1

OBJECTS AND CLASSES, CO-ALGEBRAICALLY

Bart Jacobs

CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.

Email: `bjacobs@cwi.nl`

ABSTRACT

The co-algebraic perspective on objects and classes in object-oriented programming is elaborated: classes are described as co-algebras, which may occur as models (implementations) of co-algebraic specifications. These specifications are much like deferred (or virtual) classes with assertions in Eiffel. An object belonging to a class is an element of the state space of the class, as co-algebra. We show how terminal co-algebras of co-algebraic specifications give rise to canonical models (in which all observationally indistinguishable objects are identified). We further describe operational semantics for objects, with an associated notion of bisimulation (for objects in classes modeling the same specification), expressing observational indistinguishability.

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 determine (among other things) the interface to the outside world (of the objects belonging to the class). Objects have private data, only accessible via specified operations, called "methods", which are provided by the class of the object. 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. See e.g. [5, 10, 27] for more background information. The object-oriented paradigm is both popular and successful, but a general complaint is that it lacks a proper formal foundation: it is more philosophy than mathematics. In this paper we describe a semantics for objects and classes using so-called "co-algebras". These are the formal duals of algebras. The essential difference between algebras and co-algebras is that the former have "constructors" (operations going *into* the underlying carrier set, which are used to build elements) where the latter have "destructors" or "observers" (operations going *out of* the carrier set, which allow us to observe certain behaviour). This distinction between construction and behaviour is in essence the distinction between abstract data types and procedural abstraction described in [9]. (The terminology of "constructors" and "destructors" comes from data type theory, and has no connection with constructors and destructors in C++, for example) In co-algebra one deals with state spaces as 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 and from (dynamical) system theory. The notion of bisimulation forms an intrinsic part of the co-algebraic view. It means indistinguishability of behaviour, as it can be observed via the specified (co-algebraic) operations that we have at our disposal. It arises automatically in a situation with limited access to a state space.

We shall distinguish between class *specifications*, and class *implementations*. The latter will often simply be called classes. A class specification is like an abstract class, in which methods with their signatures are given, but without their actual implementation. But assertions are there to put behavioural constraints on methods. Implementation of the methods is given in a class implementation—also called a concrete class, or simply a class. The essentials are put in the class specification, and the particulars in the class implementation (which is of no concern to a client). Such a separation is useful in situations where implementation details vary (e.g. from platform to platform, or from time to time). Also, it opens up the possibility of formal verification of classes.

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 co-algebraic description of objects. The suitability of co-algebras for the description of object-oriented features was recognized before, see e.g. [24, 15, 16]. Elements may be traced back to earlier sources, like [19, 20, 9, 23], where co-algebras are not explicitly used (in [20] one finds the phrase "abstract machine" instead). In [11] the two-level structure of specifications in the object-oriented design 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 co-algebraic (behavioural) specification. There are two similar levels in FOOPS, described by functional modules and object modules, see [14]. 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 co-algebraic approach below. This paper elaborates ideas from [24] and [16]. What we consider as the main points are the following.

(1) It describes a (set theoretic) semantics for some crucial notions of object-oriented programming: there are precise notions of object, of class (implementation) and of class specification. We focus on the meaning of the concepts, and not on syntactic details (of a particular language) but in spirit our approach is close to Eiffel [21]. It is a semantical study into object-orientation.

(2) It shows (following [24]) how behaviour can be specified co-algebraically (using conditional equations). Further, it gives operational semantics for objects (in isolation), with an associated notion of bisimulation.

(3) It describes canonical (terminal co-algebra) implementations—or "minimal realizations", in system theoretic terminology (see also [12])—of class specifications, using techniques developed in [16]. It is somewhat surprising to see that although (carriers of) terminal co-algebras obtained from methods alone are generally huge sets of infinite trees (see Lemma 6), one can cut down these sets to very reasonable size in case the behaviour is well-specified.

(4) And it makes effective use of coproduct (disjoint union) types $+, 0$ for structuring the outputs of methods. In this way the traditional distinction between functions and procedures disappears.

In a follow-up paper [17] the co-algebraic approach is used to describe inheritance, a key concept of object-oriented programming. And *temporal* co-algebraic specifications may be found in [18]. We shall make some use of elementary category theory in order to organize the concepts involved. In using 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 ALGEBRAS VERSUS CO-ALGEBRAS

Assume we wish to specify a datatype X of binary A-labeled trees, for some set of labels A. Algebraically one describes how to build up such trees by giving their "constructors" nil and node, as on the left below (where $1 = \{*\}$ is a one-element set): a binary A-labeled 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.

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

A co-algebraic specification of such trees is given on the right. It does not give the "constructors", but the "destructors" (or "observers"): 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 inside X. This X is best considered as a black box to which we only have limited access via the operations. These examples already suggest that algebra is about construction and co-algebra is about (observation of) behaviour. Mathematically, the distinguishing difference between the algebraic and the co-algebraic 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 (disjoint unions) $+$ 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] (or [8], describing an experimental programming language CHARITY with essentially only such algebras and co-algebras).

