Efficient Mendler-Style Lambda-Encodings in Cedille

Denis Firsov, Richard Blair, and Aaron Stump

Department of Computer Science
The University of Iowa

February 23, 2019
Background

- It is **possible** to encode inductive datatypes in pure type theory.
Background

- It is **possible** to encode inductive datatypes in pure type theory.
- Church-style encoding of natural numbers

\[
cNat \triangleleft \star = \forall X : \star. (X \to X) \to X \to X.
\]

\[
cZ \triangleleft cNat = \Lambda X. \lambda s. \lambda z. z.
\]

\[
cS \triangleleft cNat \to cNat = \lambda n. \Lambda X. \lambda s. \lambda z. s (n s z).
\]
It is **possible** to encode inductive datatypes in pure type theory.

Church-style encoding of natural numbers

\[
\text{cNat} \leftarrow \ast = \forall X : \ast. (X \rightarrow X) \rightarrow X \rightarrow X.
\]

\[
cZ \leftarrow \text{cNat} = \Lambda X. \lambda s. \lambda z. z.
\]

\[
cS \leftarrow \text{cNat} \rightarrow \text{cNat} = \lambda n. \Lambda X. \lambda s. \lambda z. s (n s z).
\]

Essentially, we identify each natural number \(n\) with its iterator \(\lambda s. \lambda z. s^n z\).

\[
two := \text{cS} (\text{cS} \text{cZ}) = \lambda s. \lambda z. s (s z).
\]
Background

- At the same time, it is provably impossible to derive induction principle in the second-order dependent type theory (Geuvers, 2001).
At the same time, it is provably impossible to derive induction principle in the second-order dependent type theory (Geuvers, 2001). Moreover, it is provably impossible to implement a constant-time predecessor function for cNat (Parigot, 1989).

\[
two := \text{cS (cS Z)} := \lambda s. \lambda z. s (s z).
\]

\[
three := \text{cS (cS (cS Z))} := \lambda s. \lambda z. s (s (s z)).
\]
At the same time, it is provably **impossible** to derive induction principle in the second-order dependent type theory (Geuvers, 2001).

Moreover, it is provably **impossible** to implement a constant-time predecessor function for \(c\text{Nat}\) (Parigot, 1989).

\[
two := cS (cS Z) := \lambda s. \lambda z. s (s z).
\]

\[
three := cS (cS (cS Z)) := \lambda s. \lambda z. s (s (s z)).
\]

As a consequence, most languages come with built-in infrastructure for defining inductive datatypes (data definition, pattern-matching, termination checker, negativity and strictness check, etc.).

```agda
data Nat : Set where
  zero : Nat
  suc : Nat \rightarrow Nat
pred : Nat \rightarrow Nat
  pred zero = zero
  pred (suc n) = n
```
Background

- At the same time, it is provably **impossible** to derive induction principle in the second-order dependent type theory (Geuvers, 2001).
- Moreover, it is provably **impossible** to implement a constant-time predecessor function for \( \text{cNat} \) (Parigot, 1989).

\[
\begin{align*}
two & := \text{cS} (\text{cS} \text{Z}) := \lambda s. \lambda z. s (s z). \\
three & := \text{cS} (\text{cS} (\text{cS} \text{Z})) := \lambda s. \lambda z. s (s (s z)).
\end{align*}
\]

- As a consequence, most languages come with built-in infrastructure for defining inductive datatypes (data definition, pattern-matching, termination checker, negativity and strictness check, etc.).

```agda
data Nat : Set where  
  zero : Nat  
  suc : Nat → Nat  
  pred : Nat → Nat  
  pred zero = zero  
  pred (suc n) = n
```

- In Agda, induction principle can be derived by pattern matching and explicit structural recursion.
Is it possible to extend CC with some **typing constructs** to derive induction and implement constant-time predecessor (destructor) function for some linear-space encoding of natural numbers (inductive datatypes)?
Is it possible to extend CC with some **typing constructs** to derive induction and implement constant-time predecessor (destructor) function for some linear-space encoding of natural numbers (inductive datatypes)?

