Coinductive definitions and proofs in type theory and Coq

Herman Geuvers

Radboud University Nijmegen
Thanks to: Freek Wiedijk, Robbert Krebbers, Eelis van der Weegen

Representing Streams II
Lorentz Centre Leiden
January 30, 2014

Dick de Bruijn

Dick de Bruijn

Special volume of *Indagationes Mathematicae*
Volume 24, Issue 4, Pages 647-1120 (15 November 2013)
In memory of N.G. (Dick) de Bruijn (1918 –2012)

Present state of affairs: Is formalising proofs as simple as using LaTeX?

Present state of affairs: Is formalising proofs as simple as using LaTeX?

As a kind of dream I played (in 1968) with the idea of a future where every mathematician would have a machine on his desk, all for himself, on which he would write mathematics and which would verify his work. But, by lack of experience in such matters, I expected that such machines would be available within 5 years from then. But now, 23 years later, we are not that far yet. Anyway I expected in 1968 that the memory capacity of main frame computers would grow rapidly in the next few years, but that was a deception too. Implementing Automath on the quite advanced computers available to us in the years 1970–1975 was to a large extent a struggle for living with the limitations of fast accessible memory.

Present state of affairs: Big formalisations

- Proof of the Odd Order Theorem (Walter Feit and John G. Thompson), completely machine-checked using Coq. Mathematical Components team lead by Georges Gonthier (MSR Cambridge) at the Inria Microsoft Research Joint Centre.
- Flyspeck Project to produce a formal proof of the Kepler Conjecture. Thomas Hales et al. in HOL-light.

The 100 greatest theorems, 88 formalised (Freek Wiedijk)

1.	The Irrationality of the Square Root of 2	≥ 17
2.	Fundamental Theorem of Algebra	4
3.	The Denumerability of the Rational Numbers	6
4.	Pythagorean Theorem	6
5.	Prime Number Theorem	2
6.	Gödel's Incompleteness Theorem	4
7.	Law of Quadratic Reciprocity	4
8.	The Impossibility of Trisecting the Angle and Doubling the Cube	2
9.	The Area of a Circle	4
10.	Euler's Generalization of Fermat's Little Theorem	4
11.	The Infinitude of Primes	7
12.	The Independence of the Parallel Postulate	2
13.	Polyhedron Formula	3

...

The best proof assistants . . .

Five systems seriously used for mathematics

HOL Light	86
ProofPower	42
Isabelle	49
Coq	49
Mizar	57

What is Coq?

- A combination of a proof assistant and a (functional) programming language
- Aims:
  - define arbitrary mathematical notions and prove theorems about them.
  - write programs and prove their properties
  - execute programs inside Coq (slow, but works)
  - extract programs to fast program code (in Haskell, Ocaml)

What is Coq?

- A combination of a proof assistant and a (functional) programming language
- Aims:
  - define arbitrary mathematical notions and prove theorems about them.
  - write programs and prove their properties
  - execute programs inside Coq (slow, but works)
  - extract programs to fast program code (in Haskell, Ocaml)
- Many impressive mathematical results have been formalised in Coq: Fundamental Theorem of Algebra, Fundamental Theorem of Calculus, 4 color theorem, Feit-Thompson theorem,
- efficient certified algorithms have been extracted form proofs in Coq, e.g. Buchberger's algorithm

Propositions-as-Types

- A constructive proof of a formula is itself a program
- Propositions are Types
- Proofs are Terms

Propositions-as-Types

- A constructive proof of a formula is itself a program
- Propositions are Types
- Proofs are Terms
- PAT, or in a modern setting iPAT (interpretation of P-as-T)

Propositions-as-Types

- A constructive proof of a formula is itself a program
- Propositions are Types
- Proofs are Terms
- PAT, or in a modern setting iPAT (interpretation of P-as-T)

$$M : A$$

Has two readings:

- A is a type, and M is a program (data) of type A.
- A is a proposition, and M is a proof of A.

Interactive proving in Coq

Given

```
A : Type.
P : A -> Prop.
```

We can try to prove

```
Lemma simple: forall x : A, P x -> P x.
```

Here, `simple` is the name of the proof of the Lemma, or in type theoretic terminology: the name of the proof-term we will construct.

Interactive proving in Coq

Given

```
A : Type.
P : A -> Prop.
```

We can try to prove

```
Lemma simple: forall x : A, P x -> P x.
```

Here, `simple` is the name of the proof of the Lemma, or in type theoretic terminology: the name of the proof-term we will construct.

```
simple = fun (x : A) (H : P x) => H
     : forall x : A, P x -> P x
```

Inductive types in Coq

One can define one's own inductive types

```
Inductive nat : Type :=
 | O : nat
 | S : nat -> nat
```

Inductive types in Coq

One can define one's own inductive types

```
Inductive nat : Type :=
 | O : nat
 | S : nat -> nat
```

This yields

- a type nat and terms O : nat and S : nat $\to$ nat
- a function definition principle (structural recursion)
- a proof principle (induction)

Inductive types in Coq

One can define one's own inductive types

```
Inductive nat : Type :=
 | O : nat
 | S : nat -> nat
```

This yields

- a type nat and terms O : nat and S : nat $\to$ nat
- a function definition principle (structural recursion)
- a proof principle (induction)

```
Fixpoint plus (n m : nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end.
```

Inductive types in Coq

```
Inductive nat : Type :=
 | O : nat
 | S : nat -> nat
```

This yields

- a type nat and terms O : nat and S : nat $\rightarrow$ nat
- a function definition principle (structural recursion)
- a proof principle (induction)

```
nat_ind
      : forall P : nat -> Prop,
        P O -> (forall n : nat, P n -> P (S n))
        -> forall n : nat, P n
```

Functions compute in Coq

```
Fixpoint plus (n m : nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end.
```

the function `plus` is a program that we can compute with

```
plus (S (S O)) (S O)
```

reduces to

```
plus (S (S (S O)))
```

For Coq these terms are *indistinguishable*.

```
Variable n : nat.
Eval compute in plus (S (S O)) n.
```

This yields (reduces to) `S (S n)`
Reduction in Coq always terminates! (All definable programs in

Coinductive data types

Coinductive definitions look very much like inductive definitions:

```
CoInductive Stream (T: Type): Type :=
  Cons: T -> Stream T -> Stream T.
```

`Cons` is the *constructor*.

