An Encoding of (Co)inductive Types in Coq

via W- and M-types in the category of setoids

Galaad Langlois[1], Damien Pous[2], Yannick Zakowski[3]

[1]ENS de Lyon

[2]CNRS

[3]Inria

The Coq Workshop 2023, July 31

Introduction

Goal: a library to easily define and work with datatypes such as

1. $\mu X.\ \mathbf{1} + A \times X^2$
2. $\nu X.\ A \times X^2$

Introduction

Goal: a library to easily define and work with datatypes such as

1. $\mu X.\ \mathbf{1} + A \times X^2$
2. $\nu X.\ A \times X^2$
3. $\mu X.\ A \times \mathcal{F}(X)$

Introduction

Goal: a library to easily define and work with datatypes such as

1. $\mu X.\ \mathbf{1} + A \times X^2$
2. $\nu X.\ A \times X^2$
3. $\mu X.\ A \times \mathcal{F}(X)$
4. $\nu X.\ \mu Y.\ X + Y$

Introduction

Goal: a library to easily define and work with datatypes such as

1. $\mu X.\ \mathbf{1} + A \times X^2$
2. $\nu X.\ A \times X^2$
3. $\mu X.\ A \times \mathcal{F}(X)$
4. $\nu X.\ \mu Y.\ X + Y$

Ingredients:

Goal: a library to easily define and work with datatypes such as

1. $\mu X.\ \mathbf{1} + A \times X^2$
2. $\nu X.\ A \times X^2$
3. $\mu X.\ A \times \mathcal{F}(X)$
4. $\nu X.\ \mu Y.\ X + Y$

Ingredients:

- category theory can give constructors, pattern-matching, recursion principles, induction principles

Introduction

Goal: a library to easily define and work with datatypes such as

1. $\mu X.\ \mathbf{1} + A \times X^2$
2. $\nu X.\ A \times X^2$
3. $\mu X.\ A \times \mathcal{F}(X)$
4. $\nu X.\ \mu Y.\ X + Y$

Ingredients:

- category theory can give constructors, pattern-matching, recursion principles, induction principles
- W-types and M-types: a generic family of (co)inductive types

Goal: a library to easily define and work with datatypes such as

1. $\mu X.\ \mathbf{1} + A \times X^2$
2. $\nu X.\ A \times X^2$
3. $\mu X.\ A \times \mathcal{F}(X)$
4. $\nu X.\ \mu Y.\ X + Y$

Ingredients:

- category theory can give constructors, pattern-matching, recursion principles, induction principles
- W-types and M-types: a generic family of (co)inductive types
- setoids: a necessity for an axiom-free Coq implementation

Introduction

Goal: a library to easily define and work with datatypes such as

1. $\mu X.\ \mathbf{1} + A \times X^2$
2. $\nu X.\ A \times X^2$
3. $\mu X.\ A \times \mathcal{F}(X)$
4. $\nu X.\ \mu Y.\ X + Y$

Ingredients:

- category theory can give constructors, pattern-matching, recursion principles, induction principles
- W-types and M-types: a generic family of (co)inductive types
- setoids: a necessity for an axiom-free Coq implementation

Current state: an experiment

Table of Contents

Equality for coinductive types

```
CoInductive stream := cons { hd : nat; tl : stream }.

CoFixpoint zeros := cons 0 zeros.
CoFixpoint zeros' := cons 0 (cons 0 zeros').
```

Equality for coinductive types

```
CoInductive stream := cons { hd : nat; tl : stream }.

CoFixpoint zeros := cons 0 zeros.
CoFixpoint zeros' := cons 0 (cons 0 zeros').
```

- impossible to prove `zeros = zeros'`
- need to define a bisimilarity relation by hand

```
CoInductive EqSt (s1 s2 : stream) : Prop := eqst {
  eqst_hd : hd s1 = hd s2;
  eqst_tl : EqSt (tl s1) (tl s2);
}.
```

