Coinduction in type theory: a topological connection

Wojciech Kołowski

17 June 2020

1. Review and critique

2. An intuitive approach to coinduction

3. Coinduction in type theory

4. Finite, infinite and searchable types

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

Explaining the first half of the duality

- The first half of 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.

(Co)induction, (co)limits, positive and negative types 1/2

- 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".
- For programmers this basically means that inductives are data (similar to the stuff stored in databases), whereas coinductives are interactive processes (like operating systems or web servers).

(Co)induction, (co)limits, positive and negative types 2/2

- 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` expression (but dependently typed!).
- Imagine a type of web servers that can only handle requests for pictures of funny cats (this is the elimination principle). From this description we know that there must be something in the web server that is responsible for handling these requests and thus the derived introduction principle specifies all the possible ways of constructing that thing.

Explaining the second half of the duality

- The second half of the duality is quite clear and doesn't need to be demystified as much as the first, but there are two philosophical misconceptions to be addressed.
- The first is about lazy and strict languages.
- The second is about "infinite loops".

Laziness and strictness 1/3

- 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.
- 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.
- We may think that inductive types are (or at least should) be "eager", whereas coinductive types are "lazy".

Laziness and strictness 2/3

- 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 coinductive 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”.

Lazines and strictness 3/3

- **Laziness and strictness are properties of types, not of languages**
- Haskell is usually said to be a "lazy language", but in reality it's just that its types are lazy by default. Given some strictness annotations we can define strict types or even mixed strict-lazy types.
- OCaml or StandardML are usually said to be "strict" languages, but it's just that their types are strict by default. We can make the type `'a` lazy by turning it into a function type with unit domain: `unit -> 'a`.

Termination and productivity 1/3

- In programmers' collective consciousness there is the term "infinite loop", usually applied to describe two kinds of programs.
- The first kind looks like `while(true) {...}`. In such cases the "infinite loop" was programmed intentionally. Its purpose may be to, for example, implement a server that is waiting for requests.
- The second kind looks like an ordinary loop, but the stopping condition will never be met. In such cases the "infinite loop" was programmed by mistake and thus is a bug.
- This term would also be applied to describe an erroneous implementation of recursive factorial on integers with the base case missing, even though there isn't any kind of loop going on.
- Note that this term is very one-sided – terminating programs aren't called "finite loops".

Termination and productivity 2/3

- **The term “infinite loop” is considered harmful, because it conflates two separate notions: termination and productivity.**
- Termination is a property that pertains only to recursive functions.
- Each recursive call must shrink the input and may produce a part of the output.
- Therefore all recursive functions terminate.
- Productivity is a property that pertains only to corecursive functions.
- Each corecursive call may do anything with the input and must produce a part of the output.
- Therefore all corecursive functions are productive.

Termination and productivity 3/3

- To sum it up:
- Bugged implementation of factorial on integers without the base case: recursive, nonterminating, not ok.
- Correct implementation of factorial on natural numbers: recursive, terminating, ok.
- Correct implementation of a web server that serves pictures of funny cats: corecursive, productive, ok.
- A web server that hangs for some funny cat requests: corecursive, nonproductive, not ok.

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 closer look at the duality

- Earlier we said that for an inductive type I it's easy to define functions of type $I \to X$, whereas for a coinductive type C it's easy to define functions $X \to C$.
- The induction principle for `nat` looks like this (in Coq):
  ```
  forall P : nat -> Type,
  P 0 -> (forall n : nat, P n -> P (S n)) ->
  forall n : nat, P n
  ```
- The corecursion principle for `conat` looks like this (in Coq):
  ```
  forall X : Type, (X -> option X) -> conat
  ```
- The first trouble lies in names: "induction principle" vs "corecursion principle". Where is the coinduction principle?
- The second trouble lies in the strong assymmetry between the two principles: they look nothing like each other's mirror images.

A different look at the induction principle

- To rescue the duality, we have to squint at the induction principle and reformulate it in terms more amenable to being dualized – we need to split it into a recursion principle and a uniqueness principle.
- The recursion principle consists of the recursor `rec` and its computation rules:

```
rec : forall X : Type, X -> (X -> X) -> nat -> X
uniq1 : rec X z s 0 = z
uniq2 : rec X z s (S n) = s (rec X z s n)
```

- The uniqueness principle looks like this: if a function `f` satisfies the equations

```
f 0 = z
f (S n) = s (f n)
```

  for some `z : X` and `s : X -> X`, and a function g also satisfies them, then `forall x : X, f x = g x`.

The coinduction principle

- We can now state the coinduction principle (for conatural numbers) and see the duality in full glory.
- The corecursion principle: there is a corecursor `corec` that satisfies some equations:

```
corec : forall X, (X -> option X) -> X -> conat
pred (corec X p x) =
match p x with
| None => None
| Some x' => Some (corec X p x')
end
```

- Given two functions `f` and `g` that satisfy the same corecursive equation as `corec X f`, we have `forall x : X, f x = g x`.

Meaning of the coinduction principle

- Meaning of the corecursor is quite clear – it's for making functions into coinductives. But what is the (hidden) meaning of the uniqueness principle?
- To understand it, we need to notice that, even though it is a statement about **functions** into coinductives, in reality it says something about equality of **elements** of coinductives.
- For conatural numbers, it states that two numbers are equal if they both have equal predecessors, and their predecessors have equal predecessors and so on.
- For streams, it would state that two streams are equal if they have equal heads and their tails have equal heads and so on.
- So, the uniqueness principle states that bisimilar numbers/streams/structures are equal.

Induction, recursion, uniqueness

- The recursion and uniqueness principles are independent – they can't be derived from each other.
- They can, however, be derived from the induction principle – to prove `forall i : I, f i = g i` where `I` is an inductive type, just use induction.
- For each case in the induction, we have the appropriate equations and inductive hypotheses, so it is straightforward.

Coinduction, corecursion, uniqueness

- The corecursion and uniqueness principles are independent too.
- However, because (in Coq) we only have the corecursion principle, we can't derive the uniqueness principle in any way.
- How to prove `forall x : X, f x = g x`, for `f g : X -> C` and `C` coinductive?
- Because the equality type is inductive, we can't use corecursion and because we don't know anything about `X`, we can't do anything with it.
- Therefore we are stuck without the uniqueness principle. In Coq, we can't prove bisimilar objects equal without assuming axioms.

How to get (co)inductive types?

- We saw in earlier talks that in set theory, coinductive definitions are not a basic concept and have to be derived from the ZF axioms. Inductives aren't basic either.
- What's the situation in Coq and type theory in general?
- In Coq, (co)inductives come from schematic definitions using the keywords `(Co)Inductive`. This means we can specify

- $\mathtt{Searchable}(A) :\equiv$
$$\prod_{p:A\to 2}\left(\sum_{x:A} p(x) = \mathit{true}\right) + \left(\prod_{x:A} p(x) = \mathit{false}\right)$$
- Intuition: we can find an element satisfying the boolean predicate p or get a proof that it doesn't exist.