Coinductive data types

Coinductive definitions look very much like inductive definitions:

```
CoInductive Stream (T: Type): Type :=
  Cons: T -> Stream T -> Stream T.
```

`Cons` is the *constructor*.
But the rules for what is a well-formed object of a coinductive type are different

- Terms of a coinductive type can represent infinite objects.
- These terms are evaluated lazily.
- Well-definedness is guaranteed via guardedness.

Definition of streams

```
CoInductive Stream (T: Type): Type :=
  Cons: T -> Stream T -> Stream T.
```

How to define

```
ones = 1 :: ones
```

with `ones : Stream nat`

Definition of streams

```
CoInductive Stream (T: Type): Type :=
  Cons: T -> Stream T -> Stream T.
```

How to define

```
ones = 1 :: ones
```

with `ones : Stream nat`

```
CoFixpoint ones : Stream nat :=
  Cons 1 ones.
```

The recursive call to `ones` is guarded by the constructor `Cons`.
The term `ones` does not reduce to `Cons 1 ones`.

Productivity

```
CoFixpoint ones: Stream nat :=
  Cons 1 ones.
```

The definition of `ones` is productive: it is equal to an expression of the form

$$a_0 :: a_2 \ldots :: a_n :: \texttt{ones}$$

with $n > 0$.

Productivity

```
CoFixpoint ones: Stream nat :=
  Cons 1 ones.
```

The definition of `ones` is productive: it is equal to an expression of the form

$$a_0 :: a_2 \ldots :: a_n :: \texttt{ones}$$

with $n > 0$.
In Coq, productivity is guaranteed via the syntactic criterion of guardedness.

Productivity

```
CoFixpoint ones: Stream nat :=
  Cons 1 ones.
```

The definition of ones is productive: it is equal to an expression of the form

$$a_0 :: a_2 \ldots :: a_n :: \texttt{ones}$$

with $n > 0$.
In Coq, productivity is guaranteed via the syntactic criterion of guardedness.

```
CoFixpoint simple : Stream nat := simple.
```

is not accepted by Coq.

Guardedness

Productivity is not decidable, but guardedness is.
Coq only allows corecursive definitions that are guarded.

Guardedness

Productivity is not decidable, but guardedness is.
Coq only allows corecursive definitions that are guarded.
For streams this means that a CoFixpoint definition of s should look like this:

$$\texttt{CoFixpoint}\ s\ \overrightarrow{x} := a_0 :: a_1 :: \ldots : a_n : s\,\overrightarrow{q} \quad \text{with} \quad n > 1$$

Computation

A CoFixpoint definition $\texttt{CoFixpoint}\, s\ \overrightarrow{x} := \texttt{Cons}\, a\, (s\, \overrightarrow{q})$ does *not* reduce:

$$s\ \overrightarrow{x} \not\longrightarrow \texttt{Cons}\, a\, (s\, \overrightarrow{q})$$

Computation

A CoFixpoint definition $\texttt{CoFixpoint}\, s\ \overrightarrow{x} := \texttt{Cons}\, a\, (s\, \overrightarrow{q})$ does *not* reduce:

$$s\ \overrightarrow{x} \not\longrightarrow \texttt{Cons}\, a\, (s\, \overrightarrow{q})$$

But it does "under a match":

$$\begin{array}{l} \texttt{match}\, s\ \overrightarrow{p} \qquad\quad \texttt{with} \\ \qquad\quad \texttt{Cons}\, y\, z \quad \Rightarrow \quad E(y, z) \\ \texttt{end} \\ \qquad\qquad\qquad\quad \longrightarrow \\ E(a, s\, \overrightarrow{q}) \end{array}$$

Destructors on streams

```
Definition hd (s:Stream) := match s with
                            | Cons a t => a
                            end.

Definition tl (s:Stream) := match s with
                            | Cons a t => t
                            end.
```

Destructors on streams

```
Definition hd (s:Stream) := match s with
                            | Cons a t => a
                            end.

Definition tl (s:Stream) := match s with
                            | Cons a t => t
                            end.
```

Now
`hd (Cons a t)` reduces to `a`
`tl (Cons a t)` reduces to `t`

Filtering

The following definition is not accepted by Coq

```
CoFixpoint filter (p:A->bool)(s:Stream A) : Stream A :=
  match s with
     Cons a t => if (p a)
                 then Cons a (filter p t)
                 else (filter p t)
  end.
```

The second occurrence of `filter` is not guarded by a `Cons`.

Zipping

The following definition is accepted by Coq

```
CoFixpoint zip (s t : Stream A) :=
 Cons (hd s) (zip t (tl s)).
```

Relating streams and functions

There is an isomorphism between `Stream A` and `nat -> A`.

```
CoFixpoint F (f:nat->A) : Stream A :=
  Cons (f 0)(F (fun n:nat => f (S n))).
```

This defines

```
F : (nat -> A) -> Stream A
```

by

$$F(f) := f(0) :: F(\lambda n.f(n+1))$$

which is correct, because F is guarded by the constructor.

Relating streams and functions

We compute the nth tail and the nth element of a stream.

```
Fixpoint nth_tl (n:nat) (s:Stream A) : Stream A:=
  match n with
  | O => s
  | S m => nth_tl m (tl s)
  end.

Definition nth (n:nat) (s:Stream A) : A :=
            hd (nth_tl n s).


Definition G (s :Stream A)(n:nat): A :=
            nth n s.
```

This just defines `G :Stream A -> (nat -> A)` by

$$G(s) := \lambda n.\mathrm{hd}(\mathrm{tl}^n(s))$$

Relating streams and functions

We can prove for all $f : \mathbb{N} \to A$, by induction on n

$$\forall n \, [\, G(F(f))\,(n) = f(n) \,]$$

So, f and $G(F(f))$ are extensionally equal (as functions).

Relating streams and functions

We can prove for all $f : \mathbb{N} \to A$, by induction on n

$$\forall n \, [\, G(F(f))\,(n) = f(n) \,]$$

So, f and $G(F(f))$ are extensionally equal (as functions).
How to prove

$$F(G(s)) = s??$$

Relating streams and functions

We can prove for all $f : \mathbb{N} \to A$, by induction on n

$$\forall n \, [\, G(F(f))\,(n) = f(n) \,]$$

So, f and $G(F(f))$ are extensionally equal (as functions).
How to prove

$$F(G(s)) = s??$$

What is the equality on streams?