The solution is provided by Mendler-style encoding and *The Calculus of Dependent Lambda Eliminations (CDLE)* (A. Stump, JFP 2017).
Background

- Is it possible to extend CC with some **typing constructs** to derive induction and implement constant-time predecessor (destructor) function for some linear-space encoding of natural numbers (inductive datatypes)?
- The solution is provided by Mendler-style encoding and *The Calculus of Dependent Lambda Eliminations (CDLE)* (A. Stump, JFP 2017).
- CDLE adds three typing constructs to the Curry-style Calculus of Constructions:
Is it possible to extend CC with some **typing constructs** to derive induction and implement constant-time predecessor (destructor) function for some linear-space encoding of natural numbers (inductive datatypes)?

The solution is provided by Mendler-style encoding and *The Calculus of Dependent Lambda Eliminations (CDLE)* (A. Stump, JFP 2017).

CDLE adds three typing constructs to the Curry-style Calculus of Constructions:

1. dependent intersection types,
Is it possible to extend CC with some **typing constructs** to derive induction and implement constant-time predecessor (destructor) function for some linear-space encoding of natural numbers (inductive datatypes)?

The solution is provided by Mendler-style encoding and *The Calculus of Dependent Lambda Eliminations (CDLE)* (A. Stump, JFP 2017).

CDLE adds three typing constructs to the Curry-style Calculus of Constructions:

1. dependent intersection types,
2. implicit products,
Background

- Is it possible to extend CC with some **typing constructs** to derive induction and implement constant-time predecessor (destructor) function for some linear-space encoding of natural numbers (inductive datatypes)?
- The solution is provided by Mendler-style encoding and *The Calculus of Dependent Lambda Eliminations (CDLE)* (A. Stump, JFP 2017).
- CDLE adds three typing constructs to the Curry-style Calculus of Constructions:
  1. dependent intersection types,
  2. implicit products,
  3. primitive heterogeneous equality.
Background

- Is it possible to extend CC with some **typing constructs** to derive induction and implement constant-time predecessor (destructor) function for some linear-space encoding of natural numbers (inductive datatypes)?
- The solution is provided by Mendler-style encoding and *The Calculus of Dependent Lambda Eliminations (CDLE)* (A. Stump, JFP 2017).
- CDLE adds three typing constructs to the Curry-style Calculus of Constructions:
  1. dependent intersection types,
  2. implicit products,
  3. primitive heterogeneous equality.
- Cedille is an implementation of CDLE type theory (in Agda!).
Extension: Dependent intersection types

- **Formation**
  \[
  \Gamma \vdash T : \star \quad \Gamma, x : T \vdash T' : \star
  \]
  \[
  \frac{}{\Gamma \vdash \nu x : T \cdot T' : \star}
  \]

- **Introduction**
  \[
  \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : [t_1/x]T' \quad \Gamma \vdash p : t_1 \simeq t_2
  \]
  \[
  \frac{}{\Gamma \vdash [t_1, t_2\{p\}] : \nu x : T \cdot T'}
  \]

- **Elimination**
  \[
  \Gamma \vdash t : \nu x : T \cdot T'
  \]
  \[
  \frac{}{\Gamma \vdash t.1 : T \quad \text{first view}} \quad \Gamma \vdash t : \nu x : T \cdot T'
  \]
  \[
  \frac{}{\Gamma \vdash t.2 : [t.1/x]T' \quad \text{second view}}
Extension: Dependent intersection types

- **Formation**
  \[
  \Gamma \vdash T : \star \quad \Gamma, x : T \vdash T' : \star \\
  \Gamma \vdash \nu x : T \cdot T' : \star 
  \]

- **Introduction**
  \[
  \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : [t_1/x]T' \quad \Gamma \vdash p : t_1 \simeq t_2 \\
  \Gamma \vdash [t_1, t_2\{p\}] : \nu x : T \cdot T' 
  \]

- **Elimination**
  \[
  \Gamma \vdash t : \nu x : T \cdot T' \\
  \Gamma \vdash t.1 : T \\
  \text{first view} \\
  \Gamma \vdash t.2 : [t.1/x]T' \\
  \text{second view} 
  \]

- **Erasure**
  \[
  |[t_1, t_2\{p\}]| = |t_1| \\
  |t.1| = |t| \\
  |t.2| = |t| 
  \]
Extension: Implicit products

- Formation
  \[ \Gamma, x : T' \vdash T : \star \]
  \[ \Gamma \vdash \forall x : T'. T : \star \]

- Introduction
  \[ \Gamma, x : T' \vdash t : T \quad x \not\in \text{FV}(|t|) \]
  \[ \Gamma \vdash \land x : T'. t : \forall x : T'. T \]

- Elimination
  \[ \Gamma \vdash t : \forall x : T'. T \quad \Gamma \vdash t' : T' \]
  \[ \Gamma \vdash t - t' : [t'/x]T \]
Extension: Implicit products

- **Formation**
  \[ \Gamma, x : T' \vdash T : \star \]
  \[ \Gamma \vdash \forall x : T'. T : \star \]

- **Introduction**
  \[ \Gamma, x : T' \vdash t : T \quad x \notin FV(|t|) \]
  \[ \Gamma \vdash \Lambda x : T'. t : \forall x : T'. T \]

- **Elimination**
  \[ \Gamma \vdash t : \forall x : T'. T \quad \Gamma \vdash t' : T' \]
  \[ \Gamma \vdash t - t' : [t'/x]T \]

- **Erasure**
  \[ |\Lambda x : T. t| = |t| \]
  \[ |t - t'| = |t| \]
Extension: Equality

- Formation rule
  \[ \Gamma \vdash t : T \quad \Gamma \vdash t' : T' \]
  \[ \Gamma \vdash t \simeq t' : \star \]

- Introduction
  \[ \Gamma \vdash t : T \]
  \[ \Gamma \vdash \beta : t \simeq t \]

- Elimination
  \[ \Gamma \vdash t' : t_1 \simeq t_2 \quad \Gamma \vdash t : [t_1/x]T \]
  \[ \Gamma \vdash \rho \ t' \rightarrow t : [t_2/x]T \]
Extension: Equality

- **Formation rule**
  \[
  \Gamma \vdash t : T \quad \Gamma \vdash t' : T' \\
  \Gamma \vdash t \simeq t' : \star
  \]

- **Introduction**
  \[
  \Gamma \vdash t : T \\
  \Gamma \vdash \beta : t \simeq t
  \]

- **Elimination**
  \[
  \Gamma \vdash t' : t_1 \simeq t_2 \quad \Gamma \vdash t : [t_1/x]T \\
  \Gamma \vdash \rho~t' - t : [t_2/x]T
  \]

- **Erasure**
  \[
  |\beta| = \lambda x. x \\
  |\rho~t' - t| = |t|
  \]
Categorically, inductive datatypes are modelled as initial $F$-algebras.

Mendler-style $F$-algebra is a pair of object $(\text{carrier} \ X)$ and a natural transformation $\text{C}(\ - \ , \ X) \rightarrow \text{C}(F\ - \ , \ X)$.

In Cedille, objects are types and natural transformations are polymorphic functions:

$$\text{AlgM} \triangleleft \star 
\rightarrow \star = \lambda X : \star. \forall R : \star. (R \rightarrow X) \rightarrow F\ R \rightarrow X.$$  

The object (a type) of initial Mendler-style $F$-algebra is a least fixed point of $F$:

$$\text{FixM} \triangleleft \star = \forall X : \star. \text{AlgM} X \rightarrow X.$$ 

There is a homomorphism from the carrier of initial algebra to the carrier of any other algebra (gives weak initiality):

$$\text{foldM} \triangleleft \forall X : \star. \text{AlgM} X \rightarrow \text{FixM} \rightarrow X = \langle . . \rangle.$$ 

Constructors are expressed as a Church-style algebra:

$$\text{inM} \triangleleft F\ \text{FixM} \rightarrow \text{FixM} = \lambda v. \lambda \text{alg. alg (foldM alg) v}.$$
Mendler-style inductive datatypes

- Categorically, inductive datatypes are modelled as initial F-algebras.
- Mendler-style F-algebra is a pair of object (carrier) $X$ and a natural transformation $\mathcal{C}(-, X) \rightarrow \mathcal{C}(F - , X)$.

