s} \;=\; ||| \\
\equiv & \qquad \{ \; ||| \text{ definition} \} \\
& [\!(\alpha_{|||})\!] \cdot \mathsf{s} \;=\; [\!(\alpha_{|||})\!] \\
\Leftarrow & \qquad \{ \; \text{anamorphism fusion — law (22)} \} \\
& \alpha_{|||} \cdot \mathsf{s} \;=\; \mathcal{P}(\mathsf{id} \times \mathsf{s}) \cdot \alpha_{|||}
\end{array}
$$

The last equation is justified by the following calculation:

$$
\begin{array}{cl}
& \alpha_{|||} \cdot \mathsf{s} \\
= & \qquad \{ \; ||| \text{ 'gene' definition} \} \\
& \cup \cdot (\tau_r \times \tau_l) \cdot ((\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega)) \cdot \vartriangle \cdot \mathsf{s} \\
= & \qquad \{ \; \vartriangle \text{ is natural} \} \\
& \cup \cdot (\tau_r \times \tau_l) \cdot ((\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega)) \cdot (\mathsf{s} \times \mathsf{s}) \cdot \vartriangle \\
= & \qquad \{ \; \text{s is natural} \} \\
& \cup \cdot (\tau_r \times \tau_l) \cdot (\mathsf{s} \times \mathsf{s}) \cdot ((\mathsf{id} \times \omega) \times (\omega \times \mathsf{id})) \cdot \vartriangle \\
= & \qquad \{ \; \times \text{ functor} \} \\
& \cup \cdot (\tau_r \cdot \mathsf{s} \times \tau_l \cdot \mathsf{s}) \cdot ((\mathsf{id} \times \omega) \times (\omega \times \mathsf{id})) \cdot \vartriangle \\
= & \qquad \{ \; \tau_r \text{ are } \tau_l \text{ natural} \} \\
& \cup \cdot (\mathcal{P}(\mathsf{id} \times \mathsf{s}) \cdot \tau_l \times \mathcal{P}(\mathsf{id} \times \mathsf{s}) \cdot \tau_r) \cdot ((\mathsf{id} \times \omega) \times (\omega \times \mathsf{id})) \cdot \vartriangle \\
= & \qquad \{ \; \times \text{ functor} \} \\
& \cup \cdot (\mathcal{P}(\mathsf{id} \times \mathsf{s}) \times \mathcal{P}(\mathsf{id} \times \mathsf{s})) \cdot (\tau_l \times \tau_r) \cdot ((\mathsf{id} \times \omega) \times (\omega \times \mathsf{id})) \cdot \vartriangle \\
= & \qquad \{ \; \cup \text{ is natural} \} \\
& \mathcal{P}(\mathsf{id} \times \mathsf{s}) \cdot \cup \cdot (\tau_l \times \tau_r) \cdot ((\mathsf{id} \times \omega) \times (\omega \times \mathsf{id})) \cdot \vartriangle \\
= & \qquad \{ \; \cup \text{ is commutative} \} \\
& \mathcal{P}(\mathsf{id} \times \mathsf{s}) \cdot \cup \cdot \mathsf{s} \cdot (\tau_l \times \tau_r) \cdot ((\mathsf{id} \times \omega) \times (\omega \times \mathsf{id})) \cdot \vartriangle \\
= & \qquad \{ \; \text{s is natural} \} \\
& \mathcal{P}(\mathsf{id} \times \mathsf{s}) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot ((\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega)) \cdot \mathsf{s} \cdot \vartriangle \\
= & \qquad \{ \; \text{routine verification: } \mathsf{s} \cdot \vartriangle = \vartriangle \} \\
& \mathcal{P}(\mathsf{id} \times \mathsf{s}) \cdot \cup \cdot (\tau_r \times \tau_l) \cdot ((\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega)) \cdot \vartriangle \\
= & \qquad \{ \; ||| \text{ 'gene' definition} \} \\
& \mathcal{P}(\mathsf{id} \times \mathsf{s}) \cdot \alpha_{|||}
\end{array}
$$

□

Interaction. To specify *interaction* there is a need to introduce some structure on the set Act of actions. For this purpose, we axiomatize the *interaction structure* underlying a process calculus as 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 L of labels. The intuition is that θ determines the *interaction discipline* whereas 0 represents the *absence* of interaction: a zero element is such that, for all $a \in Act$, $a\,\theta\,0 = 0$. On the other hand, a positive monoid entails $a\,\theta\,a' = 1$ iff $a = a' = 1$.

A simple example of an interaction structure captures the notion of action co-occurrence. Therefore, θ is defined as $a\,\theta\,b = \langle a, b\rangle$, for all $a, b \in Act$ different from 0 and 1. Action equality is defined as that of the 'frontiers' of Act terms, in order to assure θ associativity.