Equality on streams

```
CoInductive EqSt (s1 s2: Stream) : Prop :=
eqst :
 hd s1 = hd s2 ->
      EqSt (tl s1) (tl s2)
                -> EqSt s1 s2.
```

This is the coinductively defined equality on streams.

Equality on streams

```
CoInductive EqSt (s1 s2: Stream) : Prop :=
eqst :
 hd s1 = hd s2 ->
      EqSt (tl s1) (tl s2)
                -> EqSt s1 s2.
```

This is the coinductively defined equality on streams.
A proof of `EqSt s1 s2` must have the shape

```
CoFixpoint eq_proof (...) := eqst p eq_proof (...)
```

so the definition of `eq_proof` must be guarded by `eqst`.

Examples of Equality on streams

Equality is reflexive:

```
CoFixpoint reflex (s : Stream) : EqSt s s :=
 eqst s s (refl_equal (hd s)) (reflex (tl s))

  : forall s : Stream, EqSt s s
```

Examples of Equality on streams

Equality is reflexive:

```
CoFixpoint reflex (s : Stream) : EqSt s s :=
 eqst s s (refl_equal (hd s)) (reflex (tl s))

  : forall s : Stream, EqSt s s
```

Note that in the definition of `reflex`, we use `reflex` again, but it is guarded by `eqst`, the constructor of `EqSt`.

Relating streams and functions

```
CoInductive EqSt (s1 s2: Stream) : Prop :=
eqst :
 hd s1 = hd s2 ->
    EqSt (tl s1) (tl s2) -> EqSt s1 s2.
```

We prove $F(G(s)) = s$ by

```
Lemma ident_on_str : forall s, EqSt (F (G s)) s.
Proof with auto.
cofix ident_on_str.
destruct s.
simpl.
apply identity_on_str.
Qed.
```

Where do all these rules come from?

Type theorists get their constructions of types and terms from

- constructive logic
- functional programming
- category theory / categorical logic

Where do all these rules come from?

Type theorists get their constructions of types and terms from

- constructive logic
- functional programming
- category theory / categorical logic

For every type construction, there are four questions:

1. What is the rule for forming the type?
2. How to construct terms of that type? (introduction rule, constructors)
3. What to do with a term of that type? (elimination rule, destructors)
4. What is the reduction rule? (destructor $\circ$ constructor)

Where do all these rules come from?

Type theorists get their constructions of types and terms from

- constructive logic
- functional programming
- category theory / categorical logic

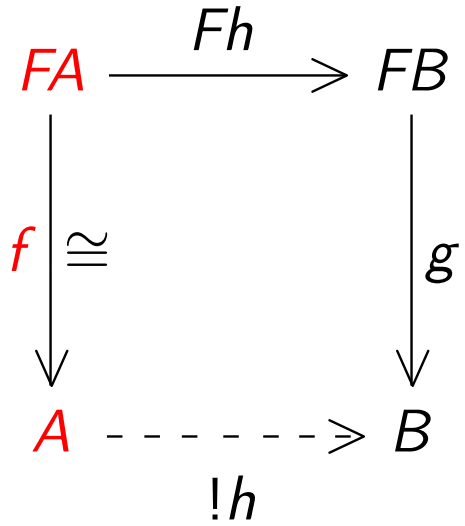
For every type construction, there are four questions:

1. What is the rule for forming the type?
2. How to construct terms of that type? (introduction rule, constructors)
3. What to do with a term of that type? (elimination rule, destructors)
4. What is the reduction rule? (destructor $\circ$ constructor)

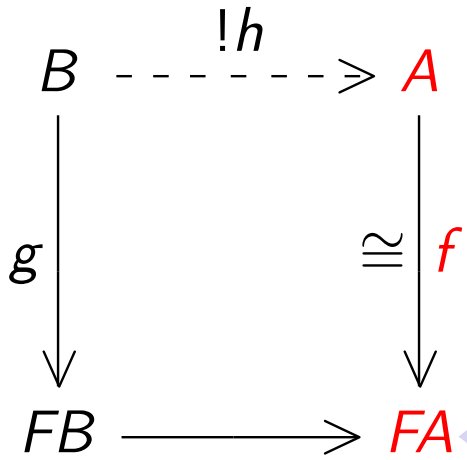
Inductive types $\rightarrow$ Initial algebras
Co-inductive types $\rightarrow$ Final co-algebras

Algebras and Coalgebras

Initial F-algebra: (A, f) s.t. $\forall (B, g)$, $\exists! h$ s.t. the diagram commutes:

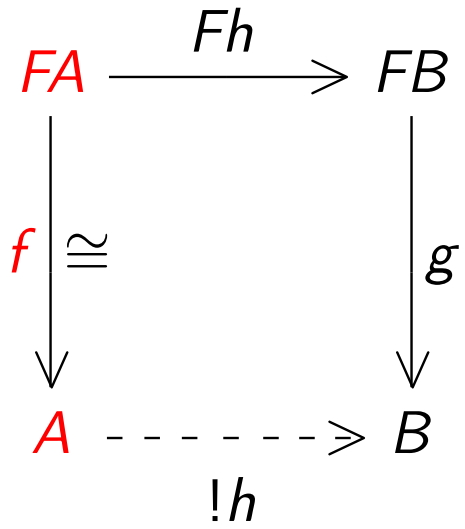


Final F-coalgebra: (A, f) s.t. $\forall (B, g)$, $\exists! h$ s.t. the diagram commutes:



Algebras

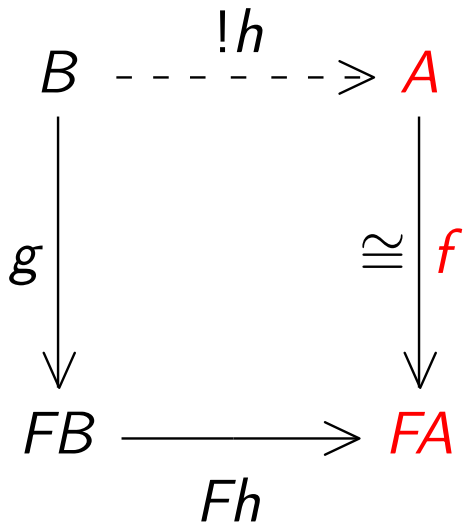
Initial F-algebra:



