Efficient Mendler-Style Lambda-Encodings in Cedille

Denis Firsov, Richard Blair, and Aaron Stump

Department of Computer Science
The University of Iowa

July 9, 2018

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 ◀ ⋆ = ∀ X : ⋆. (X → X) → X → X.

cZ ◀ cNat = Λ X. λ s. λ z. z.
cS ◀ cNat → cNat = λ n. Λ X. λ s. λ z. s (n s z).
```

Background

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

  $\texttt{cNat} \blacktriangleleft \star = \forall\ \texttt{X} : \star.\ (\texttt{X} \to \texttt{X}) \to \texttt{X} \to \texttt{X}.$

  $\texttt{cZ} \blacktriangleleft \texttt{cNat} = \Lambda\ \texttt{X}.\ \lambda\ \texttt{s}.\ \lambda\ \texttt{z}.\ \texttt{z}.$

  $\texttt{cS} \blacktriangleleft \texttt{cNat} \to \texttt{cNat} = \lambda\ \texttt{n}.\ \Lambda\ \texttt{X}.\ \lambda\ \texttt{s}.\ \lambda\ \texttt{z}.\ \texttt{s}\ (\texttt{n}\ \texttt{s}\ \texttt{z}).$
- Essentially, we identify each natural number n with its iterator $\lambda\ \texttt{s}.\ \lambda\ \texttt{z}.\ \texttt{s}^{\texttt{n}}\ \texttt{z}.$

  $\texttt{two} := \texttt{cS}\ (\texttt{cS}\ \texttt{cZ}) = \lambda\ \texttt{s}.\ \lambda\ \texttt{z}.\ \texttt{s}\ (\texttt{s}\ \texttt{z}).$

Background

- At the same time, it is provably **impossible** to derive induction principle in the second-order dependent type theory (Geuvers, 2001).

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 `cNat` (Parigot, 1989).

```
two   := cS (cS Z)       := λ s. λ z. s (s z).
three := cS (cS (cS Z)) := λ s. λ z. s (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).
- Moreover, it is provably **impossible** to implement a constant-time predecessor function for `cNat` (Parigot, 1989).

```
two   := cS (cS Z)      := λ s. λ z. s (s z).
three := cS (cS (cS Z)) := λ s. λ 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.).

```
data Nat : Set where                pred : Nat -> Nat
  zero : Nat                        pred zero = zero
  suc  : Nat → Nat                  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 `cNat` (Parigot, 1989).

```
two   := cS (cS Z)      := λ s. λ z. s (s z).
three := cS (cS (cS Z)) := λ s. λ 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.).

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

- In Agda, induction principle can be derived by pattern matching and explicit structural recursion.

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)?

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).

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:

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,

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,

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

$$\frac{\Gamma \vdash T : \star \quad \Gamma, x : T \vdash T' : \star}{\Gamma \vdash \iota\, x{:}T.\, T' : \star}$$

- Introduction

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : [t_1/x]T' \quad \Gamma \vdash p : t_1 \simeq t_2}{\Gamma \vdash [t_1, t_2\{p\}] : \iota\, x{:}T.\, T'}$$

- Elimination

$$\frac{\Gamma \vdash t : \iota\, x{:}T.\, T'}{\Gamma \vdash t.1 : T} \text{ first view} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T.\, T'}{\Gamma \vdash t.2 : [t.1/x]T'} \text{ second view}$$

Extension: Dependent intersection types

- Formation

$$\frac{\Gamma \vdash T : \star \quad \Gamma, x : T \vdash T' : \star}{\Gamma \vdash \iota\, x{:}T.\, T' : \star}$$

- Introduction

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : [t_1/x]T' \quad \Gamma \vdash p : t_1 \simeq t_2}{\Gamma \vdash [t_1, t_2\{p\}] : \iota\, x{:}T.\, T'}$$

- Elimination

$$\frac{\Gamma \vdash t : \iota\, x{:}T.\, T'}{\Gamma \vdash t.1 : T}\ \text{first view} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T.\, T'}{\Gamma \vdash t.2 : [t.1/x]T'}\ \text{second view}$$

- Erasure

$$\begin{array}{lcl} |[t_1, t_2\{p\}]| & = & |t_1| \\ |t.1| & = & |t| \\ |t.2| & = & |t| \end{array}$$

Extension: Implicit products

- Formation

$$\frac{\Gamma, x : T' \vdash T : \star}{\Gamma \vdash \forall x : T'. \, T : \star}$$

- Introduction

$$\frac{\Gamma, x : T' \vdash t : T \quad x \notin FV(|t|)}{\Gamma \vdash \Lambda x : T'. \, t : \forall x : T'. \, T}$$

- Elimination

$$\frac{\Gamma \vdash t : \forall x : T'. \, T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t \ - t' : [t'/x] T}$$

Extension: Implicit products

- Formation

$$\frac{\Gamma, x : T' \vdash T : \star}{\Gamma \vdash \forall x : T'. T : \star}$$

- Introduction

$$\frac{\Gamma, x : T' \vdash t : T \quad x \notin FV(|t|)}{\Gamma \vdash \Lambda x : T'. t : \forall x : T'. T}$$

- Elimination

$$\frac{\Gamma \vdash t : \forall x : T'. T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t \ - t' : [t'/x]T}$$

- Erasure

$$\begin{array}{lcl} |\Lambda x : T. t| & = & |t| \\ |t \ - t'| & = & |t| \end{array}$$

Extension: Equality

- Formation rule

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t \simeq t' : \star}$$

- Introduction

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \beta : t \simeq t}$$

- Elimination

$$\frac{\Gamma \vdash t' : t_1 \simeq t_2 \quad \Gamma \vdash t : [t_1/x]T}{\Gamma \vdash \rho\ t' \ - \ t : [t_2/x]T}$$

Extension: Equality

- Formation rule

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t \simeq t' : \star}$$

- Introduction

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \beta : t \simeq t}$$

- Elimination

$$\frac{\Gamma \vdash t' : t_1 \simeq t_2 \quad \Gamma \vdash t : [t_1/x]T}{\Gamma \vdash \rho\ t'\ -\ t : [t_2/x]T}$$

- Erasure

$$\begin{array}{lcl} |\beta| & = & \lambda x. x \\ |\rho\ t'\ -\ t| & = & |t| \end{array}$$

Mendler-style inductive datatypes

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

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 $\mathcal{C}(-, X) \to \mathcal{C}(F\ -, X)$.

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:

