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

1. $\mu X. 1 + A \times X^2$
2. $\nu X. A \times X^2$
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. $\mu X. \; A \times F(X)$
Goal: a library to easily define and work with datatypes such as

1. $\mu X. \ 1 + A \times X^2$
2. $\nu X. \ A \times X^2$
3. $\mu X. \ A \times F(X)$
4. $\nu X. \mu Y. \ X + Y$
Goal: a library to easily define and work with datatypes such as

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

Ingredients:
- category theory can give constructors, pattern-matching, recursion principles, induction principles
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. \( \mu X. \, 1 + A \times X^2 \)
2. \( \nu X. \, A \times X^2 \)
3. \( \mu X. \, A \times F(X) \)
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. $\mu X. \ 1 + A \times X^2$
2. $\nu X. \ A \times X^2$
3. $\mu X. \ A \times F(X)$
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>
<thead>
<tr>
<th></th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Some limits of (co)inductive types in Coq</td>
</tr>
<tr>
<td>2</td>
<td>(Co)inductive types, categorically</td>
</tr>
<tr>
<td>3</td>
<td>Polynomial functors</td>
</tr>
<tr>
<td>4</td>
<td>Coq implementation: use of setoids</td>
</tr>
</tbody>
</table>
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

\begin{verbatim}
CoInductive stream := cons { hd : nat; tl : stream }.

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

\end{verbatim}

\begin{itemize}
\item impossible to prove $\text{zeros} = \text{zeros'}$
\item need to define a bisimilarity relation by hand
\end{itemize}

\begin{verbatim}
CoInductive EqSt (s1 s2 : stream) : Prop := eqst {
  eqst_hd : hd s1 = hd s2;
  eqst_tl : EqSt (tl s1) (tl s2);
}. 
\end{verbatim}
Inductive tree (A : Type) : Type :=
  | node (label : A) (children : list (tree A)).
Inductive tree (A : Type) : Type :=
  | node (label : A) (children : list (tree A)).

▶ the automatically generated induction principle is useless
Compositionnality

\[\text{Inductive } \text{tree} (A : \text{Type}) : \text{Type} :=
\quad \mid \text{node} (\text{label} : A) (\text{children} : \text{list} (\text{tree} A)).\]

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

\textbf{Context} \ ((F : \text{Type} \rightarrow \text{Type})
\text{Context} \ (Fmap : \forall (X Y : \text{Type}), (X \rightarrow Y) \rightarrow F X \rightarrow F Y).

\textbf{Fail Inductive} \text{tree} (A : \text{Type}) : \text{Type} :=
\quad \mid \text{node} (\text{label} : A) (\text{children} : F (\text{tree} A)).
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 \textit{guard condition}.
- this condition prevents from defining some corecursive functions that are perfectly justified mathematically.