- constructor function, f to A, is basic
- function definition principle: recursion, to define a function h on A.
- proof principle: induction, proving $\forall x \in A \ldots$

Coalgebras

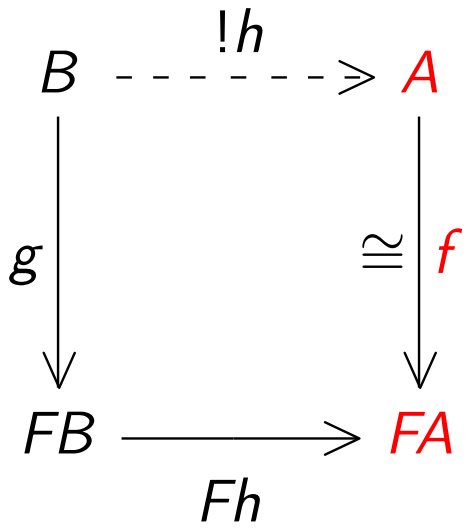
Final F-coalgebra:



- destructor function, f on A, is basic
- function definition principle: corecursion, to define a function h to A.
- proof principle: coinduction, proving equality of elements of A using bisimulation

Coalgebras

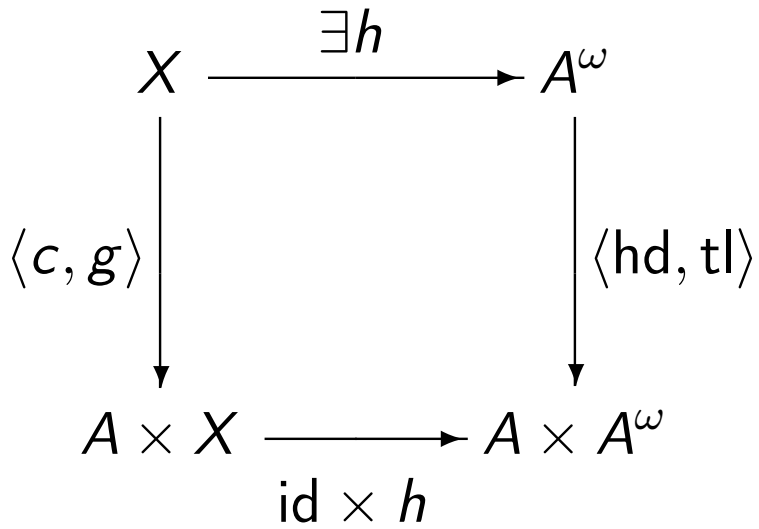
Final F-coalgebra:



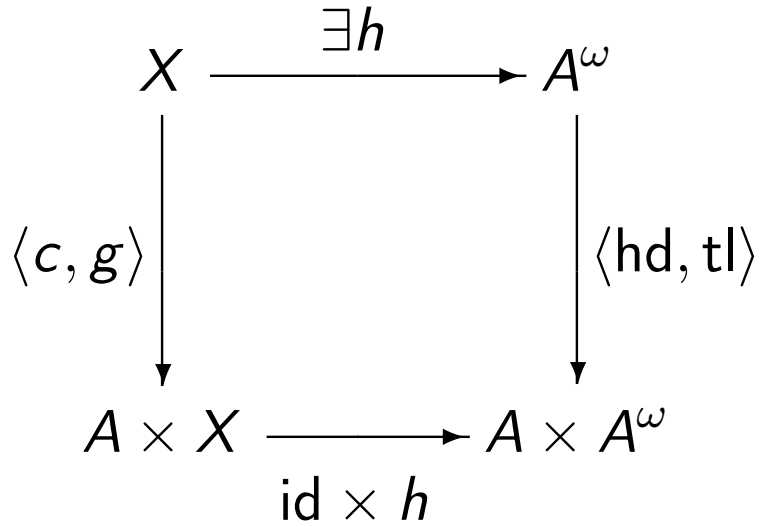
- destructor function, f on A, is basic
- function definition principle: corecursion, to define a function h to A.
- proof principle: coinduction, proving equality of elements of A using bisimulation

In Coq this seems all very different ...
Can we reconcile this??

Final Coalgebras in Coq



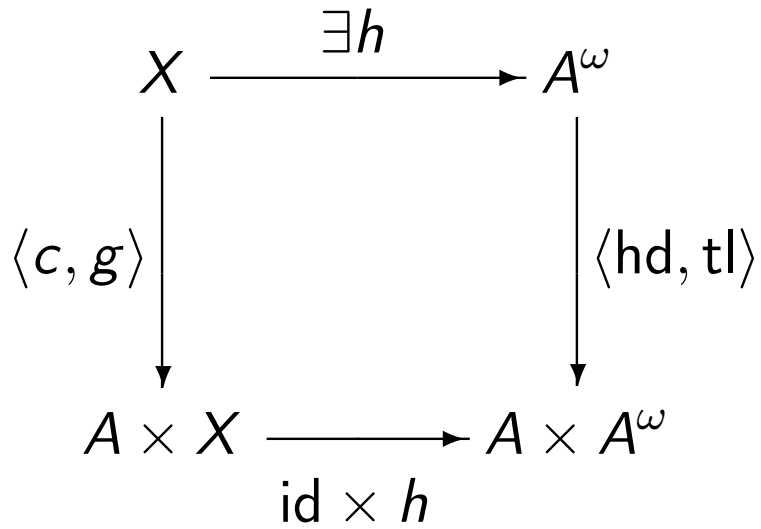
Final Coalgebras in Coq



For $c : X \to A$, $g : X \to X$, there is an $h : X \to A^\omega$ such that

$$\forall x : X, \mathsf{hd}(f(x)) = c(x)$$
$$\forall x : X, \mathsf{tl}(f(x)) = f(g(x))$$

