

Structure and Behaviour: A Coinductive Account of Processes and Components

Luís S. Barbosa

¹ CCTC — Centro de Ciências e Tecnologias da Computação
Departamento de Informática
Universidade do Minho
Braga, Portugal

lsb@di.uminho.pt

Abstract. *This text provides a brief introduction to a series of lectures to be delivered at the Shcool of Mathematical Sciences of Peking University in September of 2006. This initiative is framed in the context of a Technological and Scientific Cooperation project between Portugal and the P. R. of China (under contract GRICES-00342), on Formal Foundations for Component-based Programming.*

1. Motivation and Purpose

An increasing number of computer based systems rely on the cooperation of distributed, heterogeneous components organised into open software architectures that, moreover, can survive in loosely-coupled environments and be easily adapted to changing application requirements. Such is the case, for example, of applications designed to take advantage of the increased computational power provided by massively parallel systems or of the whole business of Internet-based software development. In order to develop such systems in a systematic way, the focus of development methods has switched, along the last decade, from functional to structural issues: both data and processes are encapsulated into software units which are connected into large systems resorting to a number of techniques intended to support reusability and modifiability. This encapsulation principle is essential to both the *object-oriented* and the more recent *component-based* software engineering paradigms.

This corresponds, at the semantic level, to an increasing emphasis put on *observation* and techniques to describe and reason about the (externally observable) behaviour of computational systems. If on data intensive applications, such as databases or data warehousing, the main element to tackle is the *structure* of information and their transformations, in dynamic, reactive computing the focus is placed on system's behaviour and their interactions. Quoting Robin Milner, in his Turing Award Lecture, computing science has become a *structural theory of interaction*: *Thus software, from being a prescription for how to do something — in Turing's terms a "list of instructions" — becomes much more akin to a description of behaviour, not only programmed on a computer, but occurring by hap or design inside or outside it.*

Both *initial* algebras and *final* coalgebras are devices which provide abstract descriptions of a variety of phenomena in programming, in particular of *data* and *behavioural* structures, respectively. As universal properties, they both entail definitional and proof principles, *i.e.*, a basis for the development of program calculi directly based on (actually driven by) type specifications. Moreover, such properties can be turned into programming *combinators* and used, not only to calculate programs, but also to program with. In functional programming the role of such universals has been fundamental to a whole discipline of algorithm derivation and transformation. On the other hand, *coalgebraic modelling* of dynamical systems and reasoning by *coinduction* has recently emerged as active area of research.

This course explores the role of such algebraic and coalgebraic structures in program development. As expected, *initial* algebras turn out to be *inductive data types*, *i.e.*, abstract descriptions of data structures. Dually, *final* coalgebras entail a notion of *coinductive, behaviour types*, representing the dynamics of systems.

Our emphasis is clearly placed on the *co*-side. As a matter of fact, there are several phenomena in computing which are hardly definable (or even simply not definable) as algebras, *i.e.*, in terms of a complete set of constructors. For example, processes, transition systems, objects, stream-like structures used in lazy programming languages, ‘infinite’ or non well-founded objects arising in semantics, and so on. Such ‘systems’ are inherently dynamic, do possess an observable behaviour, but their internal configurations remain hidden and have therefore to be identified if not distinguishable by observation.

The basic motivation for componentware is well-known: replacing conventional programming by system’s construction by composition and configuration of reusable off-the-shelf units, often regarded as ‘*abstractions with plugs*’. If all engineering disciplines rely on standard components to design and build their artifacts, software engineering should not be the exception. But, on the other hand, as it happened before with object orientation, component based design has grown up to popular technologies before consensual definitions and principles, let alone formal foundations, have been put forward.

In such a context, this course purpose is to go a few steps into the foundational direction. In particular, we intend to use coinduction to achieve a better understanding of the mathematics underlying the computational structures, such as processes and components, which form the basis of modern software design.

2. Course Plan

Outline

The development of program calculi directly based on, *i.e.*, actually driven by, type specifications has had a fundamental impact on algorithm derivation and transformation, mainly on the framework of *functional programming*. Actually, since John McCarthy original papers in the 60’s, and, later, John Backus FP proposal — a functional language based on combinators related by algebraic laws — functional programming and *program calculi* have developed in an intertwined way.

Can this program methodology be extended to the realm of *dynamic, reactive, communicating* systems?

Such is the recurring question in these lectures.

Faced with the recent paradigm shift in computing from stand-alone to distributed, open and dynamically changeable environments, computer scientists are under pressure to develop new conceptual models. On-going research on *coalgebras* suggests that, at the *specification* level, the duality between algebraic and coalgebraic structures may provide a bridge between models of *static* and *dynamical* systems. At the *programming* level such a duality, in a canonical initial-final specialisation, captures the intuitive symmetry between *data* and *behaviour*, providing the basis for more uniform and generic approaches to systems' construction.

This course is an introduction to *coinduction*, both as a *modelling* and *reasoning* tool, and to its application to the development of models and calculi for dynamic computational structures. We will consider, in particular,

- *processes* (understood as specifications of observable behaviours),
- and (state-based) *software components* (taken as the building blocks of modern software design).

From a methodological point of view, the main focus of this course is placed on *reasoning principles* for such structures, developing an entirely *calculational* approach to coinduction which avoids the explicit construction of bisimulations, and, therefore, promotes a reasoning style closer to the actual program construction practice.

Plan

1. Introduction: *modelling* is for *reasoning*
2. Coinductive reasoning: a *calculational* perspective
3. Applications: models and calculi for processes and components
 - (a) Generic *process algebra* (a reconstruction of CCS)
 - (b) A calculus of *software components*
 - (c) Componentware: from *composition* to *coordination*

Pre-requisites

These series of lectures can be attended by researchers and post-graduate students in Computer Science, preferably with some previous experience in functional programming and/or semantics of programming languages. We shall try to make the course almost self-contained. No special familiarity with category theory is assumed, elementary concepts being reviewed in the introduction.