Generic Derivation of Induction for Impredicative Encodings in Cedille

Denis Firsov
Department of Computer Science
The University of Iowa
Iowa City, IA, USA
denis-firsov@uiowa.edu

Aaron Stump
Department of Computer Science
The University of Iowa
Iowa City, IA, USA
aaron-stump@uiowa.edu

Abstract

This paper presents generic derivations of induction for impredicatively typed lambda-encoded datatypes, in the Cedille type theory. Cedille is a pure type theory extending the Curry-style Calculus of Constructions with implicit products, primitive heterogeneous equality, and dependent intersections. All data erase to pure lambda terms, and there is no built-in notion of datatype. The derivations are generic in the sense that we derive induction for any datatype which arises as the least fixed point of a signature functor. We consider Church-style and Mendler-style lambda-encodings. Moreover, the isomorphism of these encodings is proved. Also, we formalize Lambek's lemma as a consequence of expected laws of cancellation, reflection, and fusion.

CCS Concepts • **Theory of computation** → **Constructive mathematics**; **Type theory**; *Logic and verification*; Proof theory;

Keywords datatypes, lambda encodings, impredicativity, Cedille, induction

ACM Reference Format:
Denis Firsov and Aaron Stump. 2018. Generic Derivation of Induction for Impredicative Encodings in Cedille. In *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3167087

1 Introduction

Can practically useful constructive type theory be developed based on pure lambda calculus? For many decades the answer has been no. Implementations like Coq and Agda of constructive type theory augment a pure type system with a subsystem for primitive user-declared datatypes [5, 13]. This is because, among other issues, induction is provably not derivable in second-order dependent type theory [7]. In this paper, we contribute an alternative, positive answer: we show how to define a general class of inductive datatypes, with their associated induction principles, within a compact pure type theory called the Calculus of Dependent Lambda Eliminations (CDLE) [18]. The theory is pure in the sense that the language of terms is just that of pure untyped lambda calculus, with no additional term operators. This Curry-style type system extends the (Curry-style) Calculus of Constructions with a small number of additional typing primitives. Using these, the second author has already shown how to derive natural-number induction within the type theory [19]. In this paper we go much further and present a general development that derives induction abstractly, for any inductive datatype which arises as a least fixed point of a signature functor. We give separate derivations for Church-encoded datatypes and Mendler-encoded ones (these encodings are reviewed in Section 3).

The technical contributions of the paper are:

1. We present the first generic derivation of induction in a pure type theory.
2. To do this, we extend the standard notions of Church-style and Mendler-style algebra, to dependently typed versions we call **proof algebras**.
3. We show that our definitions of inductive datatypes are well-behaved. In particular, we prove the Lambek's lemma as a consequence of derived properties of reflection, cancellation, and fusion. Moreover, we prove that Church-encoded datatypes are isomorphic to Mendler-encoded datatypes. We also present the utility of our derivation on several basic examples.
4. We observe that while as expected, both the identity and composition functor laws are required for the derivation of induction based on conventional algebras (Church encoding), only the identity functor law is needed for the induction rule for Mendler encodings. To the best of our knowledge this is a novel observation which we plan to investigate in future to see if it broadens the class of definable datatypes.

Note that the first paper on CDLE includes a complex form of recursive types [18]. We have since dropped this

CPP'18, January 8–9, 2018, Los Angeles, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5586-5/18/01...$15.00
https://doi.org/10.1145/3167087

construct after discovering induction is derivable without it, in the presence of a primitive heterogeneous equality type, which we use also in this paper [19].

2 Background

The starting point for the CDLE type theory in which we work is the Curry-style Calculus of Constructions (CC). This language is defined by a type-assignment system, assigning the types of CC to pure unannotated lambda terms. These types include dependent function types $\Pi\, x : T.\, T'$ and impredicative quantification $\forall\, X : \kappa.\, T$ over types at possibly higher kind κ. Example type-assignment rules include:

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda\, x.\, t : \Pi\, x{:}T.\, T'} \qquad \frac{\Gamma, X : \kappa \vdash t : T}{\Gamma \vdash t : \forall\, X{:}\kappa.\, T}$$

Note that in the second of these rules, the subject t of the typing judgment does not change. Also note that X cannot be free in t as t is a pure untyped λ-term and hence contains no type variables.

For algorithmic typing, CDLE uses annotated terms, which contain enough information to apply the type-assignment rules deterministically. So in the implementation, one uses instead of the second rule above, this one, which is similar to the usual rule for Church-style $\forall$-introduction:

$$\frac{\Gamma, X : \kappa \vdash t : T}{\Gamma \vdash \Lambda\, X{:}\kappa.\, t : \forall\, X{:}\kappa.\, T}$$

Relatedly, when testing convertibility of types, the algorithmic type system compares the erasures of the types, where erasing a type simply erases the terms contained in it (see Figure 3). The erasure of $\Lambda\, X : \kappa.\, t$, for example, is just the erasure of t, matching up with the Curry-style version of $\forall$-introduction.

To Curry-style CC, CDLE adds three additional typing constructs:

1. implicit products $\forall\, x{:}T.\, T'$ as in the Implicit Calculus of Constructions [14],
2. a primitive heterogeneous equality type $t \simeq t'$ that expresses β-equality of two terms t and t' of possibly different types, and
3. dependent intersection types $\iota\, x{:}T.\, T'$ as introduced by Kopylov [10] (though he used notation $x : T \cap T'$)

Figure 1 gives the formation rules for these constructs, and Figure 2 the algorithmic introduction and elimination rules, showing also the syntax we use for their annotated terms. The rules for implicit products (first row of Figure 2) are essentially Miquel's [14]. We use a minus sign to indicate an erased argument. We use some arbitrary term β ($\lambda\, x.\, x$, say) as the proof for true equations. CDLE's conversion rule allows changing a term t_1 to any β-equal t_2 of the same type, so using the introduction rule we can inhabit the type $t_1 \simeq t_2$. Note, however, that in keeping with our extrinsic viewpoint, the types of the terms are not actually part of the equality type itself, nor does the elimination rule require that the types of the left- and right-hand sides are the same to do an elimination. Only upon introduction are the types required to be the same.

$$\frac{\Gamma, x : T' \vdash T : \star}{\Gamma \vdash \forall\, x{:}T'.\, T : \star} \qquad \frac{\Gamma \vdash t : T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t \simeq t' : \star}$$

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

Figure 1. Formation rules for additional type constructs of CDLE

The remaining rules of Figure 2 are for introducing and eliminating dependent intersections. These are similar to the usual (nondependent) intersection types, except that in $\iota\, x{:}T.\, T'$, the type T' may contain x free, and hence substitution of the subject of typing is required when considering this second component of the intersection. This allows the remarkable possibility to refer to a term t in its own type $[t/x]T'$, giving some form of self reference – albeit the reference x in T' is required to be at some other type T. Note that for introducing a dependent intersection, we require that the two components are provably equal. We could alternatively impose the stricter requirement that the erasures of the two components are identical; we have confirmed that the results of the paper still hold in this case.

The rules in Figure 2 are all justified by a denotational semantics for types, essentially that of [18] with a couple of straightforward modifications. This semantics justifies also the conversion rules shown in Figure 4, where $T =_\beta T'$ relates types by contracting type-level β-redexes, and $t =_\beta t'$ is standard β-equivalence of terms. In particular, the semantics interprets terms as sets of β-equivalence classes of closed terms; this explains the third premise in the rule for introducing intersection types. Finally, the erasures of annotated terms are given in Figure 3.