  ```
  AlgM ◀ ★ → ★ = λ X : ★. ∀ R : ★. (R → X) → F R → 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:

  `AlgM ◀ ⋆ → ⋆ = λ X : ⋆. ∀ R : ⋆. (R → X) → F R → X.`

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

  `FixM ◀ ⋆ = ∀ X : ⋆. AlgM X → 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:

  ```
  AlgM ◀ ⋆ → ⋆ = λ X : ⋆. ∀ R : ⋆. (R → X) → F R → X.
  ```

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

  ```
  FixM ◀ ⋆ = ∀ X : ⋆. AlgM X → X.
  ```

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

  ```
  foldM ◀ ∀ X : ⋆. AlgM X → FixM → 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:

```
AlgM ◀ ★ → ★ = λ X : ★. ∀ R : ★. (R → X) → F R → X.
```

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

```
FixM ◀ ★ = ∀ X : ★. AlgM X → X.
```

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

```
foldM ◀ ∀ X : ★. AlgM X → FixM → X = <..>
```

- Constructors are expressed as a Church-style algebra:

```
inM ◀ F FixM → FixM = λ v. λ alg. alg (foldM alg) v.
```

- There is no induction principle for `FixM`.

- There is no induction principle for `FixM`.
- We define a type `FixIndM` as an inductive subset of `FixM`:

  `FixIndM` ◀ $\star$ = ι `x : FixM. Inductive x.`

- There is no induction principle for `FixM`.
- We define a type `FixIndM` as an inductive subset of `FixM`:

  FixIndM ◀ ⋆ = ι x : FixM. Inductive x.
- For `FixIndM` to be inhabited, we must express an inductivity predicate so that the value `x : FixM` and the proof `p : Inductive x` are equal.

  FixM ◀ ⋆ = ∀ X : ⋆. AlgM X → X.

  Inductive ◀ FixM → ⋆ = λ x : FixM.
   ∀ Q : FixM → ⋆. PrfAlgM FixM Q inM → Q x.

- There is no induction principle for `FixM`.
- We define a type `FixIndM` as an inductive subset of `FixM`:

  FixIndM ◀ ⋆ = ι x : FixM. Inductive x.

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

  FixM ◀ ⋆ = ∀ X : ⋆. AlgM X → X.

  Inductive ◀ FixM → ⋆ = λ x : FixM.
   ∀ Q : FixM → ⋆. PrfAlgM FixM Q inM → Q x.

- Mendler-style proof-algebras

  AlgM ◀ ⋆ → ⋆ = λ X. ∀ R : ⋆. (R → X) → F R → X.

- There is no induction principle for `FixM`.
- We define a type `FixIndM` as an inductive subset of `FixM`:

  `FixIndM ◀ ⋆ = ι x : FixM. Inductive x.`
- For `FixIndM` to be inhabited, we must express an inductivity predicate so that the value `x : FixM` and the proof `p : Inductive x` are equal.

```
FixM ◀ ⋆ = ∀ X : ⋆. AlgM X → X.

Inductive ◀ FixM → ⋆ = λ x : FixM.
 ∀ Q : FixM → ⋆.  PrfAlgM FixM Q inM → Q x.
```

- Mendler-style proof-algebras

```
AlgM ◀ ⋆ → ⋆ = λ X. ∀ R : ⋆. (R → X) → F R → X.