Final Coalgebras in Coq

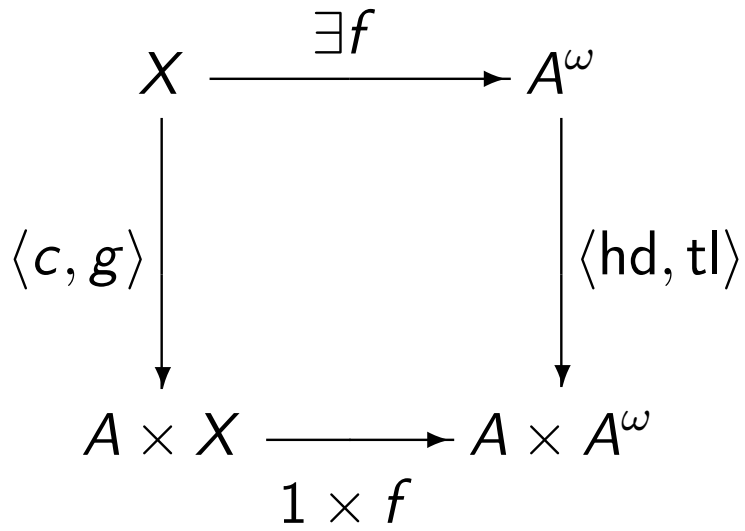


For $c : X \to A$, $g : X \to X$, there is an $h : X \to A^\omega$ such that

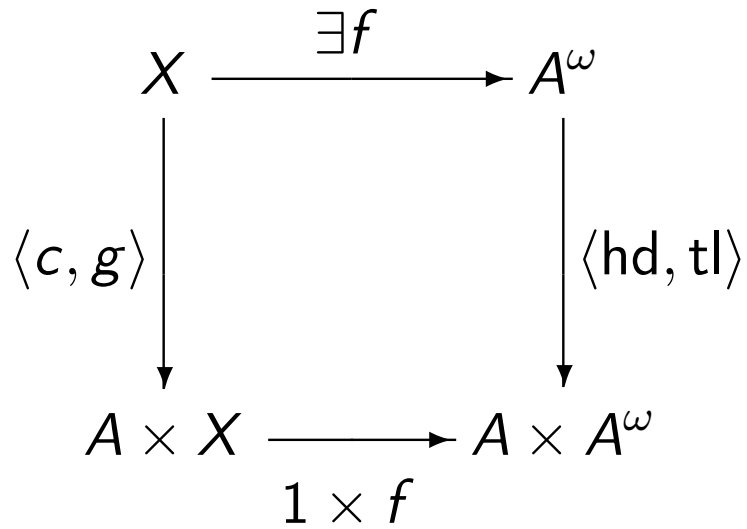
$$\forall x : X, \mathsf{hd}(f(x)) = c(x)$$
$$\forall x : X, \mathsf{tl}(f(x)) = f(g(x))$$

We call h the coiteration of $\langle c, g \rangle$, $\mathsf{coit}\, c\, g$.
This can be proven in Coq.

Final Coalgebras in Coq



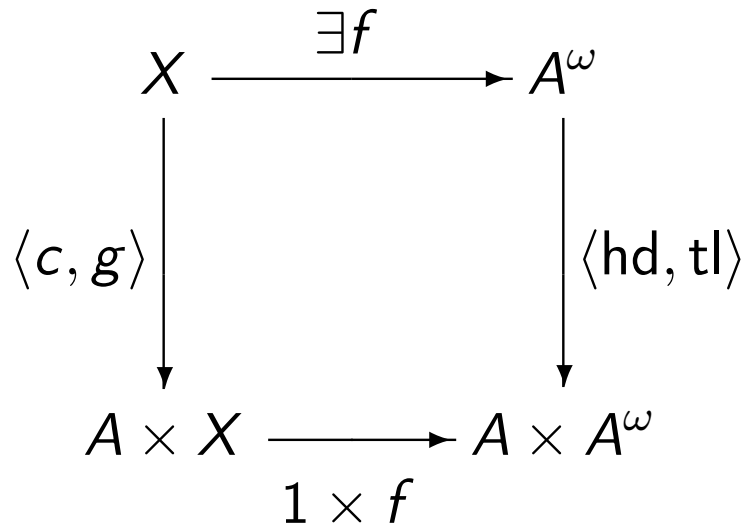
Final Coalgebras in Coq



In final coalgebras: $f := \mathsf{coit}\, c\, g$ is *unique*.

$$\frac{\forall x : X, \mathsf{hd}(f'(x)) = c(x) \quad \forall x : X, \mathsf{tl}(f'(x)) = f'(g(x))}{\forall x : X, f'(x) = \mathsf{coit}\, c\, g(x)}$$

Final Coalgebras in Coq



In final coalgebras: $f := \mathsf{coit}\, c\, g$ is *unique*.

$$\frac{\forall x : X, \mathsf{hd}(f'(x)) = c(x) \quad \forall x : X, \mathsf{tl}(f'(x)) = f'(g(x))}{\forall x : X, f'(x) = \mathsf{coit}\, c\, g(x)}$$

We can prove this in Coq.

Bisimulation and Stream equality in Coq

If two streams are bisimilar, they are stream-equal.

```
Definition bisim (R: Stream A -> Stream A -> Prop) :=
forall s t, R s t ->
    hd s = hd t /\ R(tl s)(tl t).

Lemma bisim_implies_EqSt : forall R s t, bisim R ->
    R s t -> EqSt s t.
```

Bisimulation and Stream equality in Coq

If two streams are bisimilar, they are stream-equal.

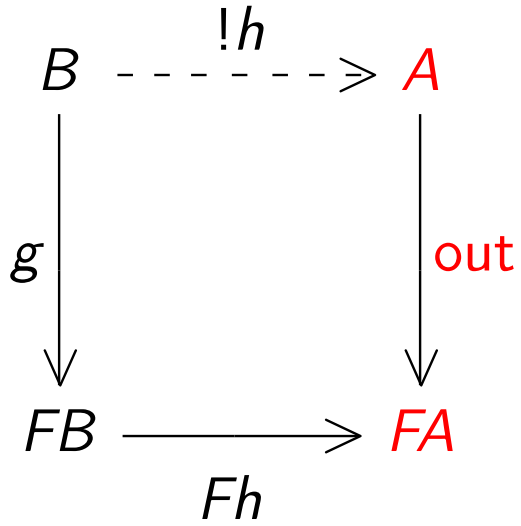
```
Definition bisim (R: Stream A -> Stream A -> Prop) :=
forall s t, R s t ->
    hd s = hd t /\ R(tl s)(tl t).

Lemma bisim_implies_EqSt : forall R s t, bisim R ->
    R s t -> EqSt s t.
```

