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 coinductive processes are (re)presented using a particular kind of machine. This machine can be in any of a number of states and can transition between states by performing an 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 it 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 for 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
term height as tree	necessarily finite	possibly infinite

A comparatory example

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

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 Polarity in Type Theory and Polarity in Proof Theory and Programming.
- 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 too amounts to pairing.
- 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`.
- In Coq, inductive types are best thought of as strict, whereas coinductive types are best thought of as lazy.

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 no loop was used there.
- 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.
- In Coq and type theory, 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.
- In Coq and type theory, 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 requests of funny cat pictures in which the cat is upside-down: corecursive, nonproductive, not ok.

A closer look at the duality 1/2

- 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 $X \to C$.
- To define functions $I \to X$, we use the elimination principle, which for inductive types is called the induction principle.
- To define functions of type $X \to C$, we use the introduction principle, which for coinductive types is called the corecursion principle.
- There are two troubles with this part of the duality.
- The first trouble lies in names: "induction principle" vs "corecursion principle". Where is the coinduction principle?

A closer look at the duality 2/2

- As our running example of I, we will use `nat`, the natural numbers.
- As our running example of C, we will use `conat`, the conatural numbers (see here for the definition).
- 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) -> X -> conat
```
- The second trouble with the duality 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 1/2

- 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 states that there is a function (called the recursor)
  `rec : forall X : Type, X -> (X -> X) -> nat -> X`
  which for all `X : Type`, `z : X` and `s : X -> X`
  satisfies the following equations definitionally:
  `rec X z s 0` $\equiv$ `z`
  `rec X z s (S n)` $\equiv$ `s (rec X z s n)`

A different look at the induction principle 2/2

- The uniqueness principle is as follows. Given `X : Type` and two functions `f : nat -> X` and `g : nat -> X`, if for all `z : X` and `s : X -> X` they satisfy the same equations as `rec X z s` (but this time propositionally, i.e. using = instead of $\equiv$), then `forall n : nat, f n = g n`.
- An alternative formulation, which involves only a single function, is as follows: if `f : nat -> X` satisfies the same equations as `rec X z s`, then
  `forall n : nat, f n = rec X z s n`.
- We could also formulate variants of the uniqueness principle that uses $\equiv$ instead of =, but this is a very confusing territory, so we won't discuss it'll omit it.

The coinduction principle 1/2

- We can now state the coinduction principle (for conatural numbers) and see the duality in full glory.
- The corecursion principle states that there is a function (called the corecursor)
  `corec : forall X, (X -> option X) -> X -> conat`
  which for all `X : Type` and `p : X -> option X`
  satisfies the following equation definitionally:

