Coalgebras in Specification and Verification for Object-Oriented Languages*

Bart Jacobs†

Department of Computer Science, University of Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
Email: `bart@cs.kun.nl` URL: `http://www.cs.kun.nl/~bart`

Abstract

The aim of this short note is to give an impression of the use of coalgebras in specification and verification for object-oriented languages. Particular emphasis will be given to the rôle of coalgebraic operations in describing state-based systems. At the end some active research topics in coalgebra will be sketched, together with pointers to the literature.

1 Introducing coalgebras

One of the new, recent developments in the area of theoretical computer science is the emergence of the field of coalgebra. One formal way of introducing coalgebras is as duals of algebras—in categorical style[1]. As such, coalgebras have been around for several decades. What is recent is the recognition that coalgebras form the underlying structure of various forms of dynamical systems / automata / transition systems. Such dynamical structures have been used for a long time in various situations in mathematics and computer science, but it is only now that people are beginning to identify them as coalgebras: it is fair to say that many people use coalgebras, but do not realise that they do.

Coalgebras may thus be seen as prototype dynamical systems. Such systems often display infinite behaviour, notably when they are supposed to be running forever, like operating systems. Coalgebras capture the idea of a system as consisting of a set of states (often called the state space), together with several operations on states. The state space should be seen as a black box, and the operations as providing interaction with the outside world. Think of yourself as sitting behind your computer: the computer has an internal state, given by, among other things, the contents of its memory cells and its registers. This state is not directly observable, but the screen provides certain observable information about your computer's state. The keyboard provides you with operations for modifying the state of your computer. You are not the only one modifying the state: the pulses of your computer's internal clock are bringing about constant state changes. The idea is that the computer (without its clock) can be seen as a coalgebra: it has a state space and operations for observation and modification.

*Appeared in: Newsletter 3 of the *Dutch Association for Theoretical Computer Science*, 1999, p.15–27.

†Research Fellow of the Royal Netherlands Academy of Arts and Sciences.

[1] A dualisation of a concept named "X" is usually called "coX" in category theory. Examples are limit and colimit, product and coproduct, monad and comonad, *etc.*

In the context of dynamical systems / automata / transition systems there are familiar notions of invariance, bisimilarity, and behaviour. An invariant is a predicate on states, which, once it is true, remains true no matter how the state is changed. Bisimilarity is the relation on states which contains all pairs of observationally indistinguishable states. The behaviour of a system can be described in terms of the possible observations after series of consecutive state changes, starting in a particular state. Two states are then bisimilar if and only if they give rise to the same observable behaviour. These notions of invariance, bisimilarity and behaviour are given a precise and uniform mathematical foundation in the theory of coalgebras. As such they can be studied at an appropriate level of abstraction. This coalgebraic framework not only provides an abstract system theoretic setting, but also associated proof techniques. Of particular importance is the technique to prove equality of (generally infinite) behaviour via bisimulation relations.

Formally, a coalgebra consists of two parts: a set and a function acting on that set. The set is the state space, and will typically be written as Self. The function (or operation) is of the following special form.

$$\mathsf{Self} \longrightarrow \boxed{\cdots \;\; \mathsf{Self} \;\; \cdots} \tag{1}$$

The box on the right-hand-side is some expression involving Self[2]. To be precise, it is the result of applying what is called a "functor" to Self. This right-hand-side describes the interface of the coalgebra to the outside world. Below we shall see several examples of how this works. What is typically coalgebraic is that the function has Self as domain, on the left-hand-side. This reflects the idea that operations act on states. In particular, there is no way to construct a new state from scratch, via such operation of a coalgebra. Initial states thus have to be given as additional structure.

(As an aside, algebras can be described as consisting of a set, say Self, together with a function pointing in the reverse direction:

$$\boxed{\cdots \;\; \mathsf{Self} \;\; \cdots} \longrightarrow \mathsf{Self} \tag{2}$$

Hence algebras allow us to construct elements in Self. In general, algebras are appropriate for describing (finite) data types, whereas coalgebras are appropriate for state-based dynamical systems, displaying possibly infinite behaviour.)

