Preface

Mathematics is about the formal structures underlying counting, measuring, transforming etc. It has developed fundamental notions like number systems, groups, vector spaces (see e.g. [345]) and has studied their properties. In more recent decades also 'dynamical' features have become a subject of research. The emergence of computers has contributed to this development. Typically, dynamics involves a 'state of affairs', which can possibly be observed and modified. For instance, the contents of a tape of a Turing machine contribute to its state. Such a machine may thus have many possible states and can move from one state to another. Also, the combined contents of all memory cells of a computer can be understood as the computer's state. A user can observe part of this state via the screen (or via the printer) and modify this state by typing commands. In reaction, the computer can display certain behaviour. Describing the behaviour of such a computer system is a non-trivial matter. However, formal descriptions of such complicated systems are needed if we wish to reason formally about their behaviour. Such reasoning is required for the correctness or security of these systems. It involves a specification describing the required behaviour, together with a correctness proof demonstrating that a given implementation satisfies the specification.

Both mathematicians and computer scientists have introduced various formal structures to capture the essence of state-based dynamics, such as automata (in various forms), transition systems, Petri nets and event systems. The area of coalgebra has emerged within theoretical computer science with a unifying claim. It aims to be the mathematics of computational dynamics. It combines notions and ideas from the mathematical theory of dynamical systems and from the theory of state-based computation. The area of coalgebra is still in its infancy but promises a perspective on uniting, say, the theory of differential equations with automata and process theory and with biological and quantum computing, by providing an appropriate semantical basis with

associated logic. The theory of coalgebras may be seen as one of the original contributions stemming from the area of theoretical computer science. The span of applications of coalgebras is still fairly limited but may in the future be extended to include dynamical phenomena in areas like physics, biology or economics – based for instance on the claim of Adleman (the father of DNA computing) that biological life can be equated with computation [32]; on [362], which gives a coalgebraic description of type spaces used in economics [211]; on [52], describing network dynamics that is common to all these areas; on [447], using coalgebras in biological modelling or on [7, 249], where coalgebras are introduced in quantum computing.

Coalgebras are of surprising simplicity. They consist of a state space, or set of states, say X, together with a structure map of the form $X \to F(X)$. The symbol F describes some expression involving X (a functor), capturing the possible outcomes of the structure map applied to a state. The map $X \to F(X)$ captures the dynamics in the form of a function acting on states. For instance, one can have F as powerset in $F(X) = \mathcal{P}(X)$ for non-deterministic computation $X \to \mathcal{P}(X)$, or $F(X) = \{\bot\} \cup X$ for possibly non-terminating computations $X \to \{\bot\} \cup X$. At this level of generality, algebras are described as the duals of coalgebras (or the other way round), namely as maps of the form $F(X) \to X$. One of the appealing aspects of this abstract view is the duality between structure (algebras) and behaviour (coalgebras).

Computer Science Is about Generated Behaviour

What is the essence of computing? What is the topic of the discipline of computer science? Answers that are often heard are 'data processing' or 'symbol manipulation'. Here we follow a more behaviouristic approach and describe the subject of computer science as *generated behaviour*. This is the behaviour that can be observed on the outside of a computer, for instance via a screen or printer. It arises in interaction with the environment, as a result of the computer executing instructions, laid down in a computer program. The aim of computer programming is to make a computer do certain things, i.e. to generate behaviour. By executing a program a computer displays behaviour that is ultimately produced by humans, as programmers.

This behaviouristic view allows us to understand the relation between computer science and the natural sciences: biology is about 'spontaneous' behaviour, and physics concentrates on lifeless natural phenomena, without autonomous behaviour. Behaviour of a system in biology or physics is often described as evolution, where evolutions in physics are transformational changes according to the laws of physics. Evolutions in biology seem to lack

inherent directionality and predictability [174]. Does this mean that behaviour is deterministic in (classical) physics, and non-deterministic in biology? And that coalgebras of corresponding kinds capture the situation? At this stage the coalgebraic theory of modelling has not yet demonstrated its usefulness in those areas. Therefore this text concentrates on coalgebras in mathematics and computer science.

The behaviouristic view does help in answering questions like: can a computer think? Or: does a computer feel pain? All a computer can do is display thinking behaviour, or pain behaviour, and that is it. But it is good enough in interactions – think of the famous Turing test – because in the end we never know for sure if other people actually feel pain. We see only pain behaviour and are conditioned to associate such behaviour with certain internal states. But this association may not always work, for instance not in a different culture: in Japan it is common to touch one's ear after burning a finger; for Europeans this is non-standard pain behaviour. This issue of external behaviour versus internal states is nicely demonstrated in [350] where it turns out to be surprisingly difficult for a human to kill a 'Mark III Beast' robot once it starts displaying desperate survival behaviour with corresponding sounds, so that people easily attribute feelings to the machine and start to feel pity.