Compositionnality

```
Inductive tree (A : Type) : Type :=
  | node (label : A) (children : list (tree A)).
```

Compositionnality

```
Inductive tree (A : Type) : Type :=
  | node (label : A) (children : list (tree A)).
```

- the automatically generated induction principle is useless

Compositionnality

```
Inductive tree (A : Type) : Type :=
  | node (label : A) (children : list (tree A)).
```

- the automatically generated induction principle is useless
- impossible to abstract over `list`

```
Context (F : Type -> Type)
Context (Fmap : ∀ (X Y : Type), (X -> Y) -> F X -> F Y).

Fail Inductive tree (A : Type) : Type :=
  | node (label : A) (children : F (tree A)).
```

The guard condition

- fixpoints and cofixpoints are accepted by Coq only if they respect a syntactic *guard condition*

The guard condition

- fixpoints and cofixpoints are accepted by Coq only if they respect a syntactic *guard condition*
- this condition prevents from defining some corecursive functions that are perfectly justified mathematically

The guard condition

- fixpoints and cofixpoints are accepted by Coq only if they respect a syntactic *guard condition*
- this condition prevents from defining some corecursive functions that are perfectly justified mathematically
- example: shuffle product on streams

```
Fail CoFixpoint shuffle (s s' : stream) : stream :=
  shuffle (tl s) s' + shuffle s (tl s').
```

Inspiration from other proof assistants

- Isabelle/HOL: AmiCo library [Blanchette et al., 2017] using *bounded natural functors* (BNF)

Inspiration from other proof assistants

- Isabelle/HOL: AmiCo library [Blanchette et al., 2017] using *bounded natural functors* (BNF)
- Lean: *quotients of polynomial functors* [Avigad et al., 2019]

Inspiration from other proof assistants

- Isabelle/HOL: AmiCo library [Blanchette et al., 2017] using *bounded natural functors* (BNF)
- Lean: *quotients of polynomial functors* [Avigad et al., 2019]
- these frameworks require quotient types, propositional and functional extensionality

Inspiration from other proof assistants

- Isabelle/HOL: AmiCo library [Blanchette et al., 2017] using *bounded natural functors* (BNF)
- Lean: *quotients of polynomial functors* [Avigad et al., 2019]
- these frameworks require quotient types, propositional and functional extensionality
- goal: a similar framework, in Coq, axiom-free

Table of Contents

Functors of types

Definition (Functor)

A functor is defined by its action F on types:

```
F : Type -> Type
```

Functors of types

Definition (Functor)

A functor is defined by its action F on types:

```
F : Type -> Type
```

and its action F^{map} on functions:

```
Fmap : ∀ X Y, (X -> Y) -> (F X -> F Y)
```

Functors of types

Definition (Functor)

A functor is defined by its action F on types:

```
F : Type -> Type
```

and its action F^{map} on functions:

```
Fmap : ∀ X Y, (X -> Y) -> (F X -> F Y)
```

Example:

$$F = \texttt{list} \qquad\qquad F^{\text{map}} = \texttt{List.map}$$

Constructors as algebras

An inductive type is the “least type” closed under some *constructors* (e.g., 0 and succ for the type nat).

Constructors as algebras

An inductive type is the "least type" closed under some *constructors* (e.g., 0 and succ for the type nat).

Definition (F-algebra)

An *F-algebra* is a pair (X, a) where X is a type and $a : F(X) \to X$.

Constructors as algebras

An inductive type is the "least type" closed under some *constructors* (e.g., 0 and succ for the type nat).

