Journal of Logical and Algebraic Methods in Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Continuity as a computational effect

Renato Neves [a,*], Luis S. Barbosa [a], Dirk Hofmann [b], Manuel A. Martins [b]

[a] INESC TEC (HASLab) & Universidade do Minho, Portugal
[b] CIDMA – Dep. of Mathematics, Universidade de Aveiro, Portugal

ARTICLE INFO

Article history:
Received 2 July 2015
Received in revised form 28 May 2016
Accepted 31 May 2016
Available online xxxx

Keywords:
Monads
Components
Hybrid systems
Control theory

ABSTRACT

The original purpose of component-based development was to provide techniques to master complex software, through composition, reuse and parametrisation. However, such systems are rapidly moving towards a level in which software becomes prevalently intertwined with (continuous) physical processes. A possible way to accommodate the latter in component calculi relies on a suitable encoding of continuous behaviour as (yet another) computational effect.

This paper introduces such an encoding through a monad which, in the compositional development of hybrid systems, may play a role similar to the one played by $1+$, powerset, and distribution monads in the characterisation of partial, nondeterministic and probabilistic components, respectively. This monad and its Kleisli category provide a universe in which the effects of continuity over (different forms of) composition can be suitably studied.

1. Introduction

1.1. Motivation and objectives

Component-based software development is often explained through a visual metaphor: a palette of computational units, and a blank canvas in which they are dropped and interconnected by drawing wires abstracting different composition and synchronisation mechanisms. More and more, however, components are not limited to traditional information processing units, but encapsulate some form of interaction with physical processes. The resulting systems, referred to as *hybrid* [1,2], exhibit a complex dynamics in which computations, coordination, and physical processes interact, become mutually constrained, and cooperate to achieve specific goals.

One generic way of looking at components, proposed in [3], emphasises an *observational* semantics, through a signature of observers and methods, that makes them amenable to a *coalgebraic* characterisation as (generalisations of) abstract Mealy machines. The resulting calculus is parametric on whatever behavioural model underlies a component specification. This captures, for example, partial, nondeterministic or probabilistic behaviour of a component's dynamics by encoding such behavioural effects as *strong monads* [4] – a pervasive mathematical structure with surprising applications in different areas of Computer Science (see *e.g.*, [5–9]).

Indeed, each monad captures a specific type of behaviour, which is then reflected in the corresponding component calculus. For example, *maybe* monad $(1+)$ introduces *partial* components; the *powerset* $(\mathcal{P})$ monad *nondeterministic* ones;

* Corresponding author.
E-mail addresses: rjneves@inescporto.pt (R. Neves), lsb@di.uminho.pt (L.S. Barbosa), dirk@ua.pt (D. Hofmann), martins@ua.pt (M.A. Martins).

http://dx.doi.org/10.1016/j.jlamp.2016.05.005
2352-2208/

Please cite this article in press as: R. Neves et al., Continuity as a computational effect, J. Log. Algebraic Methods Program. (2016), http://dx.doi.org/10.1016/j.jlamp.2016.05.005

and *distribution* monad ($\mathcal{D}$) brings (discrete) *probabilistic* evolution into the scene. Can *continuous behaviour*, prevalent in hybrid systems and control theory, be encoded in a similar way, as (yet another) computational effect? Such is the question addressed in this paper.

Monads first came in contact to Computer Science in the 80's, when E. Moggi proposed their use to structure the denotational semantics of programming languages [10,5]. Later the concept was introduced in programming practice by P. Wadler [6], leading to a rigorous style of combining purely functional programs that mimic impure (side-)effects. The key idea is that monads encode in abstract terms several kinds of computational effects, such as exceptions, state updating, nondeterminism or continuations. Such effects are represented by a type constructor $\mathcal{T}$ (an endofunctor over a suitable category) so that computations producing values of type O are regarded as terms of type $\mathcal{T}O$. In this way *values* and *computations* are explicitly distinguished and programs can be thought of as arrows $I \to \mathcal{T}O$ representing the computation of values of type O from values of type I, while producing some effect described by $\mathcal{T}$. Or, putting it in a different way, output values are encapsulated (or embedded) in the effect specified by $\mathcal{T}$. A monad comes equipped with an identity and an associative multiplication which, from a computational point of view, builds a (trivial) computation from a value, and flattens nested effects, respectively. Furthermore, if $\mathcal{T}$ is *strong* [6] additional machinery is available to distribute the computations' effect over context. The monad structure allows program composition by handling the underlying computational effect through functor $\mathcal{T}$ and the flattening operation. Actually, each monad gives rise to a so called Kleisli category in which one may study the effects of the behavioural type (as specified by the monad) over different forms of composition; ultimately, this leads to rich component calculi (as discussed in [3]).