The above algebraic specification has a canonical model given by the *initial algebra*. It consists of all finite binary A-labeled trees, and may be constructed as the set of closed terms. Also for the co-algebraic specification on the right there is a canonical model, given by the *terminal co-algebra*. It consists of the infinite binary A-labeled trees, and may be obtained as the set of "trees of observations". Initial algebras form a basis for data type semantics (see e.g. [28]), and terminal co-algebras play a similar role in an object-oriented setting. Algebraic specification is useful for the formal description of datastructures, but not of state-based systems. In contrast, states are unproblematic in co-algebraic specification.

The general definition of an **algebra** is a map of the form $T(X) \to X$, for some functor[1] $T\colon \mathbf{Sets} \to \mathbf{Sets}$ on the category **Sets** of sets and functions (or on some other category). And a **co-algebra** is a map $X \to T(X)$ in the reverse direction. Such a co-algebra $X \to T(X)$ consists of a state space X (also called the carrier) together with a transition function (or dynamics) $X \to T(X)$ acting on the state space. If we have two algebras $(c\colon T(U) \to U)$ and $(d\colon T(V) \to V)$,

[1] This means that T is an operation $X \mapsto T(X)$ acting on sets, but at the same time an operation $(f\colon X \to Y) \mapsto (T(f)\colon T(X) \to T(Y))$ acting on functions, preserving identities and composition. For simplicity one may read $T(X)$ as an expression for a set containing a variable set X, like in $(*)$ below. Functoriality will play a role later in Section 5.

then we say that an **algebra map** $c \to d$ is a morphism $f: U \to V$ 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 co-algebras of T: a **co-algebra map** $(c: U \to T(U)) \to (d: V \to T(V))$ is a morphism $f: U \to V$ with $d \circ f = T(f) \circ c$. We recall that in an arbitrary category $\mathbb{C}$ an object $1 \in \mathbb{C}$ is **terminal** if for each object $X \in \mathbb{C}$ there is precisely one arrow $X \dashrightarrow 1$. Singleton sets are terminal objects in the category **Sets**; we typically write $1 = \{*\}$. The dual notion is that of **initial** object 0, for which there is precisely one map $0 \dashrightarrow X$ 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 two projections $\pi: X \times Y \to X$, $\pi': X \times Y \to Y$ and a tupling operation $\langle -, - \rangle$. Dually, we have a coproduct $X + Y$ with the property that 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 $\kappa: X \to X + Y$, $\kappa': Y \to X + Y$ 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 $X \times 0 \cong 0$. 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 co-algebra $c: U \to T(U)$ of this functor may be identified with a collection of maps $c_1: U \times A_1 \to B_1 + C_1 \times U, \ldots, c_n: U \times A_n \to B_n + C_n \times U$. A co-algebra forms in this way a model of a certain signature of operations (i.e. methods) on a state space U. And a co-algebra map is a map between the state spaces which commutes with the operations. Note that the c_i are maps going *out of* U, with a parameter from A_i.

3 EXAMPLES OF CO-ALGEBRAIC SPECIFICATION

We start with a specification of a class of rudimentary bank accounts (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 parameter of change. We use hopefully self-explanatory notation, in the following specification—with some comments after the '#' sign.

class spec: BA	# name of the specification; BA for Bank Account
public methods:	
$\mathsf{bal}\colon X \longrightarrow \mathbb{Z}$	# this is an attribute, or instance variable
$\mathsf{ch}\colon X \times \mathbb{Z} \longrightarrow X$	# this is a procedure, with parameter from $\mathbb{Z}$;
	# it affects the local state space X.
assertions:	
$\mathsf{s.ch}(a).\mathsf{bal} =$	# in OO-style with post fix notation, where $\mathsf{s} \in X$
$\mathsf{s.bal} + a$	# with 's' for 'self' or 'state' is a local state
creation:	
$\mathsf{new.bal} = 0$	
end class spec	

In this specification we say what methods we want for our bank account and which (equational) assertions should hold. The equation $\mathsf{s.ch}(a).\mathsf{bal} = \mathsf{s.bal} + a$ should be read as: if one sends state s the change message ch with parameter a and then asks for the balance bal, then the outcome is the same as first asking state s for its balance, and then adding the amount a. The last point of the specification mentions that newly created objects of this class BA have $0 \in \mathbb{Z}$ as their balance. It describes the behaviour of the initial state. As an observer on the outside, we do not really care how the operations of such bank accounts are implemented in a class, 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 co-algebra. We notice that such a class specification is very much like a deferred (or virtual) class with assertions in Eiffel: only the methods are given with their input and output types—and not their implementation—and the behaviour of these methods is determined by assertions. The above equation may be seen as a post-condition for the change method. Such class specifications are called (behavioural) types in [2]. In the next section we shall define a class (satisfying a specification) as a co-algebra interpreting the function symbols in such a way that the assertions hold. This clear (model-theoretic) distinction between a class specification and its implementation corresponds to the distinction in actual languages between abstract classes all of whose methods are deferred and concrete classes of all whose methods are implemented.