Definition (F-algebra)

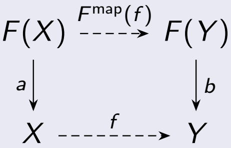
An *F-algebra* is a pair (X, a) where X is a type and $a : F(X) \to X$.

$$F(X) = \mathbf{1} + X \qquad\qquad \mathbf{1} + \mathrm{nat} \xrightarrow{[0,\mathrm{succ}]} \mathrm{nat}$$

Initial algebras

Definition (Initial F-algebra)

An *initial F-algebra* is an algebra $a : F(X) \to X$ such that for all algebra $b : F(Y) \to Y$, there exists a unique function $f : X \to Y$ such that the following diagram commutes:



Recursion via initiality

```
Fixpoint iseven (n : nat) : bool :=
  match n with
  | 0 => true
  | S n => negb (iseven n)
  end.
```

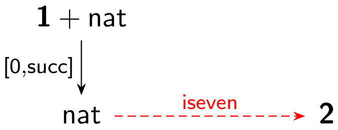
Recursion via initiality

```
Fixpoint iseven (n : nat) : bool :=
  match n with
  | 0 => true
  | S n => negb (iseven n)
  end.
```

$$\begin{array}{c} \mathbf{1} + \mathsf{nat} \\ {\scriptstyle [0,\mathrm{succ}]}\big\downarrow \\ \mathsf{nat} \end{array}$$

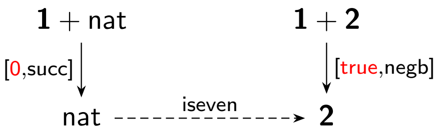
Recursion via initiality

```
Fixpoint iseven (n : nat) : bool :=
  match n with
  | 0 => true
  | S n => negb (iseven n)
  end.
```

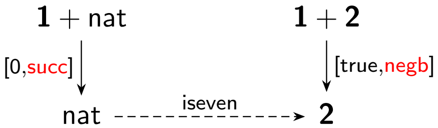


Recursion via initiality

```
Fixpoint iseven (n : nat) : bool :=
  match n with
  | 0 => true
  | S n => negb (iseven n)
  end.
```

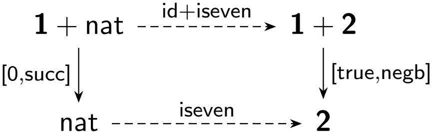


```
Fixpoint iseven (n : nat) : bool :=
  match n with
  | 0 => true
  | S n => negb (iseven n)
  end.
```



Recursion via initiality

```
Fixpoint iseven (n : nat) : bool :=
  match n with
  | 0 => true
  | S n => negb (iseven n)
  end.
```



$$\begin{cases} \text{iseven}(0) = \text{true} \\ \text{iseven}(\text{succ}(n)) = \text{negb}(\text{iseven}(n)) \end{cases}$$

Destructors as coalgebras

A coinductive type is the “greatest type” closed under some *destructors* (e.g., hd and tl for the type stream).

A coinductive type is the “greatest type” closed under some *destructors* (e.g., hd and tl for the type stream).

Definition (F-coalgebra)

An F*-coalgebra* is a pair (X, c) where X is a type and $c : X \to F(X)$.

Destructors as coalgebras

A coinductive type is the "greatest type" closed under some *destructors* (e.g., hd and tl for the type stream).

Definition (F-coalgebra)

An F-*coalgebra* is a pair (X, c) where X is a type and $c : X \to F(X)$.

$$F(X) = A \times X \qquad \mathrm{stream}_A \xrightarrow{(\mathrm{hd},\mathrm{tl})} A \times \mathrm{stream}_A$$

Final coalgebras

Definition (Final F-coalgebra)

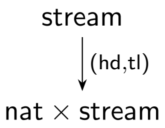
A *final F-coalgebra* is a coalgebra $c : Z \to F(Z)$ such that for all coalgebra $d : X \to F(X)$, there exists a unique function $f : X \to Z$ such that the following diagram commutes:

$$\begin{array}{ccc} X & \overset{f}{\dashrightarrow} & Z \\ {\scriptstyle d}\downarrow & & \downarrow{\scriptstyle c} \\ F(X) & \overset{F(f)}{\dashrightarrow} & F(Z) \end{array}$$