```
pred (corec X p x) ≡
match p x with
| None => None
| Some x' => Some (corec X p x')
end
```

The coinduction principle 2/2

- The uniqueness principle is as follows. Given `X : Type` and two functions `f : X -> conat` and `g : X -> conat`, if for all `p : X -> option X` they satisfy the same equations as `corec X p` (but this time propositionally, i.e. using = instead of $\equiv$), then `forall x : X, f x = g x`.
- An alternative formulation: if `f : X -> conat` satisfies the same equations as `corec X p`, then `forall x : X, f x = corec X p 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: the recursion principle is a non-dependent special case of the induction principle and the uniqueness principle can be easily proved by using the appropriate equations and induction hypotheses.
- Likewise, the induction principle can be derived when we have both the recursion principle and the uniqueness principle, but the proof is a bit more involved.

Coinduction, corecursion, uniqueness

- The corecursion and uniqueness principles are independent too. When considered together, they can be thought of as the coinduction principle.
- Therefore in Coq, because 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 either.
- Therefore we are stuck without the uniqueness principle. In Coq, we can't prove bisimilar objects equal without assuming axioms.

How to add (co)inductive types to type theory?

- 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?
- There are three approaches that I'm aware of: schematic definitions, W-types/M-types and universes of codes.

Schematic definitions

- In Coq, (co)inductives come from schematic definitions using the keyword `(Co)Inductive`.
- After issuing such a definition, appropriate rules for the defined type are added to the environment.
- Pros: direct.
- Cons: defining a (co)inductive type is a side effect! This can be seen e.g. in a parameterized module in which we define a (co)inductive type. Then two modules defined by instantiating the parameters contain two copies of the same (co)inductive type which are different.
- Another cons: schematic definitions are hard to formally describe and therefore they are usually found in implementations, not in papers.

W-types, M-types

- Another approach to adding inductives/coinductives are the so called W-types/M-types, respectively.
- The idea is to add a single parameterized type `W` whose elements are well-founded trees.
- The parameters control the shape of the trees – the branching, types of non-recursive arguments etc. Particular inductive types can be recovered by instantiating `W` with the appropriate parameters.
- The type `M` is dual to `W` and can be used to represent coinductives. Its elements are possibly non-well-founded trees.
- Pros: one ring to rule them all, simple to describe formally.
- Cons: encodings using `W` and `M` have some overhead.
- Cons: `W` and `M` are inherently higher-order, which means we need functional extensionality for equality proofs to go.

Universes of codes

- Another idea is to have an (inductive) type, whose elements are "codes", and a function that interprets these codes as real inductive/coinductive types.
- This is basically a defunctionalization of `W` and `M`.
- Pros: one ring to rule them all, first-order so no problems with functional extensionality, not an encoding.
- Cons: harder to formally describe than `W` and `M`.
- See the paper The gentle art of levitation for more.

Snippet

- Code snippets that illustrate how schematic definitions are side-effecting and how the definitions of `W` and `M` look like can be found in the file `https://github.com/wkolowski/Seminar-Bisimulation-and-Coinduction/blob/master/SchematicWM.v`.

Drinker's paradox

- There's this well-known theorem of classical logic:
- **In every non-empty bar, there is a person such that if this person drinks, then everybody drinks.**
- This is usually presented to first-year students to make them confuse if-then implication with if-then causality.
- If we rewrite $P \to Q$ as $\neg P \vee Q$, the theorem gets demystified: **in every non-empty bar there is a person that does not drink or everybody drinks.**
- An algorithmically-minded person might wonder: how to find the non-drinker? Is it even possible? After all, the bar may be infinite!
- But remember that we are using classical logic – we can find him using the Law of Excluded Middle.
- How does this translate to constructive logic and functional programming?

Searchable types

$$\texttt{Searchable}(A) :\equiv \prod_{p:A\to\mathbf{2}} \left(\sum_{x:A} p(x) = \mathit{true}\right) + \left(\prod_{x:A} p(x) = \mathit{false}\right)$$

- A type that satisfies the drinker's paradox statement is called **searchable**.
- Spelled out explicitly: a type is searchable if, given any boolean predicate, we can either find an element that satisfies it or prove that there is no such element.
- Drinker's Paradox says that in classical logic all types are searchable. But which types are searchable constructively?

Interlude: why should we care?

- Maybe you're wondering why you should care about searchable types.
- Besides their evident utility when you're looking for something, there's a very nice theorem that may convince you.

Theorem

Let A and B be types. If A is searchable and B has decidable equality, then the function type $A \to B$ has decidable equality.

Proof.

We have $f, g : A \to B$. To decide whether $f = g$ just search the domain A for an $x : A$ that satisfies $f(x) \neq g(x)$. If you find one, $f \neq g$. Otherwise, $f = g$ (assuming functional extensionality) $\square$

Which types are searchable?

- So, which types are searchable?
- Certainly every finite type is searchable – just check every element.
- If A and B are searchable, so are $A + B$ and $A \times B$.

Which types are not searchable?

- On the other hand, there are some types which should not be searchable, unless we assume excluded middle.
- The most prominent such type is $\mathbb{N}$. We should have a strong feeling that it's impossible to search its infinitely many elements in finite time.
- So, without excluded middle we can't prove that infinite types are searchable, right?
- **Yes, we can (sometimes).**
- Whether a type is searchable depends not only on the number of elements, but also on how the type is laid out as a space.

Searchability of the conaturals

- The most prominent infinite type that is searchable are the conatural numbers (like natural numbers, but defined coinductively).
- The snippet with a Coq proof of their searchability is `https://github.com/wkolowski/Seminar-Bisimulation-and-Coinduction/blob/master/Searchability.v`.
- It is based on the paper Infinite sets that Satisfy the Principle of Omniscience in any Variety of Constructive Mathematics.

Exercises 1/2

- Ex. 1: remove the axiom `sim_eq` from my Coq proof of searchability of the conaturals and prove a lemma to fill the hole.
- Ex. 2: define a type of streams, its bisimilarity relation and its corecursion principle. Then prove that the uniqueness principle implies that bisimilar streams are equal.
- Ex. 3: define the bisimilarity relation of `M`-types. Encode streams using `M`-types. Prove that the types of normal streams and `M`-encoded streams are isomorphic (i.e. there are functions going both ways, such that the composition result is bisimilar to the argument).

Exercises 2/2

- Ex. 4: for a type A its subtypes can be represented by the type $\sum_{x:A} P(x)$ where $P : A \to \text{Prop}$ is a predicate. Are subtypes of searchable types searchable? If yes, give a Coq proof. If not, give a counterexample and find a different notion of subtype for which subtypes of searchable types are searchable.
- Grading: all or nothing. These exercises were designed to look very scary but in fact they are very easy :)