Here is another example. Let A be a fixed set of data elements. We wish to specify a class of buffer objects of capacity one, which can contain a single element $a \in A$, but which may also be empty. The methods 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: (1) what happens when we send the $\mathsf{store}(a)$ message to a buffer which is already full [we choose that nothing will happen]; (2) what happens when we read from an empty buffer [the (observable) outcome will be an error value], and (3) what happens when we read from a full buffer: 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 class specifications PR for the persistent read buffers (on the left), and DR for the destructive read buffers (on the right). We emphasize that these buffers are specified co-algebraically: we only say which operations we have and how they behave (via assertions), and nothing about what is inside the (state space X of the) buffers.

class spec: PR
  public methods:
    $\mathsf{store}\colon X \times A \longrightarrow X$
    $\mathsf{read}\colon X \longrightarrow \{\mathsf{error}\} + A$
    $\mathsf{empty}\colon X \longrightarrow X$
  assertions:
    $\mathsf{s.empty.read} = \mathsf{error}$
    $\mathsf{s.read} = \mathsf{error} \vdash$
      $\mathsf{s.store}(a)\mathsf{.read} = a$
    $\mathsf{s.read} = a \vdash$
      $\mathsf{s.store}(b)\mathsf{.read} = a$
  creation:
    $\mathsf{new.read} = \mathsf{error}$
end class spec

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

The main difference between these specifications is that persistent 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 $\{\mathsf{error}\} + A \times X$ of the output of the destructive read method. We see how the traditional distinction between functions (having outputs, but no effect on the state) and procedures (having an effect on the state, but no output) disappears: the destructive read method $\mathsf{read}\colon X \to \{\mathsf{error}\} + A \times X$ yields an output error without affecting the state if the buffer is empty, and yields both an output and an effect on the state otherwise. Notice how this is reflected in the typing, via the coproduct $+$. One may wish to push this example further and specify buffers (still with capacity one) which can contain elements from a dataset A persistently and from a set B destructively. This requires a read method $\mathsf{read}\colon X \to \{\mathsf{error}\} + A + B \times X$, yielding either an error value, or an element in A (without affecting the state), or an element in B with a next state. The appropriate equations are left as an exercise in co-algebraic specification.

4 OBJECTS, CLASS IMPLEMENTATIONS AND CLASS SPECIFICATIONS

The main aspect of an object that we wish to capture co-algebraically is that it has a local state, which is only accessible via specified operations (implemented in the class of the object). Classes (or, class implementations) will be presented as co-algebraic models of class specifications, and objects (belonging to a class) as inhabitants of the carrier set of the class (as co-algebra).

Definition 1 *A* **class specification** *is a structure which has a name, and consists of three components.*

(i) *A finite set of (unary) "methods" (or "features" in Eiffel or "members" in C++) of the form*

$$X \times A_i \longrightarrow B_i + C_i \times X$$

on a local state space X. The functor T associated with this signature of, say, n such co-algebraic operations is

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

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 called* **procedures**.

(ii) *Assertions, which may be conditional. They regulate the behaviour of the objects belonging to the class.*

(iii) *The observable properties which hold for newly created objects, using* new. *These may be either with or without parameters.*

We shall use a lay-out for class specifications as in the previous section for the bank account and buffer examples. Each such specification introduces a single new type, for which we write X inside the specification, but for which one may use the specification's name outside. No binary methods (of the form $X \times X \longrightarrow B + C \times X$) are allowed in the co-algebraic approach, since they lead to contravariant functors. (But on a different level binary methods also present (typing) problems in combination with inheritance, see [4] for an extensive discussion.) In the specifications 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 to formulate the assertions. (In Eiffel the assertions should be executable, because they are used not only for specification but also for run-time monitoring.) One may distinguish between *public* and *private* methods, where one object may only send messages requiring execution of a public method in another object. But an object may send messages to itself asking for execution of its own private methods. The methods that we consider have output types $B + C \times X$. This means that they can produce either an observable element in B, or an observable element in C together with a new state in X. If $B = 0$, then we only have the second option, and if $C = 0$, only the first one remains. We can also capture methods of the form $X \times A \longrightarrow X + D \times X$ 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 $+$).

Definition 2 *Consider a class specification as in the previous definition, with functor T associated with the signature of methods.*

(i) *A* **class** *satisfying this class specification consists of three elements:*

(a) *a carrier set U, giving an interpretation of the state space;*

(b) *a co-algebra $c: U \to T(U)$ interpreting (or: implementing) the methods in such a way that the assertions are satisfied (recall that the function c can be identified with a set of functions $c_i: U \times A_i \to B_i + C_i \times U$);*

(c) *an initial state $u_0 \in U$ which satisfies the condition in the creation section of the class specification (see (ii) below).*

(ii) *An* **object** *belonging to the class $c: U \to T(U)$ in (i) is simply an element $u \in U$ of the carrier set of the class. Sending the method implemented by c_i with parameter $a \in A_i$ to the object u is interpreted via function application as*

$$u.c_i(a) \stackrel{\text{def}}{=} c_i(u, a) \in B_i + C_i \times U.$$

Coming back to (i), *the '*new*' operation applied to a class $\langle c: U \to T(U), u_0 \rangle$ yields as object of the class the initial state $u_0 \in U$.*

In this picture a class contains the code (implementation) of the methods, which is the same for all objects of the class. And an object contains the particulars, such as the data values, which can be inspected via the attributes implemented in the class. In some situations it may be more convenient to define an object as a pair $\langle u \in U, c: U \to T(U) \rangle$ consisting of an object $u \in U$ in the above sense together with its class. One may wish to add a natural number as third field, which can serve as unique identifier of the object. This is especially useful when one considers systems of objects. 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 (the co-algebra of its class) remain the same. One can 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 (see the next section) more objects are identified.