The current paper introduces a (strong) monad $\mathcal{H}$ that subsumes the typical continuous behaviour of dynamical, and hybrid systems. Intuitively, the type effect of $\mathcal{H}$ (*i.e.*, the underlying endofunctor) represents the (continuous) *evolution* over time of some value in O; the identity defines a trivial evolution (*i.e.*, with duration zero), and the flattening operation allows the control of an evolution to be passed along different systems.

Moreover, the paper explores the corresponding Kleisli category as the mathematical space in which the underlying (continuous) behaviour can be isolated and its effect over different forms of composition suitably studied. As we will see in the sequel, such a category gives rise to several forms of composition operators (*e.g.*, sequential, parallel execution), *wiring* mechanisms, and *synchronisation* techniques. Again this parallels the role that the categories of partial functions, relations and stochastic matrices have as reasoning universes for component composition under the behavioural model provided, respectively, by monads $1+$, $\mathcal{P}$ and $\mathcal{D}$ [11,12]. Similarly, this work paves the way to the development of a coalgebraic calculus of *hybrid components* in the spirit of [3].

1.2. A tribute to José Nuno Oliveira

The idea of regarding *continuity* as a computational effect, or more rigorously, a *physical* one, entailing a suitable notion of composition and a reasoning universe, in the form of a Kleisli category, owes much to the way José helped us to approach computational phenomena.

Building on the role of monads in functional programming and program calculi, as monadic inductive and coinductive schemes [13,14], José introduced us to monads both as a powerful structuring mechanism and a source of equally powerful genericity. An obsession for patterns and a sharp intuition for generic, conceptually reusable structures remain, after all, the hallmark of his illuminating, Socratic teaching.

In the late 1990's, José supervised the PhD work of the second author on the coalgebraic calculus of state-based components mentioned above [3]. This emerged from the conjunction of two key ideas; first, that a 'black-box' characterisation of software components favoured an *observational*, essentially coalgebraic, semantics; second, that the envisaged calculus had to be *generic*, in the sense that it should not depend on a particular notion of component behaviour. Monads, actually strong monads, were quickly identified as a source of such a genericity, the whole work boiling down to a calculus of *monadic* Mealy machines. Software components were thus studied as coalgebras (in a suitable category) typed as

$$S \longrightarrow \mathcal{T}(S \times O)^I$$

where S represents the (internal) state space, and I, O are respectively the input and output spaces. $\mathcal{T}$ is a strong monad that captures the intended behavioural effect.

Being generic entailed the need for an equally generic reasoning framework. By then, the adoption of a *pointfree*, essentially equational, calculational proof style, thus avoiding the somehow more standard coinductive proofs through the explicit construction of bisimulations, was understood as the price to be paid for genericity, as component laws were to be verified without fixing the working monad completely. Generic proofs performed in this style are clear and easy to follow, even if often long due to the systematic recording of almost all elementary steps.

For José, however, the way proofs are written is not a technicality. Proofs, as he taught us every day, are basically honest explanations, bearing evidence in a fixed formal context, and therefore must be conveyed in a crisp, clear, easily reproducible style, letting the underlying structure to emerge and helping to build the correct intuitions. Years later, in the context of a joint research project [15], José championed the use of calculational, pointfree reasoning as a way of reinvigorating the role of proof in elementary mathematical education. The pointfree style adopted in many proofs of this paper is also intended as a tribute to this view.

Please cite this article in press as: R. Neves et al., Continuity as a computational effect, J. Log. Algebraic Methods Program. (2016), http://dx.doi.org/10.1016/j.jlamp.2016.05.005