We have implemented CDLE in a tool called Cedille, which we have used to check all the examples in this paper. A pre-release version for use evaluating the artifacts referenced in this paper is here:

http://cs.uiowa.edu/~astump/cedille-prerelease.zip

All code referenced in this paper may be found here:

http://cs.uiowa.edu/~astump/papers/cpp2018-code.zip

2.1 Deriving natural-number induction

It is well-known that computationally, natural-number induction can be reduced to iteration (cf. Section 2 of [8]). Let us illustrate this informally. First define the type cNat of Church-encoded natural numbers as usual:

```
cNat ◂ ★ = ∀ X : ★. (X → X) → X → X.
```

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

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

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

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

Figure 2. Algorithmic introduction and elimination rules for additional type constructs of CDLE

$$\begin{array}{lcl}
|\Lambda x{:}T.\, t| & = & |t| \\
|t\ -t'| & = & |t| \\
|\beta| & = & \lambda x.\, x \\
|\rho\ t\ -\ t'| & = & |t'| \\
|[t_1, t_2\{p\}]| & = & |t_1| \\
|t.1| & = & |t| \\
|t.2| & = & |t|
\end{array}$$

Figure 3. Erasures of annotations for implicit products, primitive equality, and dependent intersections

$$\frac{\Gamma \vdash t : T' \quad T =_\beta T' \quad \Gamma \vdash T : \star}{\Gamma \vdash t : T}$$

$$\frac{\Gamma \vdash t : [t_1/x]T \quad t_1 =_\beta t_2 \quad FV(t_2) \subseteq dom(\Gamma)}{\Gamma \vdash t : [t_2/x]T}$$

Figure 4. Conversion rules

Let cZ and cS be the zero and successor constructors for this type as usually defined. Then given predicate `P : cNat → ★`, base case `b : P cZ`, and step case

```
s : Π x : cNat. P x → P (cS x)
```

we must try to inhabit `Π n : cNat. P n`. Let us use standard syntax for dependent pair types `Σ x : A. B` (though these are not primitive in CDLE). Given `n : cNat`, we apply n to `Σ x : cNat. P x` (to instantiate the type variable X in the definition of cNat), and then to `(cZ,b)` and `λ p. (cS (π₁ p) , s (π₁ p) (π₂ p))`. This constructs a proof of `P n` by iterating the step case n times starting from the base case. But crucially, at the end of this iteration, all we have is an inhabitant of `Σ x : cNat. P x`. We do not know that the first component of the pair computed for n is actually n. The identity of the n for which we have `P n` is hidden by the existential abstraction (i.e., the Σ-type).

As proposed by the second author [19], this problem can be overcome in Cedille using dependent intersection types. We first define a predicate expressing that a Church-encoded natural number (cNat) is inductive:

```
Inductive ◀ cNat → ★ = λ x : cNat.
 ∀ Q : cNat → ★.
 (∀ x : cNat. Q x → Q (cS x)) →
 Q cZ →
 Q x.
```

Now we define the "true" type of natural numbers as dependent intersection of cNat and predicate Inductive. Intuitively, Nat is a subset of cNat carved out by the inductivity predicate:

```
Nat ◀ ★ = ι x : cNat. Inductive x.
```

Moreover, this says that natural numbers are cNats which are simultaneously their own proofs of inductiveness. This builds on an observation of Leivant's that under the Curry-Howard isomorphism, proofs in second-order logic that data satisfy their type laws can be seen as isomorphic to the Church-encodings of those data [12]. Here, the data are already Church-encoded, and so they are isomorphic to the proofs of their own inductiveness. We may then define the constructors for Nat type:

```
Z ◀ Nat = [ cZ, Λ X. λ s. λ z. z { β } ].
S ◀ Nat → Nat = λ n. [ cS n.1,
  Λ P. λ s. λ z. s -n.1 (n.2 P s z) { β } ].
```

So, if n is a natural of type Nat then it can be "viewed" as a cNat by first component of the intersection type `n.1` and as a proof that `n.1` is inductive by second component, namely that `n.2 : Inductive n.1`. Critically, as noted above, the components t1 and t2 of an introduction `[ t1, t2 { p } ]` of a dependent intersection are required to be accompanied with a proof p that their erasures are equal, a requirement we refer to generally as **alignment**. Since this requirement is trivially satisfied in the definitions for Z and S then it also justifies that erasure of `n.1` and `n.2` is n and therefore `n.1 ≃ n.2` for any natural n.

Given the above definitions, we may then inhabit the following type for induction:

```
∀ Q : Nat → ★.
(∀ x : Nat. Q x → Q (S x)) →
Q Z →
Π x : Nat. Q x
```

The derivation uses `x.2` with the following predicate:

```
λ x : cNat. Σ x' : Nat. (x ≃ x'.1 × Q x')
```

This says that we will prove by induction on `x : cNat` that there exists an `x' : Nat`, a proof that x equals `x'` (since `x'.1` erases to `x'`), and a proof of `Q x'`. This is easily done based on the strategy at the start of this section. The crucial innovation allowing this strategy to go through is using dependent intersection for the definition of `Nat`, and using equality to connect the `x` that is eliminated with the `x'` that is constructed. For a more leisurely consideration of this derivation, see [19].

3 Encodings of inductive types

In this section we review the standard material on impredicative encodings of inductive datatypes [17, 24]. We also compare the Church and Mendler-style encodings.

For this and all the following sections we assume the following global parameters:

1. A functor `F` kinded as ★ → ★.
2. A function `fmap` associated with `F`:
   ```
   fmap ◂ ∀ X : ★.∀ Y : ★. (X → Y) → F X → F Y
   ```
3. The identity law for `fmap`.
   ```
   Law1 ◂ ★ = ∀ X : ★. ∀ Y : ★.
    ∀ f : X → Y.
    ∀ prf : (Π v : X. f v ≃ v).
    Π x : F X. fmap f x ≃ x.
   ```
4. The composition law for `fmap`:
   ```
   Law2 ◂ ★ = ∀ X : ★. ∀ Y : ★. ∀ Z : ★.
    ∀ f : Y → Z. ∀ g : X → Y.
    Π x : F X.
    fmap f (fmap g x) ≃ fmap (λ x. f (g x)) x.
   ```

Also we adopt some syntactical simplifications to improve readability. In particular, we hide the implicit (erased) arguments in the definitions. For example the arguments `X` and `Y` in the definition of `fmap` are quantified implicitly, so we write `fmap π₁` instead of fully annotated version `fmap (Σ A B) A (π₁ A B)`. The current version of Cedille language requires fully annotated terms.

It is important not to confuse the implicit arguments in the sense of Implicit Calculus of Constructions and "hidden" arguments as in languages like Agda and Coq. For example, in Agda the identity function has one implicit argument

```
id : {A : Set} → A → A
```

This argument may be omitted when the typechecker can infer it, e.g. `id zero`. In Cedille, the implicit arguments are ones which exist just for purposes of typing, so that equational reasoning happens on terms from which the implicit arguments have been erased (see Figure 3).

3.1 Church-style inductive types

In categorical parlance, given an endofunctor `F` the conventional (Church-style) F-algebra is a pair of object `X` (*carrier*) and an arrow `F X → X` (recall that `F` is a global parameter):

```
AlgC ◂ ★ → ★ = λ X : ★. F X → X.
```

These form a category, where an arrow between `(X,f)` and `(X',f')` is given by a homomorphism `h : X → X'` such that `∀ v : F X. f' (fmap h v) ≃ h (f v)`.

