Coinduction and topology: an (un)expected connection

Wojciech Kołowski

17 June 2020

Review of approaches to coinduction

- So far we have seen two approaches to coinduction and bisimulation:
- The LTS approach, in which a coinductive process is (re)presented using a particular kind of machine. This machine can be in any of a number of states and can transition between them by performing an appropriate action.
- The categorical approach, in which we are interested in coalgebras of endofunctors. In the final coalgebra $(\nu F, \alpha)$ the "coinductively" defined object is the carrier νF, and α takes the object apart, splitting into its constituent parts (it can also be seen as performing some observations on νF). Corecursion is a consequence of finality.

The LTS approach: a critique 1/2

- I think that the LTS approach to coinduction and bisimulation is quite bad from the explanatory point of view, for a few reasons:
- First, it obscures the very important duality between induction and coinduction, which everybody wants to learn about instantly upon seeing the name "coinduction".
- Interlude: the right notion of equality for LTSes is of course graph isomorphism, as they are nothing more than labeled graphs – static, immobile objects that prescribe actions and transitions, but don't act and don't transition.

The LTS approach: a critique 2/2

- Second, the idea of bisimulation is a bit ad hoc and circular.
- Bisimulation was advertised in the book as the right notion of behavioural equality of LTSes, but it is in fact the right notion of equality for behaviours of LTSes. The behaviour of an LTS is its dynamic aspect – where the actions and transitions take place.
- However, to formally define such a notion of behaviour, we need coinduction in the first place (or else we will miss "infinite" phenomena). Thus, there is some kind of circularity in explaining coinduction using LTSes, even if only conceptual.

The categorical approach: a critique

- The categorical approach is much better, as it makes the duality between induction and coinduction more explicit and also doesn't give the false impression that coinduction is about automata.
- However, it is not without faults:
- By using the machinery of category theory it makes coinduction seem more magical and arcane than it really is. It is unlikely to be enlightening to ordinary programmers and people with category theory disability.
- It makes the operational and computational aspects of corecursion less explicit.
- It does not provide a nice syntax/notation for corecursive definitions (even though it does provide $\nu X.F(X)$ for objects) – and that's very important! "Notation is the tool of thought", they say.

How to explain coinduction to 5 year olds

- I think that the most natural way of explaining coinduction is to refer to an informal version of the duality with induction, explain it in depth and then present lots of examples and exercises to build the right intuitions.
- So, let's do just that!

The duality

feature ↓	induction	coinduction
shape	sum (of products)	product (of sums)
basic activity	construction	deconstruction (observation)
derived activity	deconstruction (observation)	construction
easy to define functions with	inductive domain	coinductive codomain
such that every (co)recursive call	shrinks the principal argument	grows the result
thus these functions are	terminating	productive
evaluation	possibly eager	necessarily lazy
tree height	necessarily finite	possibly infinite
tree width	any	any

Scary explanations

- The first four rows in the table can be restated in terms of category theory and logic/type theory
- In categorical terms it means that inductives have a "mapping-out" universal property, i.e. they are colimits, whereas coinductives have a "mapping-in" universal property, i.e. they are limits.
- In logical terms, we can say that inductives have positive polarity and coinductives have negative polarity. For nice explanations of polarity, see `https://existentialtype.wordpress.com/2012/08/25/polarity-in-type-theory/` and `http://noamz.org/talks/logpolpro.pdf`.
- All of this can also be restated in less scary terms.

Intuitions for programmers

- For programmers this basically means that inductives are data (like the stuff you can find in databases), whereas coinductives are interactive processes (like operating systems or web servers). This basically means that:
- Inductives are determined by ways of constructing their elements and the elimination principle is a derived notion whose purpose is to say "the only things you can build come from the introduction principles (constructors)".
- Coinductives are determined by ways of observing their elements and the introduction principle is a derived notion whose purpose is to say "the only things you can observe come from the elimination principles".

Examples for programmers

- Examples for the verbiage from previous slide come in handy:
- The type `bool` is an inductive type with two constructors `true` and `false`. Knowing this we can derive an elimination principle, which amounts to an `if-then-else` construction (but dependently typed!).
- Imagine a type of web servers that can only handle requests for pictures of funny cats (this is the elimination principle). Thus we know that there must be something in the web server that is responsible for handling these requests and the introduction principle consists of all the possible ways of constructing that thing.

Philosophical points: laziness and strictness 1/2

- **Laziness and strictness are properties of types, not of languages**.
- Therefore it does not make much sense to talk about lazy or stric languages.
- Because coinductives are possibly infinite, they can't be evaluated eagerly and thus any language that incorporates them will have some form of lazy evaluation.
- Inductives can be evaluated both lazily and eagerly, buth since they are data which is most often meant to be passed to some function for further processing, it makes more sense for them to be eager.

Philosophical points: laziness and strictness 2/2

- An interesting case is the product type, which can be defined both inductively and coinductively.
- Elements of inductive products are constructed by pairing two things. The derived eliminator says that we can pattern match on the pair to get them back and then use them to construct something else.
- Elements of coinductives products are eliminated using projections. The derived introduction rule says that we must provide everything that can be projected out, so it is pairing too.
- Both product types are isomorphic, but inductive products are best thought of as "eager products", whereas coinductive products are best thought of as "lazy products".

Philosophical points: termination and productivity 1/2

- In programmers' collective consciousness lives the idea a "looping" program, exemplified by infinite loops like `while(true) {...}` or an erroneous implementation of recursive factorial on integers with the base case missing.
- **It is a conflation of two more basic concepts: nontermination and nonproductivity.**

Philosophical points: termination and productivity 2/2

- Termination and nontermination pertain only to recursive functions. The purpose of such a function is to consume its input and produce some output. If the function keeps running forever, it means it has failed to achieve its goal. In this case "looping" is nontermination and it is evil.
- Productivity is a property that only corecursive functions can possess. The purpose of such functions is, given some input, to keep producing output for as long as it is requested. If the function keeps running forever, it is not necessarily broken. After all, it may be the case that requests will never stop coming. In this case the evil thing is called "nonproductivity" and it happens when a functions hangs forever on answering a single request.

Philosophical points: termination and productivity 3/2

- To sum it up:
-

A comparatory example

- How does the duality play out in practice? Let's see an example in Coq!
- The code for this example, named `Snippet1.v`, is available from the GitHub repo of this talk: `https://github.com/wkolowski/Seminar-Bisimulation-and-Coinduction`

A comparatory example (informal)

```
data List a = Nil | Cons a (List a)

sum : List Int -> Int
sum Nil = 0
sum (Cons x xs) = x + sum xs
```

```
data List a = Nil

sum : (List a) Int ->
sum Nil = 0
sum (Cons x xs) =
```

Rescuing the duality