Let's See How Things Unfold: Reconciling the Infinite with the Intensional
(Extended Abstract)

Conor McBride

University of Strathclyde

1 Introduction

Coinductive types model infinite structures unfolded on demand, like politicians' excuses: for each attack, there is a defence but no likelihood of resolution. Representing such evolving processes coinductively is often more attractive than representing them as functions from a set of permitted observations, such as projections or finite approximants, as it can be tricky to ensure that observations are meaningful and consistent. As programmers and reasoners, we need coinductive definitions in our toolbox, equipped with appropriate computational and logical machinery.

Lazy functional languages like Haskell [18] exploit call-by-need computation to over-approximate the programming toolkit for coinductive data: in a sense, all data is coinductive and delivered on demand, or not at all if the programmer has failed to ensure the *productivity* of a program.

Tatsuya Hagino pioneered a more precise approach, separating initial data from final codata [10]. The corresponding discipline of 'coprogramming' is given expression in Cockett's work on Charity [5,6] and in the work of Turner and Telford on 'Elementary Strong Functional Programming' [22,21,23]. Crucially, all distinguish recursion (structurally decreasing on input) from *corecursion* (structurally increasing in output). As a total programmer, I am often asked 'how do I implement a *server* as a program in your terminating language?', and I reply that I do not: a server is a *coprogram* in a language guaranteeing liveness.

To combine programming and reasoning, or just to program with greater precision, we might look to the proof assistants and functional languages based on intensional type theories, which are now the workhorses of formalized mathematics and metatheory, and the mainspring of innovation in typed programming [16,4,14]. But we are in for a nasty shock if we do. Coinduction in Coq is *broken*: computation does not preserve type. Coinduction in Agda is *weak*: dependent observations are disallowed, so whilst we can unfold a process, we cannot *see* that it yields its unfolding.

At the heart of the problem is *equality*. Intensional type theories distinguish two notions of equality: the typing rules identify types and values according to an equality *judgment*, decided mechanically during typechecking; meanwhile, we can express equational *propositions* as types whose inhabitants (if we can find them) justify the substitution of like for like.

A. Kurz, M. Lenisa, and A. Tarlecki (Eds.): CALCO 2009, LNCS 5728, pp. 113–126, 2009.

In neither Coq nor Agda is a coprogram judgmentally equal to its unfolding, hence the failure in the former. That is not just bad luck: in this presentation, I check that it is impossible for any decidable equality to admit unfolding.

Moreover, neither system admits a substitutive propositional equality which identifies bisimilar processes, without losing the basic computational necessity that closed expressions compute to canonical values [13]. That is just bad luck: in this presentation, I show how to construct such a notion of equality, following earlier joint work with Altenkirch on observational equality for functions [2].

The key technical ingredient is the notion of 'interaction structure' due to Hancock and Setzer [11] — a generic treatment of indexed coinductive datatypes, which I show here to be closed under its own notion of bisimulation. This treatment is ready to be implemented in a new version of the Epigram system.

Equipped with a substitutive propositional equality that includes bisimulation, we can rederive Coq's dependent observation for codata from Agda's simpler coalgebraic presentation, whilst ensuring that what types we have, we hold. Let's see how things unfold.

2 The Problem

Eduardo Giménez pioneered Coq's treatment of coinduction [7]. It was a great step forward in its time, giving Coq access to many new application domains. Giménez was aware of the problem with type preservation, giving a counterexample in his doctoral thesis [8]. The problem did not become particularly widely known until recently, when Nicolas Oury broke an overenthusiastic early version of coinduction in Agda, then backported his toxic program to Coq, resulting in a flurry of activity on mailing lists which has not yet entirely subsided.

Presented with a categorical flavour, Coq's treatment is essentially thus: for any given strictly positive functor $F : \mathsf{Set} \to \mathsf{Set}$, we acquire a coinductive set νF equipped with a coconstructor and a coiterator (or 'unfold', or 'anamorphism') which grows νF values on demand by successive applications of a coalgebra to a 'seed' of arbitrary type. Keeping polymorphism implicit, we obtain:

$$\begin{aligned} \nu F &: \mathsf{Set} \\ \mathsf{in}_F &: F\ (\nu F) \to \nu F \\ \mathsf{coit}_F &: (S \to F\ S) \to S \to \nu F \end{aligned}$$

Of course, Coq actually provides a much richer notation for coprograms than just coit_F, but a streamlined presentation will help to expose the problem.

For the standard example of coinductive lists, this specializes (modulo high school algebra) to the traditional pair of coconstructors and unfold.

$$\begin{aligned} \mathsf{CoList}_X &: \mathsf{Set} \\ \mathsf{nil}_X &: \mathsf{CoList}_X \\ \mathsf{cons}_X &: X \to \mathsf{CoList}_X \to \mathsf{CoList}_X \\ \mathsf{unfold}_X &: (S \to 1 + X \times S) \to S \to \mathsf{CoList}_X \end{aligned}$$