Corecursion via finality

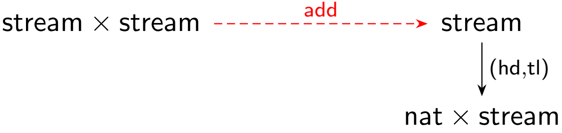
```
CoFixpoint add (s s' : stream) : stream :=
  (hd s + hd s') ::: (add (tl s) (tl s')).
```

```
CoFixpoint add (s s' : stream) : stream :=
  (hd s + hd s') ::: (add (tl s) (tl s')).
```

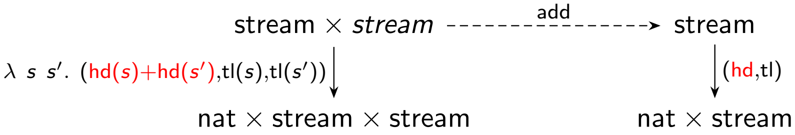


Corecursion via finality

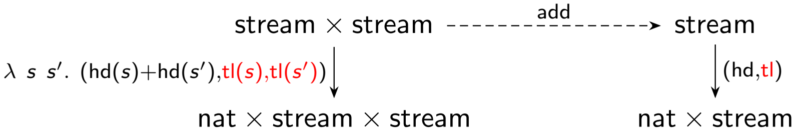
```
CoFixpoint add (s s' : stream) : stream :=
  (hd s + hd s') ::: (add (tl s) (tl s')).
```



```
CoFixpoint add (s s' : stream) : stream :=
  (hd s + hd s') ::: (add (tl s) (tl s')).
```

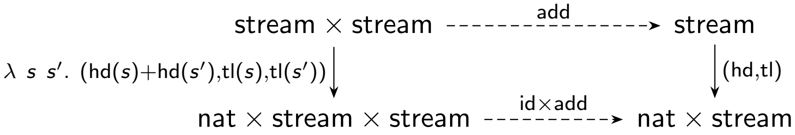


```
CoFixpoint add (s s' : stream) : stream :=
  (hd s + hd s') ::: (add (tl s) (tl s')).
```



Corecursion via finality

```
CoFixpoint add (s s' : stream) : stream :=
  (hd s + hd s') ::: (add (tl s) (tl s')).
```



$$\begin{cases} \mathrm{hd}(\mathrm{add}(s, s')) = \mathrm{hd}(s) + \mathrm{hd}(s') \\ \mathrm{tl}(\mathrm{add}(s, s')) = \mathrm{add}(\mathrm{tl}(s), \mathrm{tl}(s')) \end{cases}$$

Examples

Inductive types = initial algebras (μ) :

Examples

Inductive types = initial algebras (μ) :

- nat $= \mu X.\ \mathbf{1} + X$

Examples

Inductive types = initial algebras (μ) :

- nat $= \mu X.\ \mathbf{1} + X$
- $\text{list}_A = \mu X.\ \mathbf{1} + A \times X$

Examples

Inductive types = initial algebras (μ) :

- nat $= \mu X.\ \mathbf{1} + X$
- list$_A = \mu X.\ \mathbf{1} + A \times X$
- btree$_A = \mu X.\ A + X^2$ binary trees with leaves labelled in A

Examples

Inductive types = initial algebras (μ) :

- $\mathsf{nat} = \mu X.\ \mathbf{1} + X$
- $\mathsf{list}_A = \mu X.\ \mathbf{1} + A \times X$
- $\mathsf{btree}_A = \mu X.\ A + X^2$ binary trees with leaves labelled in A

Coinductive types = final coalgebras (ν) :

Examples

Inductive types = initial algebras (μ) :

- $\mathsf{nat} = \mu X.\ \mathbf{1} + X$
- $\mathsf{list}_A = \mu X.\ \mathbf{1} + A \times X$
- $\mathsf{btree}_A = \mu X.\ A + X^2$ binary trees with leaves labelled in A

