A Survey of Coinduction in Coq

Pierre-Marie Pédrot

PPS/\pi r^2

18 June 2015
1. A Quick Recap on Coq

2. To Infinite and Beyond: Coinduction in Coq

3. Those Infinite Spaces Frighten Me: Story of a Demise

4. Section IV: A New Hope
Both your favourite proof assistant and programming language
Based on the pCIC type theory
Famous developments: CompCert, 4-colour theorem...
Two prestigious ACM Awards last year
Both your favourite proof assistant and programming language
Based on the pCIC type theory
Famous developments: CompCert, 4-colour theorem...
Two prestigious ACM Awards last year
In the beginning was the Lambda

First versions of Coq only implemented CoC (Coquand-Huet, 1984).
In the beginning was the Lambda

First versions of Coq only implemented CoC (Coquand-Huet, 1984).

- Terms were essentially $\lambda$-terms (with rich typing)
- The only type former was $\Pi x : A. B$
- Poor expressivity as a logical system: $\nabla 0 \neq 1$
Then came the Inductive types

Inductive types were introduced by Christine Paulin-Mohring (1990).
Then came the Inductive types

Inductive types were introduced by Christine Paulin-Mohring (1990).

- New type formers: nat, list...
- New terms
  - Constructors: 0, 1, cons...
  - Destructor: \text{match } t \text{ with } \vec{p} \Rightarrow \vec{u} \text{ end}
  - Fixpoint: \text{fix } F \ n := t
- New fantasmabulous theorems: \vdash 0 \neq 1
A natural case study

Inductive nat := 0 : nat | S : nat → nat.
A natural case study

Inductive nat := 0 : nat | S : nat → nat.

Must be a positive functor! ⇝ Syntactic “positivity condition”
A natural case study

Inductive nat := 0 : nat | S : nat → nat.

Must be a **positive** functor! ⇝ Syntactic “positivity condition”

Definition nat_rect :

∀ (P : nat → Type)
   (p0 : P 0) (pS : ∀ n, P n → P (S n)) n, P n :=
fun P p0 pS ⇒
  fix F n := match n with
  | 0 ⇒ p0
  | S m ⇒ pS m (F m)
end.
A natural case study

Inductive nat := 0 : nat | S : nat → nat.

Must be a **positive** functor! ⇝ Syntactic "positivity condition"

Definition nat_rect :
∀ (P : nat → Type)
   (p0 : P 0) (pS : ∀ n, P n → P (S n)) n, P n :=
   fun P p0 pS ⇒
   fix F n := match n with
   | O ⇒ p0
   | S m ⇒ pS m (F m)
   end.

Must be a **well-founded** recursion! ⇝ Syntactic "guard condition"

Recursive calls must be “smaller”.
An aftertought on dynamics

To ensure strong normalization, one must restrict \texttt{fix} reduction.

\[
\text{(fix } F \ n := t \text{) } 0 \ \rightarrow \ \text{(fun } n \Rightarrow t[F := (\text{fix } F \ n := t)]\text{) } 0
\]

\[
\text{(fix } F \ n := t \text{) } (S \ m) \ \rightarrow \ \text{(fun } n \Rightarrow t[F := (\text{fix } F \ n := t)]\text{) } (S \ m)
\]
To ensure strong normalization, one must restrict fix reduction.

\[
(fix \ F \ n := t) \ 0 \ \rightarrow \ \ (fun \ n \Rightarrow t[F := (fix \ F \ n := t)]) \ 0
\]

\[
(fix \ F \ n := t) \ (S \ m) \ \rightarrow \ \ (fun \ n \Rightarrow t[F := (fix \ F \ n := t)]) \ (S \ m)
\]

\[
(fix \ F \ n := t) \ r \ \not\rightarrow
\]

when \( r \) is not an applied constructor.

Otherwise infinite loop due to strong reduction...
Coinduction was introduced by Eduardo Giménez (1994).

- Handling infinite datastructures as greatest fixpoints
- Kind of dual of inductive datatypes
  - Inductive objects are to be destructed
    
    \[
    \text{induction: } (FS \to S) \to \mu X. FX \to S
    \]
  
  - Coinductive objects are to be constructed
    
    \[
    \text{coinduction: } (S \to FS) \to S \to \nu X. FX
    \]
Coinduction was introduced by Eduardo Giménez (1994).