A class is often considered as a combination of two aspects: it is at the same time seen as a type and as a module, see e.g. [21]. This fits well into the above interpretation: the "class as a type" is the underlying set U, inhabitants of which are the objects belonging to the class. And the "class as a module" is the co-algebra $c: U \to T(U)$ giving us a data type structure on the type U. For convenience we often describe a class by only giving its co-algebra, without mentioning its initial state u_0 explicitly. This initial state usually arises via a special part of a class definition, called "make" in Eiffel and "constructor" in C++. In the type theoretic encoding of object-oriented constructs into second (or higher) order polymorphic lambda calculus (with subtyping), see e.g. [7, 23, 15], 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 [22]). An inhabitant of the product type $\alpha \times (\alpha \to T(\alpha))$ is a tuple consisting of a local state in α and a co-algebra $\alpha \to T(\alpha)$, like in the above definition. But in this type theoretic encoding there is no explicit way to deal with assertions; they play an essential role in the specification of behaviour. One may also view an object $u \in U$ together with its class $c: U \to T(U)$ as a particular kind of automaton, with u as current state of the automaton, and with the co-algebra 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 by a client, and with which parameter. Also the coproduct $+$ in output types introduces an element of non-determinism.

As illustration of the above definition of class and object, we shall consider the bank account specification BA from the previous section: we shall present three possible implementations, with different interpretations of the state space X and of the methods $\mathsf{bal}, \mathsf{ch}$. But these differences are not visible to clients. The functor associated with the BA-signature of methods is $T(X) = \mathbb{Z} \times X^{\mathbb{Z}}$.

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

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

These two methods together form a co-algebra $c_1: U_1 \rightarrow T(U_1)$. It obviously satisfies the equation $\mathsf{s.ch}(a).\mathsf{bal} = \mathsf{s.bal} + a$. As initial state we can take the empty sequence $\langle\rangle$ in U_1. The pair $\langle\langle\rangle, U_1 \rightarrow T(U_1)\rangle$ thus forms an example of a class satisfying the BA-specification. And an example of an object belonging to this class is the sequence $\langle 2, -3\rangle \in U_1$ containing some specific data. The balance of this bank account object is -1. This is a rather inefficient implementation: asking for the balance involves adding up all the changes that have been made. But for a client who can only access objects via the balance and change methods, these implementation details are not visible.

(2) Our second implementation 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 $\mathsf{s} = \langle a_1, \ldots, a_n\rangle \in U_2$ we define

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

This gives us a co-algebra $c_2: U_2 \rightarrow T(U_2)$, which also satisfies the equation. An object of this class consists of a non-empty sequence of integers, with the last integer in the sequence as its current balance. So as initial state one can take the sequence consisting only of $0 \in \mathbb{Z}$. (But we could also take the state $\langle 1, 0\rangle$; it is "bisimilar" to $\langle 0\rangle$, see the next section.)

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

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

A bank account object with this co-algebra, 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 implementation, containing all the information we need, and nothing else. In a mathematical sense it distinguishes itself as the "terminal co-algebra", i.e. as the terminal object in the category of co-algebras $X \rightarrow \mathbb{Z} \times X^{\mathbb{Z}}$ satisfying the bank account equation, see Section 6. The co-algebraic approach thus allows us to characterize these minimal realizations.

5 INDISTINGUISHABILITY (BISIMULATION) FOR OBJECTS

In this section we shall go deeper into the technicalities of co-algebras, using some elementary category theory. To start, consider the two bank account objects $p_1 = \langle\langle 2, -3\rangle, c_1: \mathbb{Z}^\star \rightarrow T(\mathbb{Z}^\star)\rangle$ and $p_3 = \langle -1, c_3: \mathbb{Z} \rightarrow T(\mathbb{Z})\rangle$ belonging to the first and third class at the end of the previous section. These objects p_1 and p_3 are indistinguishable from the outside because we cannot see a difference, using the public methods specified for bank accounts: 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 specification. In process theory this notion of "indistinguishability via observations" is called "bisimilarity". The two objects p_1 and p_2 are bisimilar because there is a bisimulation relation $R \subseteq \mathbb{Z}^\star \times \mathbb{Z}$ with $R(\langle 2, -3\rangle, -1)$, namely

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

Definition 3 *Let T: **Sets** $\rightarrow$ **Sets** be the general polynomial functor $T(X) = \prod_{i \leq n}(B_i + C_i \times X)^{A_i}$, and let $c: U \rightarrow T(U)$ and $d: V \rightarrow T(V)$ be two co-algebras of this functor (describing classes). A relation $R \subseteq U \times V$ on the two state spaces is called a* **bisimulation** *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 $c_i(x,a) = \langle \gamma, x'\rangle$ and $d_i(y,a) = \langle \gamma, y'\rangle$ with equal first 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 relation $R \subseteq U \times V$ with $R(u,v)$. In this case one writes $u \,\underline{\leftrightarrow}\, v$.*

Bisimulations are thus relations on carriers of co-algebras which are suitably closed under the co-algebra operations. One can describe this notion (and also the notion of mongruence in the next section) more abstractly in terms of the functor T involved, see [1, 25], or [16]. Next we define bisimilarity for objects. This notion is intended to capture observational indistinguishability and will therefore only involve the publicly available methods.