In Cedille, objects are types and natural transformations are polymorphic functions:

$$\text{AlgM} \triangleright \star \rightarrow \star = \lambda X : \star. \forall R : \star. (R \rightarrow X) \rightarrow F R \rightarrow X.$$
Mendler-style inductive datatypes

- Categorically, inductive datatypes are modelled as initial F-algebras.
- Mendler-style F-algebra is a pair of object (\textit{carrier}) \( X \) and a natural transformation \( \mathcal{C}(-, X) \rightarrow \mathcal{C}(F - , X) \).
- In Cedille, objects are types and natural transformations are polymorphic functions:

\[
\text{AlgM} : \star \rightarrow \star = \lambda X : \star. \ \forall R : \star. \ (R \rightarrow X) \rightarrow F R \rightarrow X.
\]
Mendler-style inductive datatypes

- Categorically, inductive datatypes are modelled as initial F-algebras.
- Mendler-style F-algebra is a pair of object (carrier) $X$ and a natural transformation $\mathcal{C}(\_, X) \to \mathcal{C}(F \_, X)$.
- In Cedille, objects are types and natural transformations are polymorphic functions:
  \[
  \text{AlgM} \ni \star \to \star = \lambda X : \star. \forall R : \star. (R \to X) \to F R \to X.
  \]
- The object (a type) of initial Mendler-style F-algebra is a least fixed point of $F$:
  \[
  \text{FixM} \ni \star = \forall X : \star. \text{AlgM} X \to X.
  \]
Mendler-style inductive datatypes

- Categorically, inductive datatypes are modelled as initial F-algebras.
- Mendler-style F-algebra is a pair of object (carrier) $X$ and a natural transformation $\mathcal{C}(-, X) \to \mathcal{C}(F - , X)$.
- In Cedille, objects are types and natural transformations are polymorphic functions:

  \[
  \text{AlgM} \sqcap \star \to \star = \lambda X : \star. \forall R : \star. (R \to X) \to F R \to X.
  \]

- The object (a type) of initial Mendler-style F-algebra is a least fixed point of $F$:

  \[
  \text{FixM} \sqcap \star = \forall X : \star. \text{AlgM} X \to X.
  \]

- There is a homomorphism from the carrier of initial algebra to the carrier of any other algebra (gives weak initiality):

  \[
  \text{foldM} \sqcap \forall X : \star. \text{AlgM} X \to \text{FixM} \to X = <..>
  \]
Mendler-style inductive datatypes

Categorically, inductive datatypes are modelled as initial $F$-algebras.

Mendler-style $F$-algebra is a pair of object (carrier) $X$ and a natural transformation $\mathcal{C}(\lambda X, X) \to \mathcal{C}(F \lambda X, X)$.

In Cedille, objects are types and natural transformations are polymorphic functions:

\[
\text{AlgM} \triangleleft * \to * = \lambda X : *. \forall R : *. (R \to X) \to F R \to X.
\]

The object (a type) of initial Mendler-style $F$-algebra is a least fixed point of $F$:

\[
\text{FixM} \triangleleft * = \forall X : *. \text{AlgM} X \to X.
\]

There is a homomorphism from the carrier of initial algebra to the carrier of any other algebra (gives weak initiality):

\[
\text{foldM} \triangleleft \forall X : *. \text{AlgM} X \to \text{FixM} \to X = <..>
\]

Constructors are expressed as a Church-style algebra:

\[
\text{inM} \triangleleft F \text{FixM} \to \text{FixM} = \lambda v. \lambda \text{alg. alg (foldM alg)} v.
\]
There is no induction principle for $\text{FixM}$. 

