Centrum voor Wiskunde en Informatica

REPORTRAPPORT

Objects and classes, coalgebraically

B. Jacobs

Computer Science/Department of Software Technology

CS-R9536 1995

Report CS-R9536
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Objects and Classes, Coalgebraically

Bart Jacobs

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
(bjacobs@cwi.nl)

Abstract

The coalgebraic perspective on objects and classes in object-oriented programming is elaborated: objects consist of a (unique) identifier, a local state, and a collection of methods described as a coalgebra; classes are coalgebraic (behavioural) specifications of objects. The creation of a "new" object of a class is described in terms of the terminal coalgebra satisfying the specification. We present a notion of "totally specified" class, which leads to particularly simple terminal coalgebras. We further describe local and global operational semantics for objects. Associated with the local operational semantics is a notion of bisimulation (for objects belonging to the same class), expressing observational indistinguishability.

AMS Subject Classification (1991): 18C10, 03G30
CR Subject Classification (1991): D.1.5, D.2.1, E.1, F.1.1, F.3.0
Keywords & Phrases: object, class, (terminal) coalgebra, coalgebraic specification, bisimulation

1. Introduction

Within the object-oriented paradigm the world consists of a collection of autonomous entities, called "objects", each dealing with a specific task. Coordination and communication takes place via sending of messages. Objects are grouped into certain "classes" which specify (among other things) the interface to the outside world (of the objects belonging to the class). Objects have private data, which is only accessible via specified operations, called "methods". And since each object is persistent, it can be seen as a (small) database. (But it typically has no query facilities.) There is no global state. The object-oriented paradigm is both popular and succesful, but a general complaint is that it lacks a proper formal foundation. In this paper we describe a semantics for objects and classes using so-called "coalgebras". These are the formal duals of algebras. The essential difference between algebras and coalgebras is that the former have "constructors" (operations going *into* the algebra, which are used to build things) where the latter have "destructors" (operations going *out of* the coalgebra, which allow us to observe certain behaviour). This distinction between construction and behaviour is in essence the distinction described in [9]. In coalgebra one deals with black boxes to which one only has limited access via specified operations. This aspect is important in the description of objects. It builds on ideas from automata theory. The notion of bisimulation forms an intrinsic part of the coalgebraic view. It means indistinguishability of behaviour, as it can be observed via the specified (coalgebraic) operations that we have at our disposal. It arises automatically in a situation with limited access.

There is no general agreement about what precisely constitutes an object. But there is broad agreement about the following two aspects: (1) an object has a local state, which is only accessible via the objects methods, and (2) an object combines data structure with behaviour. Precisely these two aspects are emphasized in our coalgebraic description of objects. (But there is more to say, see the last section.)

The suitability of coalgebras for the description of object-oriented features was recognized before, see e.g. [21, 20, 14, 15]. Elements may be traced back to earlier sources, like [16] or [18], where the word "coalgebra" is not yet used (in [18] one finds the phrase "abtract machine" instead). In [11] the two-level structure of specifications in the language COLD are explained: first there is a specification of one's application domain using algebraic data types, and then there is the system description in terms of "state machines". This second step corresponds to our coalgebraic (behavioural) specification. In [13, 6] the object-paradigm is explained within the algebraic world using signatures with hidden sorts. The hidden part is given a terminal interpretation in [6]. In this algebraic approach the output types of methods are unstructured, unlike in the coalgebraic approach below.

This paper elaborates ideas from [21] and [15]. What we consider as the main points are the following.

- This paper works out a (set theoretic) semantics for some crucial notions of object-oriented programming in detail. There is a precise notion of object and of class, where the latter is a suitable coalgebraic specification of the former. We do not focuss on syntactic details (or on any particular language) but on the meaning of the concepts involved. In this sense, it is a semantical study into object-orientation.
- It shows (following [21]) how behaviour can be specified coalgebraically (using conditional equations). Further, it gives a *local* operational semantics for objects (how objects behave in isolation) and a *global* operational semantics (how objects act on a configuration of messages). Associated with the local operational semantics is a notion of bisimulation for objects (belonging to the same class).
- It gives a clear meaning to "new" applied to a class, namely as the terminal coalgebra satisfying the class (as specification). This gives a canonical choice for an object belonging to a class.
- And it shows that these canonical (terminal coalgebra) models are "good" implementations. In this we use the techniques developed in [15]. It is somewhat surprising to see that although (carriers of) terminal coalgebras obtained from methods alone are generally huge sets of infinite trees (see Lemma 2.1), one can cut down these sets to very reasonable size in case one has "totally specified" behaviour; this happens in Proposition 5.2.

We shall make some use of elementary category theory in order to organize the concepts involved. In using these categories for the description of object-oriented languages one has to live with the multiple use of the word 'object'. Usually there is no confusion.

2. Preliminaries

In this first section we have collected some of the technicalities. They may be skipped at first.

2.1 Algebras versus coalgebras

Assume we wish to specify a datatype X of binary A-labelled trees, for some set of labels A. Algebraically one describes how to build up such a tree by giving its "constructors" nil and node, as on the left below (where $1 = \{*\}$ is a one-element set). In this specification one says that a binary A-labelled tree is either the empty tree nil, or of the form $\mathsf{node}(\ell, a, r)$ where ℓ and r are trees, and $a \in A$ is a label.

$$\left\{\begin{array}{rcl} \mathsf{nil}\colon 1 & \longrightarrow & X \\ \mathsf{node}\colon X \times A \times X & \longrightarrow & X \end{array}\right. \qquad\qquad \left\{\begin{array}{rcl} \mathsf{leaf}\colon X & \longrightarrow & A \\ \mathsf{left}\colon X & \longrightarrow & X \\ \mathsf{right}\colon X & \longrightarrow & X \end{array}\right.$$

A coalgebraic specification of such trees is given on the right. It does not give the "constructors", but the "destructors": it says which operations we have on our datatype of trees, namely taking off the label at a node, following the left path and following the right path. But it tells nothing about what is in X. This X is best considered as a black box to which we only have limited acces via the operations. The distinguishing difference between the algebraic and the coalgebraic description is that in the first case we have operations going *into* X and in the second case *out of* X. We can emphasize this difference even more by combining the operations into a single one by using coproducts $+$ and products $\times$. In the first case we get a single operation $1 + (X \times A \times X) \to X$ and in the second case $X \to A \times X \times X$. See also [9, 8].

The above algebraic specification has a canonical model given by the *initial algebra*. It consists of all finite binary A-labelled trees, and may be constructed as the set of closed terms. Also for the coalgebraic specification on the right there is a canonical model, given by the terminal coalgebra. It consists of the infinite binary A-labelled trees, and may be obtained as the set of "trees of observations". Initial algebras form a basis for data type semantics (see e.g. [10]), and terminal coalgebras play a similar role in an object-oriented setting.

The general definition of an **algebra** is a map of the form $T(X) \to X$, for some functor $T\colon \mathbf{Sets} \to \mathbf{Sets}$ (or some other category); and a **coalgebra** is a map $X \to T(X)$. If we have two algebras $(c\colon T(U) \to U)$ and $(d\colon V \to T(V))$, then we say that an **algebra map** $c \to d$ is a morphism $f\colon U \to V$ between the "carriers" which commutes with the operations: $f \circ c = d \circ T(f)$. This gives us a category $\mathrm{Alg}(T)$. Dually we can form a category $\mathrm{CoAlg}(T)$ of coalgebras of T: a **coalgebra map** $(c\colon U \to T(U)) \to (d\colon V \to T(V))$ is a morphism $f\colon U \to V$ with $d \circ f = T(f) \circ c$.