These wide-ranging considerations form the background for a theory about computational behaviour in which the relation between observables and internal states is of central importance.

The generated behaviour that we claim to be the subject of computer science arises by a computer executing a program according to strict operational rules. The behaviour is typically observed via the computer's input and output (I/O). More technically, the program can be understood as an element in an inductively defined set P of terms. This set forms a suitable (initial) algebra $F(P) \to P$, where the expression (or functor) F captures the signature of the operations for forming programs. The operational rules for the behaviour of programs are described by a coalgebra $P \to G(P)$, where the functor G captures the kind of behaviour that can be displayed – such as deterministic or probabilistic. In abstract form, generated computer behaviour amounts to the repeated evaluation of an (inductively defined) coalgebra structure on an algebra of terms. Hence the algebras (structure) and coalgebras (behaviour) that are studied systematically in this text form the basic matter at the heart of computer science.

One of the big challenges of computer science is to develop techniques for effectively establishing properties of generated behaviour. Often such properties are formulated positively as wanted, functional behaviour. But these properties may also be negative, such as in computer security, where unwanted

behaviour must be excluded. However, an elaborate logical view about actual program properties within the combined algebraic/coalgebraic setting has not been fully elaborated yet.

Algebras and Coalgebras

The duality with algebras forms a source of inspiration and of opposition: there is a 'hate-love' relationship between algebra and coalgebra. First, there is a fundamental divide. Think of the difference between an inductively defined data type in a functional programming language (an algebra) and a class in an object-oriented programming language (a coalgebra). The data type is completely determined by its 'constructors': algebraic operations of the form $F(X) \to X$ going *into* the data type. The class however involves an internal state, given by the values of all the public and private fields of the class. This state can be observed (via the public fields) and can be modified (via the public methods). These operations of a class act on a state (or object) and are naturally described as 'destructors' pointing *out of* the class: they are of the coalgebraic form $X \to F(X)$.

Next, besides these differences between algebras and coalgebras there are also many correspondences, analogies and dualities, for instance between bisimulations and congruences or between initiality and finality. Whenever possible, these connections will be made explicit and will be exploited in the course of this work.

As already mentioned, ultimately, stripped to its bare minimum, a programming language involves both a coalgebra and an algebra. A program is a structured element of the algebra that arises (as so-called initial algebra) from the programming language that is being used. Each construct of the language corresponds to certain dynamics (behaviour), captured via a coalgebra. The program's behaviour is thus described by a coalgebra acting on the state space of the computer. This is the view underlying the so-called structural operational semantics. Coalgebraic behaviour is generated by an algebraically structured program. This is a simple, clear and appealing view. It turns out that this approach requires a certain level of compatibility between the algebras and coalgebras involved. It is expressed in terms of so-called distributive laws connecting algebra–coalgebra pairs. These laws appear in Chapter 5.

Coalgebras Have a Black Box State Space

Coalgebra is thus the study of states and their operations and properties. The set of states is best seen as a black box, to which one has limited access –

as with the states of a computer mentioned above. As already mentioned, the tension between what is actually inside and what can be observed externally is at the heart of the theory of coalgebras. Such tension also arises for instance in quantum mechanics where the relation between observables and states is a crucial issue [365]. Similarly, it is an essential element of cryptography that parts of data are not observable – via encryption or hashing. In a coalgebra it may very well be the case that two states are internally different but are indistinguishable as far as one can see with the available operations. In that case one calls the two states *bisimilar* or *observationally equivalent*. Bisimilarity is indeed one of the fundamental notions of the theory of coalgebras; see Chapter 3. Also important are *invariant* properties of states: once such a property holds, it continues to hold no matter which of the available operations is applied; see Chapter 6. Safety properties of systems are typically expressed as invariants. Finally, specifications of the behaviour of systems are conveniently expressed using assertions and modal operators such as: for all direct successor states (nexttime), for all future states (henceforth), for some future state (eventually); see also Chapter 6. This text describes these basic elements of the theory of coalgebras – bisimilarity, invariants and assertions. It is meant as an introduction to this new and fascinating field within theoretical computer science. The text is too limited in both size and aims to justify the grand unifying claims mentioned above. But it is hoped it will inspire and generate much further research in the area.

Brief Historical Perspective

Coalgebra does not come out of the blue. Below we shall sketch several, relatively independent, developments during the last few decades that appeared to have a common coalgebraic basis and that have contributed to the area of coalgebra as it stands today. This short sketch is of course far from complete.