Definition 4 *Assume a class specification with two functors $T_{\mathrm{pu}}, T_{\mathrm{pr}}$: **Sets** $\rightrightarrows$ **Sets** describing the signatures of respectively the public and private methods. Two objects $u_1 \in U_1$ in class $c_1: U_1 \rightarrow T_{\mathrm{pu}}(U_1) \times T_{\mathrm{pr}}(U_1)$ and $u_2 \in U_2$ in class $c_2: U_2 \rightarrow T_{\mathrm{pu}}(U_2) \times T_{\mathrm{pr}}(U_2)$ with c_1 and c_2 satisfying the specification, will be called* **bisimilar** *if there is a bisimulation relation $R \subseteq U_1 \times U_2$ with respect to the co-algebras $\pi \circ c_1: U_1 \rightarrow T_{\mathrm{pu}}(U_1)$ and $\pi \circ c_2: U_2 \rightarrow T_{\mathrm{pu}}(U_2)$ of the "public" functor T_{pu}, implementing the public methods.*

The following standard result gives an equivalent description of bisimulation $\underline{\leftrightarrow}$ in terms of terminal co-algebras, see e.g. [25].

Lemma 5 *Consider two co-algebras $c: U \to T(U)$ and $d: V \to T(V)$ of the same functor T. They induce two unique co-algebra maps $!_c$ and $!_d$ to the terminal co-algebra $Z \xrightarrow{\cong} T(Z)$, 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 co-algebras are then bisimilar if and only if they have the same value on the terminal co-algebra, i.e. $u \underline{\leftrightarrow} v$ if and only if $!_c(u) = !_d(v)$. □

The terminal co-algebra in this lemma is the terminal objects in the category CoAlg(T) of co-algebras of the functor T. There is a standard construction (see e.g. [26]) to compute such a terminal co-algebra via the limit Z of the diagram $1 \leftarrow T(1) \leftarrow T^2(1) \leftarrow \cdots \leftarrow Z$. This construction applies for the above polynomial functors because they preserve limits of such chains. We shall give an explicit description of this terminal co-algebra.

Lemma 6 *The terminal co-algebra $Z \xrightarrow{\cong} T(Z)$ of the polynomial functor $T(X) = \prod_{i \leq n}(B_i + C_i \times X)^{A_i}$ on* **Sets** *can be described as a set of infinite trees. Therefore, first write $A = A_1 + \cdots + A_n$, $B = B_1 + \cdots + B_n$ and $C = C_1 + \cdots + C_n$ for the disjoint unions of the constants in the functor. We now have*

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

where $\cdot$ is concatenation of sequences (from A^+). The interpretations of the methods $Z \times A_i \to B_i + C_i \times Z$ are given by

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

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 both finite and infinite lists of C's may be identified with the set of infinite trees $\{\varphi: \mathbb{N} \to 1 + C \mid \forall n \in \mathbb{N}. \varphi(n) = * \Rightarrow \varphi(n+1) = *\}$. This is the terminal co-algebra of the functor $T(X) = 1 + C \times X$, according to the lemma.

Example 7 (See [24]) *A useful special case of the above lemma is the following: the terminal co-algebra 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*. The attribute* $Z \to B$ *is given by* $\varphi \mapsto \varphi([\,])$ *and the procedure* $Z \times A \to Z$ *by* $(\varphi, a) \mapsto \lambda\alpha \in A^\star.\, \varphi(\alpha \cdot a)$*. In [24] only these restricted signatures (without coproducts) are used. They form a special case of the polynomial functor* T *above for* $n = 2$ *and* $A_1 = 1$*,* $B_1 = B$*,* $C_1 = 0$*,* $A_2 = A$*,* $B_2 = 0$ *and* $C_2 = 1$*.*

In the remainder of 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. In such transitions the objects identifier and co-algebra 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.

Definition 8 *Consider an object* $p = \langle u \in U, c\colon U \to T(U)\rangle$*, 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 co-algebras* $Z \xrightarrow{\cong} T(Z)$ *and* $Z_{\mathrm{pu}} \xrightarrow{\cong} T_{\mathrm{pu}}(Z_{\mathrm{pu}})$ *of the entire signature, and of the public signature only. Then, by terminality, we get two co-algebra maps* $!$ *and* $!_{\mathrm{pu}}$ *in diagrams:*

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

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 co-algebra. 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), see [28]. Remember from the explicit description of terminal co-algebras in Lemma 6 that both $\mathcal{O}(p)$ and $\mathcal{O}_{\mathrm{pu}}(p)$ are infinite trees.

Lemma 9 *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. It follows from Lemma 5. 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.