The inductive type induced by the least fixed point of `F` is usually modelled as a carrier of the initial object in the category of F-algebras. We follow this definition in three steps. First, we define a carrier of initial F-algebra (which in our case is a type):

```
FixC ◂ ★ = ∀ X : ★. AlgC X → X.
```

Second, initiality tells that there must be a (unique) homomorphism from the initial one to any other F-algebra. In Cedille, this translates into a function which for an algebra `AlgC X` returns a function from `FixC` to `X`:

```
foldC ◂ ∀ X : ★. AlgC X → FixC → X
 = λ alg. λ fix. fix alg.
```

Lastly, the arrow of the initial F-algebra is a function `inC` from `F FixC` to `FixC`, which denotes the collection of constructor functions for inductive datatype `FixC`.

```
inC ◂ AlgC FixC
 = λ fix. λ alg. alg (fmap (foldC alg) fix).
```

For every F-algebra `f' : AlgC X'` the function `foldC f'` is indeed a homomorphism:

```
HomC ◂ ∀ X' : ★. Π f' : AlgC X'. Π v : F FixC.
 f' (fmap (foldC f') v) ≃ foldC f' (inC v)
 = λ f'. λ v. β.
```

The equality follows simply by beta reduction. Since we do not have a dependent elimination for `FixC` then we cannot prove that `foldC f'` is a unique homomorphism (modulo extensionality). As a result, `FixC` and `inC` form only a weakly initial F-algebra.

Categorically, one can prove *Lambek's lemma*, which states that every initial F-algebra is an isomorphism. The lemma justifies that the carrier of the initial algebra (`FixC`) is a least fixed point of the functor. Unfortunately, absence of dependent elimination (induction rule) prevents us from proving that `inC : AlgC FixC` is initial and hence the proof of the Lambek's lemma fails. We will correct this in Section 4 below.

Let us look at the example of natural numbers in terms of above definitions. Natural numbers arise as a least fixed point of functor `NatF`:

```
NatF ◂ ★ → ★ = λ X : ★. Unit + X.

natFmap ◂ ∀ X : ★. ∀ Y : ★. (X → Y)
 → NatF X → NatF Y = λ f. λ nf.
  case nf (λ unit. in1 unit)
          (λ x.    in2 (f x)).
```

`NatF X` is a disjoint sum of singleton type `Unit` and `X` (`in1` and `in2` are left and right injections of the disjoint sum). We instantiate the global functor parameter `F` with `NatF` and `fmap` with `natFmap`. Natural numbers are then the least fixed point of `NatF`:

```
NatC ◂ ★ = FixC.
```

To define the usual constructors of natural numbers we first create the values of type `NatF NatC` and then use function `inC` to "inject" them into `NatC`:

```
zeroC ◂ NatC = inC (in1 unit).

sucC ◂ NatC → NatC = λ n. inC (in2 n).
```

3.2 Mendler-style inductive types

The categorical model of the Mendler-style inductive types is more involved than the conventional one. A Mendler-style F-algebra for an endofunctor $F : \mathcal{C} \to \mathcal{C}$ is a pair (X, Φ) so that X is an object in $\mathcal{C}$ and $\Phi : \mathcal{C}(-, X) \to \mathcal{C}(F\ -, X)$ is a natural transformation [23]. In Cedille, this translates into a polymorphic function:

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

Similarly to the Church-style, Mendler-style F-algebras form a category and the inductive type induced by a signature functor `F` is modelled by the carrier of the initial object in this category. In our case, the object is a type defined as a Mendler-style least fixed point:

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

As before, folding the value of `FixM` with an algebra `AlgM X` gives the homomorphism from `FixM` to `X`:

```
foldM ◂ ∀ X : ★. AlgM X → FixM → X
 = λ alg. λ fix. fix alg.
```

In Cedille, the arrow of (weakly) initial Mendler-style F-algebra is a polymorphic function `inM`:

```
inM ◂ AlgM FixM = λ c. λ v. λ alg.
 alg (foldM alg) (fmap c v).
```

As in the case of `inC`, the purpose of `inM` is to define constructor functions for the carrier type. The example of natural numbers encoded in Mendler-style looks very similar to the Church-style approach.

```
NatM ◂ ★ = FixM.

zeroM ◂ NatM = inM (λ x. x) (in1 unit).

sucM ◂ NatM → NatM = λ n. inM (λ x. x) (in2 n).
```

In the example above, the argument `R` is implicitly instantiated with `NatM` so that `inM (λ x. x) : NatF NatM → NatM` is a Church-style F-algebra.

3.3 Comparison of approaches

As it is common to normalizing languages based on polymorphic lambda calculus, Cedille does not allow explicit recursive calls. Instead, recursive calls are encoded by means of impredicative polymorphism.

The core difference of Church-style and Mendler-style F-algebras is in how they encode the recursive calls. Let us exhibit the difference by defining the function even for `NatC` and `NatM`. In both cases we fold the input with an appropriate algebra.

```
evenC ◂ NatC → Bool = foldC evenAlgC.

evenM ◂ NatM → Bool = foldM evenAlgM.
```

The Church-style algebra is essentially a function of type `NatF Bool → Bool`. We must think of its argument `NatF Bool` as a collection of constructors of `NatC` which encapsulate the result of a recursive call of `evenC` on a previous natural number (below denoted by `b`).

```
evenAlgC ◂ AlgC Bool = λ fn.
 case fn (λ _ . true)            % zero case
         (λ b . not b).          % suc case
```

The Mendler-style `NatF`-algebra is a polymorphic function of type `∀ R : ★. (R → Bool) → NatF R → Bool`. It allows us to state the recursive calls explicitly by providing arguments `R → Bool` and `NatF R`. One can think of universally quantified `R` as `NatM` in disguise and the argument `R → Bool` is the function `evenM` in disguise. The polymorphic `R` ensures that recursive calls will be made on only the previous natural number (which ensures termination; cf. [1]).

```
evenAlgM ◂ AlgM Bool = λ rec. λ fr.
  case fr (λ _. true)            % zero case
          (λ r. not (rec r)).    % suc case
```

Delaware et al. explain that the explicit control over the recursive calls make the Mendler-style algebras behave reasonably in both *lazy* and *strict* environments. At the same time they show that the lack of control over the recursive calls in Church-style algebras leads to performance drawbacks in strict environments and subtle issues in lazy environments [2, 4].

In fact, Mendler-style and Church-style algebras are interconvertible:

```
ca2ma ◂ ∀ X : ★. AlgC X → AlgM X
 = λ algC. λ f. λ fr. algC (fmap f fr).

ma2ca ◂ ∀ X : ★. AlgM X → AlgC X
 = λ algM. algM (λ x. x).
```

Hence, `FixC` and `FixM` are interconvertible as well. Moreover, both encodings are isomorphic, but we cannot formally prove that without induction.

4 Induction principle

The goal of this section is to employ dependent intersection types to define inductive types for which the induction principle is provable.