2.2 Finite products, coproducts and exponents

We recall that in a category $\mathbb{C}$ an object $1 \in \mathbb{C}$ is **terminal** if for each $X \in \mathbb{C}$ there is precisely one arrow $X \to 1$. Singleton sets are terminal objects in **Sets**; we typically write $1 = \{*\}$. The dual notion is that of **initial** object 0, for which there is precisely one $0 \to X$ to any X. In **Sets** we have $0 = \emptyset$. The binary product $X \times Y$ is characterized by the property that maps $Z \to X \times Y$ are in (natural) bijective correspondence with pairs of maps $Z \to X$ and $Z \to Y$. This gives us to projections $\pi\colon X \times Y \to X$, $\pi'\colon X \times Y \to Y$ and a tupling operation $\langle -, - \rangle$. Dually, we have a coproduct $X + Y$ with the property maps $X + Y \to Z$ out of it correspond (naturally) to pairs of maps $X \to Z$ and $Y \to Z$. This gives us coprojections $\kappa\colon X \to X + Y$, $\kappa'\colon Y \to X + Y$ and a cotupling operation $[-, -]$. In **Sets** $\times$ is the usual cartesian product of pairs of elements, and $+$ is the disjoint union. Finally we use an exponent construction Y^X, with the property that maps $Z \to Y^X$ correspond (naturally) to maps $Z \times X \to Y$. In presence of these exponents we get the familiar distributivities $X \times (Y + Z) \cong (X \times Y) + (X \times Z)$ and $X \times 0 \cong 0$. All told, we will be using the structure of a bicartesian closed category (BiCCC). But (higher type) exponents do not play an essential role: in principle we could also use distributive categories.

We use these constructions $1, \times, 0, +, (-)^{(-)}$ to build up so-called **polynomial** functors. We shall restrict ourselves to functors of the form

$$T(X) = (B_1 + C_1 \times X)^{A_1} \times \cdots \times (B_n + C_n \times X)^{A_n} \qquad (*)$$

for certain (constant) sets A_i, B_i, C_i—which may be 0 or 1 so that parts of this functor become simpler. A coalgebra $c\colon U \to T(U)$ of this functor may be identified with a set of maps $c_1\colon U \times A_1 \to B_1 + C_1 \times U$, $\ldots$, $c_n\colon U \times A_n \to B_n + C_n \times U$. A coalgebra forms in this way a model of a certain signature of operations (i.e. methods). And a coalgebra map is a map between the carrier sets which commutes with the operations. Note that the c_i are maps going *out of* U, with a parameter from A_i.

2.3 Terminal coalgebras

For reasons that will become clear below we shall be especially interested in terminal objects in categories $\mathrm{CoAlg}(T)$ of coalgebras of functors T. These will be called terminal coalgebras. There is a standard construction (see e.g. [22]) to compute such a coalgebra as a limit Z of the diagram $1 \leftarrow T(1) \leftarrow T^2(1) \leftarrow \cdots \leftarrow Z$. This construction applies for the above functors $(*)$, because they preserve limits of such chains. We like to have an explicit description of this terminal coalgebra.

2.1. Lemma. *The terminal coalgebra $Z \xrightarrow{\sim} T(Z)$ of the above functor $(*)$ in* **Sets** *can be described as a set of infinite trees. Therefore, first write*

$$A = A_1 + \cdots + A_n, \qquad B = B_1 + \cdots + B_n, \qquad C = C_1 + \cdots + C_n$$

for the disjoint union of the constants in the functor. We now have

$$\begin{aligned} Z = \{\varphi \colon A^+ \to B + C \mid & \forall \alpha \in A^+ . \forall i \leq n . \forall a \in A_i . \varphi(\langle i, a\rangle \cdot \alpha) \in B_i + C_i \\ & \text{and } \varphi(\langle i, a\rangle \cdot \alpha) \in B_i \;\Rightarrow\; \forall i' \leq n . \forall a' \in A_{i'} . \varphi(\langle i', a'\rangle \cdot \langle i, a\rangle \cdot \alpha) = \varphi(\langle i', a'\rangle \cdot \alpha)\}. \end{aligned}$$

where $\cdot$ is concatenation of sequences (from A^+).

The methods $Z \times A_i \to B_i + C_i \times Z$ are given by

$$(\varphi, a) \mapsto \begin{cases} \varphi(\langle i, a\rangle) & \text{if } \varphi(\langle i, a\rangle) \in B_i \\ \langle \varphi(\langle i, a\rangle), \lambda\alpha \in A^+ . \varphi(\alpha \cdot \langle i, a\rangle)\rangle & \text{otherwise.} \end{cases} \qquad \square$$

Notice that elements of this set Z are infinite trees. This infinity is achieved by repetition in case an "attribute value" in a B_i comes out. For example, the set Z of finite and infinite lists of C's is the (carrier of the) terminal coalgebra of the functor $T(X) = 1 + C \times X$. Explicitly, it is described as $Z = \{\varphi \colon \mathbb{N} \to 1 + C \mid \forall n \in \mathbb{N} . \varphi(n) = * \Rightarrow \varphi(n+1) = *\}$.

2.2. Example (See [21]). A useful special case of the above lemma is the following: the terminal coalgebra in **Sets** of the functor $T(X) = B \times X^A$ associated with the signature $X \to B$, $X \times A \to X$ is the set $Z = B^{A^\star}$ of functions from the set $A^\star$ of finite sequences of A's to B, with methods $Z \to B$ given by $\varphi \mapsto \varphi([\,])$ and $Z \times A \to Z$ by $(\varphi, a) \mapsto \lambda\alpha \in A^\star . \varphi(\alpha \cdot a)$. In [21] only these restricted signatures (without coproducts) are used. They form a special case of $(*)$ above for $n = 2$ and $A_1 = 1$, $B_1 = B$, $C_1 = 0$, $A_2 = A$, $B_2 = 0$ and $C_2 = 1$.

2.4 Bisimulations and mongruences

Bisimulations and mongruences are relations and predicates on carriers of coalgebras which are suitably closed under the coalgebra operations. One can describe these notions in terms of the functor involved, see [1], or [15]. Here we shall describe these notions concretely for functors $(*)$ as above.

2.3. Definition. Let $T \colon \mathbf{Sets} \to \mathbf{Sets}$ be the above polynomial functor $(*)$, and let $c \colon U \to T(U)$ and $d \colon V \to T(V)$ be two coalgebras of this functor.

(i) A relation $R \subseteq U \times V$ is called a **bisimulation** (on these coalgebras) if for $x \in U$ and $y \in V$, in case $R(x, y)$ holds, then for each $i \leq n$ and $a \in A_i$ we have one of the following two cases:

- both $c_i(x, a)$ and $d_i(y, a)$ are in B_i, and they are equal.
- both $c_i(x, a)$ and $d_i(y, a)$ are tuples, of the form $c_i(x, a) = \langle c, x'\rangle$ and $d_i(y, a) = \langle c, y'\rangle$ with equal components in C_i and with $R(x', y')$.

Two elements $u \in U$ and $v \in V$ are called **bisimilar** if there is a a bisimulation $R \subseteq U \times V$ with $R(u, v)$. In this case one writes $u \,\underline{\leftrightarrow}\, v$.

(ii) A **mongruence** on $c \colon U \to T(U)$ is a predicate $P \subseteq U$ for which: if $P(x)$ holds, then also $P(x')$ for each $x' \in U$ occurring as "new state" $c_i(x, a) = \langle c, x'\rangle \in C_i \times U$, for $i \leq n$ and $a \in A_i$.

The following standard result gives an equivalent description of bisimulation $\underline{\leftrightarrow}$ in terms of terminal coalgebras.