If two streams are stream-equal, they are bisimilar.

```
Lemma EqSt_implies_bisim :
    forall s t, EqSt s t ->  exists R, bisim R /\ R s t.
```

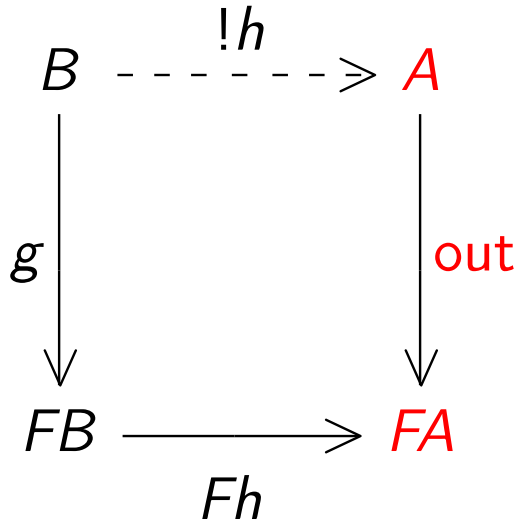
Deriving Coq's coinductive types from final coalgebras?



For a co-inductive type definition, Coq gives the following

- Cons : $F\ A \to A$

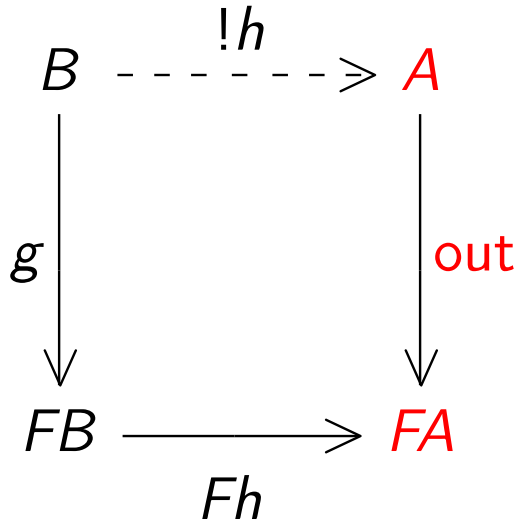
Deriving Coq's coinductive types from final coalgebras?



For a co-inductive type definition, Coq gives the following

- $\mathsf{Cons} : F\ A \to A$
- $\mathsf{out} \circ \mathsf{Cons} = \mathsf{id}$
  $(\mathsf{hd}(\mathsf{Cons}\ a\ s) = a$ and $\mathsf{tl}(\mathsf{Cons}\ a\ s) = s)$.
- $\forall x : A, \exists y : F\ A, x = \mathsf{Cons}\ y$
- A guarded definition principle

Deriving Coq's coinductive types from final coalgebras?

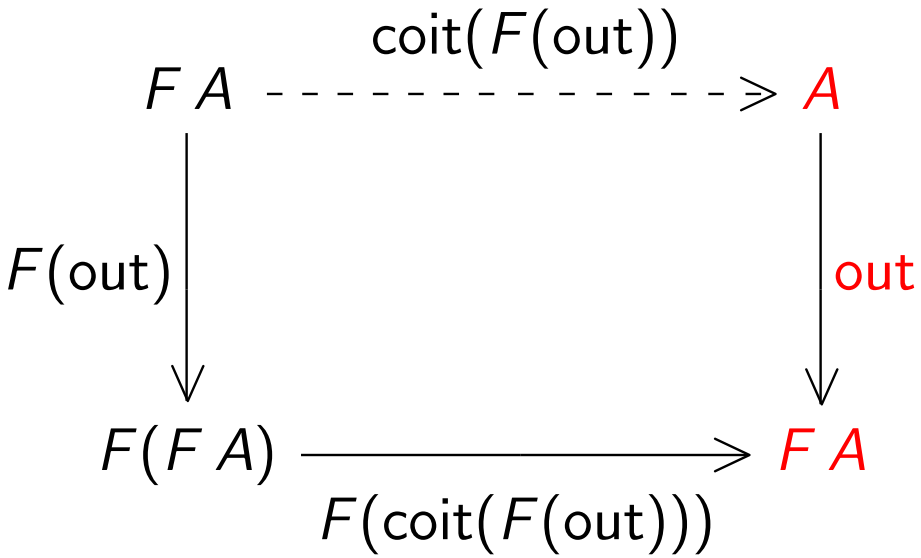


For a co-inductive type definition, Coq gives the following

- Cons : $F\,A \to A$
- out $\circ$ Cons = id
  (hd(Cons $a\,s$) = a and tl(Cons $a\,s$) = s).
- $\forall x : A, \exists y : F\,A, x = \text{Cons}\,y$
- A guarded definition principle

Can we recover these from the final algebra diagram?

Coq's coinductive types from final coalgebras



- We define Cons := coit(F(out))
- Then out ∘ Cons = id (By Lambek's Lemma)
- From this one can prove

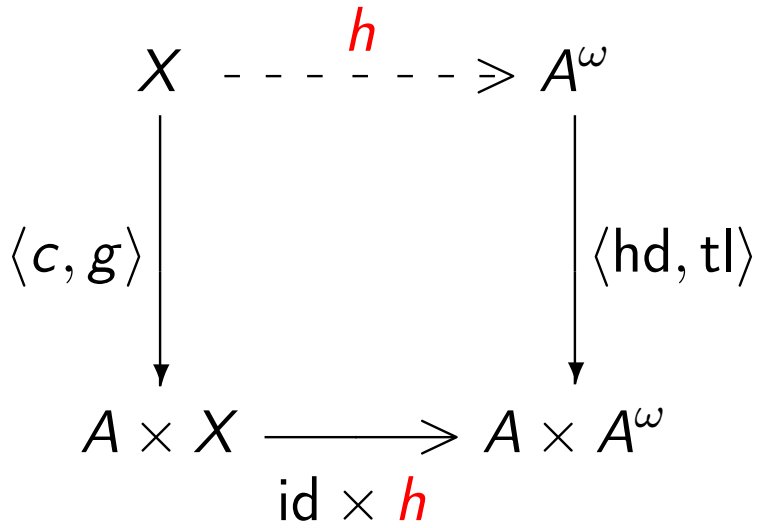
$$\text{Cons} \circ \text{out} = \text{id}$$

.

- Then for $\forall x : A, \exists y : F\,A, x = \text{Cons}\,y$, by taking $y := \text{out}\,x$.