CCS synchronisation discipline [Milner, 1989] provides another example. In this case the set L of labels carries an involutive operation represented by an horizontal bar as in $\overline{a}$, for $a \in L$. Two actions a and $\overline{a}$ are said to be complementary. A special action, denoted by $\tau \notin L$, is introduced to represent the result of a synchronisation between a pair of complementary actions. Therefore, θ evaluates to τ whenever applied to a pair of complementary actions and to 0 in all other cases (except, obviously, if one of the arguments is 1).

Restriction and Renaming. The *restriction* combinator $\backslash_K$, for each subset $K \subseteq L$, forbids the occurrence of actions in K. Formally, $\backslash_K = [\![(\alpha_{\backslash_K})]\!]$ where

$$\alpha_{\backslash_K} \;=\; \nu \xrightarrow{\omega} \mathcal{P}(Act \times \nu) \xrightarrow{\mathsf{filter}_K} \mathcal{P}(Act \times \nu)$$

where $\mathsf{filter}_K = \lambda s\,.\ \{t \in s |\ \pi_1\ t \notin K\}$.

Once an interaction structure is fixed, any homomorphism $f : Act \longrightarrow Act$ lifts to a *renaming* combinator $[f]$ between processes defined as $[f] \;=\; [\![(\alpha_{[f]})]\!]$, where

$$\alpha_{[f]} \;=\; \nu \xrightarrow{\omega} \mathcal{P}(Act \times \nu) \xrightarrow{\mathcal{P}(f \times \mathsf{id})} \mathcal{P}(Act \times \nu)$$

Synchronous Product. This static operator models the simultaneous execution of its two arguments. At each step the resulting action is determined by the interaction structure for the calculus. Formally, $\otimes \;=\; [\![(\alpha_\otimes)]\!]$ where

$$\begin{aligned}\alpha_\otimes \;=\; & \nu \times \nu \xrightarrow{(\omega \times \omega)} \mathcal{P}(Act \times \nu) \times \mathcal{P}(Act \times \nu)\\ & \xrightarrow{\delta_r} \mathcal{P}(Act \times (\nu \times \nu)) \xrightarrow{\mathsf{sel}} \mathcal{P}(Act \times (\nu \times \nu))\end{aligned}$$

where $\mathsf{sel} = \mathsf{filter}_{\{0\}}$ filters out all synchronisation failures. The fundamental observation here concerns the way interaction is catered by δ_r — the distributive law for the strong monad $\mathcal{P}(Act \times \mathsf{Id})$ [17]. Again, for our purposes here it is enough a direct pointwise definition:

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

[17]Note that the monoidal structure in Act extends functor $\mathcal{P}(Act \times \mathsf{Id})$ to a strong monad, δ_r being the Kleisli composition of the left and right strengths. This, on its turn, involves the application of the monad multiplication to 'flatten' the result and this, for a monoid monad, requires the suitable application of the underlying monoidal operation. This, in our case, fixes the interaction discipline.

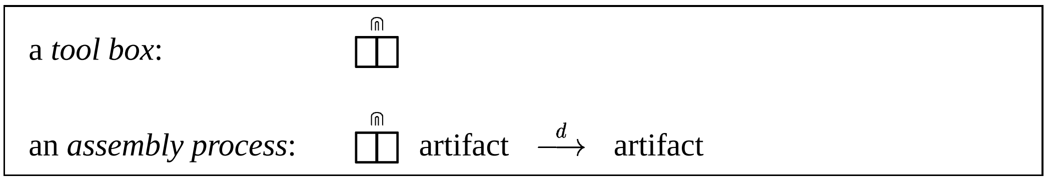
Parallel Composition. *Parallel* composition arises as a combination of *interleaving* and *synchronous product*, in the sense that the evolution of $p \mid q$, for processes p and q, consists of all possible derivations of p and q plus the ones associated to the synchronisations allowed by the particular interaction structure for the calculus. This cannot be achieved by mere composition of the corresponding combinators $|||$ and $\otimes$: it has to be performed at the 'genes' level for $|||$ and $\otimes$. Formally, $| \ = \ [\![(\alpha_{|})]\!]$, where

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

What's Next? Due to space limitations we have omitted here some of the CHARITY implementations and the discussion of how this approach to the design of process calculi extends smoothly to different behaviour models (*e.g.*, probabilistic processes) and process structures (*e.g.*, reactive processes). These extensions add genericity to the approach in ways which are complementary to the already discussed parametrization by the interaction discipline. The reader is referred to [Barbosa and Oliveira, 2002], for details on the CHARITY implementation of a generic process interpreter, and to [Barbosa, 2001] if interested in semantic issues. This last reference includes a *conditional fusion* theorem to derive conditional laws, *i.e.*, equalities depending on the fulfilment of some side conditions which are very common in process calculi. The novelty of the approach is that such conditions are *derived* rather than postulated and proved.