\texttt{Fail CoFixpoint shuffle \( (s \ s') : \text{stream} \) :=
\quad \text{shuffle} \ (\text{tl} \ s) \ s' + \text{shuffle} \ s \ (\text{tl} \ s').}
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

```coq
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 \textit{bounded natural functors} (BNF)
- Lean: \textit{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
1. Some limits of (co)inductive types in Coq

2. (Co)inductive types, categorically

3. Polynomial functors

4. Coq implementation: use of setoids
A functor is defined by its action \( F \) on types:

\[
F : \text{Type} \rightarrow \text{Type}
\]
Definition (Functor)

A functor is defined by its action $F$ on types:

$$F : \text{Type} \rightarrow \text{Type}$$

and its action $F^\text{map}$ on functions:

$$F^{\text{map}} : \forall X Y, (X \rightarrow Y) \rightarrow (F X \rightarrow F Y)$$
Functors of types

Definition (Functor)

A functor is defined by its action $F$ on types:

$$F : \text{Type} \rightarrow \text{Type}$$

and its action $F^{\text{map}}$ on functions:

$$F^{\text{map}} : \forall X \ Y, (X \rightarrow Y) \rightarrow (F \ X \rightarrow F \ Y)$$

Example:

$$F = \text{list} \quad F^{\text{map}} = \text{List.\ map}$$
An inductive type is the “least type” closed under some constructors (e.g., 0 and succ for the type nat).
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 $a : F(X) \to X$. 

Galaad Langlois
An Encoding of (Co)inductive Types in Coq 10 / 31
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 \(a : F(X) \rightarrow X\).

\[ F(X) = 1 + X \]

\[ 1 + \text{nat} \xrightarrow{[0,\text{succ}]} \text{nat} \]
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:

\[
\begin{array}{ccc}
F(X) & \xrightarrow{F^{\text{map}}(f)} & F(Y) \\
\downarrow a & & \downarrow b \\
X & \xrightarrow{f} & Y
\end{array}
\]
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.

\[
\begin{array}{c}
\text{1} + \text{nat} \\
\Downarrow [0, \text{succ}] \\
\text{nat}
\end{array}
\]
Fixpoint iseven (n : nat) : bool :=
  match n with
  | 0 => true
  | S n => negb (iseven n)
end.

1 + nat
[0,succ]↓
iseven

nat ────> 2
Fixpoint iseven (n : nat) : bool :=
  match n with
  | 0 => true
  | S n => negb (iseven n)
  end.
Recursion via initiality

\[\text{Fixpoint iseven (n : nat) : bool :=}
\begin{align*}
\text{match } n \text{ with} \\
\mid 0 & \Rightarrow \text{true} \\
\mid S\ n & \Rightarrow \neg\text{b (iseven } n) \\
\text{end.}
\end{align*}\]
Recursion via initiality

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

\[
\begin{align*}
1 + \text{nat} & \xrightarrow{\text{id+iseven}} 1 + 2 \\
[0,\text{succ}] & \downarrow \\
\text{nat} & \xrightarrow{\text{iseven}} 2 \\
\end{align*}
\]

\[
\begin{aligned}
\{ 
\text{iseven}(0) &= \text{true} \\
\text{iseven}(\text{succ}(n)) &= \neg\text{b}(\text{iseven}(n)) \\
\} 
\end{aligned}
\]
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 \(c : X \rightarrow F(X)\).
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 \(c : X \to F(X)\).

\[
F(X) = A \times X \quad \text{stream}_A \xrightarrow{(\text{hd}, \text{tl})} A \times \text{stream}_A
\]
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:

\[
\begin{array}{c}
X \xrightarrow{f} Z \\
\downarrow d \quad \downarrow c \\
F(X) \xrightarrow{F(f)} F(Z)
\end{array}
\]
Corecursion via finality

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’)).

stream

(hd,tl)

nat × stream
Corecursion via finality

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

\[
\text{stream } \times \text{stream} \xrightarrow{\text{add}} \text{stream}
\]

\[
\begin{align*}
\text{nat } \times \text{stream} & \quad \text{(hd,tl)}
\end{align*}
\]
Corecursion via finality

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

\begin{center}
\begin{tikzpicture}
  \node (stream) at (0,0) {\textit{stream \times \textit{stream}}};
  \node (out) at (6,0) {\textit{stream}};
  \draw[->] (stream) -- node[above] {\textit{add}} (out);
  \draw[->] (stream) -- node[below] {\textit{nat \times \textit{stream} \times \textit{stream}}} (out);
  \draw[->] (stream) -- node[below] {\textit{nat \times \textit{stream}}} (out);
\end{tikzpicture}
\end{center}
CoFixpoint add (s s' : stream) : stream :=
  (hd s + hd s') ::: (add (tl s) (tl s')).

\[\text{stream} \times \text{stream} \xrightarrow{\text{add}} \text{stream}\]

\[\lambda s s'. (\text{hd}(s) + \text{hd}(s'), \text{tl}(s), \text{tl}(s'))\]

\[\text{nat} \times \text{stream} \times \text{stream}\]
Corecursion via finality

\textbf{CoFixpoint} \texttt{add} (s s' : \texttt{stream}) : \texttt{stream} :=
\hspace{1cm}
(hd s + hd s') ::: (\texttt{add} (tl s) (tl s')).

\[
\begin{array}{c}
\text{stream} \times \text{stream} \ \overset{\text{add}}{\longrightarrow} \ \text{stream} \\
\lambda \ s \ s'. \ (\text{hd}(s) + \text{hd}(s'), \text{tl}(s), \text{tl}(s')) \downarrow \\
\text{nat} \times \text{stream} \times \text{stream} \ \overset{\text{id} \times \text{add}}{\longrightarrow} \ \text{nat} \times \text{stream} \\
\begin{cases}
\text{hd}(\text{add}(s, s')) = \text{hd}(s) + \text{hd}(s') \\