- Handling infinite datastructures as greatest fixpoints
- Kind of dual of inductive datatypes
  - Inductive objects are to be destructed
    \[
    \text{induction: } (FS \rightarrow S) \rightarrow \mu X.FX \rightarrow S
    \]
  - Coinductive objects are to be constructed
    \[
    \text{coinduction: } (S \rightarrow FS) \rightarrow S \rightarrow \nu X.FX
    \]
- New term: \text{cofix} \ F \ := \ t \ constructs \ a \ coinductive
- Otherwise use the same constructions as for inductive types
  - Constructors
  - Pattern-matching
“That’s easy!”
What is your favourite coinductive?

CoInductive stream := cons : nat → stream → stream.
What is your favourite coinductive?

CoInductive stream := cons : nat → stream → stream.

Same syntactic “positivity condition” as for inductive types
What is your favourite coinductive?

CoInductive stream := cons : nat → stream → stream.

Same syntactic “positivity condition” as for inductive types

Definition nats : stream :=
  (cofix F := fun n : nat ⇒ cons n (F (S n))) 0.
What is your favourite coinductive?

CoInductive stream := cons : nat → stream → stream.

Same syntactic “positivity condition” as for inductive types

Definition nats : stream :=
  (cofix F := fun n : nat ⇒ cons n (F (S n))) 0.

Must be a anti-founded corecursion! ⇝ Syntactic “guard condition”

- Corecursive calls must be “blocked”.
- Fairness assumption: the cofix must be productive at each unfolding
As for inductive types one must restrict \texttt{cofix} reduction.

- \texttt{fix} was restricted by arguments being constructors
- Dually \texttt{cofix} is restricted by surrounding context:

\[
\text{(cofix } F := t \text{) } \not\rightarrow \text{ } t[F := (\text{cofix } F := t)]]
\]
As for inductive types one must restrict \texttt{cofix} reduction.

- \texttt{fix} was restricted by arguments being constructors
- Dually \texttt{cofix} is restricted by surrounding context:

\[
\text{(cofix } F := t \text{) } \not\rightarrow \text{ } t[F := (\text{cofix } F := t)]
\]

\[
E[\text{cofix } F := t] \rightarrow E[t[F := (\text{cofix } F := t)]]
\]

only when the innermost component of \(E\) is a pattern-matching.
Some Examples

Definition hd (s : stream) : nat :=
    match s with cons n _ ⇒ n end.

Definition tl (s : stream) : stream :=
    match s with cons _ s' ⇒ s' end.

Definition X : stream := (cofix F := cons 1 (cons 2 F)).
Definition \texttt{hd} (s : stream) : nat :=
\hspace{1em} \text{match } s \text{ with } \text{cons } n \_ \Rightarrow n \text{ end.}

Definition \texttt{tl} (s : stream) : stream :=
\hspace{1em} \text{match } s \text{ with } \text{cons } \_ s' \Rightarrow s' \text{ end.}

Definition \texttt{X} : stream := (cofix F := cons 1 (cons 2 F)).

The reduction behaviour forces to write unfolding lemmas.

Lemma \texttt{stream_unfold} : \text{forall } s : \text{stream}, s = \text{cons } (\text{hd } s) (\text{tl } s).

This is provable thanks to the fact \texttt{hd} and \texttt{tl} are pattern-matchings.

\(
\leadsto \) \text{ without such unfoldings, proofs are horrendous (if doable).}
More Examples

Luckily or not, manipulating coinductive objects foregoes equality.

Definition ones : stream := (cofix F := cons 1 F).
Definition onesones : stream := (cofix F := cons 1 (cons 1 F)).

One cannot prove that \( \text{ones} = \text{onesones} \).

... only that they are bisimilar.

\[
\text{CoInductive bisimilar : stream -> stream -> Prop :=}
\]
\[
\text{bisim : forall x s1 s2, bisimilar s1 s2 ->}
\]
\[
\text{bisimilar (cons x s1) (cons x s2).}
\]

Lemma ones_onesones : bisimilar ones onesones.

(By a proof by coinduction.)
More Examples

Luckily or not, manipulating coinductive objects foregoes equality.

Definition ones : stream := (cofix F := cons 1 F).
Definition onesones : stream := (cofix F := cons 1 (cons 1 F)).

One cannot prove that \( \text{ones} = \text{onesones} \).
... only that they are bisimilar.

CoInductive bisimilar : stream -> stream -> Prop :=
  bisim : forall x s1 s2, bisimilar s1 s2 ->
  bisimilar (cons x s1) (cons x s2).

Lemma ones_onesones : bisimilar ones onesones.
(By a proof by coinduction.)
A Theoretical Failure

CoInductive tick := Tick : tick -> tick.
CoFixpoint loop := Tick loop.
Definition etaeq : loop = loop :=
match loop with
| Tick t ⇒ eq_refl (Tick t)
end.

Definition BOOM := Eval compute in (etaeq : loop = loop).
Error: Found type Tick loop = Tick loop
but expected type loop = loop.

Failure of subject reduction
(a serious matter)
CoInductive tick := Tick : tick -> tick.
CoFixpoint loop := Tick loop.
Definition etaeq : loop = loop :=
match loop with
| Tick t ⇒ eq_refl (Tick t)
end.

Definition BOOM := Eval compute in (etaeq : loop = loop).
A Theoretical Failure

CoInductive tick := Tick : tick -> tick.
CoFixpoint loop := Tick loop.
Definition etaeq : loop = loop :=
match loop with
| Tick t ⇒ eq_refl (Tick t)
end.

Definition BOOM := Eval compute in (etaeq : loop = loop).

Error: Found type Tick loop = Tick loop
but expected type loop = loop.
A Theoretical Failure

CoInductive tick := Tick : tick -> tick.
CoFixpoint loop := Tick loop.
Definition etaeq : loop = loop :=
match loop with
| Tick t ⇒ eq_refl (Tick t)
end.

Definition BOOM := Eval compute in (etaeq : loop = loop).

Error: Found type Tick loop = Tick loop
but expected type loop = loop.

Failure of subject reduction
(a serious matter)
“I didn’t know that.”
Analysis of the failure

The problem stems from the use of pattern-matching in etaeq.

- The reduction rule allows for more precise information about `loop`.
- The dependency of the matching allows this information to escape.
- Reducing the matching loses this information.

```coq
Definition etaeq : loop = loop :=
match loop with
| Tick t ⇒ eq_refl (Tick t)
end.
```

```
etaeq → eq_refl (Tick loop)
```
Analysis of the failure

The problem stems from the use of pattern-matching in etaeq.

- The reduction rule allows for more precise information about loop
- The dependency of the matching allows this information to escape
- Reducing the matching loses this information

Definition etaeq : loop = loop :=
match loop with
| Tick t ⇒ eq_refl (Tick t)
end.

etaeq → eq_refl (Tick loop)

Dependent pattern-matching on coinductive types is evil.
(we’re doing it wrong)
A More Practical Issue

The current handling of guardedness is also problematic in practice.

- Inductive proofs allowed by an induction principle
  - abstract over the guard condition
  - forces at least one step of the fixpoint
  - modularizes proofs: the induction tactic
A More Practical Issue

The current handling of guardedness is also problematic in practice.

- **Inductive proofs allowed by an induction principle**
  - abstract over the guard condition
  - forces at least one step of the fixpoint
  - modularizes proofs: the induction tactic

- **No such abstraction for coinductive proofs**
  - cofixpoints are built by hand (the infamous cofix tactic)
  - steps must be provided as syntactic constructors
  - cannot abstract over them (no functions, no opaque terms)
  - thus no granularity

In theory: no problem.
In practice: Really, really painful.
A More Practical Issue

The current handling of guardedness is also problematic in practice.

- Inductive proofs allowed by an induction principle
  - abstract over the guard condition
  - forces at least one step of the fixpoint
  - modularizes proofs: the induction tactic

- No such abstraction for coinductive proofs
  - cofixpoints are built by hand (the infamous cofix tactic)
  - steps must be provided as syntactic constructors
  - cannot abstract over them (no functions, no opaque terms)
  - thus no granularity

One cannot chain coinductive lemmas in proofs.
Everything must be done in one go.
A More Practical Issue

The current handling of guardedness is also problematic in practice.

- Inductive proofs allowed by an induction principle
  - abstract over the guard condition
  - forces at least one step of the fixpoint
  - modularizes proofs: the induction tactic

- No such abstraction for coinductive proofs
  - cofixpoints are built by hand (the infamous cofix tactic)
  - steps must be provided as syntactic constructors
  - cannot abstract over them (no functions, no opaque terms)
  - thus no granularity

One cannot chain coinductive lemmas in proofs.
Everything must be done in one go.

In theory: no problem.
In practice: Really, really painful.
The failure of subject reduction is due to a misinterpretation.

“Coinductives are like inductives, save that they’re greatest fixpoints.”
The failure of subject reduction is due to a misinterpretation.

“Coinductives are like inductives, save that they’re greatest fixpoints.”

* ¡No! *
The failure of subject reduction is due to a misinterpretation.

“Coinductives are like inductives, save that they’re greatest fixpoints.”

* ¡No! *

+ Inductives lie on the positive side: $\bigoplus, \bigotimes, \mu$
  - Built out of constructors
  - Destructed by fixpoint + pattern-matching
  - Normal inhabitants have a constrained form

− Coinductives lie on the negative side: $\&$, $\forall$, $\nu$
  - Built out of cofixpoints + records
  - Destructed by projections
  - Normal inhabitants can be about anything
Matthieu Sozeau introduced in Coq 8.5 the so-called primitive projections.

- Records defined by projection rather than pattern-matching
  - True negative products
  - Projections are first-class terms
- Originally for efficiency and semi-theoretical ($\eta$-equivalence) purposes
- Happens to solve the subject reduction issue
  - “Copattern”-style coinduction
  - Can only observe projections, not the object itself
  - When Coq turns Object-Oriented?
CoInductive stream := { hd : nat; tl : stream }.
Definition cons n s := { hd := n; tl := s }.

Definition nats :=
  (cofix F := fun n => { hd := n; tl := F (S n) } ) 0.

Definition ones := cofix F := { hd := 1; tl := F }.
CoFixpoint ones2 :=
  cofix F := { hd := 1; tl := { hd := 1; tl := F } }.
Drastic Changes

- Positivity condition is similar
- Guardedness is adapted as:

  \( \text{Corecursive calls must be under a record field} \)

- Reduction is adapted as:

  \[
  (\text{cofix } F := t).p \rightarrow (t[F := (\text{cofix } F := t)]).p
  \]

- Bisimilarity becomes essential
  \( \leadsto \) One cannot prove anymore that \( s = \text{cons (hd } s \text{) (tl } s \text{)} \)
  \( \leadsto \) Equality over coinductives becomes trivial
  \( \leadsto \) Coinductives as blackboxes

- The problematic example is not writable anymore
  \( \leadsto \) Less equality for a safer world!
A solution to the practical problem

The cofixpoint abstraction problem can be worked around.

- The notions of “positive” or “being productive” are too syntactical.
- Let’s make them semantical!
A solution to the practical problem

The cofixpoint abstraction problem can be worked around.

- The notions of “positive” or “being productive” are too syntactical.
- Let’s make them semantical!

We will use Mendler-style coinduction.

- A program translation making everything positive for free
- Works by expliciting the inner state of the inductive
- A technique successfully used by the Paco library (Chung-Kil Hur)
- “Open recursion approach”
Let $H$ be the complete lattice of propositions.
Let $F : H \to H$ be some function and pose

$$\lceil F \rceil : (H \to H) \to H \to H$$

$$:= \lambda G X. \exists Y. (FY) \land (Y \to X \lor GX)$$

**Theorem**

- $\lceil F \rceil$ is syntactically positive in $G$.
- In particular $\nu \lceil F \rceil : H \to H$ exists.
- Moreover, if $F$ is monotone, $\nu \lceil F \rceil (Y) \equiv \nu X. F(X \lor Y)$.
- In particular $\nu \lceil F \rceil (\bot) \equiv \nu F$.

Here $\nu \lceil F \rceil$ acts as “$\nu F$ with explicit inner state”.
By applying the previous results, one gets for free three principles.

(Init) $\nu F \equiv \nu [F](\bot)$

(Unfold) $\nu[F](X) \equiv F(X \lor \nu[F](X))$

(Coiter) $(Y \rightarrow \nu[F](X)) \equiv (Y \rightarrow \nu[F](X \lor Y))$
The Mendlerified Example

CoInductive stream :=
    cons : nat -> stream -> stream.

CoInductive stream (R : Type) :=
    cons : nat -> (R + stream R) -> stream R.

Definition coiter :
    forall L R, (L -> stream (L + R)) -> L -> stream R.

The corecursion combinator allows for cofix-free reasoning.

Definition nats : stream False :=
    coiter (fun n => cons n (inr (inl (S n)))) 0.

Definition ones : stream False :=
    coiter (fun _ => cons 0 (inr (inl tt))) tt.

Definition ones2 : stream False :=
    coiter (fun _ => cons 0 (inl (cons 0 (inr (inl tt))))) tt.
Coinduction is a bit tricky in Coq
... but things are getting better
... and we have kludges to work around its defects
Thanks for your attention.