6. Conclusions

Construction Strikes Back. The *observational* standpoint adopted here as the framework to model and reason about state-based systems, later formalised in coalgebraic terms, has to be contrasted again to the 'engineer's view' referred in section 2.. The latter emphasises the possibility of at least some (essentially finite) things being not only observed, but actually *built*. Therefore, one works not with a 'lens' but with a 'toolbox'. Then the *assembly process* is specified in a similar (but dual) way to the one used to define observation structures. I.e, the engineer will equip her/himself with,



Notice that in the picture 'artifact' replaces 'universe', to stress that one is now dealing with 'culture' (as opposed to 'nature') and, which is far more relevant, that the arrow has been *reversed*. Formally, the ‘toolbox’ is again a functor. The *assembly process*, however, is a T-algebra. As a function this amounts to a collection of *constructors*. For example, for the binary tree ‘artifact’, the suitable toolbox will be

$$\overset{\Cap}{\boxed{}\!\boxed{}}\ D \;=\; \mathbf{1} + Data \times D \times D$$

This means that binary trees can either be built via a constant constructor, yielding the trivial, empty tree, or via the aggregation some data to two previously constructed trees, thus building a larger one.

Both ‘observation structures’ and ‘assembly processes’ can be related and compared. ‘Lens’ and ‘toolboxes’ are *functors*. In several cases, if the ‘lens’ or the ‘toolbox’ is smooth enough, there exist a canonical representative of the respective ‘observation structures’ or ‘assembly processes’. In the latter case this is known as the *term* algebra, which may be regarded as the formal analog to the ‘smallest’ machinery able to produce all possible $\overset{\Cap}{\boxed{}\!\boxed{}}$-artifacts. Its carrier is composed by all the terms generated by the constructors.

Just as the collection of all possible observed behaviours, with respect to a given interface, forms a (quite special) coalgebra (the *final* coalgebra), the set of *terms* gives rise to a ‘syntactic’ algebra (called the *initial* algebra) which also satisfies an universal, but *dual*, property [18]: there is a unique morphism from the term algebra to any other algebra d for the same functor. This is called a *catamorphism* and represented by $(\![d]\!)$; not only its definition, but also its properties are dual to those of an anamorphism.

Coalgebra Theory and Applications. Initial algebras correspond to *inductive data types*, *i.e.*, abstract descriptions of data structures. Dually, *final* coalgebras entail a notion of *coinductive*, *behaviour types*, representing the dynamics of systems. While data entities in an algebra are built by constructors and considered to be different if differently constructed, coalgebras deal with entities which are observed, or decomposed, by observers (or ‘destructors’). Internal configurations are identified if they cannot be distinguished by observation. As a consequence, equality has to be replaced by *bisimilarity* and *coinduction* replaces induction as a proof principle.

Although previously known in Universal Algebra, coalgebras began to be seriously considered only after the categorical account of both algebraic and coalgebraic structures of a *type* T has provided the right generic framework in which several phenomena and theories fit. The relevance of coalgebraic concepts and tools was first recognised in programming semantics — see, for example, P. Aczel foundational work on ‘non well-founded sets’ and the semantics of processes [Aczel, 1988, Aczel, 1993]; H. Reichel characterisation of behavioural satisfaction [Reichel, 1981] and J. Rutten, G. Plotkin and D. Turi work on final semantics [Rutten and Turi, 1994, Turi, 1996, Turi and Plotkin, 1997]. The systematic study of their theory, essentially along the lines of Universal Algebra, was initiated by J. Rutten in [Rutten, 1996], which later appeared in [Rutten, 2000]. This rapidly expanded to a broad research topic, as an increasing number of contributions

[18] A really exciting thing in category theory is the fact that universal constructions and properties always come in pairs.

appeared on both the theory, applications and rephrasing of 'old' results in seemingly unrelated areas. Part of this research is documented in the proceedings of the *Coalgebraic Methods in Computer Science* workshop series, starting in 1998.

In fact, only recently coalgebra theory itself has received enough attention and eventually emerged as a common framework to describe 'state based', dynamical, systems. Since then, coalgebraic modelling and reasoning principles have been applied in several areas. Examples range from automata [Rutten, 1998] to objects [Reichel, 1995, Jacobs, 1996b], from process semantics [Lenisa, 1998, Schamschurko, 1998, Wolter, 1999] to hybrid transition systems [Jacobs, 1996a]. B. Jacobs and his group, following earlier work by H. Reichel [Reichel, 1995, Hensel and Reichel, 1995] have coined the term *coalgebraic specification* [Jacobs, 1997, Jacobs, 2002] to denote a style of axiomatic specification involving equations up to bisimilarity acting as constraints on system observable behaviour. Models of coalgebraic specifications are subcoalgebras of the final coalgebra, just as models for algebraic specifications are quotients of initial algebras.

