A Survey of Coinduction in Coq

Pierre-Marie Pédrot

PPS/πr^2

18 June 2015

Plan

Coq

- 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

Coq

- 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 λ-terms (with rich typing)
- The only type former was $\Pi x : A.B$
- Poor expressivity as a logical system: $\nvdash 0 \neq 1$

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: `match` t `with` $\vec{p} \Rightarrow \vec{u}$ `end`
  - Fixpoint: `fix` $F\ n := t$
- New fantasmabulous theorems: $\vdash 0 \neq 1$

A natural case study

```
Inductive nat := O : nat | S : nat → nat.
```

A natural case study

`Inductive nat := O : nat | S : nat → nat.`

Must be a positive functor! $\rightsquigarrow$ Syntactic "positivity condition"

A natural case study

```
Inductive nat := O : nat | S : nat → nat.
```

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

```
Definition nat_rect :
  ∀ (P : nat → Type)
    (p0 : P O) (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.
```

A natural case study

```
Inductive nat := O : nat | S : nat → nat.
```

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

```
Definition nat_rect :
  ∀ (P : nat → Type)
    (p0 : P O) (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 `fix` reduction.

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

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

An aftertought on dynamics

To ensure strong normalization, one must restrict `fix` reduction.

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

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

$$\texttt{(fix F n := t) r} \quad \not\rightarrow$$

when `r` is not an applied constructor.

Otherwise infinite loop due to strong reduction...

Enters Coinduction

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

$$\text{induction:} \quad (FS \to S) \to \mu X.FX \to S$$

  - Coinductive objects are to be constructed

$$\text{coinduction:} \quad (S \to FS) \to S \to \nu X.FX$$

Enters Coinduction

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

$$\text{induction:} \quad (FS \to S) \to \mu X.FX \to S$$

  - Coinductive objects are to be constructed

$$\text{coinduction:} \quad (S \to FS) \to S \to \nu X.FX$$

- New term: `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

To infinity and beyond

As for inductive types one must restrict `cofix` reduction.

- `fix` was restricted by arguments being constructors
- Dually `cofix` is restricted by surrounding context:

$$\texttt{(cofix F := t)} \not\longrightarrow \texttt{t[F := (cofix F := t)]}$$

To infinity and beyond

As for inductive types one must restrict `cofix` reduction.

- `fix` was restricted by arguments being constructors
- Dually `cofix` is restricted by surrounding context:

$$\texttt{(cofix F := t)} \;\not\longrightarrow\; \texttt{t[F := (cofix F := t)]}$$

$$E\texttt{[cofix F := t]} \longrightarrow E\texttt{[t[F := (cofix F := t)]]}$$

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)).
```

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)).
```

The reduction behaviour forces to write unfolding lemmas.

```
Lemma stream_unfold : forall s : stream, s = cons (hd s) (tl s).
```

This is provable thanks to the fact `hd` and `tl` are pattern-matchings.

$\rightsquigarrow$ 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 `ones = onesones`.

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 `ones = 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.
```

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).
```

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

```
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` $\longrightarrow$ `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

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.

Through the Looking Glass

The failure of subject reduction is due to a misinterpretation.

"Coinductives are like inductives, save that they're greatest fixpoints."

Through the Looking Glass

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: $\oplus, \otimes, \mu$

- Built out of constructors
- Destructed by fixpoint + pattern-matching
- Normal inhabitants have a constrained form

$-$ Coinductives lie on the negative side: $\&, \parr, \nu$

- Built out of cofixpoints + records
- Destructed by projections
- Normal inhabitants can be about anything

A Solution to the Subject Reduction Issue

Matthieu Sozeau introduced in Coq 8.5 the so-called primitive projections.

- Records defined by projection rather than pattern-matching
  - $\rightsquigarrow$ True negative products
  - $\rightsquigarrow$ Projections are first-class terms
- Originally for efficiency and semi-theoretical (η-equivalence) purposes
- Happens to solve the subject reduction issue
  - $\rightsquigarrow$ "Copattern"-style coinduction
  - $\rightsquigarrow$ Can only observe projections, not the object itself
  - $\rightsquigarrow$ When Coq turns Object-Oriented?

Revisiting the Example

```
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:

  Corecursive calls must be under a record field

- Reduction is adapted as:

  `(cofix F := t).p` $\longrightarrow$ `(t[F := (cofix F := t)]).p`

- Bisimilarity becomes essential
  - $\rightsquigarrow$ One cannot prove anymore that `s = cons (hd s) (tl s)`
  - $\rightsquigarrow$ Equality over coinductives becomes trivial
  - $\rightsquigarrow$ Coinductives as blackboxes
- The problematic example is not writable anymore
  - $\rightsquigarrow$ 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”

The Underlying Mathematical Justification

Let $\mathbf{H}$ be the complete lattice of propositions.
Let $F : \mathbf{H} \to \mathbf{H}$ be some function and pose

$$\begin{array}{rcl} \lceil F \rceil & : & (\mathbf{H} \to \mathbf{H}) \to \mathbf{H} \to \mathbf{H} \\ & := & \lambda G X.\, \exists Y.\, (F\, Y) \wedge (Y \to X \vee G\, X) \end{array}$$

Theorem

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

Here $\nu \lceil F \rceil$ acts as "νF with explicit inner state".

By applying the previous results, one gets for free three principles.

$$\begin{array}{rl}
\textbf{(Init)} & \nu F \equiv \nu\lceil F\rceil(\bot) \\
\textbf{(Unfold)} & \nu\lceil F\rceil(X) \equiv F(X \vee \nu\lceil F\rceil(X)) \\
\textbf{(Coiter)} & (Y \to \nu\lceil F\rceil(X)) \equiv (Y \to \nu\lceil F\rceil(X \vee Y))
\end{array}$$

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.
```

Conclusion

- 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.