4.1 Induction for Mendler-style types

In this section, our goal is to define a type which will represent a subset of `FixM` for which the induction principle is derivable. We define the subset as an intersection type of `FixM` with the "inductivity" predicate on it. Also, we are constrained by an introduction rule of the intersection types, which requires a proof that the terms which are involved in an intersection have the same erasures (see Figure 2). To satisfy this condition we express the inductivity of `FixM` as a "dependently-typed" version of `FixM`. Recall, that `FixM` is defined in terms of Mendler-style algebra:

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

FixM ◀ ★ = ∀ X : ★. AlgM X → X.
```

Hence, we start by introducing the dependent version of Mendler algebra, a *Q-proof F-algebra*, which is parameterized by an algebra and the predicate on its carrier. (Note that our notion of proof algebra differs from that of [4].) But first, to aid the reader, here is an overview of the central concepts that will be defined below:

- `PrfAlgM` – a dependently typed version of `AlgM`, but with some extra explicit arguments that may be helpful for users of induction (but hinder alignment with `AlgM`).
- `PrfAlgM'` – like `PrfAlgM` but with those arguments made implicit (and so not obstructing alignment with `AlgM`); this version is used internally in the development of induction, but we will see at the end of the section how to return to `PrfAlgM`.
- `IsIndFixM` – a predicate stating that an element of type `FixM` satisfies induction for predicates on `FixM`. Induction here is phrased using the function `inM` (which denotes the constructors of `FixM`).
- `FixIndM` – the subset of `FixM` satisfying `IsIndFixM`; this is the type for which we prove induction.
- `IsIndFixIndM` – a predicate stating that an element of `FixIndM` satisfies induction for predicates on `FixIndM`. Induction is phrased using the function `inFixIndM`, which denotes the constructors of `FixIndM`.
- `allIndFixIndM` – the proof that every element of type `FixIndM` indeed satisfies the predicate `IsIndFixIndM`. Deriving this is the main result of this section.

To return to proof algebras: as we saw above (Section 3.2), a Mendler-style F-algebra provides a function to make explicit recursive calls. Correspondingly, we define proof algebras to provide a function to use for explicitly invoking the inductive hypothesis. Therefore, the inductive hypothesis is a dependent function of type `Π r : R. Q (cast r)`, where `cast` converts polymorphic `R` to `X`. For the inductive hypothesis to be strong enough, `cast` must not change the value it is being applied to.

```
PrfAlgM ◀ Π X : ★. (X → ★) → AlgM X → ★
 = λ X : ★. λ Q : X → ★. λ alg : AlgM X.
   ∀ R : ★. Π cast : R → X.
   Π _ : ∀ r : R. cast r ≃ r.
   (Π r : R. Q (cast r)) →
   Π fr : F R.  Q (alg cast fr).
```

Given the inductive hypothesis for every `R`, the proof algebra must conclude that `alg cast fr` satisfies `Q`. Since `PrfAlgM` has more explicit parameters than `AlgM`, the erasures of their values can never be the same (align)—this will prevent us from defining the inductive subset of `FixM` as the intersection type. For that reason we give an alternative definition of proof algebra so that the function `cast` and the proof that it is the identity function are implicit:

```
PrfAlgM' ◀ Π X : ★. (X → ★) → AlgM X → ★
 = λ X : ★. λ Q : X → ★. λ alg : AlgM X.
   ∀ R : ★. ∀ cast : R → X.
   ∀ _ : ∀ r : R. cast r ≃ r.
   (Π r : R. Q (cast r)) →
   Π fr : F R.  Q (alg cast fr).
```

Implicitly quantified `cast` might appear as a restriction on a derivation of `Q (alg cast fr)`. However, later we will observe that both types of algebras are equivalent in the context of the induction rule.

Next, to stay close to the definition of `FixM` we say that the value of `x : FixM` is inductive if a `Q`-proof algebra implies `Q x`:

```
IsIndFixM ◀ FixM → ★ = λ x : FixM.
 ∀ Q : FixM → ★.
 PrfAlgM' FixM Q inM → Q x.
```

If `x` satisfies `IsIndFixM` then to show that the particular `x` satisfies `Q` it is enough to do a proof by induction—prove that for any `fr : F R` we can conclude `Q (inM cast fr)` given the premise that every `r : R` satisfies `Q (cast r)` and `cast r ≃ r`.

It is crucially important to maintain a similarity in the definition of `FixM` and the inductivity predicate `IsIndFixM`.

```
FixM         = AlgM X              → X.
IsIndFixM x  = PrfAlgM' FixM Q inM → Q x.
```

The analogy of definitions allows us to internalize the fact that induction can be reduced to iteration. Namely, that the inductive value `x : FixM` and the proof that `x` is inductive (`IsIndFixM x`) could be represented by terms with provably equal erasures—the property which is required by introduction rule of intersection types.

Let us then define the inductive subset of `FixM` as a dependent intersection of `FixM` and predicate `IsIndFixM`:

```
FixIndM ◀ ★ = ι x : FixM. IsIndFixM x.
```

Similarly to the function `inM`, the function `inFixIndM` constructs the values of `FixIndM` from polymorphic `R : ★`, function `f : R → FixIndM`, and value `fr : F R`. The implementation combines these arguments into value `v : F FixIndM` by mapping `f` over `fr`:

```
inFixIndM ◀ AlgM FixIndM
 = λ f. λ fr. let v = fmap f fr in
 [ tm1 v, tm2 v { eqm v } ].
```

Then the resulting value `FixIndM` is an intersection of `tm1 v` and `tm2 v`. The first component of intersection must be a value of FixM derived from F FixIndM in terms of previously defined function inM.

```
tm1 ◂ F FixIndM → FixM
 = λ v. inM (λ x. x) (fmap (λ x. x.1) v).
```

The second component (tm2 v) is a proof that every tm1 v is inductive:

```
tm2 ◂ Π v : F FixIndM. IsIndFixM (tm1 v)
 = λ v. Λ Q. λ q. q FixIndM
 -(λ z. z.1)
 -(Λ r. β)
  (λ r. r.2 Q q) (fmap FixIndM FixIndM (λ x. x) v).
```

(For better intuition the implicit arguments are shown.)

Now let us look at the unfolded erasures of tm1 and tm2

```
tm1 = λ v. λ q. q (λ r. (r q))
               (fmap (λ x. x) (fmap (λ x. x) v))
tm2 = λ v. λ q. q (λ r. (r q))(fmap (λ x. x) v)
```

The third component of intersection (eqm v) proves that erasures of tm1 and tm2 are equal by applying the identity law of F.

Now we can turn our attention to the derivation of induction for FixIndM. Similarly to FixM, the value of x : FixIndM is inductive if we can derive Q x from the respective proof algebra (note a similarity of IsIndFixM and IsIndFixIndM).

```
IsIndFixIndM ◂ FixIndM → ★ = λ x : FixIndM.
 ∀ Q : FixIndM → ★.
 PrfAlgM' FixIndM Q inFixIndM → Q x.
```

Our goal is to prove that all FixIndM are inductive in this sense. Note that since the predicate Q ranges over FixIndM instead of FixM (as in IsIndFixM), we cannot simply use the form of inductivity for x.1 arising from x : FixIndM (namely, x.2 : IsIndFixM x.1) as a proof of inductivity of x itself (namely, IsIndFixIndM x).

Let us start the derivation by assuming the existence of a predicate Y : FixM → ★ with the property that Y x.1 implies Q x for any x. Then we can reduce the derivation of Q x to Y x.1 and prove Y x.1 by using the fact that x.1 is inductive. However, to do that we must convert a proof algebra of FixIndM to a proof algebra of FixM. In other words, we need a function from PrfAlgM' FixIndM Q to PrfAlgM' FixM Y. For that purpose we also need an implication from Q x to Y x.1. The most important part of the derivation is to show how to convert a predicate on FixIndM to a predicate on FixM satisfying both the above properties:

```
WithWitness ◂ Π X : ★. Π Y : ★.
 (X → ★) → (X → Y) → Y → ★
 = λ X : ★. λ Y : ★. λ Q : X → ★.
   λ cast : X → Y. λ y : Y.
   Σ x : X. (y ≃ cast x) × Q x.

WithFixIndM ◂ (FixIndM → ★) → FixM → ★
 = λ Q : FixIndM → ★.
   WithWitness FixIndM FixM Q (λ x. x.1).
```

The predicate WithWitness X Y Q cast is satisfied by value y : Y iff there exists a value x : X so that Q x holds and y ≃ cast x. Therefore, the predicate WithFixIndM Q is satisfied by value y : FixM iff there exists a value x : FixIndM so that Q x holds and y ≃ x.1. The key role in this definition is played by heterogeneous equality on erasures. Since the erasure of x.1 is x then the equality y ≃ x.1 is equivalent to y ≃ x. Hence, it becomes easy to verify that Q x holds iff WithFixIndM Q x.1 does.