Coinductive types = final coalgebras (ν) :

- $\mathsf{stream}_A = \nu X.\ A \times X$

Examples

Inductive types = initial algebras (μ) :

- nat $= \mu X.\ \mathbf{1} + X$
- list$_A = \mu X.\ \mathbf{1} + A \times X$
- btree$_A = \mu X.\ A + X^2$ binary trees with leaves labelled in A

Coinductive types = final coalgebras (ν) :

- stream$_A = \nu X.\ A \times X$
- colist$_A = \nu X.\ \mathbf{1} + A \times X$ potentially infinite lists

Examples

Inductive types = initial algebras (μ) :

- $\mathsf{nat} = \mu X.\ \mathbf{1} + X$
- $\mathsf{list}_A = \mu X.\ \mathbf{1} + A \times X$
- $\mathsf{btree}_A = \mu X.\ A + X^2$ binary trees with leaves labelled in A

Coinductive types = final coalgebras (ν) :

- $\mathsf{stream}_A = \nu X.\ A \times X$
- $\mathsf{colist}_A = \nu X.\ \mathbf{1} + A \times X$ potentially infinite lists
- $\mathsf{fbtree}_A = \nu X.\ A \times \mathsf{list}(X)$ finitely branching trees

Table of Contents

Polynomial functors

Not all functors have an initial algebra/final coalgebra.

- in Coq: strict positivity condition

Polynomial functors

Not all functors have an initial algebra/final coalgebra.

- in Coq: strict positivity condition
- in category theory: polynomial functors

Polynomial functors

Not all functors have an initial algebra/final coalgebra.

- in Coq: strict positivity condition
- in category theory: polynomial functors

$$P, Q ::= \mathrm{id} \mid \mathrm{cst}_S \mid P + Q \mid P \times Q \mid P^S$$

Definitions

Definition (Container)

A *container* is a pair noted $(A \triangleright B)$ where A : Type and $B : A \to Type$.

Definition (Container)

A *container* is a pair noted $(A \triangleright B)$ where A : Type and $B : A \to Type$.

Definition (Polynomial functor)

A *polynomial functor* is, up to equivalence, a functor of the form:

$$P(X) = \sum_{a:A} X^{B(a)}$$

Definitions

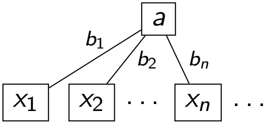
Definition (Container)

A *container* is a pair noted $(A \triangleright B)$ where A : Type and $B : A \to \mathit{Type}$.

Definition (Polynomial functor)

A *polynomial functor* is, up to equivalence, a functor of the form:

$$P(X) = \sum_{a:A} X^{B(a)}$$



$a : A$

$f : B(a) \to X$

Examples

$$A : \mathrm{Type} \qquad B : A \to \mathrm{Type}$$

$$P(X) = \sum_{a:A} X^{B(a)}$$

Examples

$$A : \text{Type} \qquad B : A \to \text{Type} \qquad P(X) = \sum_{a:A} X^{B(a)}$$

$P(X) = X \times X$	$(\mathbf{1} \triangleright \star \mapsto \mathbf{2})$

Examples

$$A : \mathsf{Type} \qquad B : A \to \mathsf{Type} \qquad P(X) = \sum_{a:A} X^{B(a)}$$

$P(X) = X \times X$	$(\mathbf{1} \triangleright \star \mapsto \mathbf{2})$
$P(X) = \mathsf{option}(X) \simeq \mathbf{1} + X$	$\left(\mathbf{2} \triangleright \begin{array}{l} \mathsf{true} \mapsto \mathbf{0} \\ \mathsf{false} \mapsto \mathbf{1} \end{array}\right)$

Examples

$$A : \mathsf{Type} \qquad B : A \to \mathsf{Type} \qquad P(X) = \sum_{a:A} X^{B(a)}$$