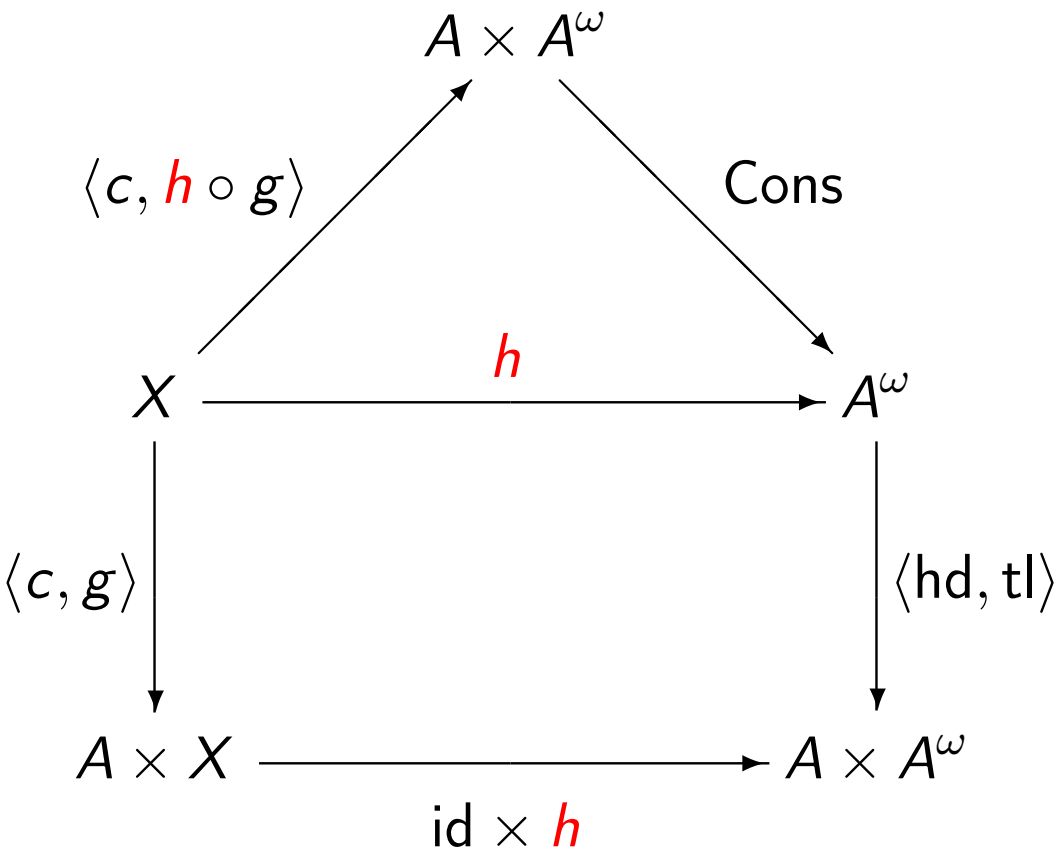
Coq's guarded definitions from final coalgebras

We consider the case of streams.

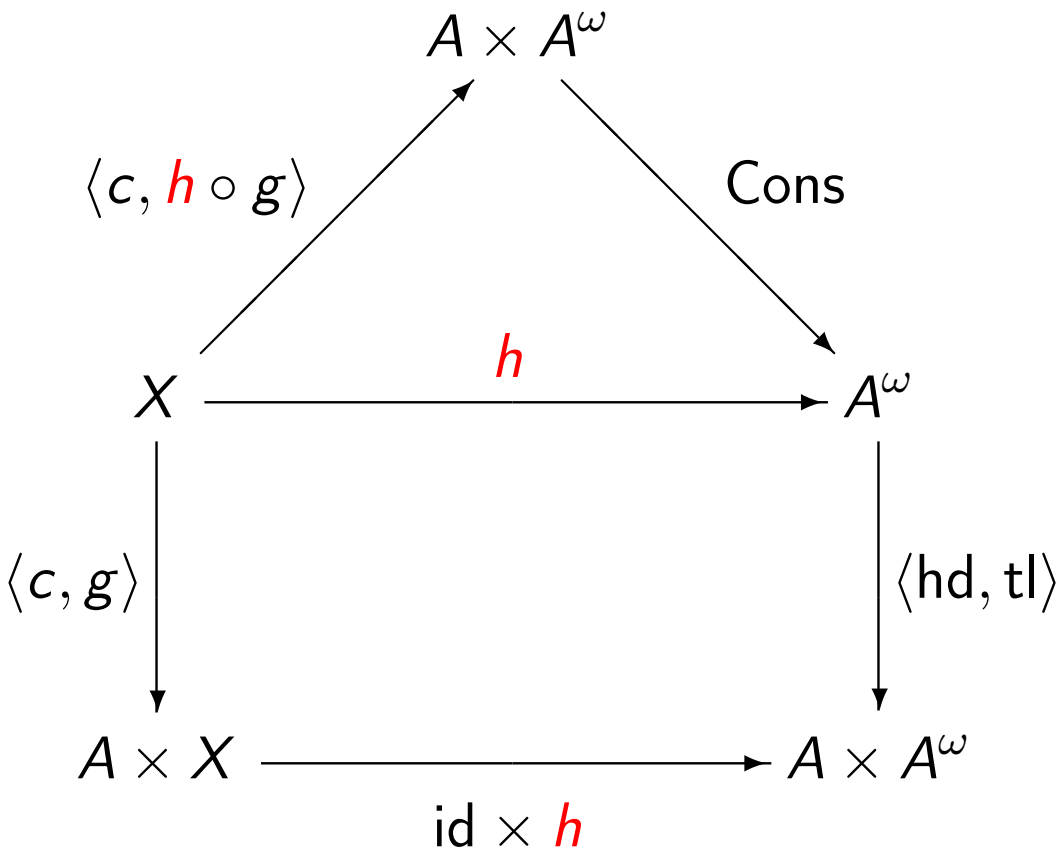


The top of the diagram can be further decomposed.

Coq's guarded definitions from final coalgebras



Coq's guarded definitions from final coalgebras



Coq actually uses this property to define the function h.

```
CoFixpoint h (x:A) := Cons (c x) (h (g x))
```

The Hamming Stream in Coq (Eelis vd Weegen)

- All natural numbers of the form $2^i 3^j 5^k$
- . . . in increasing order.