- We define a type $\text{FixIndM}$ as an inductive subset of $\text{FixM}$:

$$\text{FixIndM} \triangleq \exists \star \equiv \iota x : \text{FixM}. \text{Inductive } x.$$ 

- For $\text{FixIndM}$ to be inhabited, we must express an inductivity predicate so that the value $x : \text{FixM}$ and the proof $p : \text{Inductive } x$ are equal.

$$\text{FixM} \triangleq \forall X : \star. \text{AlgM } X \rightarrow X.$$ 

- Mendler-style proof-algebras $\text{AlgM}$:

$$\text{AlgM} \triangleq \lambda X. \forall R : \star. (R \rightarrow X) \rightarrow \text{F R} \rightarrow X.$$ 

- Proof-algebras $\text{PrfAlgM}$:

$$\text{PrfAlgM} \triangleq \lambda A : \star. (A \rightarrow \star) \rightarrow (\text{F A} \rightarrow A) \rightarrow \star.$$ 

- We define $\text{PrfAlgM}$ as:

$$\text{PrfAlgM} \triangleq \lambda A. \lambda Q. \lambda \text{alg}. \forall R : \star. \forall c : R \rightarrow A. \forall e : (\Pi r : R. c r \simeq r). (\Pi r : R. Q (c r)) \rightarrow \Pi \text{fr : F R. Q (alg (fmap c fr))).}$$
There is no induction principle for FixM.

We define a type FixIndM as an inductive subset of FixM:

\[
\text{FixIndM} \triangleleft \star = \iota \; x : \text{FixM}. \text{Inductive } x.
\]
There is no induction principle for \( \text{FixM} \).

We define a type \( \text{FixIndM} \) as an inductive subset of \( \text{FixM} \):
\[
\text{FixIndM} = \{ x : \text{FixM} . \text{Inductive } x \}.
\]

For \( \text{FixIndM} \) to be inhabited, we must express an inductivity predicate so that the value \( x : \text{FixM} \) and the proof \( p : \text{Inductive } x \) are equal.

\[
\text{FixM} = \forall X : \star . \text{AlgM } X \rightarrow X.
\]

\[
\text{Inductive } \text{FixM} \rightarrow \star = \lambda x : \text{FixM} .
\]
\[
\forall Q : \text{FixM} \rightarrow \star . \text{PrfAlgM } \text{FixM } Q \text{ inM } \rightarrow Q x.
\]
There is no induction principle for FixM.

We define a type FixIndM as an inductive subset of FixM:

\[
\text{FixIndM} \triangleleft \star = \iota \ x : \text{FixM}. \text{Inductive } x.
\]

For FixIndM to be inhabited, we must express an inductivity predicate so that the value \( x : \text{FixM} \) and the proof \( p : \text{Inductive } x \) are equal.

\[
\text{FixM} \triangleleft \star = \forall \ X : \star. \text{AlgM } X \to X.
\]

Inductive \( \triangleleft \text{FixM} \to \star = \lambda \ x : \text{FixM}. \forall \ Q : \text{FixM} \to \star. \ \text{PrfAlgM } \text{FixM } Q \ \text{inM} \to Q \ x. \)

Mendler-style proof-algebras

\[
\text{AlgM} \triangleleft \star \to \star = \lambda \ X. \forall \ R : \star. (R \to X) \to F \ R \to X.
\]
There is no induction principle for \( \text{FixM} \).

We define a type \( \text{FixIndM} \) as an inductive subset of \( \text{FixM} \):

\[
\text{FixIndM} \triangleq \forall x : \text{FixM}. \text{Inductive} \ x.
\]

For \( \text{FixIndM} \) to be inhabited, we must express an inductivity predicate so that the value \( x : \text{FixM} \) and the proof \( p : \text{Inductive} \ x \) are equal.

\[
\text{FixM} \triangleq \forall X : \star. \text{AlgM} \ X \to X.
\]

\[
\text{Inductive} \triangleq \text{FixM} \to \star = \lambda x : \text{FixM}. \\
\forall Q : \text{FixM} \to \star. \text{PrfAlgM} \ \text{FixM} \ Q \ \text{inM} \to Q \ x.
\]

Mendler-style proof-algebras

\[
\text{AlgM} \triangleq \star \to \star = \lambda X. \forall R : \star. (R \to X) \to F R \to X.
\]

\[
\text{PrfAlgM} \triangleq \Pi A : \star. (A \to \star) \to (F A \to A) \to \star \\
= \lambda A. \lambda Q. \lambda alg. \\
\forall R : \star. \forall c : R \to A. \forall e : (\Pi r : R. c r \simeq r). \\
(\Pi r : R. Q (c r)) \to \\
\Pi fr : F R. Q (\text{alg} (\text{fmap} \ c \ fr)).
\]
Mendler-style induction principle