In a broader perspective, coalgebraic modelling explores the close relationship between coalgebras and *modal* and *temporal* logics, whose role in the specification of system's dynamics is well-established. The basic idea, developed namely in [Moss, 1999] and [Jacobs, 1999], consists of associating a modal language to the interface functor and having the logic assertions interpreted over the (Kripke models defined by the) transition systems induced by T-coalgebras.

Some Work Directions. This tutorial explored the role of coalgebraic structures in modelling and reasoning about dynamical, state-based systems. The underlying research agenda in *program calculi* is broad enough to include a variety of semantic formulations and applications, but, at the same time, the quite pragmatic goal of developing useful tools for the *working software engineer*. In this context, we are currently working on the following directions:

- Development of coalgebraic models and calculi for coordination languages and middleware, building on previous work on generic state-based software components [Barbosa, 2000].
- Extension of the work reported here on process calculi design to cope with naming and mobility.
- Systematic study of co-recursion schemes, and their interpretation in different categories.
- Development of suitable notions of *refinement* in a coalgebraic setting and their application to *process re-engineering* of legacy information systems.

References

Aczel, P. (1988). *Non-Well-Founded Sets*. CSLI Lecture Notes (14), Stanford.

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

Aczel, P. and Mendler, N. (1988). A final coalgebra theorem. In Pitt, D., Rydeheard, D., Dybjer, P., Pitts, A., and Poigne, A., editors, *Proc. Category Theory and Computer Science*, pages 357–365. Springer Lect. Notes Comp. Sci. (389).

Backus, J. (1978). Can programming be liberated from the Von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21:613–641.

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

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

Barbosa, L. S. and Oliveira, J. N. (2002). Coinductive interpreters for process calculi. In *Proc. of FLOPS'02*, Aizu, Japan. Springer Lect. Notes Comp. Sci. (to appear).

Barr, M. (1970). Coequalizers and cofree cotriples. *Mathematische Zeitschrift*, 166:307–322.

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

Bird, R. S. (1987). An introduction to the theory of lists. In Broy, M., editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer-Verlag.

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.

Cockett, R. and Fukushima, T. (1992). About Charity. Yellow Series Report No. 92/480/18, Dep. Computer Science, University of Calgary.

Cockett, R. and Spencer, D. (1992). Strong categorical datatypes I. In Seely, R. A. G., editor, *Proceedings of Int. Summer Category Theory Meeting, Montréal, Québec, 23–30 June 1991*, pages 141–169. AMS, CMS Conf. Proceedings 13.

Cockett, R. and Spencer, D. (1995). Strong categorical datatypes II: A term logic for categorical programming. *Theor. Comp. Sci.*, 139:69–113.

Hagino, T. (1987). A typed lambda calculus with categorical type constructors. In Pitt, D. H., Poigné, A., and Rydeheard, D. E., editors, *Category Theory and Computer Science*, pages 140–157. Springer Lect. Notes Comp. Sci. (283).

Hensel, U. and Reichel, H. (1995). Defining equations in terminal coalgebras. In Astesiano, E., Reggio, G., and Tarlecki, A., editors, *Recent Trends in Data Type Specification*, pages 307–318. Springer Lect. Notes Comp. Sci. (906).

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

Hudak, P., Peyton Jones, S. L., and Wadler, P. (1992). Report on the programming language Haskell, a non-strict purely-functional programming language, version 1.2. *SIGPLAN Notices*, 27(5).

Jacobs, B. (1996a). Object-oriented hybrid systems of coalgebras plus monoid actions. In Wirsing, M. and Nivat, M., editors, *Algebraic Methodology and Software Technology (AMAST)*, pages 520–535. Springer Lect. Notes Comp. Sci. (1101).

Jacobs, B. (1996b). Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. L. and Schek, H.-J., editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers.

Jacobs, B. (1997). Behaviour-refinement of coalgebraic specifications with coinductive correctness proofs. In *TAPSOFT'97: Theory and Practice of Software Development*, pages 787–802. Springer Lect. Notes Comp. Sci. (1214).

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

Jacobs, B. (2002). Exercises in coalgebraic specification. In Crole, R., Backhouse, R., and Gibbons, J., editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Constuction*. Springer Lect. Notes Comp. Sci. (2297).

Kieburtz, R. B. (1998). Reactive functional programming. In Gries, D. and de Roever, W.-P., editors, *Programming Concepts and Methods (PROCOMET'98)*, pages 263–284. Chapman and Hall.

Kurz, A. (2001). *Logics for Coalgebras and Applications to Computer Science*. Ph.D. Thesis, Fakultat fur Mathematik, Ludwig-Maximilians Univ., Muenchen.

Lawvere, F. W. and Schanuel, S. H. (1997). *Conceptual Mathematics*. Cambridge University Press.