```
prop1 ◂ Π x : FixIndM. ∀ Q : FixIndM → ★.
 Q x → WithFixIndM Q x.1 = <..>.

prop2 ◂ Π x : FixIndM. ∀ Q : FixIndM → ★.
 WithFixIndM Q x.1 → Q x = <..>.


convIH ◂ ∀ Q : FixIndM → ★.
 PrfAlgM' FixIndM Q inFixIndM →
 PrfAlgM' FixM (WithFixIndM Q) inM
 = <..>.
```

(convIH is implemented in terms of prop1 and prop2.)

This is enough to show that all FixIndM are inductive:

```
allIndFixIndM ◂ Π x : FixIndM. IsIndFixIndM x.
 = λ x. Λ Q. λ algQ. prop2 x Q
     (x.2 (WithFixIndM Q) (convIH Q algQ)).
```

Unfolding the definition of IsIndFixIndM, we may rearrange premises in the above statement to highlight that any Q-proof algebra implies that Q holds for every FixIndM.

```
inductionM' ◂ ∀ Q : FixIndM → ★.
 PrfAlgM' FixIndM Q inFixIndM →
 Π x : FixIndM. Q x
 = λ algQ. λ x. allIndFixIndM x algQ.
```

Recall that we designed PrfAlgM' to align with AlgM. Since PrfAlgM has more explicit parameters, it is more convenient for the user to define. In the context of the induction rule the original PrfAlgM is equivalent to PrfAlgM'. The central idea is that proof algebra PrfAlgM' for lifted predicate WithWitness X X Q (λ x. x) is equivalent to PrfAlgM for Q. But since lifted Q is logically equivalent to Q then we can state the final version of induction in terms of original "strong" proof algebra PrfAlgM:

```
inductionM ◂ ∀ Q : FixIndM → ★.
 PrfAlgM FixIndM Q inFixIndM →
 Π x : FixIndM. Q x = <..>.
```

4.2 Induction for Church-style types

Similarly to the previous section, our goal is to define a type which will represent a subset of FixC for which the induction principle is derivable. We define this subset as an intersection type of FixC with the "inductivity" predicate IsIndFixC. As before, the erasures of x : FixC and IsIndFixC x must align. We define IsIndFixC by following the definition of FixC:

```
AlgC ◂ ★ → ★ = λ X : ★. F X → X.

FixC ◂ ★ = ∀ X : ★. AlgC X → X.
```

The first question is how to define the dependent version of Church-style algebra. The main difficulty of this task is in expressing the inductive hypothesis. The immediate idea is to use the dependent product of type `Σ X Q`. In other words, we pair the values of `X` and proofs that they satisfy `Q`. Then the `Q`-proof algebra is simply a dependent function from inductive hypothesis `x : F (Σ X Q)` to `Q (alg (fmap π₁ x))`:

```
PrfAlgC ◂ Π X : ★. (X → ★) → AlgC X → ★
 = λ X : ★. λ Q : X → ★. λ alg : AlgC X.
 Π ih : F (Σ X Q). Q (alg (fmap π₁ ih)).
```

`FixC` is inductive if given a `Q`-proof algebra we can conclude that it satisfies `Q` (analogously to the definition of `FixC`):

```
IsIndFixC ◂ FixC → ★ = λ x : FixC.
 ∀ Q : FixC → ★. PrfAlgC FixC Q inC → Q x.
```

The inductive subset of `FixC` consists of values which satisfy `IsIndFixC`:

```
FixIndC ◂ ★ = ι x : FixC. IsIndFixC x.
```

Next, we implement a function for constructing the values of `FixIndC` from `F FixIndC`:

```
inFixIndC ◂ AlgC FixIndC
 = λ v. [ tc1 v, tc2 v { eqc v } ].
```

The function `tc1` must convert `F FixIndC` to `FixC`. Since `F` is a functor and we already defined function `inC` then `tc1` is implemented in terms of it:

```
tc1 ◂ F FixIndC → FixC =
 λ v. inC (fmap (λ x. x.1) v).
```

The function `tc2` must prove that every `tc1 v` is inductive:

```
tc2 ◂ Π v : F FixIndC. IsIndFixC (tc1 v)
 = λ v. Λ Q. λ k. k (fmap FixIndC (Σ FixC Q)
 (λ q . sigma q.1 (q.2 Q k)) v).
```

To finalize the definition of `inFixIndC` we must show that the erasure of `tc1` and `tc2` are the same. Unfortunately, this is not the case. The fully unfolded and erased terms look as follows:

```
tc1 v = λ k. k (fmap (λ q.            q k)
                                (fmap (λ x. x) v))
tc2 v = λ k. k (fmap (λ q. λ c. (c q (q k))) v)
```

The variable `q` in the erasure of `tc1` represents the value of `FixC` and value `k` represents the F-algebra. Hence, `q k` delivers a recursive call (`q k ≃ foldC k q`). The variable `q` in the erasure of `tc2` represents the value of `FixIndC` and `k` represents the proof algebra. Hence, `q k` delivers the inductive hypothesis `Q q` (`q.2 Q k ≃ q k`). Since the value of `Q q` depends on `q`, the sigma type is being created (`sigma q.1 (q.2 Q k) ≃ λ c. c q (q k)`).

The problem is that the F-algebra and proof algebra differ in the representation of recursive call and representation of inductive hypothesis. The recursive call is simply a value `X` while the inductive hypothesis is a dependent pair `Σ X Q`. To force the equality between the erasures of `tc1` and `tc2` we must adjust the algebras. To achieve that we wrap the recursive call into the unary product and use a "weak" sigma type for the inductive hypothesis in the proof algebra. The definition of unary product is simple:

```
Unary ◂ ★ → ★
 = λ A : ★. ∀ X : ★. (A → X) → X.

unary ◂ ∀ X : ★. X → Unary X
 = Λ X. λ x. Λ Y. λ c. c x.
```

The weak sigma type represents the "dependent" version of unary product. In other words, one can think of `WΣ` as usual sigma type but with the first projection being implicit (erased).

```
WΣ ◂ Π A : ★ . (A → ★) → ★
 = λ A : ★. λ B : A → ★.
 ∀ X : ★. (∀ a : A. B a → X) → X.

wsigma ◂ ∀ X : ★. ∀ Y : X → ★.
 ∀ x : X. Y x → WΣ X Y
 = Λ X. Λ Y. Λ x. λ y. Λ Z. λ c. c -x y.
```

Observe, that erasure of `wsigma` is equal to `λ a. λ c. c a` which is the same as the erasure of `unary`. Hence, if we wrap the recursive call into unary product `unary (foldC k q)` and wrap the inductive hypothesis into weak sigma type `wsigma -q.1 (q.2 Q k)` then the erasures will be equal to `λ c. c (q k)` in both cases and we can fix the problem with alignment described above.

Unfortunately, in the general case it is impossible to implement projection functions from `WΣ A B`. We can implement both projections for the special case `WΣ A (WWId A B)`, where `WWId` lifts the predicate `B` to the logically equivalent one that also stores the witness `A`:

```
WWId ◂ Π X : ★. (X → ★) → X → ★ =
 λ X : ★. λ Q : X → ★. WithWitness X X Q (λ x. x).

wsPrj1 ◂ ∀ X : ★. ∀ Y : X → ★.
 WΣ X (WWId X Y) → X = <..>.
```

Now, to guarantee the alignment of algebras we can redefine Church F-algebra in terms of `Unary` and proof algebra in terms of `WΣ X (WWId X Q)`:

```
AlgC' ◂ ★ → ★ = λ X : ★. F (Unary X) → X.