Definition 10 *Consider two objects $u, u' \in U$ belonging to the same class $c\colon U \to T(U)$, where T is the functor $X \mapsto \prod_{i \leq n}(B_i + C_i \times X)^{A_i}$ as used before. The single transition step*

$$u \xrightarrow[y]{x} u'$$

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

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

More explicitly, we have the following two possible transitions.

$$\begin{array}{cc} u \xrightarrow[\beta]{\langle i,a\rangle} u & \qquad u \xrightarrow[\gamma]{\langle i,a\rangle} u' \\ \text{if } c_i(u,a) = \beta \in B_i & \qquad \text{if } c_i(u,a) = \langle \gamma, u'\rangle \in C_i \times U. \end{array}$$

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 next local state u', then the value is visible, but the next local state gives us a different object with the original identifier and co-algebra, but with this new local state.

Lemma 11 *The operational semantics $\mathcal{O}(p)$ of an object p with local state $u \in U$ as an element of the set Z of trees $A^+ \to B + C$ from Lemma 6 may be described explicitly as:*

$$\mathcal{O}(p)\left(\langle x_n, x_{n-1}, \ldots, x_1\rangle\right) = y \Leftrightarrow \begin{cases} \text{there are objects } u_1, \ldots, u_n \in U \\ \text{and outputs } y_1, \ldots, y_{n-1} \in B + C \text{ with} \\ u \xrightarrow[y_1]{x_1} u_1 \xrightarrow[y_2]{x_2} \cdots \xrightarrow[y_{n-1}]{x_{n-1}} u_{n-1} \xrightarrow[y_n]{x_n} u_n \end{cases}$$

Proof. This is because the description in the lemma is the unique map ! to the terminal co-algebra, applied to u. $\square$

6 TERMINAL CO-ALGEBRAS SATISFYING ASSERTIONS

In Lemma 6 we have described terminal co-algebras of functors associated with signatures of methods, whereby the assertions in a specification were ignored. The carrier sets of these terminal co-algebras are rather large sets of infinite trees. It turns out that assertions cut down such sets considerably. One then considers the terminal co-algebra which satisfies these assertions. It gives us a canonical model (class implementation) for a specification. These terminal co-algebras are comparable to initial algebras in algebraic specification, in the sense that they form "best possible" models. For suitably specified classes the creation conditions determine an element of the carrier of this terminal co-algebra, which can serve as interpretation of the initial state. We start by sketching the approach from [16] to "carve out" terminal co-algebras satisfying equations. It is a two-step approach, like in algebra. There, one first forms the initial algebra of operations only, and then takes a quotient with respect to the least congruence relation induced by the equations, see e.g. [28]. By definition, a congruence relation is closed under the (algebraic) operations. In co-algebra one first takes the terminal co-algebra of operations and then one carves out the subco-algebra given by the greatest "mongruence" induced by the equations. And a mongruence is a predicate which is suitably closed under co-algebraic operations.

Definition 12 *A* **mongruence** *on a co-algebra* $c: U \to T(U)$ *of a functor* $T(X) = \prod_{i \leq n}(B_i + C_i \times X)^{A_i}$ *is a predicate* $P \subseteq U$ *satisfying: if* $P(x)$ *holds, then also* $P(x')$ *for each* $x' \in U$ *occurring as next state in* $c_i(x, a) = \langle y, x' \rangle \in C_i \times U$*, for some* $i \leq n$ *and* $a \in A_i$*.*

To find the terminal co-algebra for a specification, consider the terminal co-algebra $Z \xrightarrow{\cong} T(Z)$ of a polynomial functor T induced by the signature, and let $E \subseteq Z$ be the subset induced by the assertions. Let $\underline{E}$ be the greatest mongruence on $Z \to T(Z)$ which is contained in E. Then $\underline{E}$ inherits a co-algebra structure, and is the terminal co-algebra satisfying E.

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 7, as terminal co-algebra the set of functions $\varphi \in (1 + A)^{(A+1)^\star}$ with operations defined by $\varphi.\mathsf{store}(a) = \lambda\alpha.\, \varphi(\alpha \cdot \kappa a)$, $\varphi.\mathsf{read} = \varphi([\,])$, and $\varphi.\mathsf{empty} = \lambda\alpha.\, \varphi(\alpha \cdot \kappa' *)$. The three equations in the persistent read class gives us a subset $E \subseteq (1 + A)^{(A+1)^\star}$ consisting of those φ satisfying (1) $\varphi.\mathsf{empty}.\mathsf{read} = *$, i.e. $\varphi(\kappa' *) = *$;