$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36, \ldots$

The Hamming Stream in Coq (Eelis vd Weegen)

- All natural numbers of the form $2^i 3^j 5^k$
- . . . in increasing order.

$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36, \ldots$

Popularized by Edsger Dijkstra

Formalisation

We would like to

- Write a formal specification for the Hamming stream
- Give an implementation of the Hamming stream (an algorithm, e.g. Dijkstra's)
- Prove that the implementation satisfies the specification
- All this formally in Coq ...

Dijkstra's Hamming-Stream Algorithm

Algorithm uses the `merge` function

```
merge :: [Integer] -> [Integer] -> [Integer]
merge (x:xs) (y:ys)
   | x < y    = x : merge xs (y:ys)
   | x > y    = y : merge (x:xs) ys
   | x == y   = x : merge xs ys
```

Dijkstra's Hamming-Stream Algorithm

Algorithm uses the merge function

```
merge :: [Integer] -> [Integer] -> [Integer]
merge (x:xs) (y:ys)
   | x < y    = x : merge xs (y:ys)
   | x > y    = y : merge (x:xs) ys
   | x == y   = x : merge xs ys
```

```
hamming :: [Integer]
hamming = 1 :
  merge (map (* 2) hamming) (
    merge (map (* 3) hamming)
      (map (* 5) hamming) )
```

Dijkstra's Hamming-Stream Algorithm

```
hamming :: [Integer]
hamming = 1 :
  merge (map (* 2) hamming) (
    merge (map (* 3) hamming)
      (map (* 5) hamming) )
```

- Functional
- Efficient
- Elegant(?)
- Representation of streams via the usual list datatype

Formal specification

A stream s is the Hamming stream if

- when a number occurs in s, it has the right shape ("soundness");
- when a number has the right shape, it occurs in s ("completeness");
- s is increasing.

Dijkstra's Hamming-stream algorithm

It is not possible to use Dijkstra's Hamming-stream algorithm directly in Coq.
Problem: Coq requires a CoFixpoint definition to be *explicitly guarded*. Dijkstra's Hamming-stream algorithm not guarded:

Dijkstra's Hamming-stream algorithm

It is not possible to use Dijkstra's Hamming-stream algorithm directly in Coq.
Problem: Coq requires a CoFixpoint definition to be *explicitly guarded*. Dijkstra's Hamming-stream algorithm not guarded:

```
hamming = 1 :
  merge (map (* 2) hamming) (
    merge (map (* 3) hamming)
      (map (* 5) hamming) )
```

Dijkstra's Hamming-stream algorithm

It is not possible to use Dijkstra's Hamming-stream algorithm directly in Coq.
Problem: Coq requires a CoFixpoint definition to be *explicitly guarded*. Dijkstra's Hamming-stream algorithm not guarded:

```
hamming = 1 :
  merge (map (* 2) hamming) (
    merge (map (* 3) hamming)
      (map (* 5) hamming) )
```

hamming is "under" map and merge, which could do anything, e.g. take the tail of a list ...

```
CoFixpoint ff (s:Stream nat) :=
  Cons 1 (tl (ff s)).
```

is ill-formed.

Another Hamming-stream algorithm

How to formalize the Hamming stream?

- Use another algorithm that is (syntactically) guarded.
- Tricks to use Dijkstra's algorithm after all.

Another Hamming-stream algorithm

```
CoFixpoint ham_from (l: ne_list nat): Stream nat :=
  Cons (hd l) (ham_from (
   enqueue (hd l * 2)(
     enqueue (hd l * 3)(
       enqueue (hd l * 5)(tl l)))))
```

where enqueue takes an n and an increasing list l and puts n "at the right place" in l:

Another Hamming-stream algorithm

```
CoFixpoint ham_from (l: ne_list nat): Stream nat :=
  Cons (hd l) (ham_from (
   enqueue (hd l * 2)(
     enqueue (hd l * 3)(
       enqueue (hd l * 5)(tl l)))))
```

where enqueue takes an *n* and an increasing list *l* and puts *n* "at the right place" in *l*:

```
Fixpoint enqueue(n: nat)(l: list nat) : ne_list nat :=
  match l with
  | nil => [n]
  | h :: t =>
    if n<h then n :: l else
     if n=h then h :: t else
       h :: (enqueue n t)
    end.
```

The orginal Hamming-stream algorithm?

Define hamming and merge on lists:

```
Definition ham_body (l: list nat): list nat :=
cons 1 (merge (map (mult 2) l)
(merge (map (mult 3) l) (map (mult 5) l))).
```

The orginal Hamming-stream algorithm?

Define hamming and merge on lists:

```
Definition ham_body (l: list nat): list nat :=
cons 1 (merge (map (mult 2) l)
(merge (map (mult 3) l) (map (mult 5) l))).
```

```
n | repeat_apply ham_body nil n
-----------------------------
0 | 1
1 | 1, 2, 3, 5
2 | 1, 2, 3, 4, 5, 6, 9, 10, 15, 25
3 | 1, 2, 3, 4, 5, 6, 8, 9,  10, 12, 15, 18, 20, 25, 27, 30
```

Taking the "diagonal"produces the Hamming stream.

Conclusion

- Coq can deal with coinductive types in a natural way, but ...
- The strong guardedness requirements gets in the way when defining objects in a coinductive type, so one cannot "just" write a Haskell-like algorithm in Coq.
- When a definition is proved productive, it should be accepted.
- First define a stream, and then prove it is productive?
- Stay closer to the final coalgebra definition of coinductive types, with destructors as basic?

Related work

- Th. Coquand (1994) – Infinite Objects in Type Theory.
- Eduardo Giménez (1994) – Codifying Guarded Definitions with Recursive Schemes
- Yves Bertot (2005) – Filters on CoInductive Streams, an Application to Eratosthenes' Sieve
- M. Niqui (2009) – Coalgebraic Reasoning in Coq: Bisimulation and the λ-Coiteration Scheme.
- Nils Anders Danielsson (2010) – Beating the Productivity Checker Using Embedded Languages
- Jörg Endrullis, Dimitri Hendriks, Martin Bodin (2013) – Circular Coinduction in Coq using Bisimulation-Up-To Techniques
- ... see the talk by Andreas Abel.