\text{tl}(\text{add}(s, s')) = \text{add}(\text{tl}(s), \text{tl}(s'))
\end{cases}
\end{array}
\]
Inductive types = initial algebras ($\mu$):

$\text{nat} = \mu X. 1 + X$ 

$\text{list} A = \mu X. 1 + A \times X$ 

$\text{btree} A = \mu X. A + X$ 

Coinductive types = final coalgebras ($\nu$):

$\text{stream} A = \nu X. A \times X$ 

$\text{colist} A = \nu X. 1 + A \times X$ 

$\text{fbtree} A = \nu X. A \times \text{list}(X)$ 

potentially infinite lists

finitely branching trees
Examples

Inductive types = initial algebras (μ):

▶ nat = μX. 1 + X
Examples

Inductive types = initial algebras ($\mu$):

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

Inductive types = initial algebras ($\mu$):

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

Inductive types = initial algebras ($\mu$) :

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

Coinductive types = final coalgebras ($\nu$) :

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

Inductive types = initial algebras ($\mu$):

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

Coinductive types = final coalgebras ($\nu$):

- $\text{stream}_A = \nu X. \ A \times X$
- $\text{colist}_A = \nu X. \ 1 + A \times X$
- $\text{fbtree}_A = \nu X. \ A \times \text{list}(X)$ finitely branching trees
Examples

Inductive types = initial algebras ($\mu$):

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

Coinductive types = final coalgebras ($\nu$):

- $\text{stream}_A = \nu X. \ A \times X$
- $\text{colist}_A = \nu X. \ 1 + A \times X$  potentially infinite lists
Inductive types = initial algebras ($\mu$) :
▶ $\text{nat} = \mu X. \ 1 + X$
▶ $\text{list}_A = \mu X. \ 1 + A \times X$
▶ $\text{btree}_A = \mu X. \ A + X^2$ binary trees with leaves labelled in $A$

Coinductive types = final coalgebras ($\nu$) :
▶ $\text{stream}_A = \nu X. \ A \times X$
▶ $\text{colist}_A = \nu X. \ 1 + A \times X$ potentially infinite lists
▶ $\text{fbtree}_A = \nu X. \ A \times \text{list}(X)$ finitely branching trees
Table of Contents

1. Some limits of (co)inductive types in Coq
2. (Co)inductive types, categorically
3. Polynomial functors
4. Coq implementation: use of setoids
Polynomial functors

Not all functors have an initial algebra/final coalgebra.

▶ in Coq: strict positivity condition
Not all functors have an initial algebra/final coalgebra.

- in Coq: strict positivity condition
- in category theory: polynomial functors
Not all functors have an initial algebra/final coalgebra.  

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

\[ P, Q ::= \text{id} \mid \text{cst}_S \mid P + Q \mid P \times Q \mid P^S \]
Definition (Container)

A container is a pair noted \((A \searrow B)\) where \(A : \text{Type}\) and
\(B : A \rightarrow \text{Type}\).
Definitions

Definition (Container)

A container is a pair noted \((A \rhd B)\) where \(A : \text{Type}\) and \(B : A \rightarrow \text{Type}\).

Definition (Polynomial functor)

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

\[
P(X) = \sum_{a:A} X^{B(a)}
\]
**Definition (Container)**

A *container* is a pair noted \((A ▷ B)\) where \(A : \text{Type}\) and \(B : A \rightarrow \text{Type}\).

**Definition (Polynomial functor)**

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

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

\[
\begin{array}{c}
  a \\
  \downarrow b_1 \\
  \downarrow b_2 \\
  \downarrow b_n \\
  \hline
  x_1
  x_2
  \ldots
  x_n
  \ldots
\end{array}
\]

\[a : A\]

\[f : B(a) \rightarrow X\]
Examples

\[ A : \text{Type} \]
\[ B : A \rightarrow \text{Type} \]
\[ P(X) = \sum_{a : A} X^{B(a)} \]
Examples

\[ A : \text{Type} \]
\[ B : A \to \text{Type} \]

\[ P(X) = X \times X \]

\[ P(X) = \sum_{a : A} X^{B(a)} \]
### Examples

$$A : \text{Type}$$
$$B : A \rightarrow \text{Type}$$

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

<table>
<thead>
<tr>
<th>$P(X)$</th>
<th>$(1 \triangleright \star \mapsto 2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(X) = X \times X$</td>
<td></td>
</tr>
<tr>
<td>$P(X) = \text{option}(X) \simeq 1 + X$</td>
<td>$\left(2 \triangleright \begin{array}{c} \text{true} \mapsto 0 \ \text{false} \mapsto 1 \end{array} \right)$</td>
</tr>
</tbody>
</table>
Examples

\[
\begin{align*}
A &: \text{Type} \\
B &: A \to \text{Type} \\
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>(P(X))</th>
<th>(\text{Table})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P(X) = X \times X)</td>
<td>(1 (\triangleright) (*) (\mapsto) 2)</td>
</tr>
<tr>
<td>(P(X) = \text{option}(X) \approx 1 + X)</td>
<td>(2 (\triangleright) (\text{true}) (\mapsto) 0)</td>
</tr>
<tr>
<td>(P(X) = \text{list}(X) \approx \sum_{n:\text{nat}} X^n)</td>
<td>(nat (\triangleright) (n) (\mapsto) (n))</td>
</tr>
</tbody>
</table>
Closure properties

Polynomial functors are closed under:

▶ sum
▶ product
▶ composition
▶ \((\mu \text{ and } \nu \text{ in the multivariate case})\)

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.
Context \((A : \text{Type}) \ (B : A \rightarrow \text{Type})\).