- The collection of constructors of type FixIndM is expressed by Church-algebra

\[
in\text{FixIndM} \leftrightarrow F \text{FixIndM} \rightarrow \text{FixIndM} = <..>
\]
Mendler-style induction principle

- The collection of constructors of type $\text{FixIndM}$ is expressed by Church-algebra
  \[
  \text{inFixIndM} \triangleleft F \text{FixIndM} \rightarrow \text{FixIndM} = \langle..\rangle
  \]

- Induction principle
  \[
  \text{induction} \triangleleft \forall Q : \text{FixIndM} \rightarrow \star.
  \quad \text{PrfAlgM} \text{FixIndM} \ Q \ \text{inFixIndM} \rightarrow
  \quad \prod x : \text{FixIndM}. \ Q x = \langle..\rangle
  \]
Mendler-style induction principle

- The collection of constructors of type FixIndM is expressed by Church-algebra
  \[ \text{inFixIndM} \iff F \text{FixIndM} \rightarrow \text{FixIndM} = \langle \ldots \rangle \]
- Induction principle
  \[ \text{induction} \iff \forall Q : \text{FixIndM} \rightarrow \star. \]
  \[ \text{PrfAlgM FixIndM Q inFixIndM} \rightarrow \]
  \[ \Pi x : \text{FixIndM}. \; Q x = \langle \ldots \rangle \]
- Cancellation law:
  \[ \text{indHom} \iff \forall Q \text{ palg x}. \]
  \[ \text{induction palg (inFixInd x)} \simeq \text{palg (induction palg)} x \]
  \[ = \Lambda Q. \; \Lambda \text{palg}. \; \Lambda x. \; \beta. \]
- Can we define a a proof-algebra which erases to lambda term \( \lambda x. \lambda y. \; y \)?
Constant-time destructor

\[
\text{outAlgM} \triangleq \text{PrfAlgM FixIndM} \ (\lambda \_ \cdot \ F \ \text{FixIndM}) \ \text{inFixIndM} = \Lambda \ R. \ \Lambda \ c. \ \Lambda \ e. \ \lambda \ x. \ \lambda \ y. \ \ [\ y, \ c \ y \ {\ e \ y} \ ].2.
\]
Constant-time destructor

- outAlgM ◁ PrfAlgM FixIndM (λ _. F FixIndM) inFixIndM
  = Λ R. Λ c. Λ e. λ x. λ y. [ y , c y { e y } ]].2.

- Finally, we arrive at the generic constant-time linear-space destructor of inductive datatypes:
  outFixIndM ◁ FixInd → F FixInd = induction outAlgM.
Constant-time destructor

- outAlgM $\triangleleft$ PrfAlgM FixIndM ($\lambda _\_. F$ FixIndM) inFixIndM
  $= \Lambda R. \Lambda c. \Lambda e. \lambda x. \lambda y. [y, c\ y\ {e\ y}]$.2.

- Finally, we arrive at the generic constant-time linear-space destructor of inductive datatypes:
  outFixIndM $\triangleleft$ FixInd $\rightarrow F$ FixInd $= \text{induction outAlgM}$.

- Since outFixIndM is constant-time then we get Lambek’s Lemma as an easy consequence
  lambek1 $\triangleleft \prod x: F$ FixInd. outFixIndM (inFixIndM x) $\simeq x$
  $= \lambda x. \beta$.

  lambek2 $\triangleleft \prod x: \text{FixIndM}. \text{inFixIndM (outFixIndM x)} \simeq x$
  $= \lambda x. \text{induction (}\Lambda R. \Lambda c. \Lambda e. \lambda ih. \lambda fr. \beta) x$. 
Example: Natural numbers

- Natural numbers arise as least fixed point of a scheme NF
  \[ \text{NF} \triangleleft * \rightarrow * = \lambda X : * \cdot \text{Unit} + X. \]

- Constructor `zero` for `Nat` is
  \[ \text{zero} \triangleleft \text{Nat} = \text{inFixIndM} \ (\text{in1} \ \text{unit}). \]