(2) if φ.read $= *$, then φ.store(a).read $= a$, i.e. if $\varphi([\,]) = *$, then $\varphi(\kappa a) = a$; and (3) if φ.read $= a$, then φ.store(b).read $= a$, i.e. if $\varphi([\,]) = a$, then $\varphi(\kappa b) = a$. The greatest mongruence $\underline{E} \subseteq (1 + A)^{(A+1)^\star}$ contained in E is the greatest set $\underline{E} \subseteq E$ satisfying: if $\varphi \in \underline{E}$, then also φ.store$(a) \in \underline{E}$ and φ.empty $\in \underline{E}$. It is easy to see that $\underline{E}$ is then the set of those $\varphi \in (1 + A)^{(A+1)^\star}$ satisfying for all $\alpha \in (A + 1)^\star$, (1) $\varphi((\kappa' *) \cdot \alpha) = *$, (2) if $\varphi(\alpha) = *$ then $\varphi((\kappa a) \cdot \alpha) = a$, and (3) if $\varphi(\alpha) = a$ then $\varphi((\kappa b) \cdot \alpha) = a$. Each tree $\varphi \in \underline{E}$ is thus determined by its value $\varphi([\,]) \in 1 + A$ at the root. Hence the set $\underline{E}$ is isomorphic to $1 + A$, and this is the (carrier of the) terminal co-algebra satisfying the persistent read equations. It is the minimal realization of an A-buffer with capacity one, since the state space $1 + A = \{*\} \cup A$ contains an error element $* \in 1 + A$ and single data elements $a \in 1 + A$, for $a \in A$. Nothing else is needed.

We have two side remarks about this terminal co-algebra.

(1) We note that the two states s.empty.empty and s.empty are indistinguishable (bisimilar), and indeed have equal interpretations in the terminal co-algebra $1 + A$. But there is no way that we can prove from the equations in the persistent read class that s.empty.empty and s.empty are equal since we have no equations between states. Hence in the minimal realization more than just the derivable equations are valid.

(2) One may be tempted from an algebraic perspective to see the "creation" part in a class as the description of a constant new: $1 \to X$. One can then investigate what the *initial* model of the specification is. In the persistent read example it is not the (minimal) set $1 + A$ of internal states that comes out in the co-algebraic approach. Algebraically one gets more, since one cannot show that the closed terms new and new.empty are the same.

Essential in the elimination of these trees φ is that they are determined by their output $\varphi([\,])$ at the root. This happens when the behaviour is "totally" specified: the future behaviour of an arbitrary state is determined by the immediate observations (via attribute outputs). The state space of the terminal co-algebra is then a subset of the set $T(1) = \prod_{i \leq n}(B_i + C_i)^{A_i}$ of attribute outputs. And the initial state (describing new) is an element of this set.

7 EQUATIONS BETWEEN STATES?

In the examples so far we have been careful to describe equations only between (observable) attribute values, and not between local states (elements of X). This reflects the idea that we do not have direct access to local states. The most we can say in comparing two states is that they are bisimilar (indistinguishable). This is as close as we can get to equality. The question arises: can we, or do we want to, always avoid equations between states in co-algebraic specifications. The answer seems to be *no*. It turns out to be convenient to have equations between states, for example if one wishes an "undo" operation for a certain procedure, which restores the original state. But in line with our earlier mentioned view that states themselves are inaccessible, we shall write bisimilarity $\underline{\leftrightarrow}$ instead of equality $=$ between states. As example, we consider the following co-algebraic queue specifications, in last-in-first-out (LIFO) and in first-in-first-out (FIFO) form. A fixed set A of data elements used.

class spec: LIFO
  public methods:
    $\mathsf{in}\colon X \times A \longrightarrow X$
    $\mathsf{out}\colon X \longrightarrow 1 + A \times X$
  assertions:
    $\mathsf{s.in}(a)\mathsf{.out} \neq *$
    $\mathsf{s.in}(a)\mathsf{.out.fst} = a$
    $\mathsf{s.in}(a)\mathsf{.out.snd} \;\underline{\leftrightarrow}\; \mathsf{s}$
  creation:
    $\mathsf{new.out} = *$
end class spec

class spec: FIFO
  public methods:
    $\mathsf{in}\colon X \times A \longrightarrow X$
    $\mathsf{out}\colon X \longrightarrow 1 + A \times X$
  assertions:
    $\mathsf{s.in}(a)\mathsf{.out} \neq *$
    $\mathsf{s.out} = * \vdash$
      $\mathsf{s.in}(a)\mathsf{.out.fst} = a$
      $\wedge\; \mathsf{s.in}(a)\mathsf{.out.snd} \;\underline{\leftrightarrow}\; \mathsf{s}$
    $\mathsf{s.out} \neq * \vdash$
      $\mathsf{s.in}(a)\mathsf{.out.fst} = \mathsf{s.out.fst}$
      $\wedge\; \mathsf{s.in}(a)\mathsf{.out.snd} \;\underline{\leftrightarrow}\; \mathsf{s.out.snd.in}(a)$
  creation:
    $\mathsf{new.out} = *$
end class spec

(We use the inequality $\mathsf{s.out} \neq *$ above for convenience; it can be replaced in a more elaborate formulation by an equality: mapping the output $\mathsf{s.out} \in 1 + A \times X$ to $1 + 1$ via the function $\mathrm{id} + !\colon 1 + A \times X \to 1 + 1$, we can use the equality $(\mathrm{id} + !)(\mathsf{s.out}) = \kappa' *$ to check that the result is in the right alternative of $+$.) One can derive in the FIFO case

$$\begin{array}{rcl} \mathsf{new.in}(a)\mathsf{.in}(b)\mathsf{.out.fst} & = & a \\ \mathsf{new.in}(a)\mathsf{.in}(b)\mathsf{.out.snd} & \underline{\leftrightarrow} & \mathsf{new.in}(b). \end{array}$$