Let us see some examples of coalgebras. Consider a (silly) system consisting of a light bulb together with a button. The idea is that if the button is pushed twice, the light changes from on to off, and vice-versa. The state space will thus have to incorporate information about whether the light is on or off, and about whether the next push will result in a change of light. One way of realising such a system is to take as state space (set of states):

$$S_1 = \{0,1\} \times \{0,1\}$$

Then, for a state $(x, y) \in S_1$, the first component $x \in \{0,1\}$ tells whether the light is on or off, and the second component y tells whether a next push will result in a change of light. We thus define two operations on S_1, capturing the light bulb:

$$\mathsf{light}_1 \colon S_1 \longrightarrow \{0,1\} \qquad \text{is} \qquad (x,y) \longmapsto x$$

and the push button:

$$\mathsf{push}_1 \colon S_1 \longrightarrow S_1 \qquad \text{is} \qquad (x,y) \longmapsto \begin{cases} (x,1) & \text{if } y = 0 \\ (1-x,0) & \text{if } y = 1. \end{cases}$$

[2] In positive position.

The two operations $\mathsf{light}_1, \mathsf{push}_1$ can be combined into a single function, giving an example of a coalgebra:

$$S_1 \xrightarrow{\langle \mathsf{light}_1, \mathsf{push}_1 \rangle} \{0,1\} \times S_1 \tag{3}$$

Notice that the operation light_1 is for observation, and push_1 is for modification.

A completely different realisation of this same system consists of a state space

$$S_2 = \mathbb{N}$$

with light operation:

$$\mathsf{light}_2 \colon S_2 \longrightarrow \{0,1\} \qquad \text{is} \qquad n \longmapsto \begin{cases} 0 & \text{if } \exists m \in \mathbb{N}.\ m \text{ is even, and} \\ & \qquad n = 2m \text{ or } n = 2m+1 \\ 1 & \text{otherwise.} \end{cases}$$

The realisation of the push button is now very simple:

$$\mathsf{push}_2 \colon S_2 \longrightarrow S_2 \qquad \text{is} \qquad n \longmapsto n+1.$$

This second realisation forms a coalgebra, with the same interface as the first realisation:

$$S_2 \xrightarrow{\langle \mathsf{light}_2, \mathsf{push}_2 \rangle} \{0,1\} \times S_2 \tag{4}$$

But the second state space S_2 is clearly different from the first one (S_1). For a user on the outside, these differences are not noticeable: both systems display the same observable behaviour.

The idea that will be elaborated in these notes is that coalgebras share certain essential aspects with classes in object-oriented languages. A class has an implicit state space, given, for example, by the Cartesian product of the types of its instance variables (both public and private). Also, it has certain operations (called methods in this context) for observing or possibly modifying the state. The types of the instance variables and of the methods together determine the interface of the class, corresponding to the box in (1), see Section 3 for more details. As a suggestive example, we describe the first realisation (3) as a class in Java.

```
class LightPush {
  private boolean light_status, next_change;
  public boolean light() {
    return light_status;
  }
  public void push() {
    if ( next_change )
      { light_status = !light_status; }
    next_change = !next_change;
  }
}
```

Now that we have seen some examples of coalgebras, we proceed with their use in specification and verification.

2 Coalgebraic class specifications

In the previous section we have given an informal description of a system with a light and a push button, such that pushing the button twice changes the status of the light. What would be useful is a formal way of describing such a system. This can be done in what is called a coalgebraic specification. It is a specification based on coalgebras, that may be understood as a specification of a class in an object-oriented language. It consists of several coalgebraic operations (which may be combined into a single coalgebra operation), together with certain assertions. The latter restrict the behaviour of the operations in an appropriate manner. Additionally, a coalgebraic specification may contain (parametrised) initial states, typically written as new, with their own assertions. For example, the light bulb system mentioned earlier can be specified as:

$$
\begin{array}{l}
\mathsf{ATTRIBUTE} \\
\qquad \mathsf{light} \colon \mathsf{Self} \longrightarrow \{0, 1\} \\
\mathsf{METHOD} \\
\qquad \mathsf{push} \colon \mathsf{Self} \longrightarrow \mathsf{Self} \\
\mathsf{ASSERTION} \\
\qquad \forall x \colon \mathsf{Self}.\ \mathsf{light}(\mathsf{push}(\mathsf{push}(x))) = 1 - \mathsf{light}(x) \\
\mathsf{CONSTRUCTOR} \\
\qquad \mathsf{new} \colon \mathsf{Self} \\
\mathsf{CREATION} \\
\qquad \mathsf{light}(\mathsf{new}) = 0 \wedge \mathsf{light}(\mathsf{push}(\mathsf{new})) = 0
\end{array}
$$