- Constructor `suc` for `Nat` is
  \[ \text{suc} \triangleleft \text{Nat} \rightarrow \text{Nat} = \lambda n. \text{inFixIndM} \ (\text{in2} \ n). \]

- Constructor `suc` has the following underlying lambda-term
  \[ \text{suc} \ n \approx \lambda \text{alg.} \ (\text{alg} \ (\lambda f. \ (f \ \text{alg}))) \ (\lambda i. \lambda j. \ (j \ n))). \]

- Constant-time predecessor
  \[ \text{pred} \triangleleft \text{Nat} \rightarrow \text{Nat} \]
  \[ = \lambda n. \text{case} \ (\text{outFixIndM} \ n) \ (\lambda _. \text{zero}) \ (\lambda m. \ m). \]
Identity mappings instead of functors

- The described developments are well-justified for any functor

\[
\text{Functor } \updownarrow (\star \to \star) \to \star = \lambda F. \\
\sum \text{fmap} : \forall X Y : \star. (X \to Y) \to F X \to F Y. \\
\text{IdentityLaw fmap} \times \text{CompositionLaw fmap}.
\]
Identity mappings instead of functors

- The described developments are well-justified for any functor
  \[ \text{Functor} \mapsto (\star \to \star) \to \star = \lambda F. \]
  \[ \Sigma \text{fmap} : \forall X Y : \star. (X \to Y) \to F X \to F Y. \]
  \[ \text{IdentityLaw fmap \times CompositionLaw fmap.} \]

- Surprisingly, the construction can be easily generalized to the larger class of schemes we call identity mappings
  \[ \text{IdMapping} \mapsto (\star \to \star) \to \star = \lambda F. \]
  \[ \forall X Y : \star. \text{Id} X Y \to \text{Id} (F X) (F Y). \]
Identity mappings instead of functors

- The described developments are well-justified for any functor
  \[ \text{Functor } \lambda (\star \to \star) \to \star = \lambda F. \]
  \[ \sum \text{fmap} : \forall X Y : \star. (X \to Y) \to F X \to F Y. \]
  \[ \text{IdentityLaw fmap } \times \text{CompositionLaw fmap}. \]

- Surprisingly, the construction can be easily generalized to the larger class of schemes we call identity mappings
  \[ \text{IdMapping } \lambda (\star \to \star) \to \star = \lambda F. \]
  \[ \forall X Y : \star. \text{Id} X Y \to \text{Id} (F X) (F Y). \]

- Every functor is identity mapping
  \[ \text{fm2im } \lambda \forall F : \star \to \star. \text{Functor} F \to \text{IdMapping} F = <..> \]
Identity mappings instead of functors

- The described developments are well-justified for any functor
  \[
  \text{Functor} \; \lambda F. \; \Sigma \text{fmap} : \forall X Y : \star. (X \rightarrow Y) \rightarrow F X \rightarrow F Y.
  \]
  IdentityLaw fmap \times CompositionLaw fmap.

- Surprisingly, the construction can be easily generalized to the larger class of schemes we call identity mappings
  \[
  \text{IdMapping} \; \lambda F. \; \forall X Y : \star. \; \text{Id} X Y \rightarrow \text{Id} (F X) (F Y).
  \]

- Every functor is identity mapping
  \[
  \text{fm2im} \; \forall F : \star \rightarrow \star. \; \text{Functor} F \rightarrow \text{IdMapping} F = <..>
  \]

- Converse is not true
  \[
  \text{UneqPair} \; \lambda X. \; \Sigma x_1 x_2 : X. \; x_1 \neq x_2.
  \]

- Identity mappings induce a large class of datatypes (including infinitary and non-strictly positive datatypes).
There is more!

- We generically define course-of-value datatypes and implement dependent histomorphisms. We do this by defining a least fixed point of a coend of “negative” scheme.

\[
\text{Lift} \downarrow (\star \to \star) \to \star \to \star = \lambda F. \lambda X. F X \times (X \to F X).
\]

\[
\text{FixCoV} \downarrow (\star \to \star) \to \star = \lambda F. \text{FixIndM} \left(\text{Coend} \ (\text{Lift} \ F)\right).
\]

- In a similar way, we generically derive (small) inductive-recursive datatypes and derive the respective dependent elimination.
Thank you!