$P(X) = X \times X$	$(\mathbf{1} \triangleright \star \mapsto \mathbf{2})$
$P(X) = \mathsf{option}(X) \simeq \mathbf{1} + X$	$\left(\mathbf{2} \triangleright \begin{array}{l} \mathsf{true} \mapsto \mathbf{0} \\ \mathsf{false} \mapsto \mathbf{1} \end{array}\right)$
$P(X) = \mathsf{list}(X) \simeq \sum_{n:\mathsf{nat}} X^n$	$(\mathsf{nat} \triangleright n \mapsto \mathbf{n})$

Closure properties

Polynomial functors are closed under:

- sum
- product
- composition
- (μ and ν in the multivariate case)

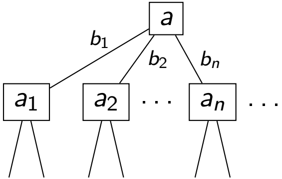
which means providing the corresponding constructions on containers and establishing the associated functor equivalences

W-types and M-types

```
Context (A : Type) (B : A -> Type).
Inductive W : Type :=
| sup (a : A) (f : B a -> W) : W.
```

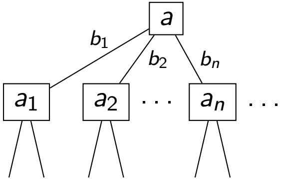
W-types and M-types

```
Context (A : Type) (B : A -> Type).
Inductive W : Type :=
| sup (a : A) (f : B a -> W) : W.
```



W-types and M-types

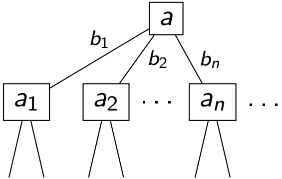
```
Context (A : Type) (B : A -> Type).
Inductive W : Type :=
| sup (a : A) (f : B a -> W) : W.
```



- well-founded trees: $W = \mu X.\ P(X)$

W-types and M-types

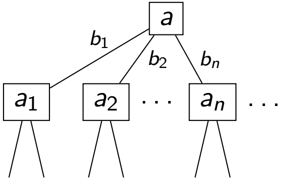
```
Context (A : Type) (B : A -> Type).
Inductive W : Type :=
| sup (a : A) (f : B a -> W) : W.
```



- well-founded trees: $W = \mu X.\ P(X)$
- non-well-founded trees: $M = \nu X.\ P(X)$

W-types and M-types

```
Context (A : Type) (B : A -> Type).
Inductive W : Type :=
| sup (a : A) (f : B a -> W) : W.
```



- well-founded trees: $W = \mu X.\ P(X)$
- non-well-founded trees: $M = \nu X.\ P(X)$

"one (co)inductive to rule them all"

Table of Contents

Extensionality problems

- goal: axiom-free implementation

Extensionality problems

- goal: axiom-free implementation
- functional extensionality is necessary for the proof that W-types carry a structure of initial algebras

Extensionality problems

- goal: axiom-free implementation
- functional extensionality is necessary for the proof that W-types carry a structure of initial algebras
- quotient types are necessary to define coinductive types with the appropriate notion of equality, namely bisimilarity

Extensionality problems

- goal: axiom-free implementation
- functional extensionality is necessary for the proof that W-types carry a structure of initial algebras
- quotient types are necessary to define coinductive types with the appropriate notion of equality, namely bisimilarity
- these are *extensional* concepts, while Coq is based on an *intensional* type theory

Extensionality problems

- goal: axiom-free implementation
- functional extensionality is necessary for the proof that W-types carry a structure of initial algebras
- quotient types are necessary to define coinductive types with the appropriate notion of equality, namely bisimilarity
- these are *extensional* concepts, while Coq is based on an *intensional* type theory
- solution : setoids [Hofmann 1995]

Solution : shift of category

Types $\dashrightarrow$ Setoids

Definition (Setoid)

A *setoid* is a pair $(X, \equiv_X)$ where X is a type and $\equiv_X$ is an equivalence relation on X.