The terminal co-algebras (minimal realizations) of these specifications both have the set $A^\infty = A^\star + A^{\mathbb{N}}$ of finite and infinite sequences as carrier set. Also the implementations of the out methods are the same: for $\alpha \in A^\infty$

$$\alpha.\mathsf{out} = \begin{cases} * & \text{if } \alpha \text{ is the empty sequence} \\ \langle a, \beta \rangle & \text{if } \alpha = a \cdot \beta. \end{cases}$$

But the implementations of the in methods differs: in the LIFO case

$$\alpha.\mathsf{in}(a) = a \cdot \alpha.$$

And in the FIFO case

$$\alpha.\mathsf{in}(a) = \begin{cases} \alpha \cdot a & \text{if } \alpha \in A^\star \text{, i.e. if } \alpha \text{ is finite} \\ \alpha & \text{otherwise.} \end{cases}$$

Infinite sequences are included in the terminal co-algebras since nothing in the LIFO and FIFO specifications tells us that $\mathsf{out}\colon X \to 1 + A \times X$ will end in 1 at some stage. But surely all reachable states (from new) will have finite output.

Acknowledgement. Thanks are due to Jan Rutten and Horst Reichel for clarifying discussions.

REFERENCES

[1] P. Aczel and N. Mendler, A final co-algebra theorem, In: D.H. Pitt and A. Poigné and D.E. Rydeheard (eds.), *Category Theory and Computer Science* Springer LNCS 389, 1989, 357–365.

[2] P. America, Designing an object-oriented language with behavioural subtyping. In: [3], 60–90.

[3] J.W. de Bakker, W.P. de Roever and G. Rozenberg (eds.), *Foundations of Object-Oriented Languages* (Springer LNCS 489, 1990).

[4] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. Leavens and B. Pierce, On binary methods, *Theory and Practice of Object Systems*, to appear.

[5] T. Budd, *An Introduction to Object-Oriented Programming* (Addison-Wesley, 1991).

[6] R. Burstall and R. Diaconescu, Hiding and behaviour: an institutional approach. In: A.W. Roscoe (ed.), *A Classical Mind. Essays in honour of C.A.R. Hoare* (Prentice Hall, 1994), 75–92.

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

[8] J.R.B. Cockett and D. Spencer, Strong categorical datatypes I. In: R.A.G. Seely (ed.), *Category Theory 1991* (CMS Conference Proceedings **13**, 1992), 141–169.

[9] W.R. Cook, Object-oriented programming versus abstract data types. In: [3], 151–178.

[10] A. Eliëns, *Principles of Object-Oriented Software Development* (Addison-Wesley, 1995).

[11] L.M.G. Feijs and H.B.M. Jonkers, *Formal Specification and Design* (Cambridge Univ. Press, Tracts in Theor. Comp. Sci. **5**, 1992).

[12] J.A. Goguen, Realization is universal, *Math. Syst. Theory* **6**(**4**) (1973), 359–374.

[13] J.A. Goguen, Types as Theories. In: G.M. Reed. A.W. Roscoe and R.F. Wachter (eds.), *Topology and Category Theory in Computer Science* (Oxford Univ. Press, 1991), 357–390.

[14] J.A. Goguen and J. Meseguer, Unifying functional, object-oriented and relational programming with logical semantics. In: B. Shriver and P. Wegner (eds.), *Research Directions in Object-Oriented Programming* (The MIT Press series in computer systems, 1987), 417–477.

[15] M. Hofmann and B. Pierce, A unifying type-theoretic framework for objects, *Journ. Funct. Progr.*, to appear.

[16] B. Jacobs, Mongruences and cofree co-algebras. In: V.S. Alagar and M. Nivat (eds., *Algebraic Methods and Software Technology* (Springer LNCS 936, 1995), 245–260.

[17] B. Jacobs, Inheritance and cofree constructions. In: P. Cointe (ed.), *European Conference on Object-Oriented Programming* (Springer LNCS, 1996, to appear).

[18] B. Jacobs, Co-algebraic specifications and models of deterministic hybrid systems. In: M. Wirsing (ed.), *Algebraic Methods and Software Technology* (Springer LNCS, 1996, to appear).

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

[20] J. Meseguer and J.A. Goguen, Initiality, induction and computability. In: M. Nivat and J.C. Reynolds (eds.), *Algebraic Methods in Semantics* (Cambridge Univ. Press, 1985), 459–541.

[21] B. Meyer, *Object-Oriented Software Construction* (Prentice Hall, 1988).

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

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

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

[25] J. Rutten and D. Turi, On the foundations of final semantics: non-standard sets, metric spaces and partial orders. In: J.W. de Bakker, W.P. de Roever, and G. Rozenberg (eds.), *Semantics: Foundations and Applications* (Springer LNCS 666, 1993), 477–530.

[26] M.B. Smyth and G.D. Plotkin, The category theoretic solution of recursive domain equations, *SIAM Journ. Comput.* **11** (1982) 761–783.

[27] P. Wegner, Concepts and paradigms of object-oriented programming, *OOPS Messenger* **1**(**1**) (1990), 7–87.

[28] M. Wirsing, Algebraic Specification. In: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier/MIT Press 1990, Volume B, 673–788.