PrfAlgM ◀ Π A : ⋆. (A → ⋆) → (F A → A) → ⋆
 = λ A. λ Q. λ alg.
   ∀ R : ⋆. ∀ c : R → A. ∀ e : (Π r : R. c r ≃ r).
   (Π r : R. Q (c r)) →
   Π fr : F R.  Q (alg (fmap c fr)).
```

Mendler-style induction principle

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

  `inFixIndM ◀ F FixIndM → FixIndM = <..>`

Mendler-style induction principle

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

```
inFixIndM ◀ F FixIndM → FixIndM = <..>
```

- Induction principle

```
induction ◀ ∀ Q : FixIndM → ⋆.
 PrfAlgM FixIndM Q inFixIndM →
 Π x : FixIndM. Q x = <..>
```

Mendler-style induction principle

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

  ```
  inFixIndM ◀ F FixIndM → FixIndM = <..>
  ```

- Induction principle

  ```
  induction ◀ ∀ Q : FixIndM → ⋆.
   PrfAlgM FixIndM Q inFixIndM →
   Π x : FixIndM. Q x = <..>
  ```

- Cancellation law:

  ```
  indHom ◀ ∀ Q palg x.
   induction palg (inFixInd x) ≃ palg (induction palg) x
   = Λ Q. Λ palg. Λ x. β.
  ```

- Can we define a a proof-algebra which erases to lambda term `λ x. λ y. y`?

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.`

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 ◂ 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.`
- Since `outFixIndM` is constant-time then we get Lambek's Lemma as an easy consequence

  ```
  lambek1 ◂ Π x: F FixInd. outFixIndM (inFixIndM x) ≃ x
   = λ x. β.

  lambek2 ◂ Π x: FixIndM. inFixIndM (outFixIndM x) ≃ x
   = λ x. induction (Λ R. Λ c. Λ e. λ ih. λ fr. β) x.
  ```

Example: Natural numbers

- Natural numbers arise as least fixed point of a scheme `NF`

```
NF ◀ ⋆ → ⋆ = λ X : ⋆. Unit + X.

Nat ◀ ⋆ = FixIndM NF.
```

- Constructors

```
zero ◀ Nat         =       inFixIndM (in1 unit).
suc  ◀ Nat → Nat = λ n. inFixIndM (in2 n).
```

- Constructor `suc` has the following underlying lambda-term
  `suc n ≃ λ alg. (alg (λ f. (f alg)) (λ i. λ j. (j n)))`.
- Constant-time predecessor

```
pred ◀ Nat → Nat
 = λ n. case (outFixIndM n) (λ _. zero) (λ m. m).
```

Identity mappings instead of functors

- The described developments are well-justified for any functor

```
Functor ◀ (⋆ → ⋆) → ⋆ = λ F.
  Σ fmap : ∀ X Y : ⋆. (X → Y) → F X → F Y.
  IdentityLaw fmap × CompositionLaw fmap.
```

Identity mappings instead of functors

- The described developments are well-justified for any functor

```
Functor ◀ (⋆ → ⋆) → ⋆ = λ F.
  Σ fmap : ∀ X Y : ⋆. (X → Y) → F X → F Y.
  IdentityLaw fmap × CompositionLaw fmap.
```

- Surprisingly, the construction can be easily generalized to the larger class of schemes we call **identity mappings**

```
IdMapping ◀ (⋆ → ⋆) → ⋆ = λ F.
  ∀ X Y : ⋆. Id X Y → Id (F X) (F Y).
```

Identity mappings instead of functors

- The described developments are well-justified for any functor

```
Functor ◀ (⋆ → ⋆) → ⋆ = λ F.
  Σ fmap : ∀ X Y : ⋆. (X → Y) → F X → F Y.
  IdentityLaw fmap × CompositionLaw fmap.
```

- Surprisingly, the construction can be easily generalized to the larger class of schemes we call **identity mappings**

```
IdMapping ◀ (⋆ → ⋆) → ⋆ = λ F.
  ∀ X Y : ⋆. Id X Y → Id (F X) (F Y).
```

- Every functor is identity mapping

```
fm2im ◀ ∀ F : ⋆ → ⋆. Functor F → IdMapping F = <..>
```

Identity mappings instead of functors

- The described developments are well-justified for any functor

```
Functor ◀ (⋆ → ⋆) → ⋆ = λ F.
  Σ fmap : ∀ X Y : ⋆. (X → Y) → F X → F Y.
  IdentityLaw fmap × CompositionLaw fmap.
```

- Surprisingly, the construction can be easily generalized to the larger class of schemes we call **identity mappings**

```
IdMapping ◀ (⋆ → ⋆) → ⋆ = λ F.
  ∀ X Y : ⋆. Id X Y → Id (F X) (F Y).
```

- Every functor is identity mapping

```
fm2im ◀ ∀ F : ⋆ → ⋆. Functor F → IdMapping F = <..>
```

- Converse is not true

```
UneqPair ◀ ⋆ → ⋆ = λ X. Σ x_1 x_2 : X. x_1 ≠ 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.

  $\texttt{Lift} \blacktriangleleft (\star \to \star) \to \star \to \star = \lambda\ \texttt{F}.\ \lambda\ \texttt{X}.\ \texttt{F X} \times (\texttt{X} \to \texttt{F X}).$

  $\texttt{FixCoV} \blacktriangleleft (\star \to \star) \to \star = \lambda\ \texttt{F}.\ \texttt{FixIndM (Coend (Lift F))}.$

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

Thank you!