PrfAlgC' ◂ Π X : ★. (X → ★) → AlgC' X → ★
 = λ X : ★. λ Q : X → ★. λ alg : AlgC' X.
   Π ih : F (WΣ X (WWId X Q)).
     (WWId X Q)
        (alg (fmap (λ x. unary (wsPrj1 x) ih)).
```

(The predicate `IsIndFixC` must be adjusted to `PrfAlgC'`) By using the adjusted definitions of algebras we developed functions `tc1'` and `tc2'` so that their erasures are equal.

```
tc1'◂ F (Unary FixIndC) → FixC = <..>.

tc2'◂ Π v : F (Unary FixIndC). IsIndFixC (tc1' v)
 = <..>.
```

Then it becomes possible to implement a function:

```
inFixIndC' ◂ AlgC' FixIndC = <..>.
```

Since `Unary X` is isomorphic to `X` then we get previously desired `AlgC FixIndC`:

```
inFixIndC ◂ AlgC FixIndC = <..>.
```

Next, by following exactly the same steps as in the previous section we derive the induction principle for the lifted predicates `WWId FixIndC Q`:

```
inductionC' ◂ ∀ Q : FixIndC → ★.
 PrfAlgC' FixIndC Q inFixIndC' →
 Π x : FixIndC. WWId FixIndC Q x
  = <..>.
```

Observe that `WWId FixIndC Q` is logically equivalent to `Q`, `Unary X` is isomorphic to `X`, and `(WΣ X (WWId X Q))` is isomorphic to `Σ X Q`. Therefore, we can state the induction principle in terms of the original tidier definition of proof algebra `PrfAlgC`:

```
inductionC ◂ ∀ Q : FixIndC → ★.
 PrfAlgC FixIndC Q inFixIndC →
 Π x : FixIndC. Q x = <..>.
```

4.3 Discussion

We discovered that it was simpler to derive the generic induction rule in Mendler-style than in Church-style. Recall, that the Church-style F-algebras provide access to the results of recursive calls. By analogy, the Church-style proof algebra must provide access to the results of the invocation of the inductive hypothesis on "previous" elements. This inevitably couples these elements with proofs that they satisfy a property. The coupling between elements and proofs in proof algebras hinders alignment with F-algebras. To overcome this issue we adjusted both algebras by wrapping the results of recursive calls in unary product and using specifically tuned "weak" sigma types for representation of inductive hypothesis.

The derivation of induction for Mendler-style datatypes is simpler. Recall, that Mendler-style algebras allow the explicit recursive calls by providing the function `R → X` and elements of `F R`, where `R` is a polymorphic type. Analogously, a Mendler-style proof algebra expresses its inductive hypothesis on elements of `F R` as a dependent function `Π r : R. Q (cast r)`, where `cast` is an implicit identity function from `R` to `X`. Therefore proof algebras perfectly align with the respective F-algebras.

The unexpected aspect of our derivation of induction is that in Mendler-style it only relies on the first functor law. We plan to investigate this aspect further to find if it broadens the class of definable datatypes.

5 Properties

In this section we show that the inductive datatypes defined by our generic development are well-behaved and satisfy the expected properties. The same set of properties holds for both encodings.

5.1 Initiality

`FixIndM` is a weakly initial Mendler-style F-algebra since there is an algebra homomorphism from it to any other algebra.

```
foldIndM ◂ ∀ X : ★. AlgM X → FixIndM → X
 = λ alg. λ fix. foldM alg fix.1.
```

To show that `FixIndM` is initial we must prove that given an algebra `algM : AlgM X` the homomorphism `foldIndM algM` is unique (modulo extensionality). This is known as *universal* property of folds [9]:

```
universal' ◂ ∀ X : ★. Π h : FixIndM → X.
 Π algM : AlgM X.
 (Π y : F FixIndM.
  h (inFixIndM (λ x. x) y) ≃ algM h y) →
 Π x : FixIndM. h x ≃ foldIndM algM x = <..>.
```

The proof of the above lemma does not succeed because there are two ways of "using" Mendler-style F-algebra. First, we can specify `R` to `X` and then construct the value of `X` as follows: `algM X (λ x. x) (fmap h e)`. The second possibility is to specify `R` to `FixIndM` and then construct the same value differently—`algM FixIndM h e`. In a categorical setting the equality of both values follows from naturality conditions on `algM` [23]. In Cedille, we cannot prove that all Mendler-style F-algebras are natural. Instead, we define a predicate:

```
Natural ◂ Π X : ★. AlgM X → ★ =
 λ X : ★. λ algM : AlgM X.
 ∀ R : ★. ∀ f : R → X. ∀ fr : F R.
 algM f fr ≃ algM (λ x. x) (fmap f fr).
```

(Church encodings do not require any extra assumptions.) Now, if we assume that the given algebra is natural then we can prove universality of `foldIndM` by induction:

```
universalM ◂ ∀ X : ★ . Π h : FixIndM → X.
 Π algM : AlgM X. Natural X algM →
 (Π y : F FixIndM.
  h (inFixIndM (λ x. x) y) ≃ algM h y) →
 Π x : FixIndM. h x ≃ foldIndM algM x = <..>.
```

This property justifies that `FixIndM` and `inFixIndM` form an initial Mendler-style F-algebra.

5.2 Reflection, cancellation, and fusion

The three best-known consequences of initiality are the reflection, cancellation, and fusion laws.

The reflection property states that folding the value with its constructors does not change it:

```
reflectionM ◂ Π x : FixIndM.
 foldIndM inFixIndM x ≃ x = <..>.
```

Reflection is a direct consequence of previously proved initiality. Since `inFixIndM` is natural and `foldIndM inFixIndM` is an F-algebra homomorphism from `FixIndM` to `FixIndM` then it must be the identity homomorphism.

The cancellation property can be viewed as the reduction rule where the fold is applied to a data constructor. The reduction recursively replaces the constructors of `FixIndM` with given F-algebra.

```
cancellationM  ◂ ∀ X : ★.
 Π algM : AlgM X. Natural X algM →
 Π x : F FixIndM.
 foldIndM algM (inFixIndM (λ x. x) x) ≃
    algM (foldIndM algM) x = <..>.
```

The fusion law describes the composition of fold with another function. It gives conditions under which the intermediate values produced by folding can be eliminated.