1. **The categorical approach to mathematical system theory.** During the 1970s several people, notably Arbib, Manes and Goguen, and also Adámek, have analysed Kalman's [287] work on linear dynamical systems, in relation to automata theory. They realised that linearity does not really play a role in Kalman's famous results about minimal realisation and duality and that these results could be reformulated and proved more abstractly using elementary categorical constructions. Their aim was 'to place sequential machines and control systems in a unified framework' (abstract of [40]), by developing a notion of 'machine in a category' (see also [13, 14]). This led to general notions of state, behaviour, reachability, observability and

realisation of behaviour. However, the notion of coalgebra did not emerge explicitly in this approach, probably because the setting of modules and vector spaces from which this work arose provided too little categorical infrastructure (especially, no cartesian closure) to express these results purely coalgebraically.

2. **Non-well-founded sets.** Aczel [9] formed a next crucial step with his special set theory that allows infinitely descending $\in$-chains, because it used coalgebraic terminology right from the beginning. The development of this theory was motivated by the desire to provide meaning to concurrent processes with potentially infinite behaviour in Milner's calculus of communicating systems (CCS). Therefore, the notion of bisimulation from process theory played a crucial role. An important contribution of Aczel is that he showed how to treat bisimulation in a coalgebraic setting, especially by establishing the first link between proofs by bisimulations and finality of coalgebras; see also [12, 10].
3. **Data types of infinite objects.** The first systematic approach to data types in computing [165] relied on initiality of algebras. The elements of such algebraic structures are finitely generated objects. However, many data types of interest in computer science (and mathematics) consist of infinite objects, such as infinite lists or trees (or even real numbers). The use of (final) coalgebras in [462, 41, 192, 378] to capture such structures provided a next important step. Such infinite structures can be represented in functional programming languages (typically with lazy evaluation) or in logical programming languages [431, 189, 190].
4. **Initial and final semantics.** In the semantics of program and process languages it appeared that the relevant semantical domains carry the structure of a final coalgebra (sometimes in combination with an initial algebra structure [144, 132]). Especially in the metric space–based tradition (see e.g. [50]) this insight was combined with Aczel's techniques by Rutten and Turi. It culminated in the recognition that 'compatible' algebra–coalgebra pairs (called bialgebras) are highly relevant structures, described via distributive laws. The basic observation of [452, 451], further elaborated in [58], is that such laws correspond to specification formats for operational rules on (inductively defined) programs (see also [298]). These bialgebras satisfy elementary properties such as: observational equivalence (i.e. bisimulation with respect to the coalgebra) is a congruence (with respect to the algebra).
5. **Behavioural approaches in specification.** Reichel [397] was the first to use so-called behavioural validity of equations in the specification of algebraic structures that are computationally relevant. The basic idea is to

divide types (also called sorts) into 'visible' and 'hidden' ones. The latter are supposed to capture states and are not directly accessible. Equality is used only for the 'observable' elements of visible types. For elements of hidden types (or states) one uses behavioural equality instead: two elements x_1 and x_2 of hidden type are behaviourally equivalent if $t(x_1) = t(x_2)$ for each term t of visible type. This means that they are equal as far as can be observed. The idea is further elaborated in what has become known as hidden algebra [164] (see for instance also [152, 418, 68]) and has been applied to describe classes in object-oriented programming languages, which have an encapsulated state space. But it was later realised that behavioural equality is essentially bisimilarity in a coalgebraic context (see e.g. [339]), and it was again Reichel [399] who first used coalgebras for the semantics of object-oriented languages. Later on they have been applied also to actual programming languages such as Java [264, 245].

6. **Modal logic.** A more recent development is the connection between coalgebras and modal logics. In general, such logics qualify the truth conditions of statements concerning knowledge, belief and time. In computer science such logics are used to reason about the way programs behave and to express dynamical properties of transitions between states. Temporal logic is a part of modal logic which is particularly suitable for reasoning about (reactive) state-based systems, as argued for example in [389, 390], via its nexttime and lasttime operators. Since coalgebras give abstract formalisations of such state-based systems one expects a connection. It was Moss [359] who first associated a suitable modal logic to coalgebras – which inspired much subsequent work [403, 404, 322, 230, 243, 373, 317]; see [318] for an overview. The idea is that the role of equational formulas in algebra is played by modal formulas in coalgebra.

Position of This Text

There are several recent texts presenting a synthesis of several of the developments in the area of coalgebra [266, 453, 176, 411, 320, 374, 178, 16, 269]. This text is a first systematic presentation of the subject in the form of a book. Key points are: coalgebras are general dynamical systems; final coalgebras describe behaviour of such systems (often as infinite objects) in which states and observations coincide; bisimilarity expresses observational indistinguishability; the natural logic of coalgebras is modal logic; etc.