2.4. Lemma. *Consider two coalgebras $c: U \to T(U)$ and $d: V \to T(V)$ of the same functor T. They induce two unique coalgebra maps $!_c$ and $!_d$ to the terminal coalgebra $Z \xrightarrow{\sim} T(Z)$, in:*

$$\begin{tikzcd}
T(U) \arrow[r, dashed, "T(!_c)"] & T(Z) & T(V) \arrow[l, dashed, "T(!_d)"'] \\
U \arrow[u, "c"] \arrow[r, dashed, "!_c"'] & Z \arrow[u, "\cong"'] & V \arrow[u, "d"'] \arrow[l, dashed, "!_d"]
\end{tikzcd}$$

Two elements $u \in U$ and $v \in V$ of the carriers of these coalgebras are then bisimilar if and only if they have the same value on the terminal coalgebra, i.e. $u \underline{\leftrightarrow} v$ if and only if $!_c(u) = !_d(v)$. □

3. Objects and Classes (locally)

This section contains the main definitions, which will be illustrated subsequently by a series of examples. The main aspect of an object that we are capturing coalgebraically is that it has a local state, which is only accessible via specified operations. Classes will be presented as specifications of objects. We consider objects "locally" in the sense that we describe single objects on their own, and not systems of objects (as in Section 6).

3.1. Definition. (i) Let $\mathcal{C}$ be a set of class names. In a particular program, this set will be fixed. We write $\mathrm{OiD} = \mathcal{C} \times \mathbb{N}$ for the set of **object identifiers** (names of objects). Each object has a unique identifier $o = \langle N, i \rangle$ where N is the name of the class to which the object belongs, and i is a natural number used to distinguish different objects of the same class.

(ii) An **object** is a 3-tuple $\langle o, u \in U, c: U \to T(U) \rangle$ where o is the objects identifier, u is its **local state**, U is its **local state space**, and $c: U \to T(U)$ is a coalgebra structure on U, that gives the operations (methods) of the object on the local state space U. This coalgebra is determined by the identifier, in the sense that equality of identifiers (names) for objects means equality of coalgebras.

The identifier is not so important at first, when we consider objects locally (in isolation). But later on, when we consider collections of objects globally, it plays a role because it is used to denote the target object of a message. Having an explicit name component is mathematically not so elegant, since the standard naming convention (in mathematics) is to use the tuple as a name for the object.

3.2. Example. At this stage we only mention a trivial example, where we ignore the identifier. Every value $b \in B$ forms an object, namely with trivial local state space $1 = \{*\}$ and with b as coalgebra $b: 1 \to B = T(1)$ for the constant functor $T(X) = B$. This constant object returns the value b when it is sent a message. There is no possibility to change the local state of this object.

In the type theoretic encoding of object-oriented features into second (or higher) order polymorphic lambda calculus (with subtyping), see e.g. [7, 20, 14], one uses the type $\exists \alpha \colon \mathsf{Type}.\, \alpha \times (\alpha \to T(\alpha))$ for objects with "interface" T. One thus has an encoding which involves hiding the local state space α via an existential quantifier (as in [19]). An inhabitant of the product type $\alpha \times (\alpha \to T(\alpha))$ is a tuple consisting of a local state in α and a coalgebra $\alpha \to T(\alpha)$, like in the above definition. But in this type theoretic encoding there is no explicit way to deal with equations; they play an essential role below in the specification of behaviour.

One may also view an object with local state $u \in U$ and coalgebra $c: U \to T(U)$ as a particular kind of automaton, with u as current state of the automaton, and with the coalgebra c as transition function. From an object-oriented perspective there is some degree of non-determinism in the sense that the transition function c is a tuple of methods c_i, and the object itself does not know which of these components is selected, and with which parameter.

We shall describe classes as specifications of objects. Thus an object belongs to a class, when it is a model of (or, when it meets) the specification.

3.3. Definition. (i) A **class** is a structure which has a name, and consists of three components.

- A finite set of (unary) methods

$$\left\{\begin{array}{lcl} X \times A_1 & \longrightarrow & B_1 + C_1 \times X \\ & \vdots & \\ X \times A_n & \longrightarrow & B_n + C_n \times X \end{array}\right.$$

  on a local state space X. The functor associated with this signature of coalgebraic operations is

$$T(X) = (B_1 + C_1 \times X)^{A_1} \times \cdots \times (B_n + C_n \times X)^{A_n}.$$

  In a class one should explicitly say which of these methods are **public** and which are **private**. If some C_i is the empty set 0, then the associated method gets the form $X \times A_i \longrightarrow B_i$, and may be called an **attribute**, since it yields an "observable element" in B_i and does not change the local state space. Methods which do affect the local state may be seen as procedures.

- Equations, which may be conditional. These equations are between observable outcomes (the B's and C's above), not between local states (elements of X). This reflects the idea that we do not have direct access to local states. These equations regulate the behaviour of the objects belonging to the class. Both public and private methods may occur in the equations.

- The attributes which hold for newly created objects, using new. These may be either without parameters, or be parametrized.

(ii) An object $p = \langle o, u \in U, c\colon U \to T(U)\rangle$ **belongs to** (or simply: **is in**) a class with name N if

- p's identifier o is of the form $\langle N, i\rangle$, where $i \in \mathbb{N}$;
- the functor T occurring in p is the same as the one in the specification of the methods of N;
- the coalgebra c in p satisfies the equations of N.

(iii) Objects belonging to a particular class N can be organized in a category $\mathrm{Obj}\,N$. Then objects of $\mathrm{Obj}\,N$ (in a categorical sense) are objects (in an object-oriented sense). As morphisms $\langle o_1, u_1 \in U_1, c_1\colon U_1 \to T(U_1)\rangle \longrightarrow \langle o_2, u_2 \in U_2, c_2\colon U_2 \to T(U_2)\rangle$ in $\mathrm{Obj}\,N$ we simply take the coalgebra maps $c_1 \to c_2$ (consisting of functions $f\colon U_1 \to U_2$ between the local state spaces with $c_2 \circ f = T(f) \circ c_1$). Morphisms between objects do not take identifiers and local states into account. (The dynamics of objects is thus ignored; but one can choose to define another category of objects, in which maps do preserve local states.)

(iv) For a class N, the object new N will be the terminal object in $\mathrm{Obj}\,N$. It will have the local state determined by the attribute values which are specified in the third point in (i) above. Its identifier will be $\langle N, n+1\rangle$ where n is the number of already existing objects belonging to the class N. (This latter aspect is determined during run time; it guarantees uniqueness of object identifiers.)

In the classes that we consider in this paper we shall only use equational logic, but from a semantical point of view there is no objection against using a more expressive logic in the second point. For example, temporal logic is used in [12]. The difference between public and private methods is that one object may only send messages requiring execution of a public method in another object. But it may send messages to itself asking for execution of its own private methods. Private atributes (i.e. methods

$X \longrightarrow B$ in the private section) may play the role of instance variables. Also attributes in the public method section may be seen as instance variables, but these variables can then be read from outside.

The methods that we consider have output types $B_i + C_i \times X$. This means that they can produce an observable element in B_i, or an observable element in C_i together with a new state in X. We can also have methods $X \times A_i \longrightarrow X + D_i \times X$ by using the isomorphism $X + D \times X \cong (1 + D) \times X \cong 0 + (1 + D) \times X$, so that we have an isomorphic output of the required format. But notice that at most one new state can be produced (in every alternative of $+$).

Next we define bisimilarity for objects. This notion is intended to capture observational indistinguishability for objects. It will therefore only involve the publicly available methods.