The distinction between attributes and methods is not essential at this stage, but the idea is that an attribute is an operation that is used for observation and that does not change the state. Notice that this specification does not tell us anything about the internal structure of the state space Self. In coalgebraic specification this structure is irrelevant, since the state space is considered to be a black box. The specification only requires the presence of certain operations, displaying certain behaviour. How that behaviour is realised internally should not be part of a specification. For example, both the state spaces S_1 and S_2 from the previous section may be used to obtain models of this specification. The light and push operartions from the above specification are interpreted as the functions light_1 and push_1 on S_1, or as light_2 and push_2 on S_2. The initial state new from the specification is interpreted as $\mathsf{new}_1 = (0, 0) \in S_1$, or as $\mathsf{new}_2 = 0 \in S_2$. It is not hard to see that the assertion and creation condition from the specification hold, both for S_1 and for S_2.

We have seen in (1) that the operation of a coalgebra has a simple domain (*viz.* Self), but a structured codomain. This gives great expressivity. For instance, a *partial* operation from Self to Self can be described coalgebraically as

$$\mathsf{Self} \longrightarrow 1 + \mathsf{Self}$$

where 1 is a singleton set, say $1 = \{*\}$, and $+$ is disjoint union. Thus, $1 + \mathsf{Self}$ is the "lift of Self", obtained by adding an extra element. Next, suppose one wishes to specify some store system with an operation to read certain data, say in a set D, from the (reading position in the) store. This reading should proceed in such a way that reading means at the same time removing the element that is being read from the store. The coalgebraic way to describe such an operation is as a function

$$\mathsf{Self} \xrightarrow{\quad \mathsf{read} \quad} 1 + (D \times \mathsf{Self})$$

Notice that this read is regarded as a partial operation: if it produces an output in the singleton set $1 = \{*\}$ this means that the store is empty and that there is nothing to read. If the store is non-empty, the read operation produces a pair (d, y)

consisting of the data $d \in D$ that has been read, together with the successor state y, which is the store state from which d is removed.

Let us turn to assertions in coalgebraic specifications. In principle ordinary logical (first- or higher-order) language may be used here, with one important restriction: coalgebraic specifications should not contain equations between states. This goes against the underlying idea that states are basically inaccessible, so that one is never able to establish actual equality of states. What should be used instead of equality on states is bisimilarity $\underline{\leftrightarrow}$. Bisimilarity is behavioural indistinguishability, and is equality "as far as we can see". For example, in a system with a do and an undo operation, say both of type $\mathsf{Self} \to \mathsf{Self}$, one should *not* write an assertion

$$\forall x \colon \mathsf{Self}.\ \mathsf{undo}(\mathsf{do}(x)) = x \tag{5}$$

but:

$$\forall x \colon \mathsf{Self}.\ \mathsf{undo}(\mathsf{do}(x)) \mathrel{\underline{\leftrightarrow}} x \tag{6}$$

Writing an equality as in (5) severely limits the possible models of the specification. For example, it excludes "history" models in which it is internally recorded which operations have been applied. In that case one can still have validity of the assertion (6), if the internal history record is not observable.

There is more to be said about bisimilarity in specifications.

1. The theory of coalgebras provides for each interface (formally, for each functor) a tailored definition of bisimilarity[3] which precisely captures indistinguishability for that interface. This is used in each coalgebraic specification: the operations are given first, and determine the interface, and thus the precise formulation of bisimilarity $\underline{\leftrightarrow}$ for the specification. This relation $\underline{\leftrightarrow}$ can then be used in the subsequent assertions.