Solution : shift of category

Types ⇢ Setoids

Definition (Setoid)

A *setoid* is a pair $(X, \equiv_X)$ where X is a type and $\equiv_X$ is an equivalence relation on X.

Functions ⇢ Extensional functions

Definition (Extensional function)

An *extensional function* between two setoids $(X, \equiv_X)$ and $(Y, \equiv_Y)$ is a function $f : X \to Y$ such that if $x \equiv_X x'$ then $f(x) \equiv_Y f(x')$.

Solution : shift of category

Types --→ Setoids

Definition (Setoid)

A *setoid* is a pair $(X, \equiv_X)$ where X is a type and $\equiv_X$ is an equivalence relation on X.

Functions --→ Extensional functions

Definition (Extensional function)

An *extensional function* between two setoids $(X, \equiv_X)$ and $(Y, \equiv_Y)$ is a function $f : X \to Y$ such that if $x \equiv_X x'$ then $f(x) \equiv_Y f(x')$.

0 1 2 $+$ $\times$ $\surd$

Solution : shift of category

Types $\dashrightarrow$ Setoids

Definition (Setoid)

A *setoid* is a pair $(X, \equiv_X)$ where X is a type and $\equiv_X$ is an equivalence relation on X.

Functions $\dashrightarrow$ Extensional functions

Definition (Extensional function)

An *extensional function* between two setoids $(X, \equiv_X)$ and $(Y, \equiv_Y)$ is a function $f : X \to Y$ such that if $x \equiv_X x'$ then $f(x) \equiv_Y f(x')$.

$\boxed{\mathbf{0}\ \mathbf{1}\ \mathbf{2}\ +\ \times}\ \surd \qquad \boxed{A \to \text{Type}\ \sum\ \prod}\ ?$

Setoid families

- what is a setoid family on A?

Setoid families

- what is a setoid family on A?
- $B : A \to \text{Setoid}$

Setoid families

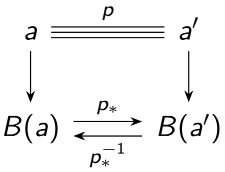
- what is a setoid family on A?
- $B : A \to \text{Setoid}$
- $a \equiv a' \implies B(a) \approx B(a')$

Setoid families

- what is a setoid family on A?
- $B : A \to \text{Setoid}$
- $a \equiv a' \implies B(a) \approx B(a')$
- *proof-irrelevant setoid families* [Palmgren 2012]

Setoid families

- what is a setoid family on A?
- $B : A \to \text{Setoid}$
- $a \equiv a' \implies B(a) \approx B(a')$
- *proof-irrelevant setoid families* [Palmgren 2012]
- *transport* function along an equivalence $p : a \equiv a'$, $p_* : B(a) \to B(a')$, isomorphism $B(a) \cong B(a')$



Polynomial functors of setoids

```
Structure container := {
  A : Setoid;
  B : setoid_family A }.
```

Polynomial functors of setoids

```
Structure container := {
  A : Setoid;
  B : setoid_family A }.

Structure PFUNCTOR := {
  pf_func :> FUNCTOR SETOIDS SETOIDS;
  pf_cont : container;
  pfE : pf_func ≃ extension pf_cont }.
```

W-setoids and M-setoids

- counterparts of W- and M-types in the category of setoids

W-setoids and M-setoids

- counterparts of W- and M-types in the category of setoids
- W- and M-types enriched with an equivalence relation

W-setoids and M-setoids

- counterparts of W- and M-types in the category of setoids
- W- and M-types enriched with an equivalence relation
- extensionnal

W-setoids and M-setoids

- counterparts of W- and M-types in the category of setoids
- W- and M-types enriched with an equivalence relation
- extensionnal
- initiality of the algebra of W-setoids: very challenging, already done in Coq [Palmgren 2015]