Inductive \(W : \text{Type} :=\)
| sup \((a : A) \ (f : B\ a \rightarrow W) : W\).
W-types and M-types

Context \( (A : \text{Type}) \) \( (B : A \to \text{Type}) \).

Inductive \( W : \text{Type} \) :=
\( \mid \text{sup} \ (a : A) \ (f : B \ a \to W) : W. \)

- well-founded trees: \( W = \mu X. P(X) \)
W-types and M-types

Context $(A : Type) (B : A \to Type)$.
Inductive $W : Type :=$
\[| \text{sup } (a : A) (f : B a \to W) : W. \]

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

Context \((A : \text{Type}) (B : A \rightarrow \text{Type})\).

Inductive \(W : \text{Type} := \)

| sup \((a : A) (f : B a \rightarrow W) : W\).

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

“one (co)inductive to rule them all”
1. Some limits of (co)inductive types in Coq

2. (Co)inductive types, categorically

3. Polynomial functors

4. Coq implementation: use of setoids
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]
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]
Types $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ Extensional functions

**Definition (Extensional function)**

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

Types \( \rightarrow \) 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 \( \rightarrow \) Extensional functions

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

\[
\begin{array}{ccc}
0 & 1 & 2 \\
+ & x & \sqrt{}
\end{array}
\]
Solution: shift of category

Types $\rightarrow$ 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 $\rightarrow$ Extensional functions

**Definition (Extensional function)**

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

\[\begin{array}{c c}
0 & 1 & 2 & + & \times & \sqrt \\
A & \rightarrow & \text{Type} & \sum & \prod & ?
\end{array}\]
what is a setoid family on $A$?
what is a setoid family on $A$?

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

$B : A \rightarrow \text{Setoid}$

$a \equiv a' \implies B(a) \simeq B(a')$
Setoid families

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

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

$$
\begin{array}{c}
 a \equiv a' \\
\downarrow p \downarrow \\
 B(a) \simeq B(a')
\end{array}
$$
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
- extensional
- 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

(implmented in the code)
1. Define a **PFUNCTOR** using the provided DSL.

   ```coq
   Definition P := nat * X.
   ```
1. Define a \texttt{PFUNCTOR} using the provided DSL.

   \texttt{Definition P := nat * X.}

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

   \texttt{Definition stream\textunderscore coalg := nu\textunderscore coalg P.}
   \texttt{Definition stream := coalg\textunderscore car stream\textunderscore coalg.}
   \texttt{Definition final\textunderscore stream\textunderscore coalg : final stream\textunderscore coalg := final\textunderscore nu P.}
1. Define a \texttt{PFUNCTOR} using the provided DSL.

\begin{verbatim}
Definition P := nat * X.
\end{verbatim}

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

\begin{verbatim}
Definition stream_coalg := nu_coalg P.
Definition stream := coalg_car stream_coalg.
Definition final_stream_coalg : final stream_coalg := final_nu P.
\end{verbatim}

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

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

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

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

5. A (co)induction principle is provided for proofs.
3. The (Co)Lambek lemma provides a constructor and a destructor.

\[
\text{Definition iso_fix : stream } \leadsto \text{nat } \ast \text{ stream } := \\
\text{CoLambek final_stream_coalg.}
\]

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

\[
\text{Definition stream_corec (X : Setoid) (c : X } \rightarrow \text{nat } \ast \text{ X) } : X \rightarrow \text{stream } := \\
\text{corec final_stream_coalg c.}
\]

\[
\text{Definition add : stream } \ast \text{ stream } \rightarrow \text{stream } := \\
\text{stream_corec (stream } \ast \text{ stream) } \\
(\text{fun s s'} \Rightarrow (\text{hd s } + \text{ hd s'}, (\text{tl s, tl s'}))).
\]
3. The (Co)Lambek lemma provides a constructor and a destructor.

\[
\text{Definition iso\_fix : stream} \cong \text{nat} \times \text{stream} := \text{CoLambek}\ \text{final\_stream\_coalg}.
\]

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

\[
\text{Definition stream\_corec (X : Setoid) (c : X \to \text{nat} \times X) : X \to \text{stream} := corec final\_stream\_coalg c.}
\]

\[
\text{Definition add : stream} \times \text{stream} \to \text{stream} := \text{stream\_corec (stream} \times \text{stream) (fun s s' => (hd s + hd s', (tl s, tl s'))).}
\]

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 $\rightarrow$ multivariate polynomial functors
Future work

- automation and syntactic sugar
- nested and mixed inductive-coinductive types → multivariate polynomial functors
- mutually defined (co)inductive types → 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