2. Bisimilarity $\underline{\leftrightarrow}$ on Self is not enough in assertions, and must be "lifted" to a new relation $\underline{\underline{\leftrightarrow}}$, called observational equality, on arbitrary type expressions. This is done by induction on the structure of these type expressions, in such a way that $\underline{\underline{\leftrightarrow}}$ is equality on constants, and $\underline{\leftrightarrow}$ on Self. On constructed types, say via $\times$ and $+$, the relation $\underline{\underline{\leftrightarrow}}$ is structurally determined by $\underline{\underline{\leftrightarrow}}$ on the components. To see an example, consider the previously mentioned read operation of a store system. Suppose the system also has a insert operation, of type $\mathsf{Self} \to \mathsf{Self}^D$, for inserting an element from D into the store—where Self^D is the set (or type) of functions from D to Self. If the system under consideration is a LIFO queue, one expects an assertion of the form

$$\forall x \colon \mathsf{Self}, d \colon D.\ \mathsf{read}(\mathsf{insert}(x)(d)) \mathrel{\underline{\underline{\leftrightarrow}}} (d, x)$$

It tells that reading after inserting an element d in a state x produces output containing the element d together with a state which is bisimilar to the original state x. This involves observational equality $\underline{\underline{\leftrightarrow}}$ lifted to the type expression $1 + (D \times \mathsf{Self})$. For elements $z, w \colon 1 + (D \times \mathsf{Self})$, $z \mathrel{\underline{\underline{\leftrightarrow}}} w$ means: either both z and w are in $1 = \{*\}$ and thus equal, or both z and w are in $D \times \mathsf{Self}$ and their first components in D are equal and their second components in Self are bisimilar.

Once a coalgebraic specification has been written down, it can be used for (at least) three important purposes. First, one can describe a concrete (intended) model of the specification, *e.g.* to test its consistency[4]. This involves defining a concrete

[3] and also of invariance.

[4] A more demanding challenge is identify the final model of the specification, given by all possible behaviours. It can be used to see that unintended behaviour does not occur in models of the specification.

coalgebra (like in Section 1) together with initial states, and showing that the required assertions hold. Experience shows that such checks often reveal errors in the specification. Secondly, one can do theory development for a coalgebraic specification. This means deriving consequences from the assertions, using ordinary logical rules, plus some rules for bisimilarity. Important to establish are invariants of specifications, telling for example that a certain integer attribute has a positive value. Such invariants express properties that are supposed to hold forever, and trying to prove them sometimes reveals unforeseen behaviour. Thirdly, one can establish refinements between coalgebraic specifications. In that case, one defines "relative models" of one (abstract) specification, assuming a model of another (concrete) specification.

3 Coalgebraic program verification

We shall sketch some ingredients of the use of coalgebras in program verification for the object-oriented programming language Java. What we tell here is not very specific for Java, and may apply, in slightly modified form, to other (imperative) languages.

The foundation of coalgebraic program verification for Java involves modeling Java classes as suitable coalgebras, acting on a common global memory. The basic components of Java classes are instance variables, methods and constructors. All of these are modeled as coalgebraic operations[5]—as was already suggested in Section 1. An instance variable, say `boolean b`, gives rise to an operation $b\colon \mathsf{Self} \to \mathsf{bool}$. Methods and constructors give rise to more complicated operations. We concentrate on methods—or more generally, on statements—with return type `void`, which do not produce a result. Statements in Java may hang, *e.g.* because of an infinite while loop or recursion. Also, they may terminate abruptly, *e.g.* because of an exception, say thrown by a division by zero. But other forms of abrupt termination can also occur, caused by a return, a break or a continue statement. And of course a statement can also terminate normally. Coalgebraically, all these different result options can be captured naturally. Statements are coalgebras of the form

$$\mathsf{Self} \xrightarrow{\quad \mathsf{statement} \quad} \mathsf{StatResult}(\mathsf{Self})$$

where $\mathsf{StatResult}(\mathsf{Self})$ is an abbreviation:

$$\mathsf{StatResult}(\mathsf{Self}) \quad = \quad 1 + \mathsf{Self} + \mathsf{StatAbnormal}(\mathsf{Self})$$

and where $\mathsf{StatAbnormal}(\mathsf{Self})$ is another abbreviation, to be described shortly. The first $+$-component $1 = \{*\}$ in $\mathsf{StatResult}(\mathsf{Self})$ corresponds to non-termination (hanging); in this case no next state is produced. The second component Self corresponds to normal termination; in this case a successor state in Self is produced. The third component $\mathsf{StatAbnormal}(\mathsf{Self})$ corresponds to abrupt termination; it contains the four possible forms of abrupt termination for statements in Java:

$$\begin{aligned}&\mathsf{StatAbnormal}(\mathsf{Self})\\ &\quad = \quad (\mathsf{Self} \times \mathsf{Exception}) + \mathsf{Self} + (\mathsf{Self} \times (1 + \mathsf{String})) + (\mathsf{Self} \times (1 + \mathsf{String}))\end{aligned}$$

The first of the (four) $+$-components in $\mathsf{StatAbnormal}(\mathsf{Self})$ corresponds to an exception abnormality; in that case the state in which the exception occurred is returned,

[5]Even constructors are coalgebras in this approach, since they modify the global state space, in allocating storage for the newly created object.

together with (a reference to) the relevant exception object. The second component captures a return abnormality; then, only the state is returned, appropriately tagged. The third and fourth option deal with break and continue abnormalities. In Java both the break and the continue statement may occur with or without label, given as string. Hence, a break or continue abnormality consists of a state, possibly with a string. Thus, all possible termination options of statements can be captured in an appropriate type, which is used as codomain type of statements, like in the general pattern (1).

Expressions in Java, say of return type Out, are handled similarly. They are captured as coalgebras of the form

$$\mathsf{Self} \xrightarrow{\quad \mathsf{expression} \quad} \mathsf{ExprResult}(\mathsf{Self}, \mathsf{Out})$$

where $\mathsf{ExprResult}(\mathsf{Self}, \mathsf{Out})$ is an abbreviation:

$$\mathsf{ExprResult}(\mathsf{Self}, \mathsf{Out}) \quad = \quad 1 + (\mathsf{Self} \times \mathsf{Out}) + \mathsf{ExprAbnormal}(\mathsf{Self})$$

with $\mathsf{ExprAbnormal}(\mathsf{Self})$ incorporating abrupt termination for expressions[6]. In the normal termination case—the second component of $\mathsf{ExprResult}(\mathsf{Self}, \mathsf{Out})$—both a state in Self and a value in Out are returned. The state that is produced captures the possible side-effect of the expression.

The interface of a class in Java can be described as a type, in a similar manner. We only give an example, involving a simple class:

```
class A {
  int i;
  void m1(boolean b) { ... }
  float m2(float x, byte a) { ... }
}
```

For such a class we construct an "interface type", containing[7] the interface types of the members of this class in a Cartesian product.

$$\begin{aligned} &\mathsf{AIFace}(\mathsf{Self}) \\ &\quad = \quad \mathsf{int} \times (\mathsf{Self}^{\mathsf{int}}) \times (\mathsf{StatResult}(\mathsf{Self})^{\mathsf{boolean}}) \times (\mathsf{ExprResult}(\mathsf{Self}, \mathsf{float})^{(\mathsf{float} \times \mathsf{byte})}) \end{aligned}$$

(The second $\times$-component $\mathsf{Self}^{\mathsf{int}}$ corresponds to a generated assignment operation for the instance variable `i`. It takes an integer, say a, and yields a new state in which the value of `i` is set to a, and everything else is unchanged.)

A coalgebra of the form

$$\mathsf{Self} \longrightarrow \mathsf{AIFace}(\mathsf{Self}) \tag{7}$$

then incorporates all the operations of the Java class. A further step is to link the method bodies—occurring in the above class `A` only as "`...`"—into this approach. This is done by using suitable predicates on coalgebras like (7), telling that the methods correspond to the translated method bodies (which are not discussed here). These predicates are much like the assertions used in the coalgebraic specifications in the previous section, since they take the form: "method call = method body".

Once this translation is completed, it can be used to reason with standard type theoretic and logical means about the original Java class, possibly making use of proof tools. Typical properties to be proved are normal / abnormal termination

[6] which can only happen because of an exception.
[7] For simplicity we omit the default constructor.

resulting in certain outcomes, for individual methods, and invariants, for classes as a whole. Given the present space restrictions, details have to be skipped. What we would like to emphasise is the natural coalgebraic representation of classes, via a precise interface type as codomain, leading to tailor-made definitions of bisimilation and invariance, for particular (Java) classes. In alternative approaches based on automata or transition systems, the structure of the interface to the outside world is usually lost in an unstructured set of actions.

4 Conclusions and pointers

We have sketched some ideas behind the use of coalgebras in specification and verification for state-based systems, described in object-oriented style. Emphasis was put on the rôle of coalgebraic operations, having the state space as domain, and a structured type as codomain. Several exemplary systems were described using such operations.