W-setoids and M-setoids

- counterparts of W- and M-types in the category of setoids
- W- and M-types enriched with an equivalence relation
- extensionnal
- initiality of the algebra of W-setoids: very challenging, already done in Coq [Palmgren 2015]
- finality of the coalgebra of M-setoids: easier

W-setoids and M-setoids

- counterparts of W- and M-types in the category of setoids
- W- and M-types enriched with an equivalence relation
- extensionnal
- initiality of the algebra of W-setoids: very challenging, already done in Coq [Palmgren 2015]
- finality of the coalgebra of M-setoids: easier

(implemented in the code)

In practice (1)

1. Define a `PFUNCTOR` using the provided DSL.

```
Definition P := nat * X.
```

In practice (1)

1. Define a PFUNCTOR using the provided DSL.

```
Definition P := nat * X.
```

2. Define the desired (co)inductive setoid as the W-setoid associated to the functor.

```
Definition stream_coalg := nu_coalg P.
Definition stream := coalg_car stream_coalg.
Definition final_stream_coalg : final stream_coalg :=
  final_nu P.
```

In practice (1)

1. Define a PFUNCTOR using the provided DSL.

```
Definition P := nat * X.
```

2. Define the desired (co)inductive setoid as the W-setoid associated to the functor.

```
Definition stream_coalg := nu_coalg P.
Definition stream := coalg_car stream_coalg.
Definition final_stream_coalg : final stream_coalg :=
  final_nu P.
```

2'. Use your own type, enriched with an equivalence relation, a (co)algebra structure and a proof that it is an initial/final (co)algebra of the functor.

In practice (2)

3. The (Co)Lambek lemma provides a constructor and a destructor.

```
Definition iso_fix : stream ≃ nat * stream :=
  CoLambek final_stream_coalg.
```

In practice (2)

3. The (Co)Lambek lemma provides a constructor and a destructor.

```
Definition iso_fix : stream ≃ nat * stream :=
  CoLambek final_stream_coalg.
```

4. The initiality/finality property provides a (co)recursor to define (co)recursive functions.

```
Definition stream_corec (X : Setoid) (c : X -·-> nat * X)
    : X -·-> stream :=
  corec final_stream_coalg c.

Definition add : stream * stream -·-> stream :=
  stream_corec (stream * stream)
    (fun s s' => (hd s + hd s', (tl s, tl s'))).
```

In practice (2)

3. The (Co)Lambek lemma provides a constructor and a destructor.

```
Definition iso_fix : stream ≃ nat * stream :=
  CoLambek final_stream_coalg.
```

4. The initiality/finality property provides a (co)recursor to define (co)recursive functions.

```
Definition stream_corec (X : Setoid) (c : X ⟶ nat * X)
    : X ⟶ stream :=
  corec final_stream_coalg c.

Definition add : stream * stream ⟶ stream :=
  stream_corec (stream * stream)
    (fun s s' => (hd s + hd s', (tl s, tl s'))).
```

5. A (co)induction principle is provided for proofs.

Future work

- automation and syntactic sugar

Future work

- automation and syntactic sugar
- nested and mixed inductive-coinductive types $\to$ multivariate polynomial functors

Future work

- automation and syntactic sugar
- nested and mixed inductive-coinductive types $\rightarrow$ multivariate polynomial functors
- mutually defined (co)inductive types $\rightarrow$ dependent polynomial functors

Future work

- automation and syntactic sugar
- nested and mixed inductive-coinductive types $\rightarrow$ multivariate polynomial functors
- mutually defined (co)inductive types $\rightarrow$ dependent polynomial functors
- quotients of polynomial functors

Future work

- automation and syntactic sugar
- nested and mixed inductive-coinductive types $\rightarrow$ multivariate polynomial functors
- mutually defined (co)inductive types $\rightarrow$ dependent polynomial functors
- quotients of polynomial functors
- more powerful (co)recursion principle $\rightarrow$ up-to techniques