Lenisa, M. (1998). *Themes in Final Semantics*. PhD thesis, Universita de Pisa-Udine.

Mac Lane, S. (1971). *Categories for the Working Mathematician*. Springer Verlag.

Malcolm, G. R. (1990). Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM*, 3(4):184–195.

McCarthy, J. (1963). A basis for a mathematical theory of computation. In Braffort, P. and Hirshberg, D., editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland.

McLarty, C. (1992). *Elementary Categories, Elementary Toposes*, volume 21 of *Oxford Logic Guides*. Clarendon Press.

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

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).

Milner, A. J. R. G. (1980). *A Calculus of Communicating Systems*. Springer Lect. Notes Comp. Sci. (92).

Milner, A. J. R. G. (1989). *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International.

Moore, E. F. (1966). Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton University Press.

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

Park, D. (1981). Concurrency and automata on infinite sequences. pages 561–572. Springer Lect. Notes Comp. Sci. (104).

Power, J. and Watanabe, H. (1998). An axiomatics for categories of coalgebras. In *ETAPS'98, Workshop on Coalgebraic Methods in Computer Science, Lisbon*. ENTCS, volume 11, Elsevier.

Reichel, H. (1981). Behavioural equivalence — a unifying concept for initial and final specifications. In *Third Hungarian Computer Science Conference*. Akademiai Kiado, Budapest.

Reichel, H. (1995). An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152.

Rutten, J. (1996). Universal coalgebra: A theory of systems. Technical report, CWI, Amsterdam.

Rutten, J. (1998). Automata and coinduction (an exercise in coalgebra). In *Proc. CONCUR' 98*, pages 194–218. Springer Lect. Notes Comp. Sci. (1466).

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

Rutten, J. and Turi, D. (1994). Initial algebra and final co-algebra semantics for concurrency. In *Proc. REX School: A Decade of Concurrency*, pages 530–582. Springer Lect. Notes Comp. Sci. (803).

Schamschurko, D. (1998). Modeling process calculi with PVS. In *I Workshop on Coalgebraic Methods in Computer Science, Lisbon*. ENTCS, volume 11, Elsevier.

Turi, D. (1996). *Functorial Operational Semantics and its Denotational Dual*. PhD thesis, Free University of Amsterdam.

Turi, D. and Plotkin, G. (1997). Towards a mathematical operational semantics. In *Proc. 12th LICS Conf.*, pages 280–291. IEEE, Computer Society Press.

Turi, D. and Rutten, J. (1998). On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces. *Mathematical Structures in Computer Science*, 8(5):481–540.

Turner, D. (1995). Elementary strong functional programming. In *Proc. Inter. Sym. on Functional Programming Languages in Education*, pages 1–13. Springer Lect. Notes Comp. Sci. (1022).

Winskel, G. (1993). *The Foraml Semantics of Programming Languages*. Foundations of Computing Series. The MIT Press.

Wolter, U. (1999). A coalgebraic introduction to CSP. In *II Workshop on Coalgebraic Methods in Computer Science*. ENTCS, volume 19, Elsevier.

A Where Do Models Live?

The success of functional programming, which literally means *programming with functions*, is due, to a great extent, to its roots on a basic mathematical intuition. The notion of an arrow connecting two objects (as we usually depict the signature of a function) and representing some kind of transformation of one into the other is pervasive both in mathematics and computer science. Therefore, some sort of space of types and (composable) typed arrows is usually taken as the underlying semantic universe for systems' specifications. In this text types are simply identified with sets and arrows with set-theoretic functions. This appendix reviews some canonical constructions on such a universe — *product*, *sum* and *exponentials* — and takes the opportunity to fix some notation used in the main text.

Product. The product of two sets A and B can be characterised either concretely (as the set of all pairs that can formed by elements of A and B) or in terms of an abstract specification. In this case, we say set $A \times B$ is defined as the source of two functions $\pi_1 : A \times B \longrightarrow A$ and $\pi_2 : A \times B \longrightarrow B$, called the *projections*, which satisfy the following property: for any other set Z and arrows $f : Z \longrightarrow A$ and $g : Z \longrightarrow B$, there is a unique arrow $\langle f, g\rangle : Z \longrightarrow A \times B$, usually called the *split* of f and g, that makes the following diagram to commute:

$$\begin{tikzcd}
 & Z \arrow[dl, "f"'] \arrow[d, "\langle f,g\rangle"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B
\end{tikzcd}$$

This is again an *universal* property [19], entailing both an *existence* and a *uniqueness* assertion. Such an abstract characterisation turns out to be more generic and suitable for conducting calculations. Note that it can be written as

$$k = \langle f, g\rangle \;\Leftrightarrow\; \pi_1 \cdot k = f \;\wedge\; \pi_2 \cdot k = g \tag{35}$$

where $\Rightarrow$ means *existence* and $\Leftarrow$ means *uniqueness*.

We illustrate this claim with a very simple example. Suppose we want to show that pairing the projections of a cartesian product has no effect, *i.e.*, $\langle \pi_1, \pi_2\rangle = \mathsf{id}$. If we proceed in a concrete way we first attemp to convince ourselves that the unique possible definition for *split* is as a pairing function, *i.e.*, $\langle f, g\rangle\ z = \langle f\ z, g\ z\rangle$. Then, instantiating the definition for the case at hands, conclude

$$\langle \pi_1, \pi_2\rangle\ \langle x, y\rangle = \langle \pi_1\ \langle x, y\rangle, \pi_2\ \langle x, y\rangle\rangle = \langle x, y\rangle$$

[19] Recall discussion on universal properties in the main text.

Using the universal property, instead, the result follows immediately and in a *pointfree* way:

$$\mathsf{id} = \langle \pi_1, \pi_2 \rangle \;\equiv\; \pi_1 \cdot \mathsf{id} = \pi_1 \wedge \pi_2 \cdot \mathsf{id} = \pi_2$$

Equation

$$\langle \pi_1, \pi_2 \rangle \;=\; \mathsf{id}_{A \times B} \tag{36}$$

is called the *reflection* law for products. Similarly the following laws (known as $\times$ *cancellation* and *fusion*) are derivable from (35):

$$\pi_1 \cdot \langle f, g \rangle = f \ , \ \pi_2 \cdot \langle f, g \rangle = g \tag{37}$$

$$\langle g, h \rangle \cdot f \;=\; \langle g \cdot f, h \cdot f \rangle \tag{38}$$

$$(i \times j) \cdot \langle g, h \rangle \;=\; \langle i \cdot g, j \cdot h \rangle \tag{39}$$

The same applies to *structural equality*:

$$\langle f, g \rangle = \langle k, h \rangle \;\equiv\; f = k \;\wedge\; g = h \tag{40}$$

Finally note that the product construction applies not only to sets but also to functions, yielding, for $f : A \longrightarrow B$ and $g : A' \longrightarrow B'$, function $f \times g : A \times A' \longrightarrow B \times B'$ defined as the split $\langle f \cdot \pi_1, g \cdot \pi_2 \rangle$. This equivales to the following pointwise definition: $f \times g = \lambda \, \langle a, b \rangle \, . \, \langle f\ a, g\ b \rangle$.

Sum. The *sum* $A + B$ (or *coproduct*) of A and B corresponds to their disjoint union. The construction is dual to the product one. From a programming point of view it corresponds to the aggregation of two entities in *time* (as in a `union` construction in C), whereas product entails an aggregation in *space* (as a `record`). It also arises by universality: $A + B$ is defined as the target of two arrows $\iota_1 : A \longrightarrow A + B$ and $\iota_2 : B \longrightarrow A + B$, called the *injections*, which satisfy the following universal property: for any other set Z and functions $f : A \longrightarrow Z$ and $g : B \longrightarrow Z$, there is a unique arrow $[f, g] : A + B \longrightarrow Z$, usually called the *either* (or *case*) of f and g, that makes the following diagram to commute:

$$\begin{tikzcd}
A \arrow[r, "\iota_1"] \arrow[dr, "f"'] & A + B \arrow[d, "{[f,g]}"] & B \arrow[l, "\iota_2"'] \arrow[dl, "g"] \\
& Z &
\end{tikzcd}$$

Again this universal property can be written as

$$k = [f, g] \;\Leftrightarrow\; k \cdot \iota_1 = f \;\wedge\; k \cdot \iota_2 = g \tag{41}$$

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

$$[f, g] \cdot \iota_1 = f \ , \ [f, g] \cdot \iota_2 = g \tag{42}$$

$$[\iota_1, \iota_2] \;=\; \mathsf{id}_{X+Y} \tag{43}$$

$$f \cdot [g, h] \;=\; [f \cdot g, f \cdot h] \tag{44}$$

Products and sums interact through the following *exchange* law

$$[\langle f, g\rangle, \langle f', g'\rangle] \;=\; \langle [f, f'], [g, g']\rangle \tag{45}$$

provable by either $\times$ (35) or $+$ (41) universality. The *sum* combinator also applies to functions yielding $f + g : A + B \longrightarrow A' + B'$ defined as $[\iota_1 \cdot f, \iota_2 \cdot g]$.

Basic Sets. We should also assume some basic sets, namely $\emptyset$, the empty set and $\mathbf{1}$, the singleton set. Note they are both 'degenerate' cases of, respectively, *sum* and *product* (obtained by applying the iterated version of those combinators to a nullary argument). The reader is invited to specialise the corresponding universal properties to these particular cases. Of course all our constructions are made up to isomorphism. Therefore, set $\mathbf{2} = \mathbf{1} + \mathbf{1}$ is taken as the set of boolean values. Similarly the infinite sum of $\mathbf{1}$ gives the set of natural numbers.

Exponential. Notation B^A is used to denote *function space, i.e.,* the set of (total) functions from A to B. It is also characterised by an universal property: for all function $f : A \times C \longrightarrow B$, there exists a unique $\overline{f} : A \longrightarrow B^C$ such that $f = \mathsf{ev} \cdot (\overline{f} \times C)$. Diagrammatically,

$$\begin{tikzcd}
A \arrow[d, "\overline{f}"'] & & A \times C \arrow[d, "\overline{f} \times \mathsf{id}_C"'] \arrow[dr, "f"] & \\
B^C & & B^C \times C \arrow[r, "\mathsf{ev}"'] & B
\end{tikzcd}$$

i.e.,

$$k = \overline{f} \;\Leftrightarrow\; f = \mathsf{ev} \cdot (k \times \mathsf{id}) \tag{46}$$

Remark. It should be remarked that the set-theoretic foundations assumed here are not enough when dealing with programming languages which, like HASKELL [Hudak et al., 1992], incorporate partial functions. Some order-enriched semantic setting applies then. In some cases, however, such an universe will 'collapse' initial algebras and final coalgebras, making obscure, or even preventing, the direct 'manipulation' of 'infinite', 'circular' objects. By contrast, the basic symmetry between initial and final types makes itself more intuitive in languages, like CHARITY, in which all programs are guaranteed to terminate (in the sense that only total functions can be programmed). Reference [Turner, 1995] provides a lively discussion on the merits of such a discipline of typed *total* functional programming. This is certainly not the place to go further into semantic considerations (the interested reader is referred to [Winskel, 1993] for a first introduction). The point we would like to stress, however, is that constructions introduced in this appendix (and, in general, coalgebra theory [20]) come from category theory [Mac Lane, 1971], where they can be appreciated in their full genericity and instantiated

[20]there is a remarkable amount of recent work on axiomatic, set-theoretic free, coalgebra theory [Power and Watanabe, 1998, Kurz, 2001].

to a myriad of semantic universes. The reader is referred to [McLarty, 1992] for a concise introduction to categories and to [Lawvere and Schanuel, 1997] for building the intuitions behind this exciting way of *thinking mathematically*.

B Programming in Charity

As a programming language CHARITY [Cockett and Fukushima, 1992] makes only a few assumptions on the underlying semantics universe (*cf.*, appendix A): it assumes the existence of products, (the possibility of defining) sums and their distributivity. It also provides a mechanism for defining both *inductive* and *coinductive* types, *i.e.*, initial algebras and final coalgebras for a class of so-called *strong* functors [21].

CHARITY primitive types are, then, the nullary (denoted by `1`) and binary product types (denoted by the infix operator `*`, with projections `p0` and `p1`). The absence of exponentials at the basic level of the language gives to programming in CHARITY a rather different flavour when compared with more traditional functional languages. In particular, functions are not values and function composition, instead of function application, is taken as the fundamental primitive in the language. This does not mean, however, that CHARITY lacks support for higher-order types: simply they have to be explicitly declared.

In this context, CHARITY may be classified as a polymorphic, strongly-typed language which is functional in style. In particular, any program has a guarantee of 'termination', in the sense that the term representing it always reduces to a head normal form and, therefore, a 'response' is produced. Such a 'response' is computed either lazily or eagerly depending on the types involved being coinductive or inductive, respectively. In any case, the type system simply blocks the possibility of writing functions that may never terminate. CHARITY programs are written in a particular term logic [Cockett and Spencer, 1995] which allows the use of variables in combinator expressions and pattern matching.

Let us briefly review the main 'building blocks' of a CHARITY program, starting with type declarations. The declaration of a coinductive type in CHARITY has the following format:

$$\texttt{data } S \texttt{ -> } T\texttt{(}A\texttt{) = } o_1 \texttt{:} S \texttt{ -> } E_1(A,S) \texttt{ | } \ldots \texttt{ | } o_n \texttt{:} S \texttt{ -> } E_n(A,S)\texttt{.}$$

which introduces the type $T\texttt{(}A\texttt{)}$, parametric on A. The declaration format conveys the idea that morphisms from any type S to $T\texttt{(}A\texttt{)}$ are solely determined by morphisms from S to each $E_i(A,S)$, the output type of observer o_i. Formally, this defines $T\texttt{(}A\texttt{)}$ as the final coalgebra for a functor $\mathsf{T_A}$ determined by the observers signature, *i.e.*,

$$\langle \nu_{\mathsf{T_A}}, \langle o_1, \ldots o_n \rangle : \nu_{\mathsf{T_A}} \longrightarrow \prod_i E_i(A, \nu_{\mathsf{T_A}}) \rangle$$

[21]The *strong* qualification means that the underlying functor T possess a *strength*, *i.e.*, for each type C transformations $\tau_r^{\mathsf{T}} : \mathsf{T}X \times C \longrightarrow \mathsf{T}(X \times C)$ subject to certain conditions. Its effect is to *distribute* context C along functor T. When types are modeled in such a setting, the universal combinators (such as anamorphisms) will possess a somewhat more general shape, able to deal with the presence of extra parameters in the functions being defined.