We conclude with a brief survey of some research lines in the new field of coalgebra, together with selected pointers to the rapidly growing literature. The paper [JR97] forms an introduction to the field of coalgebras. Part of the ongoing research may be found in the proceedings of the first two workshops on coalgebras [JMRR98, JR99]—selected papers from the first of which will appear in a special issue of *Theoretical Computer Science.*

Structure on coalgebras

The abstract study of products, coproducts, homomorphisms *etc.* for coalgebras can be done as "universal coalgebra" [Rut99] or also as a categorical study [JPT$^+$98]. There is a famous result in universal algebra, due to Birkhoff [Bir35], stating that a class of algebras is equationally definable if and only if it is closed under subalgebras, quotients and products. A dual result for coalgebras is investigated in [GS98, Roş98].

Coalgebras and data type theory

Early work on using coalgebras in data type theory to capture infinite data structures occurs in [AM82, Hag87, Hag89]. The experimental programming language Charity only contains algebras and coalgebras [CF92]. Programs can be constructed via various laws for initial algebras and final coalgebras, as studied *e.g.* in [MFP91, Par98]. As an example, the theory of the data type of both finite and infinite sequences is developed on a coalgebraic basis in [HJ99a]. Sometimes, initial algebra and final coalgebra solutions of data type equations coincide, giving rise to special theories [Fre91, Fre92].

Reasoning principles for coalgebras

For initial algebras there is a standard reasoning principle called induction. The analogous, but dual, principle for final coalgebras is called coinduction. It can also be formulated in terms of bisimulations [Fio96, TR98]. Coinduction is useful for reasoning about (infinite) behaviours of systems, by reducing global arguments over infinite structures to single-step arguments—like induction does. The coinduction principle is currently formalised in the proof tools Coq [BBC$^+$97] and Isabelle [Pau97], and can be encoded in second order type theory [Wra89, RP93].

Coalgebraic specification

The area of coalgebraic specification of classes (in object-oriented languages) was introduced in [Rei95], and continued in [Jac96c, Jac95, Jac96b, Jac97]. In [HHJT98] a formal language CCSL for coalgebraic class specifications is described, together with a translation tool for generating logical theories for the proof tool PVS [ORSvH95]. Connections with (specifications in) hidden algebra (see [GM96]) are investigated in [Mal96, Cîr98].

Coalgebraic program verification

Coalgebras can provide a basis for Java program verification, in the style described in Section 3. An elaboration of such a semantics, together with an associated Hoare logic, is described in [HJ99b]. A tool that translates Java classes to logical theories (in PVS [ORSvH95]) may be found in [JvdBH+98]. A major case study is presented in [HBJ99]. In this context it is worth pointing out that the recent textbook [Rey98] on program semantics systematically describes denotations $[\![c]\!]_{\mathrm{comm}}$ of programs c as coalgebras, acting on a state space of mappings from variables to values—without, by the way, ever calling these denotations coalgebras.

Coalgebras and continuous mathematics

Infinite sequences play an important rôle in analysis. Describing these coalgebraically, leads to a new, uniform framework in which one can describe analytic functions and various associated results [PE98], including solutions to differential equations [Pav98]. Such solutions give rise to coalgebras in hybrid systems [Jac96a], combining discrete and continuous dynamics. Also, probabilistic bisimulation for Markov processes may be studied from a coalgebraic perspective [VR98, DEP98].

Coalgebras and modal logic

Modal logic [Eme90, Gol92] may be seen as the logic of dynamics. Therefore, it combines very well with coalgebras, as witnessed by several recent studies [Mos99, Röß99, Kur98].

Coalgebras and non well-founded set theory

In the original monograph [Acz88] on non well-founded sets, coalgebras are already prominently present, since they are fundamental to the idea of a non well-founded set. Further investigations occur in [BM96, TR98].

Coalgebras and automata / transition systems

Automata and transition systems form well-established structures for describing dynamical systems. They form instancs of coalgebras. As such, they are studied in [Rut98, TR98, JPT+98].

Acknowledgements

Thanks to Joachim van den Berg, Marieke Huisman, Erik Poll, Jan Rutten and Hendrik Tews for commenting on the first draft.

References

[Acz88] P. Aczel. *Non-well-founded sets.* CSLI Lecture Notes 14, Stanford, 1988.