```
fusionM ◂ ∀ X : ★. ∀ Y : ★.
 Π f : X → Y.
 Π alg1 : AlgM X. Natural X alg1 →
 Π alg2 : AlgM Y. Natural Y alg2 →
 (Π fx : F X.
  f (alg1 (λ x. x) fx) ≃ alg2 f fx) →
 Π x : FixIndM.
  f (foldIndM alg1 x) ≃ foldIndM alg2 x
 = <..>.
```

5.3 Lambek's lemma

The Lambek's lemma says that if `in : F (Fix F) → Fix F` forms an initial F-algebra then `in` is an isomorphism with inverse being `fold (fmap in)` [11]. In this section we formalize the Lambek's lemma for Mendler-style types. In particular we show that `FixIndM` is isomorphic to `F FixIndM` (the same holds for `FixIndC`). The proof becomes possible due to derived initiality (which itself depends on induction principle).

To start with we convert the initial Mendler-style F-algebra to the Church-style F-algebra:

```
inFixIndM' ◂ F FixIndM → FixIndM
 = ma2ca inFixIndM.
```

As mentioned previously, the categorical model of inductive types gives the exact recipe on how to implement the inverse of `inFixIndM'`, namely:

```
outFixIndM ◂ FixIndM → F FixIndM
 = foldIndM (ca2ma (fmap inFixIndM')).
```

We show that it is a pre-inverse:

```
inoutM ◂ Π x : FixIndM.
 inFixIndM' (outFixIndM x) ≃ x = <..>.
```

Definitionally, `inFixIndM' (outFixIndM x)` is equal to `inFixIndM' (foldIndM (ca2ma (fmap inFixIndM')) x)`, therefore, by fusion law it is equal to `foldIndM inFixIndM x` which by reflection law is `x`.

The function `outFixIndM` is also a post-inverse:

```
outinM ◂ Π x : F FixIndM.
 outFixIndM (inFixIndM' x) ≃ x = <..>.
```

Since, `FixIndM` is isomorphic to `F FixIndM` then we are justified in calling it a fixed point of `F`. Initiality justifies in calling it a least fixed point.

5.4 Isomorphism of encodings

In this section, we show that Church-style and Mendler-style encodings are isomorphic. Recall, that in Section 3.3 we discussed how to convert between Church and Mendler-style algebras (functions `ca2ma` and `ma2ca`). Hence, to convert between encodings of fixed points we must fold the original value with the constructors (initial algebras) of the target encoding:

```
c2m ◂ FixIndC → FixIndM
 = foldIndC (ma2ca inFixIndM).


m2c ◂ FixIndM → FixIndC
 = foldIndM (ca2ma inFixIndC).
```

The composition of `c2m` with `m2c` is an F-algebra homomorphism from `FixIndM` to `FixIndM`. Therefore, by initiality and reflection property of `FixIndM` it must be the identity homomorphism:

```
isoM ◂ Π x : FixIndM. c2m (m2c x) ≃ x = <..>.
```

The same reasoning applies for the opposite direction:

```
isoC ◂ Π x : FixIndC. m2c (c2m x) ≃ x = <..>.
```

6 Examples

We instantiate the generic development for natural numbers and polymorphic lists.

6.1 Natural numbers

In Section 2.1 we showed a specific definition of natural numbers and derivation of induction principle for it. Let us list the main steps we took:

1. Defining the "simply" typed natural numbers `cNat`.
2. Implementing constructors `cZ` and `cS` for `cNat`.
3. Defining the inductivity predicate `Inductive` in terms of constructor functions `cZ` and `cS`.
4. Defining the inductive subset of `cNat` as the intersection type of `cNat` and `Inductive`.
5. Implementing the constructors `Z` and `S` for `Nat`.
6. Stating and deriving induction for `Nat`.

The definition of inductive datatypes in terms of generic development parameterized by a functor allows us to derive most of these steps automatically.

As was mentioned previously, natural numbers arise as a least fixed point of functor `NatF`:

```
NatF ◂ ★ → ★ = λ X : ★. Unit + X.
```

So, to define natural numbers we must instantiate the functor F of generic development with NatF, fmap with natFmap, and prove the functor laws. Then we define Church-style natural numbers as a least fixed point of NatF:

```
Nat ◂ ★ = FixIndC.
```

Even before we defined the constructors of Nat the generic development provides the induction rule inductionC for Nat (the Church-style proof algebra argument is unfolded):

```
inductionNatGen ◂ ∀ Q : Nat → ★.
 (Π ih : NatF (Σ Nat Q).
   Q (inFixIndC (fmap π₁ ih))) →
 Π x : Nat. Q x = inductionC.
```

After defining usual constructors for Nat we can derive the equivalent "flat" version of induction rule:

```
zero ◂ Nat = inFixIndC (in1 unit).

suc ◂ Nat → Nat = λ n. inFixIndC (in2 n).

inductionNat ◂ ∀ Q : Nat → ★.
 Q zero → (Π n : Nat. Q n → Q (suc n)) →
 Π x : Nat. Q x = λ qz. λ qs. λ x.
 inductionNatGen (λ ih. case ih
 (λ u'. ρ (eta-unit u') - qz)  % zero case
 (λ b. qs (π₁ b) (π₂ b)))      % suc case
 x.
```

In the zero case we use the fact (eta-unit) that type Unit has the unique inhabitant unit, so ρ (eta-unit u') rewrites the goal by equation u' ≃ unit.

6.2 Lists

In this section we use Mendler-style encoding to define polymorphic lists. Lists of elements of type A arise as a least fixed point of functor ListF A:

```
ListF ◂ ★ → ★ → ★ = λ A : ★. λ X : ★.
 Unit + (A × X).
```

We skip the obvious proofs that ListF A is a functor which satisfies the required laws. Since ListF is a family of functors then we must parameterize the combinators of generic development explicitly depending on A. Then List A is FixIndM (ListF A) (fmap A) (law1 A) (law2 A). However, for the readability purposes we only write the first argument:

```
List ◂ ★ → ★ = λ A : ★. FixIndM (ListF A).
```

The previously developed function inductionM immediately provides the generic induction principle for List A (the proof algebra argument is unfolded):

```
inductionListGen ◂ ∀ A : ★.
 (∀ R : ★. Π cast : R → List A.
  Π _ : ∀ r : R. cast r ≃ r.
  (Π r : R. Q (cast r)) →
  Π fr : ListF A R. Q (inFixIndM cast fr)) →
 Π x : List A. Q x = <..>.
```

We define constructors and the flat version of induction rule:

```
nil ◂ ∀ A : ★. List A
 = inFixIndM (λ x. x) (in1 unit).

cons ◂ ∀ A : ★. A → List A → List A
 = λ x. λ xs. inFixIndM (λ x. x) (in2 (pair x xs)).