3.4. Definition. Consider a class N with two functors $T_{\mathrm{pu}}, T_{\mathrm{pr}} \colon \mathbf{Sets} \rightrightarrows \mathbf{Sets}$ describing the signatures of respectively the public and private methods of N. Two objects $\langle o_1, u_1 \in U_1, c_1 \colon U_1 \to T_{\mathrm{pu}}(U_1) \times T_{\mathrm{pr}}(U_1)\rangle$, $\langle o_2, u_2 \in U_2, c_2 \colon U_2 \to T_{\mathrm{pu}}(U_2) \times T_{\mathrm{pr}}(U_2)\rangle$ belonging to this class N will be called **bisimilar** if there is a bisimulation $R \subseteq U_1 \times U_2$ with respect to the coalgebras $\pi \circ c_1 \colon U_1 \to T_{\mathrm{pu}}(U_1)$ and $\pi \circ c_2 \colon U_2 \to T_{\mathrm{pu}}(U_2)$ of the "public" functor T_{pu}, implementing the public methods.

During the lifetime of an object its local state may change through the execution of its methods (as a result of incoming messages), but its identifier and its methods (coalgebra) remain the same. We shall often call two objects **identical** if they only differ in their local state. Thus, execution of methods does not change the identity of objects. Under bisimilarity more objects are identified.

3.5. Remarks. (i) Classes are described above as specifications of objects. Thus there is a sharp distinction between specification and implementation. One of the strong points of this approach is that it provides a clear semantics for "new": the canonical implementation of the specification is taken. Often there is not such a clear separation of specification and implementation, e.g. when classes contain certain implementation details about the precise way in which a specific method is written as a program. Such "classes" may be seen as bodies for our "classes as specifications".

(ii) There are similarities between the approach presented here and the one in Wieringa [23]. For example, a 'class' here is an 'object specification' there, and a 'conditional equation' here is a 'local event constraint' (with pre- and post-conditions) there. A difference is that Wieringa incorporates some process algebra into his specification formalism. But the main difference is that he works in an algebraic world, using Kripke semantics, and not in a coalgebraic one.

In the remainder of this section we shall consider examples of classes and objects. Object identifiers will play a minor role in these local investigations.

A rudimentary bank account

We consider a bank account (of a single person) for which we only have methods bal giving the balance of the account, and ch with which we can change the amount of money in the account. An obvious equation should then be satisfied, describing the balance after the change in terms of the balance before, and the change. We use hopefully self-explanatory notation, in the following specification—

with some comments afther the '#' sign.

```
class: BA                            # name of the class; 'BA' for 'bank account'
    public methods:
        bal: $X \longrightarrow \mathbb{Z}$              # this is an attribute
        ch: $X \times \mathbb{Z} \longrightarrow X$          # this is a method, with parameter from $\mathbb{Z}$;
                                     # it affects the local state space $X$.
    equations:
        $x.\mathsf{ch}(a).\mathsf{bal} = x.\mathsf{bal} + a$   # in OO-style with post fix notation
                                     # (where $x: X$ a variable for the local state)
    creation:
        new.bal = 0
endclass
```

In this specification we say what methods we want for our bank account and what equations should hold. The equation $x.\mathsf{ch}(a).\mathsf{bal} = x.\mathsf{bal} + a$ should be read as: if one sends x the change message ch with parameter a and then asks for the balance bal, then the outcome is the same as first asking x for its balance, and then adding the amount a. The last point of the specification mentions that newly created objects (written as new BA) of this class BA have $0 \in \mathbb{Z}$ as balance.

As an observer on the outside, we do not really care how objects belonging to this bank account class are implemented, as long as they meet the specification. We have no access to the local state space X except via the above two methods. This is coalgebra. We shall present some possible implementations, which give examples of objects belonging to this class, with different interpretations of X and of bal, ch. But these differences are not visible to users. Notice that the functor associated with the signature of methods is $T(X) = \mathbb{Z} \times X^{\mathbb{Z}}$.

A first try is to take a bank account as a sequence consecutive changes. Thus we take as local state space $U_1 = \mathbb{Z}^\star$, the set of finite sequences of integers. For $x = \langle a_0, \ldots, a_n \rangle \in U_1$ we define methods:

$$x.\mathsf{bal} = a_0 + \cdots + a_n \qquad \text{and} \qquad x.\mathsf{ch}(a) = \langle a_0, \ldots, a_n, a \rangle.$$

These two methods together form a coalgebra $c_1: U_1 \rightarrow T(U_1)$. It obviously satisfies the equation $x.\mathsf{ch}(a).\mathsf{bal} = x.\mathsf{bal} + a$. We can thus form an object $\langle\langle \mathrm{BA}, 1\rangle, \langle 2, -3\rangle \in U_1, c_1: U_1 \rightarrow T(U_1)\rangle$ belonging to the class BA. The balance of this bank account is -1. One could note that this is a rather inefficient implementation: asking for the balance involves adding up all the changes that have been made. But for a user of the object on the outside—who can only access the object via the balance and change methods—these implementation details are not visible.

Our second try also involves an implementation which keeps a record of changes, but this time the additions are done immediately so that taking the balance gives a more direct answer. So we now take as local state space $U_2 = \mathbb{Z}^+$, the set of non-empty sequences of integers. For an element $x = \langle a_1, \ldots, a_n \rangle \in \mathbb{Z}^+$ we define

$$x.\mathsf{bal} = a_n \qquad \text{and} \qquad x.\mathsf{ch}(a) = \langle a_1, \ldots, a_n, a_n + a \rangle.$$

This gives us a coalgebra $c_2: U_2 \rightarrow T(U_2)$, which also satisfies the equations.

We mention a third implementation which simply has a local state space the set $U_3 = \mathbb{Z}$ of integers. For $x \in \mathbb{Z}$ we define

$$x.\mathsf{bal} = x \qquad \text{and} \qquad x.\mathsf{ch}(a) = x + a.$$

A bank account object with this coalgebra, call it $c_3: U_3 \rightarrow T(U_3)$, has as local state an integer that represents the current balance. In a sense this is the most efficient implemention. In a mathematical sense it distinguishes itself as the terminal coalgebra, i.e. as the terminal object in the category of coalgebras $X \to \mathbb{Z} \times X^{\mathbb{Z}}$ satisfying the bank account equation.

Consider the two bank account objects $p_1 = \langle o_1, \langle 2, -3\rangle, c_1 \colon \mathbb{Z}^\star \to T(\mathbb{Z}^\star)\rangle$ and $p_3 = \langle o_3, -1, c_3 \colon \mathbb{Z} \to T(\mathbb{Z})\rangle$ using the above first and third implementation. These are bisimilar, because we cannot see a difference, using the public methods specified in the bank account class: they have the same balance, namely -1, and by using the change method we cannot create a difference, since the balance after a change is determined by the equation in the class. More technically, we have a bisimulation $R \subseteq \mathbb{Z}^\star \times \mathbb{Z}$ with $R(\langle 2, -3\rangle, -1)$ namely

$$R = \{\langle a_0, \ldots, a_n, a\rangle \in \mathbb{Z}^\star \times \mathbb{Z} \mid a_0 + \cdots + a_n = a\}.$$

There is one more aspect of classes that we can illustrate in this bank account example, namely the difference between creation with or without parameters. The line $\mathsf{new.bal} = 0$ in the above class describes creation without parameters. Its result is that newly created bank accounts of this class have balance 0. One may wish to have some more flexibility here—for example, some banks encourage opening of accounts by giving a starters premium—and to be able to specify the amount of money that should already be there at creation. This is creation with a parameter. The syntax one could then use is $\mathsf{new}\,\mathrm{BA}(10)$ to indicate that the initial balance should be 10. In the class itself one should indicate this option of creation with a parameter, for example by writing $\mathsf{new}(z).\mathsf{bal} = z$ (instead of the line $\mathsf{new.bal} = 0$ for unparametrized creation, as above).

Two buffers with capacity one

Let A be a fixed set of data elements. We wish to describe a class of buffer objects, which can contain an element $a \in A$. The methods that it should have are $\mathsf{store}(a)$, to put an element $a \in A$ in a buffer, and read to read the content of a buffer. We should decide explicitly:

- what happens when we send the $\mathsf{store}(a)$ message to a buffer which is already full; [we choose that nothing will happen];
- what happens when we read from an empty buffer; [the (observable) outcome will be an error value];
- what happens to a buffer when we read from it: one can have a **destructive read** (DR), which means that after reading an element a buffer will be empty, or a **persistent read** (PR), which means that reading does not affect the content of a buffer; in that case one needs an explicit method empty for emptying the buffer.

Below we shall present two classes PR for the persistent read buffers (on the left), and DR for the destructive read buffers (on the right).

```
class: PR
    public methods:
        store: $X \times A \longrightarrow X$
        read: $X \longrightarrow \{\mathsf{error}\} + A$
        empty: $X \longrightarrow X$
    equations:
        $x.\mathsf{empty.read} = \mathsf{error}$
        $x.\mathsf{read} = \mathsf{error} \vdash x.\mathsf{store}(a).\mathsf{read} = a$
        $x.\mathsf{read} = a \vdash x.\mathsf{store}(b).\mathsf{read} = a$
    creation:
        $\mathsf{new.read} = \mathsf{error}$
endclass
```

```
class: DR
    public methods:
        store: $X \times A \longrightarrow X$
        read: $X \longrightarrow \{\mathsf{error}\} + A \times X$
    equations:      # in sloppy notation
        $x.\mathsf{read} = \mathsf{error} \vdash x.\mathsf{store}(a).\mathsf{read.fst} = a$
        $x.\mathsf{read} = a \vdash x.\mathsf{store}(b).\mathsf{read.fst} = a$
        $x.\mathsf{read} = \langle a, y\rangle \vdash y.\mathsf{read} = \mathsf{error}$
    creation:
        $\mathsf{new.read} = \mathsf{error}$
endclass
```

The main difference between the persistent read class and the destructive read class is that in the former the read method is an attribute: it does not change the local state space. The destructive read method does have an effect on the local state space—it empties the buffer—which is reflected in the type of this method: the X occurs in the type of the output of the read method. The functor describing the signature of persistent read methods is $X \mapsto X^A \times (1 + A) \times X \cong X^{(A+1)} \times (1 + A)$. For the destructive read we have $X \mapsto X^A \times (1 + A \times X)$, where $1 = \{\mathsf{error}\}$ is the terminal set.

The equations may equivalently be expressed via diagrams. For example, the three equations for persistent read may be expressed via two diagrams:

$$\begin{tikzcd}
X \arrow[r, "\mathsf{empty}"] \arrow[d] & X \arrow[d, "\mathsf{read}"] \\
1 \arrow[r, "\kappa"'] & 1 + A
\end{tikzcd}
\qquad
\begin{tikzcd}
X \times A \arrow[rr, "\mathsf{store}"] \arrow[d, "\mathsf{read} \times id"'] & & X \arrow[rr, "\mathsf{read}"] & & 1 + A \\
(1 + A) \times A \arrow[rr, "\cong"'] & & A + (A \times A) \arrow[rr, "{[id, \pi]}"'] & & A \arrow[u, "\kappa'"']
\end{tikzcd}$$

The equations for the destructive read class have been expressed in a somewhat sloppy way: the $+$-component of the read output is left implicit. More formally, we need a "case" construction (as in a type theory with coproduct types $+$), so that we can write the equations as:

$$\begin{array}{rcl}
\mathsf{case\ of}\ x.\mathsf{read} \left\{ \begin{array}{ccc} \mathsf{error} & \mapsto & \kappa' b \\ \langle a, y \rangle & \mapsto & \kappa' a \end{array} \right\} & = & \mathsf{case\ of}\ x.\mathsf{store}(b).\mathsf{read} \left\{ \begin{array}{ccc} \mathsf{error} & \mapsto & \kappa\mathsf{error} \\ \langle a, y \rangle & \mapsto & \kappa' a \end{array} \right\} \\
\mathsf{case\ of}\ x.\mathsf{read} \left\{ \begin{array}{ccc} \mathsf{error} & \mapsto & \kappa\mathsf{error} \\ \langle a, y \rangle & \mapsto & y.\mathsf{read} \end{array} \right\} & = & \kappa\mathsf{error}.
\end{array}$$

But of course, they can also be written diagrammatically.

The terminal coalgebras satisfying these specifications have in both the persistent and in the destructive case as local state space the set $1 + A$. This set can contain an error value in $1 = \{\mathsf{error}\}$ representing that the buffer is empty, and it can contain an element $a \in A$. It thus contains the minimal information need for a buffer of capacity one. The store method is in both cases implemented as the composite

$$\mathsf{store} = \Big((1 + A) \times A \cong A + (A \times A) \xrightarrow{[id, \pi]} A \xrightarrow{\kappa'} 1 + A \Big)$$

It sends $(z, a) \in (1 + A) \times A$ to $a \in A$ if $z = \mathsf{error}$ and to $b \in A$ if $z = b$. The read methods are of course different. The persistent read is simply the identity function $1 + A \to 1 + A$, whereas the desctructive read is the composite

$$\mathsf{read} = \Big(1 + A \cong 1 + A \times 1 \xrightarrow{id + id \times \kappa} 1 + A \times (1 + A) \Big)$$

Finally, the empty method from the persistent read always gives an error element, via

$$\mathsf{empty} = \Big(1 + A \longrightarrow 1 \xrightarrow{\kappa} 1 + A \Big)$$

Other implementations are possible. For example, one can have as local state space $(1 + A) \times A^\star \cong A^\star + A^+$, so that one can use the $1 + A$ component as the actual buffer (like above), and the $A^\star$ component as history of elements that have been stored. The actual implementation of the methods on this alternative local state space (for both the persistent and the destructive read) is left to the interested reader.

Two final points. (1) We note that the two states $x.\mathsf{empty}.\mathsf{empty}$ and $x.\mathsf{empty}$ are indistinguishable (bisimilar), and indeed have equal interpretations in the terminal coalgebra $1 + A$. But there is no way that we can prove from the equations in the persistent read class that $x.\mathsf{empty}.\mathsf{empty}$ and $x.\mathsf{empty}$ are equal since we have no equations between states. (2) One may be tempted from an algebraic perspective to see the "creation" part in a class as the description of a constant $\mathsf{new}\colon 1 \rightarrow X$. One can then investigate what the initial model of the specification is. In the above persistent read example it is not the (minimal) set $1 + A$ of internal states that comes out in the coalgebraic approach. We algebraically one gets more, since one cannot show that the closed terms new and new.empty are the same.