[AM82] M.A. Arbib and E.G. Manes. Parametrized data types do not need highly constrained parameters. *Inf. & Contr.*, 52:139–158, 1982.

[BBC+97] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-Chr. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant User's Guide Version 6.1. Technical Report 203, INRIA Rocquencourt, France, May 1997.

[Bir35] G. Birkhoff. On the structure of abstract algebras. *Proc. Cambridge Phil. Soc.*, 31:433–454, 1935.

[BM96] J. Barwise and L.S. Moss. *Vicious Circles: On the Mathematics of Non-wellfounded Phenomena.* CSLI Lecture Notes 60, Stanford, 1996.

[CF92] J.R.B. Cockett and T. Fukushima. About Charity. Technical Report 92/480/18, Dep. Comp. Sci., Univ. Calgary, 1992.

[Cîr98] C. Cîrstea. Coalgebra semantics for hidden algebra: parametrised objects and inheritance. In F. Parisi Presicce, editor, *Recent Trends in Data Type Specification*, number 1376 in Lect. Notes Comp. Sci., pages 174–189. Springer, Berlin, 1998.

[DEP98] J. Desharnais, A. Edalat, and P. Panangaden. A logical characterization of bisimulation for labeled markov processes. In *Logic in Computer Science*, 1998.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 997–998. Elsevier/MIT Press, 1990.

[Fio96] M.P. Fiore. A coinduction principle for recursive data types based on bisimulation. *Inf. & Comp.*, 127(2):186–198, 1996.

[Fre91] P.J. Freyd. Algebraically complete categories. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Como Conference on Category Theory*, number 1488 in Lect. Notes Math., pages 95–104. Springer, Berlin, 1991.

[Fre92] P.J. Freyd. Remarks on algebraically compact categories. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of Categories in Computer Science*, number 177 in LMS, pages 95–106. Cambridge Univ. Press, 1992.

[GM96] J.A. Goguen and G. Malcolm. An extended abstract of a hidden agenda. In J. Meystel, A. Meystel, and R. Quintero, editors, *Proceedings of the Conference on Intelligent Systems: A Semiotic Perspective*, pages 159–167. Nat. Inst. Stand. & Techn., 1996.

[Gol92] R. Goldblatt. *Logics of Time and Computation.* CSLI Lecture Notes 7, Stanford, 2nd rev. edition, 1992.

[GS98] H.P. Gumm and T. Schröder. Covarieties and complete covarieties. In B. Jacobs, L. Moss, H. Reichel, and J. Rutten, editors, *Coalgebraic Methods in Computer Science*, number 11 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 1998.

[Hag87] T. Hagino. A typed lambda calculus with categorical type constructors. In D.H. Pitt, A Poigné, and D.E. Rydeheard, editors, *Category and Computer Science*, number 283 in Lect. Notes Comp. Sci., pages 140–157. Springer, Berlin, 1987.

[Hag89] T. Hagino. Codatatypes in ML. *Journ. Symb. Computation*, 8:629–650, 1989.

[HBJ99] M. Huisman, J. van den Berg, and B. Jacobs. A case study in class library verification: Java's Vector class. Manuscript, 1999.

[HHJT98] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Ch. Hankin, editor, *European Symposium on Programming*, number 1381 in Lect. Notes Comp. Sci., pages 105–121. Springer, Berlin, 1998.

[HJ99a] U. Hensel and B. Jacobs. Coalgebraic theories of sequences in PVS. *Journ. of Logic and Computation*, 1999. To appear.

[HJ99b] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. Manuscript, 1999.

[Jac95] B. Jacobs. Mongruences and cofree coalgebras. In V.S. Alagar and M. Nivat, editors, *Algebraic Methodology and Software Technology*, number 936 in Lect. Notes Comp. Sci., pages 245–260. Springer, Berlin, 1995.

[Jac96a] B. Jacobs. Coalgebraic specifications and models of deterministic hybrid systems. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, number 1101 in Lect. Notes Comp. Sci., pages 520–535. Springer, Berlin, 1996. The full version is to appear in *Theor. Comp. Sci.*

[Jac96b] B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on Object-Oriented Programming*, number 1098 in Lect. Notes Comp. Sci., pages 210–231. Springer, Berlin, 1996.

[Jac96c] B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ., 1996.