During the last decade a 'coalgebraic community' has emerged, centered around the workshops Coalgebraic Methods in Computer Science, see the proceedings [260, 267, 400, 109, 361, 180, 22, 151, 19, 262, 420, 82];

the conferences Coalgebra and Algebra in Computer Science (CALCO), see [130, 363, 326, 108, 209]; and the associated special journal issues [261, 268, 110, 181, 23, 20, 263, 375, 83]. Publications on coalgebras have also appeared outside these dedicated fora, in essentially all conference proceedings and journals in theoretical computer science – and also as a cover story in the *Communications of the ACM* [79]. This book is specifically not solely focused on the coalgebra community but tries to reach a wider audience. This means that the emphasis lies – certainly in the beginning – on explaining the theory via concrete examples and on motivation rather than on generality and (categorical) abstraction.

Coalgebra and Category Theory

Category theory is a modern, abstract mathematical formalism that emerged in the 1940s and 1950s in algebraic topology. It has become the preferred formalism in the area of semantics of datatypes and programming languages: it adequately captures the relevant phenomena and makes it possible to express similarities between different structures (such as sets, domains and metric spaces). The field of coalgebra requires the theory of categories already in the definition of the notion of coalgebra itself – since it uses the concept of a functor. However, the reader is not assumed to know category theory: in this text the intention is not to describe the theory of coalgebras in its highest form of generality, making systematic use of category theory right from the beginning. After all, this is only an introduction. Rather, the text starts from concrete examples and introduces the basics of category theory as it proceeds. Categories will thus be introduced gradually, without making it a proper subject matter. Hopefully, readers unfamiliar with category theory can thus pick up the basics along the way, seeing directly how it is used. Anyway, most of the examples that are discussed live in the familiar standard setting of sets and functions, so that it should be relatively easy to see the underlying categorical structures in a concrete setting. Thus, more or less familiar set-theoretic language is used in the first half of the book, but with a perspective on the greater generality offered by the theory of categories that is more prominent in the second half. In this way we hope to serve the readers without background in category theory, and at the same time offer the more experienced cognoscenti an idea of what is going on at a more abstract level – which they can find to a limited extent in the Exercises but to a greater extent in the literature. Clearly, this is a compromise which runs the risk of satisfying no one: the description may be too abstract for some, and too concrete for others. The hope is that it does have something to offer for everyone.

In the first half of the book (Chapters 1–3) the formalism of categories will not be very prominent, for instance, in the restriction to so-called polynomial functors which can be handled rather concretely. This is motivated by our wish to produce an introduction that is accessible to non-specialists. Certainly, the general perspective is always right around the corner and we hope it will be appreciated once this more introductory material has been digested. Certainly in the second half of the book, starting from Chapter 4, the language of category theory will be inescapable.

Often the theory of categories is seen as a very abstract part of mathematics that is not very accessible. However, this theory is essential in this text, for several good reasons.

1. It greatly helps to properly organise the relevant material on coalgebras.
2. Only by using categorical language can the duality between coalgebra and algebra be fully seen – and exploited.
3. Almost all of the literature on coalgebra uses category theory in one way or another. Therefore, an introductory text that wishes to properly prepare the reader for further study cannot avoid the language of categories.
4. Category helps you to structure your thinking and to ask relevant questions: ah, this mapping is a functor, so what structure does it preserve? Does it have an adjoint?

In the end, we think that coalgebras form a very basic and natural mathematical concept and that their identification is real step forward. Many people seem to be using coalgebras in various situations, without being aware of it. It is hoped that this text will make them aware and will contribute to a better understanding and exploitation of these situations. And it is hoped that many more such application areas will be identified, further enriching the theory of coalgebras.

Intended Audience

This text is written for everyone with an interest in the mathematical aspects of computational behaviour. This probably includes primarily mathematicians, logicians and (theoretical) computer scientists, but hopefully also an audience with a different background such as mathematical physics, biology or even economics. A basic level of mathematical maturity is assumed, for instance familiarity with elementary set theory and logic (and its notation). The examples in the text are taken from various areas. Each section is accompanied by a series of exercises, to facilitate teaching – typically at a late Bachelor's or early Master's level – and for testing one's own understanding in self-study.

Acknowledgements

An earlier version of this book has been on the web for quite some time. This generated useful feedback from many people. In fact, there are too many of them to mention individually here. Therefore I would like to thank everyone in the coalgebra community (and beyond) for their cooperation, feedback, help, advice, wisdom, insight, support and encouragement.