A coffee and tea machine
As third example we sketch a class of objects that can be understood as elementary machines handling coffee and tea requests. There are methods coin for inserting a coin, liq for making a choice between coffee and tea, and add to choose whether one wishes the coffee or tea to be black (b), with milk (m), with sugar (s), or both with milk and sugar (ms). The interesting aspect is that we use a fourth *private* method status to describe the internal state of the machine. The user of objects belonging to this class is not supposed to have access to this method. With this status method we can express how the public methods change the local state. This a crucial technique in coalgebraic specification. For convenience we assume that only one type of coin is used.

```
class: CTM
    public methods:
        $\mathsf{coin}\colon X \longrightarrow X$
        $\mathsf{liq}\colon X \times \{\mathrm{c}, \mathrm{t}\} \longrightarrow \{\mathsf{error}\} + X$
        $\mathsf{add}\colon X \times \{\mathrm{b}, \mathrm{m}, \mathrm{s}, \mathrm{ms}\} \longrightarrow \{\mathsf{error}\} + \{\mathrm{bc}, \mathrm{mc}, \mathrm{sc}, \mathrm{msc}, \mathrm{bt}, \mathrm{mt}, \mathrm{st}, \mathrm{mst}\} \times X$
    private methods:
        $\mathsf{status}\colon X \longrightarrow \{0, 1, \mathrm{c}, \mathrm{t}\}$
    equations:
        $x.\mathsf{status} = 0 \vdash x.\mathsf{coin}.\mathsf{status} = 1$
        $x.\mathsf{status} = s \vdash x.\mathsf{coin}.\mathsf{status} = s$            for $s \in \{1, c, t\}$
        $x.\mathsf{status} = 1 \vdash x.\mathsf{liq}(a).\mathsf{status} = a$
        $x.\mathsf{status} = s \vdash x.\mathsf{liq}(a) = \mathsf{error}$             for $s \in \{0, c, t\}$
        $x.\mathsf{status} = s \vdash x.\mathsf{add}(a).\mathsf{fst} = ‘as'$            for $s \in \{c, t\}$
        $x.\mathsf{status} = s \vdash x.\mathsf{add}(a).\mathsf{snd}.\mathsf{status} = 0$   for $s \in \{c, t\}$
        $x.\mathsf{status} = s \vdash x.\mathsf{add}(a) = \mathsf{error}$             for $s \in \{0, 1\}$
    creation:
        $\mathsf{new}.\mathsf{status} = 0$
endclass
```

One sees how the private status method describes the four different internal states that are relevant: status 0 means waiting for a coin, status 1 means waiting for a choice of coffee or tea, and status c/t means waiting for a choice of additive, to be combined with the already known choice for coffee (c) or tea (t). In this sense we can describe what is the order in which the messages should be sent to get appropriate results. But of course, the objects in this class are able to handle messages coming in any order—by possibly giving error outcomes. Coalgebraic specification is quite flexible in this sense.

The terminal coalgebra is in this case precisely this (minimal) set of internal status $\{0, 1, \mathrm{c}, \mathrm{t}\}$. Alternatively, it may be seen as the number 4. We leave it to the reader to implement the above methods on this carrier set.

For an example of an elementary, coalgebraically specified, database, see [21, 5.4], where there is a method $\mathsf{store}\colon X \times K \times A \longrightarrow X$ which allows one to store data from A under a key from K.

4. Local operational semantics

In this section we describe the operational semantics $\mathcal{O}(p)$ of a single object p as the tree of all possible transitions that start from p. (We thus use a "branching" semantics, as opposed to a "linear" semantics of traces.) In such transitions the objects identifier and coalgebra remain unaltered, but its local state may change. We shall distinguish between the transitions caused by public methods, and transitions by both public and private methods.

4.1. Definition. Consider an object $p = \langle o, u \in U, c\colon U \to T(U)\rangle$, where $T(-) = T_{\mathrm{pu}}(-) \times T_{\mathrm{pr}}(-)$ is the functor combining the signatures of public and private methods. We take the two terminal coalgebras $Z \xrightarrow{\sim} T(Z)$ and $Z_{\mathrm{pu}} \xrightarrow{\sim} T_{\mathrm{pu}}(Z_{\mathrm{pu}})$ of the entire signature, and of the public signature only. Then, by terminality, we get two coalgebra maps ! and $!_{\mathrm{pu}}$ in diagrams:

$$\begin{tikzcd}
T(U) \arrow[r, dashed, "T(!)"] & T(Z) \\
U \arrow[u, "c"] \arrow[r, dashed, "!"'] & Z \arrow[u, "\cong"']
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
T_{\mathrm{pu}}(U) \arrow[r, dashed, "T_{\mathrm{pu}}(!_{\mathrm{pu}})"] & T_{\mathrm{pu}}(Z_{\mathrm{pu}}) \\
U \arrow[u, "\pi \circ c"] \arrow[r, dashed, "!_{\mathrm{pu}}"'] & Z_{\mathrm{pu}} \arrow[u, "\cong"']
\end{tikzcd}$$

We then assign operational meanings $\mathcal{O}(p) \in Z$ and $\mathcal{O}_{\mathrm{pu}}(p) \in Z_{\mathrm{pu}}$ to the object p by putting $\mathcal{O}(p) = !(u)$ and $\mathcal{O}_{\mathrm{pu}}(p) = !_{\mathrm{pu}}(u)$.

The operational semantics is thus obtained ("by coinduction") via the unique map into a terminal coalgebra. This is dual to the usual way a denotational semantics is defined, namely ("by induction") as unique map going out of an initial algebra (of terms). Remember from the explicit description of terminal coalgebras in Lemma 2.1 that both $\mathcal{O}(p)$ and $\mathcal{O}_{\mathrm{pu}}(p)$ are infinite trees.

The standard result Lemma 2.4 gives us the following.

4.2. Lemma. *Two objects p, q belonging to the same class are bisimilar if and only if they have the same public operational semantics, i.e. if and only if $\mathcal{O}_{\mathrm{pu}}(p) = \mathcal{O}_{\mathrm{pu}}(q)$.* □

This means that two objects are indistinguishable by using their public methods if and only if the associated trees of public observations are equal. We can give an explicit description of these trees $\mathcal{O}(p)$ and $\mathcal{O}_{\mathrm{pu}}(p)$ via single transition steps for objects. For convenience, we shall do this for $\mathcal{O}(p)$ only.

4.3. Definition. Consider an object $p = \langle o, u \in U, c\colon U \to T(U)\rangle$, where T is the functor $X \mapsto \prod_{i \le n}(B_i + C_i \times X)^{A_i}$ as used before. The single transition steps

$$\langle o, u \in U, c\colon U \to T(U)\rangle \xrightarrow[y]{x} \langle o, u' \in U, c\colon U \to T(U)\rangle$$

where $x \in A = A_1 + \cdots + A_n$ is an input, and $y \in B + C = (B_1 + \cdots + B_n) + (C_1 + \cdots + C_n)$ is an output, is defined as follows. For $x = \langle i, a\rangle \in A$ with $a \in A_i$ one has

$$\begin{cases} y = c_i(u, a) \in B_i \text{ and } u' = u & \text{if } c_i(u, a) \in B_i \\ \langle y, u'\rangle = c_i(u, a) \in C_i \times U & \text{otherwise} \end{cases}$$

So if the outcome of applying the i-th component c_i of c to the local state u with parameter a is a value in B_i, then the local state does not change; but if it yields both a value in C_i and a new local state u', then the value is visible, but the new local state gives us a different object with the original identifier and coalgebra, but with this new local state.

Notice that the idenitity of an object (as described after Definition 3.4) does not change under transition. Thus we have that objects are *persistent* entities.

4.4. Lemma. *The operational semantics $\mathcal{O}(p)$ as an element of the set Z of trees $A^+ \to B+C$ from Lemma 2.1 may be described explicitly as:*

$$\mathcal{O}(p)\big(\langle x_n, x_{n-1}, \ldots, x_1\rangle\big) = y \;\Leftrightarrow\; \left\{ \begin{array}{l} \text{there are objects } p_1, \ldots, p_n \text{ and } y_1, \ldots, y_{n-1} \in B+C \text{ with} \\ p \xrightarrow[y_1]{x_1} p_1 \xrightarrow[y_2]{x_2} \cdots \xrightarrow[y_{n-1}]{x_{n-1}} p_{n-1} \xrightarrow[y]{x_n} p_n. \end{array} \right.$$

Proof. This is because the description in the lemma is the unique map to the terminal coalgebra, applied to the local state of p. □

5. Terminal coalgebras satisfying equations

In Lemma 2.1 we have described terminal coalgebras of functors associated with signatures of methods, whereby the equations were ignored. The carrier sets of these terminal coalgebras are rather large sets of infinite trees. It turns out that in many cases one can cut down this set considerably by imposing certain additional (behavioural) conditions, such as equations in classes. One then considers the terminal coalgebra which satisfies these conditions.

The following comes from [15]. Consider the terminal coalgebra $Z \xrightarrow{\sim} T(Z)$ of a polynomial functor T, and let $E \subseteq Z$ be a subset induced by certain equations. Let $\underline{E}$ be the greatest mongruence (see Definition 2.3 (ii)) on $Z \to T(Z)$ which is contained in E. Then $\underline{E}$ inherits a coalgebra structure, and is the terminal coalgebra satisfying E. (This procedure is like in algebra, where one cannot just quotient by the relation given by the equations, but one first has to take the associated least congruence relation, and then form the quotient algebra.)

We illustrate this with the example of the persistent read class from Section 3. The associated functor is $T(X) = X^{(A+1)} \times (1+A)$, which has, by Example 2.2, as terminal coalgebra the set of functions $\varphi \in (1+A)^{(A+1)^\star}$ with operations:

$$\varphi.\mathsf{store}(a) = \lambda\alpha.\,\varphi(\alpha \cdot \kappa a), \qquad \varphi.\mathsf{read} = \varphi([\,]), \qquad \varphi.\mathsf{empty} = \lambda\alpha.\,\varphi(\alpha \cdot \kappa' *).$$

The three equations in the persistent read class gives us a subset $E \subseteq (1+A)^{(A+1)^\star}$ consisting of those φ satisfying:

- $\varphi.\mathsf{empty}.\mathsf{read} = *$; i.e. $\varphi(\kappa' *) = *$;
- if $\varphi.\mathsf{read} = *$, then $\varphi.\mathsf{store}(a).\mathsf{read} = a$; i.e. if $\varphi([\,]) = *$, then $\varphi(\kappa a) = a$;
- if $\varphi.\mathsf{read} = a$, then $\varphi.\mathsf{store}(b).\mathsf{read} = a$; i.e. if $\varphi([\,]) = a$, then $\varphi(\kappa b) = a$.

The greatest mongruence $\underline{E} \subseteq E$ is the greatest set $\underline{E} \subseteq E$ satisfying: if $\varphi \in \underline{E}$, then also $\varphi.\mathsf{store}(a) \in \underline{E}$ and $\varphi.\mathsf{empty} \in \underline{E}$. It is easy to see that $\underline{E}$ is then the set of $\varphi \in (1+A)^{(A+1)^\star}$ satisfying for all $\alpha \in (A+1)^\star$, $\varphi((\kappa' *) \cdot \alpha) = *$, if $\varphi(\alpha) = *$ then $\varphi((\kappa a) \cdot \alpha) = a$, and if $\varphi(\alpha) = a$ then $\varphi((\kappa b) \cdot \alpha) = a$. But then we have that each tree $\varphi \in \underline{E}$ is determined by its value $\varphi([\,]) \in 1 + A$. Hence $\underline{E} \cong 1 + A$, and this is the (carrier of the) terminal coalgebra satisfying the equations.

The essential element in the elimination of these trees φ is that they are determined by their output $\varphi([\,])$ at the root. This will be formalized using the operational semantics $\mathcal{O}(-)$ from the previous section.

5.1. Definition. A class will be called **total** (or **totally specified**) when for each object p belonging to the class the following holds: for each $n \geq 1$, if $\mathcal{O}(p)(\langle x_m, \ldots, x_1\rangle)$ is known for each $m \leq n$ and input sequence $\langle x_m, \ldots, x_1\rangle$, then also the outcome of the next step $\mathcal{O}(p)(\langle x_{n+1}, x_n, \ldots, x_1\rangle)$ is known, for each input sequence $\langle x_{n+1}, x_n, \ldots, x_1\rangle$. This means that the entire tree $\mathcal{O}(p)$ is determined by the set of outputs $\mathcal{O}(p)(\langle x\rangle)$ on singleton input sequences.

A class is thus total when we can deduce what the outcome of a next step is (in terms of observable outputs), from what we already know. This means that the equations cover all possible situations that may occur. It may be clear that the persistent read class is total: the output $\mathcal{O}(p)(\langle x_{n+1}, x_n, \ldots, x_1\rangle) \in 1 + A$ is determined by $x_{n+1} \in A + 1$ and by $\mathcal{O}(p)(\langle x_n, \ldots, x_1\rangle) \in 1 + A$, according to the following table.

	$\mathcal{O}(p)(\langle x_n, \ldots, x_1\rangle) = * \in 1$	$\mathcal{O}(p)(\langle x_n, \ldots, x_1\rangle) = a \in A$
$x_{n+1} = * \in 1$	$*$	$*$
$x_{n+1} = b \in A$	b	a

Notice that the carrier of the terminal coalgebra in this situation is the set $T(1) = 1^{(A+1)} \times (1 + A) \cong 1 + A$. The same analysis may be applied to the destructive read class; it gives the same carrier set, but with different operations.

5.2. Proposition. *The carrier of the terminal coalgebra of a* total *class involving a polynomial functor* $T(X) = \prod_{i \leq n}(B_i + C_i \times X)^{A_i}$ *is a subset of* $T(1) = \prod_{i \leq n}(B_i + C_i)^{A_i}$.

Proof. Like in the persistent bank account example, the trees $\varphi\colon A^+ \to B + C$ in the carrier of the terminal coalgebra are determined by their values $\varphi(\langle i, a\rangle) \in B_i + C_i$. These can be described as n functions $A_i \to B_i + C_i$. They combine into an element of $T(1)$. □

This result thus gives us a superset for the carrier of the terminal coalgebra. That we can really get a proper subset of $T(1)$ can be seen in the example of the (total) class of the coffee and tea machine in Section 3, where the coin, liq and add methods on $T(1)$ are determined by the current status. This allows a further simplification of $T(1)$.

5.3. Rule of thumb. The carrier of the terminal coalgebra for a total class is the minimal set of internal states needed to carry out the specified task.

For a total class, the observable output values at creation (for new) should be specified as an element of $T(1)$—or of an appropriate subset of this product of function spaces.

6. Global operational semantics

So far we have only considered objects in isolation. In order to communicate, objects should be able to send messages to each other (including to itself). In this final section we briefly sketch how such communtication may take place via a global transition relation. Many details are left out.

A **message** is a 3-tuple of the form

$$\langle o, m, a\rangle \qquad \text{where} \qquad \begin{cases} o \text{ is an object identifier, representing the target} \\ m \text{ is a method name (occuring in the class in } o) \\ a \text{ is a parameter for } m. \end{cases}$$

For example, we may have a message $\langle\langle \mathrm{BA}, 1\rangle, \mathsf{ch}, 5\rangle$ which, when received by object 1 belonging to the bank account class BA, is intended to cause execution of the method $\mathsf{ch}(5)$ with parameter 5.

Let us write $\mathcal{M}$ for the set of all possible messages (given a certain collection $\mathcal{C}$ of classes). From now on we let $\mathcal{M}$ occur explicitly in the output types in signatures of methods in classes, like in

$$X \times A \longrightarrow B + (C \times \mathcal{M} \times X).$$

An output message $\langle o, m, a\rangle \in \mathcal{M}$ will be understood as an act of sending this message. (This should not be regarded as visible on the outside, so we have to adapt the definition of bisimilation by eliminating $\mathcal{M}$ from the associated functor, and by projecting it away from coalgebras.) For an object p with an input $\langle i, a\rangle$ execution of the i-th method in p on the local state with parameter a may now result in a number of messages as output. We will write this outcome as a (multi-) set $\mathrm{mess}(p, \langle i, a\rangle)$ of messages.

In concurrent object-oriented programming there is no global state containing values of global variables, through which local entities may communicate. One may describe communication via synchronous message passing, as in the language POOL, see [3]. Here we sketch asynchronous communication where there is a global collection of messages waiting to be executed. Related ideas are expressed in [4, 17, 2]. This collection—in a sense—is a substitute for the global state; it may be depicted as a "sea" or "chemical soup" of messages, in which each object can recognize the messages directed at it through uniqueness of identifiers. Thus each object can pick out the relevant messages from this soup, and execute the method in the message. Such executions may be performed concurrently, since there is *no* interference. This is because (1) objects have their own local state, and (2) objects have unique identifiers, so there is no possibility that one object handles a message aimed at another object. The absence of interference is one of the selling points of concurrent object-oriented programming. Of course, there are some (operational) scheduling problems in this set-up. For example, one has to specify how (global) execution proceeds when there is more than one message for a particular object. An obvious approach is to have messages waiting in queues for their target objects, in the (temporal) order in which they arrive.

Since there may be multiple copies of the same message waiting to be executed, we have to take our configurations of messages as multisets. We take as set $\mathcal{S}$ of all these configurations the space $\mathcal{S} = \mathbb{N}^{\mathcal{M}}$ of functions from messages to natural numbers. For $\sigma \in \mathcal{S}$ and $k \in \mathcal{M}$ we see $\sigma(k) \in \mathbb{N}$ to be the number of messages k in the configuration σ. We extend inhabitation $\in$ and union $\cup$ to $\mathcal{S}$ in the obvious way. For example $k \in \sigma$ stands for $\sigma(k) \geq 1$.

We have now prepared the grounds for the description of the global operational semantics. It involves the local transition steps between objects from Definition 4.3, but also transition steps between configurations (multisets) of messages. For a configuration $\sigma \in \mathcal{S} = \mathbb{N}^{\mathcal{M}}$ we have a rule:

$$\frac{\langle o, m, a\rangle \in \sigma \qquad\qquad \mathrm{identifier}(p) = o}{p \xrightarrow[y]{\langle m, a\rangle} p' \qquad\qquad \sigma \longrightarrow (\sigma - \langle 0, m, a\rangle) \cup \mathrm{mess}(p, \langle i, a\rangle)}$$

This rule should be read as follows. If a message $\langle o, m, a\rangle$ occurs in the current configuration of messages σ and if p is the object with identifier o at which this message is targeted, then p can make a single transition step with input $\langle m, a\rangle$—that is, p can execute message m with parameter a, which yields object p' with possibly different local state, see Definition 4.3—and a single occurrence of the message $\langle o, m, a\rangle$ is removed from the configuration σ, while the output messages in $\mathrm{mess}(p, \langle i, a\rangle)$ produced in the transition $p \rightarrow p'$ are added to the configuration. We thus have both a local and a global step.

7. Concluding remarks

We have presented a coalgebraic formalism to describe some of the basic concepts of object-oriented programming. Subtyping and inheritance do not form part of the picture (so far). Some of the characteristics of the coalgebraic perspective are listed below.

1. An object has a local state to which one only has access via the public methods of the object. We do not know anything about this local state, except what these methods tell us. This emphasis on observation is characteristic of coalgebra, as opposed to algebra where construction is the key aspect.

2. An object combines both data structure and behaviour; the former is explicit in the signature of operations in its class, and the latter in the operational semantics.

3. One only has *unary* methods, acting on a single local state. Thus there are no binary methods, of the form
$$X \times X \longrightarrow B + C \times X$$
Such binary methods are excluded in the coalgebraic approach, since they lead to contravariant functors. But on a different level binary methods also present problems in combination with inheritance, see [5] for an extensive discussion.

4. An object has no autonomous activity: it acts only in reaction to incoming messages. But an object may send messages to itself.

5. Parallellism only occurs at a global level between objects, and not within objects. But there is some degree of non-determinism within objects, since an object does not know which method will be executed next, and with which parameter. Also, the output type of a method can contain a coproduct $+$ so that it may not be known in advance which alternative is selected.

6. A "two-layered" semantics has been described involving local and global phenomena. For the language POOL a "three-layered" semantics is given in [3], where meanings are assigned to methods in terms of a third level, containing the meanings of elementary program statements.

Much further work remains to be done. For example investigation of some serious examples involving communication and of modularisation mechanisms for coalgebraic specifications, and comparison to other specification formalisms.

Acknowledgement

Thanks are due to Jan Rutten for clarifying discussions.

References

1. P. Aczel and N. Mendler. A final coalgebra theorem. In D.H. Pitt, A. Poigné, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, number 389 in Lect. Notes Comp. Sci., pages 357–365. Springer, Berlin, 1989.
2. G. Agha. *Actors.* MIT Press, 1986.

3. P. America and J. Rutten. A layered semantics for a parallel object-oriented language. *Formal Aspects of Comp.*, 4:376–408, 1992.

4. G. Berry and G. Boudol. The chemical abstract machine. In *Princ. of Progr. Lang.*, pages 81–94. ACM, 1990.

5. K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. Leavens, and B. Pierce. On binary methods. Manuscript, May 1995.

6. R. Burstall and R. Diaconescu. Hiding and behaviour: an instiutional approach. In A.W. Roscoe, editor, *A Classical Mind. Essays in honour of C.A.R. Hoare*, pages 75–92. Prentice Hall, 1994.

7. L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Comp. Surv.*, 4:471–522, 1985.

8. J.R.B. Cockett and D. Spencer. Strong categorical datatypes I. In R.A.G. Seely, editor, *Category Theory 1991*, number 13 in CMS Conference Proceedings, pages 141–169, 1992.

9. W.R. Cook. Object-oriented programming versus abstract data types. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in Lect. Notes Comp. Sci., pages 151–178. Springer, Berlin, 1990.

10. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Number 6 in EATCS Monographs. Springer, Berlin, 1985.

11. L.M.G. Feijs and H.B.M. Jonkers. *Formal Specification and Design.* Number 35 in Tracts in Theor. Comp. Sci. Cambridge Univ. Press, 1992.

12. J. Fiadeiro and T. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal Aspects of Comp.*, 4:239–272, 1992.

13. J.A. Goguen. Types as theories. In G.M. Reed, A.W. Roscoe, and R.F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford Univ. Press, 1991.

14. M. Hofmann and B. Pierce. An abstract view of objects and subtyping. *Journ. Funct. Progr.*, 1995, to appear.

15. B. Jacobs. Mongruences and cofree coalgebras. In ???, editor, *Algebraic Methods and Software Technology*, Lect. Notes Comp. Sci., page ??? Springer, Berlin, 1995, to appear.

16. S. Kamin. Final data types and their specification. *ACM Trans. on Progr. Lang. and Systems*, 5(1):97–123, 1983.

17. J. Meseguer. A logical theory of concurrent objects. In N. Meyrowitz, editor, *OOPSLA, ECOOP'90 Proceedings*, pages 101–115. ACM, 1990.

18. J. Meseguer and J. Goguen. Initiality, induction and computability. In M. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge Univ. Press, 1985.

19. J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Trans. on Progr. Lang. and Systems*, 10(3):470–502, 1988.

20. B.C. Pierce and D.N. Turner. Simple type theoretic foundation for object-oriented programming. *Journ. Funct. Progr.*, 4(2):207–247, 1994.

21. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. Comp. Sci.*, 1995, to appear.

22. M.B. Smyth and G.D. Plotkin. The category theoretic solution of recursive domain equations. *SIAM Journ. Comput.*, 11:761–783, 1982.

23. R. Wieringa. Equational specification of dynamic objects. In R.A. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction (DS-4)*, IFIP, pages 415–438. Elsevier Sci. Publ., 1991.