[Jac97] B. Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, number 1349 in Lect. Notes Comp. Sci., pages 276–291. Springer, Berlin, 1997.

[JMRR98] B. Jacobs, L. Moss, H. Reichel, and J. Rutten, editors. *Coalgebraic Methods in Computer Science (CMCS'98)*, number 11 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 1998. Available from URL: `http://www.elsevier.nl/locate/entcs/volume11.html`.

[JPT+98] P.T. Johnstone, A.J. Power, T. Tsujishita, H. Watanabe, and J. Worrell. An axiomatics for categories of transition systems as coalgebras. In *Logic in Computer Science*. IEEE, Computer Science Press, 1998.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[JR99] B. Jacobs and J. Rutten, editors. *Coalgebraic Methods in Computer Science (CMCS'99)*, number 19 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 1999. Available (soon) from URL: `http://www.elsevier.nl/locate/entcs/volume19.html`.

[JvdBH+98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.

[Kur98] A. Kurz. Specifying coalgebras with modal logic. In B. Jacobs, L. Moss, H. Reichel, and J. Rutten, editors, *Coalgebraic Methods in Computer Science*, number 11 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 1998.

[Mal96] G. Malcolm. Behavioural equivalence, bisimulation and minimal realisation. In M. Haveraaen, O. Owe, and O.J. Dahl, editors, *Recent Trends in Data Type Specification*, number 1130 in Lect. Notes Comp. Sci., pages 359–378. Springer, Berlin, 1996.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, number 523 in Lect. Notes Comp. Sci., pages 215–240. Springer, Berlin, 1991.

[Mos99] L.S. Moss. Coalgebraic logic. *Ann. Pure & Appl. Logic*, 1999. To appear.

[ORSvH95] S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.

[Par98] A. Pardo. Monadic corecursion—definition, fusion laws, and applications—. In B. Jacobs, L. Moss, H. Reichel, and J. Rutten, editors, *Coalgebraic Methods in Computer Science*, number 11 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 1998.

[Pau97] L.C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journ. of Logic and Computation*, 7:175–204, 1997.

[Pav98] D. Pavlović. Guarded induction on final coalgebras. In B. Jacobs, L. Moss, H. Reichel, and J. Rutten, editors, *Coalgebraic Methods in Computer Science*, number 11 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 1998.

[PE98] D. Pavlović and M. Escardó. Calculus in coinductive form. In *Logic in Computer Science*, pages 408–417. IEEE, Computer Science Press, 1998.

[Rei95] H. Reichel. An approach to object semantics based on terminal coalgebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.

[Rey98] J.C. Reynolds. *Theories of Programming Languages.* Cambridge Univ. Press, 1998.

[Roş98] G. Roşu. A Birkhoff-like axiomatizability result for hidden algebra and coalgebra. In B. Jacobs, L. Moss, H. Reichel, and J. Rutten, editors, *Coalgebraic Methods in Computer Science*, number 11 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 1998.

[Röß99] M. Rößiger. From modal logic to terminal coalgebras. Manuscript, 1999.

[RP93] J.C. Reynolds and G.D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Inf. & Comp.*, 105:1–29, 1993.

[Rut98] J. Rutten. Automata and coinduction (an exercise in coalgebra). In D. Sangiorigi and R. de Simone, editors, *Concur'98: Concurrency Theory*, number 1466 in Lect. Notes Comp. Sci., pages 194–218. Springer, Berlin, 1998.

[Rut99] J. Rutten. Universal coalgebra: a theory of systems. *Theor. Comp. Sci.*, 1999. To appear.

[TR98] D. Turi and J. Rutten. On the foundations of final semantics: non-standard sets, metric spaces and partial orders. *Math. Struct. in Comp. Sci.*, 8(5):481–540, 1998.

[VR98] E.P. de Vink and J.J.M.M. Rutten. Bisimulation for probabilistic transition systems: a coalgebraic approach (extended abstract). In

P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *International Colloquium on Automata, Languages and Programming*, number 1256 in Lect. Notes Comp. Sci., pages 460–470. Springer, Berlin, 1998.

[Wra89] G.C. Wraith. A note on categorical datatypes. In D.H. Pitt, A. Poigné, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, number 389 in Lect. Notes Comp. Sci., pages 118–127. Springer, Berlin, 1989.