Each o_i identifies one of such observers whose type is obtained by setting $S = T(A)$ in the declaration. Therefore, $T(A)$ models $\nu_{\mathsf{T_A}}$.

Dually, an inductive type is declared as

$$\texttt{data}\ T(A)\ \texttt{->}\ S\ \texttt{=}\ c_1 \texttt{:} E_1(A,S)\ \texttt{->}\ S\ \texttt{|}\ \ldots\ \texttt{|}\ c_n \texttt{:} E_n(A,S)\ \texttt{->}\ S\texttt{.}$$

Such a declaration introduces type $T(A)$, again parametric on A, as the initial algebra

$$\langle \mu_{\mathsf{T_A}}, [c_1, \ldots, c_n] : \sum_i E_i(A, \mu_{\mathsf{T_A}}) \longrightarrow \mu_{\mathsf{T_A}} \rangle$$

The basic combinator associated to a coinductive type is `unfold`, *i.e.* an anamorphism in a strong setting. In CHARITY, this is specified by supplying, for each observer o_i the corresponding component p_i of the source coalgebra. As expected, strongness requires that each p_i be typed as $p_i : S \times C \longrightarrow E_i(A, S)$, assuming S as the carrier of the source coalgebra and C the context type. The concrete syntax for an `unfold` expression is as follows:

$$(s,c)\ \texttt{=>}\ (\texttt{|}\ s \texttt{=>} o_1 \texttt{:} p_1(s,c)\ \texttt{|}\ \cdots\ \texttt{|}\ o_n \texttt{:} p_n(s,c)\ \texttt{|})$$

where s and c denote variables of type S and C, respectively. A 'degenerate', *i.e.*, non recursive, `unfold`, is the `record` combinator which provides a way of populating a coinductive type by specifying particular values for the observers. The general pattern for the `record` combinator is

$$c\ \texttt{=>}\ (\ o_1 \texttt{:} f_1(c)\ \cdots\ \texttt{|}\ o_n \texttt{:} f_n(c)\ )$$

For inductive types the duals of these two combinators are `fold` and `case`, respectively. The `fold` combinator is specified by introducing, for each constructor c_i, the corresponding constructor, d_i, of the target algebra. Each of them is typed as $d_i : E_i(A, S) \times C \longrightarrow S$, where C is the type of the context and S the carrier of the target algebra. The target algebra is, of course, just 'the *either* of all such d_i'. The concrete syntax for a `fold` expression is:

$$(s,c)\ \texttt{=>}\ \{\texttt{|}\ c_1 \texttt{:} s_1 \texttt{=>} d_1(s_1,c)\texttt{|}\ \cdots\ \texttt{|}\ c_n \texttt{:} s_n \texttt{=>} d_n(s_n,c)\ \texttt{|}\}\ s$$

As mentioned above, the `record` combinator provides a canonical way of specifying (generalized) elements of a coinductive type. Dually, (generalized) elements of any type S, having an inductive type as domain of variation, can arise in a simple (non recursive) way by defining its value on each constructor of the domain. This construction is known in CHARITY as the `case` combinator whose syntax, in the general case, is

$$(s,c)\ \texttt{=>}\ \{\ c_1 s_1 \texttt{=>} d_1(s_1,c)\texttt{|}\ \cdots\texttt{|}\ c_n s_n \texttt{=>}\ \ d_n(s_n,c)\ \}\ s$$

Each d_i is a function from $E_i(A, \mu_{\mathsf{T_A}}) \times C$, to the target type. Notice that $E_i(A, \mu_{\mathsf{T_A}})$ is the domain of constructor c_i and C denotes the type of the context. Therefore, the domain of the case combinator is $\mathsf{T_A} \times C$. Using strength, context is pushed inside the $\mathsf{T_A}$ outermost (coproduct) structure and, therefore, even in this general case, the combinator is still determined by $[d_1, \ldots, d_n]$.

For each inductive or coinductive type, the `map` combinator denotes the action on morphisms, with strength, of the corresponding type functor. Its general format, for $h_i : A_i \times C \longrightarrow A'_i$, is:

$$\texttt{(}t\texttt{,}c\texttt{) => } T\{\ x_1 \texttt{=>} h_1\texttt{(}x_1\texttt{,}c\texttt{)|}\ \cdots\ \texttt{|}\ x_m \texttt{=>} h_m\texttt{(}x_m\texttt{,}c\texttt{)}\ \}\, t$$