inductionListM ◂ ∀ A : ★. ∀ Q : List A → ★.
 Q nil →
 (Π x : A. Π xs : List A. Q xs → Q (cons x xs)) →
 Π xs : List A. Q xs = λ qnil. λ qcons.
 inductionListGen (λ cast. λ eq. λ ih. λ fr.
  case fr
   (λ u'. ρ (eta-unit u') - qnil)   % nil case
   (λ p. (qcons (π₁ p)              % cons case
              (cast (π₂ p))
              (ih   (π₂ p))))).
```

It is worth noting that in the "cons" case the inductive hypothesis Q (cast (π₂ p)) is produced explicitly by invoking function ih : Π r : R → Q (cast r).

7 Related work

Swierstra showed how to solve the famous expression problem stated by Wadler [25]. His technique allows to assemble datatypes and functions from isolated individual components [21]. The key idea is to define datatypes as fixed points of a functor. Most importantly, he observes that if F and F' are functors then the pointwise coproduct F :+: F is also a functor. This allows to derive function Fix (F :+: F') → X from independently defined functions Fix F → X and Fix F' → X.

Delaware et al. extended the idea of Swierstra to modular proofs [4]. They developed an approach to deriving induction for impredicative encodings based on universal property of folds. The value v : Fix F is universal if h v ≃ fold alg v for any algebra alg and homomorphism h. Then, it is shown how to derive the induction principle for values which satisfy universality. The induction principle allows to derive properties for Fix (F :+: F') from properties of Fix F and Fix F'. Also, it is important to note that the proof of induction relies on functional extensionality. Our approach does not require extra axioms or assumptions.

Initially, the Coq proof assistant was based on the Calculus of Constructions. It also used the impredicative encodings to model inductive datatypes [17]. The induction principles for those encodings were added axiomatically which endangered normalization properties of the calculus. The calculus of inductive constructions (CIC) extends CC with built-in inductive datatypes and serves as a basis for later versions of Coq [16].

Ghani et al. described the derivation of induction principle for inductive types in fibrational setting [6]. For example, the described approach allows to derive induction for hyperfunctions which arise as a fixed point of functor `F X = (X → Int) → Int` (this fixed point cannot be interpreted as a set). Since their approach is purely categorical then it is also inherently extensional.

In some ways closest to the present work is a recent series of papers on adding foundational support for datatypes and co-datatypes based on category theory, to Isabelle/HOL. This line of numerous papers is summarized in [3]; the initiating paper is [22]. Like the present work, foundational (co)datatypes for Isabelle/HOL is based on a categorical view of algebras (and coalgebras). At a high level, the main point in favor of our approach is that we achieve a single generic derivation of induction within our theory. In contrast, the Isabelle/HOL work has developed a package which, given suitable user specifications of (co)datatypes, can generate, in a foundational way, the requisite definitions and proofs of various desired theorems. So they produce derivations of induction and related constructs automatically for each datatype presented, while we give a single generic derivation once and for all. While the Isabelle/HOL work derives more than our approach (e.g., we have not treated codatatypes, nor do we integrate with a complex ecosystem of theorem-proving plugins and packages), their package weighs in at a hefty 29,000 lines of Standard ML [3]. Our developments are an order of magnitude smaller (and carried out within the theory itself).

8 Conclusions and future work

We showed that the Calculus of Constructions extended with implicit products, intersection types, and heterogeneous equality allows to generically derive an induction rule for impredicatively encoded inductive datatypes. In our work we considered Church-style and Mendler-style encodings. We observed that Mendler-style representation of recursive calls (inductive hypothesis) makes the derivation of induction simpler than the Church-style representation. Also we proved the Lambek's lemma and showed that Church-style and Mendler-style encodings are isomorphic.

Even with many explicit type annotations required by the current early-stage implementation of Cedille, our developments are very compact. The entire code for deriving induction, proving the discussed properties, and the examples is, for Church-encoding, 800 lines of Cedille, and for Mendler-encoding, it is just 600 lines. Thus we have achieved one of the goals of the Cedille project, to give a compact core type theory in which we can derive inductive types in a concise way.

In future, we consider to explore richer classes of datatypes in Cedille. For example, it should be straightforward to extend our development to indexed datatypes by defining them as least fixed points of indexed functors.

Another interesting direction is investigation of inductive-recursive datatypes in Cedille. Uustalu and Vene described a construction which allows to turn any scheme `S : ★ → ★` (`S` can be mixed-variant—an argument can appear on covariant and contravariant positions) into an isomorphic scheme `S^e : ★ → ★` which is a functor. Then they showed how to use this construction for taking a least fixed point of a mixed-variant scheme to implement a course-of-value natural numbers (natural numbers paired with predecessor function) [23]. We conjecture that the same construction could be used for expressing the inductive-recursive datatypes in Cedille.

Unfortunately, Church-style and Mendler-style encodings suffer from the linear time predecessor function. The possible alternatives are Parigot and Stump-Fu encodings [15, 20]. Parigot encoding represents datatypes as their own recursors which allows to have a constant time predecessor. The drawback of this is that the representation of natural `n` is exponential in call-by-value setting. More recent Stump-Fu encoding improves the Parigot representation by requiring only quadratic space for representation of natural `n`. We plan to investigate if the induction principle is derivable for these encodings.

Acknowledgments

The first author thanks Anna, Albert, and Eldar for their support and patience.

We gratefully acknowledge NSF support under award 1524519, and DoD support under award FA9550-16-1-0082 (MURI program).

References

[1] Ki Yung Ahn and Tim Sheard. 2011. A Hierarchy of Mendler Style Recursion Combinators: Taming Inductive Datatypes with Negative Occurrences. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 234–246.

[2] Patrick Bahr. 2011. Evaluation à la carte: Non-strict evaluation via compositional data types. In *Proceedings of the 23rd Nordic Workshop on Programming Theory (NWPT '11)*. 38–40.

[3] Julian Biendarra, Jasmin Christian Blanchette, Aymeric Bouzy, Martin Desharnais, Mathias Fleury, Johannes Hölzl, Ondrej Kuncar, Andreas Lochbihler, Fabian Meier, Lorenz Panny, Andrei Popescu, Christian Sternagel, René Thiemann, and Dmitriy Traytel. 2017. Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic. In *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Clare Dixon and Marcelo Finger (Eds.), Vol. 10483. Springer, 3–21.

[4] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à La Carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 207–218.

[5] The Agda development team. 2016. *Agda*. http://wiki.portal.chalmers.se/agda/pmwiki.php Version 2.5.1.

[6] Clément Fumex, Neil Ghani, and Patricia Johann. 2011. Indexed Induction and Coinduction, Fibrationally. In *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings.* 176–191.

[7] Herman Geuvers. 2001. Induction Is Not Derivable in Second Order Dependent Type Theory. In *Typed Lambda Calculi and Applications (TLCA)*. 166–181.

[8] Neil Ghani, Patricia Johann, and Clément Fumex. 2010. Fibrational Induction Rules for Initial Algebras. In *Proceedings of the 24th International Conference/19th Annual Conference on Computer Science Logic (CSL)*. Springer-Verlag, 336–350.

[9] Graham Hutton. 1999. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.* 9, 4 (July 1999), 355–372.

[10] Alexei Kopylov. 2003. Dependent Intersection: A New Way of Defining Records in Type Theory. In *18th IEEE Symposium on Logic in Computer Science (LICS)*. 86–95.

[11] Joachim Lambek. 1968. A Fixpoint Theorem for complete Categories. *Mathematische Zeitschrift* 103 (1968), 151–161.

[12] Daniel Leivant. 1983. Reasoning about functional programs and complexity classes associated with type disciplines. In *24th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 460–469.

[13] The Coq development team. 2016. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.5.

[14] Alexandre Miquel. 2001. The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In *Typed Lambda Calculi and Applications (TLCA)*, Samson Abramsky (Ed.). 344–359.

[15] Michel Parigot. 1988. *Programming with proofs: A second order type theory*. Springer Berlin Heidelberg, Berlin, Heidelberg, 145–159.

[16] Christine Paulin-Mohring. 1993. Inductive Definitions in the System Coq - Rules and Properties. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA '93)*. Springer-Verlag, London, UK, UK, 328–345.

[17] Frank Pfenning and Christine Paulin-Mohring. 1989. Inductively Defined Types in the Calculus of Constructions. In *Proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics, Tulane University, New Orleans, Louisiana*, M. Main, A. Melton, M. Mislove, and D. Schmidt (Eds.). Springer-Verlag LNCS 442, 209–228.

[18] Aaron Stump. [n. d.]. The Calculus of Dependent Lambda Eliminations. *Journal of Functional Programming* 27 ([n. d.]), e14.

[19] Aaron Stump. 2017. From Realizability to Induction via Dependent Intersection. Under consideration for Annals of Pure and Applied Logic.

[20] Aaron Stump and Peng Fu. 2016. Efficiency of lambda-encodings in total type theory. *Journal of Functional Programming* 26 (2016).

[21] Wouter Swierstra. 2008. Data Types à La Carte. *J. Funct. Program.* 18, 4 (July 2008), 423–436.

[22] Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. 2012. Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*. IEEE Computer Society, 596–605.

[23] Tarmo Uustalu and Varmo Vene. 1999. Mendler-style Inductive Types, Categorically. *Nordic J. of Computing* 6, 3 (Sept. 1999), 343–361.

[24] Philip Wadler. 1990. *Recursive types for free!* http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt

[25] Philip Wadler. 2016. *The Expression Problem*. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt