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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

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
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 \( \kappa \). Example type-assignment rules include:

\[
\begin{align*}
\Gamma, x : T & \vdash t : T' \\
\Gamma & \vdash \lambda x. t : \Pi x : T. T'
\end{align*}
\]

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 \( \lambda \)-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:

\[
\begin{align*}
\Gamma, X : \kappa & \vdash t : T \\
\Gamma & \vdash \Lambda X : \kappa.t : \forall X : \kappa. T
\end{align*}
\]

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 \( t \equiv t' \) that expresses \( \beta \)-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 \( \beta (\lambda x. x, \text{say}) \) as the proof for true equations. CDLE’s conversion rule allows changing a term \( t_1 \) to any \( \beta \)-equal \( t_2 \) of the same type, so using the introduction rule we can inhabit the type \( t_1 \equiv t_2 \). 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.

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 \( \beta \)-redexes, and \( t = \beta t' \) is standard \( \beta \)-equivalence of terms. In particular, the semantics interprets terms as sets of \( \beta \)-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 \( \text{cNat} \) of Church-encoded natural numbers as usual:

\[
\text{cNat} : \mathbb{N} = \forall X : \star. (X \rightarrow X) \rightarrow X \rightarrow X.
\]
We must try to inhabit \( \Pi n : \text{cNat} \) (though these are not primitive in CDLE). Given \( X \) variable the base case. But crucially, at the end of this iteration, all we have is an inhabitant of \( \Sigma x : \text{cNat} \). \( P \ x \). 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 \( \Sigma \)-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:

\[
\text{Inductive} \; p \; \text{cNat} \rightarrow * = \lambda \ x : \text{cNat}. \\
(\forall \ Q : \text{cNat} . \; \text{Q} \ x \rightarrow \text{Q} (\text{cS} \ x)) \\nQ \; \text{cZ} \rightarrow \\
Q \; x.
\]

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:

\[
\text{Nat} \; \triangleleft \; * \; = \; \{ x : \text{cNat} . \; \text{Inductive} \; x \}.
\]

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:

\[
\begin{align*}
Z & \triangleleft \text{Nat} = [ \; \text{cZ} , \; \Lambda \; x . \; \lambda \; z . \; z \{ \; \beta \; \} \; ] . \\
S & \triangleleft \text{Nat} \rightarrow \text{Nat} = \lambda \; n . \; [ \; \text{cS} \; n . 1 , \\
& \; \Lambda \; p . \; \lambda \; z . \; s \; n . 1 \; (n . 2 \; p \; s \; z) \{ \; \beta \; \} \; ].
\end{align*}
\]

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:

\[
\begin{align*}
(\forall \; Q : \text{Nat} . \; Q \; x \rightarrow \text{Q} (S \; x)) & \rightarrow \\
Q \; Z & \rightarrow \\
\Pi \; x : \text{Nat} . \; Q \; x
\end{align*}
\]

The derivation uses \( x . 2 \) with the following predicate:

\[
\begin{align*}
\lambda \; x : \text{cNat} . \; \Sigma \; x' : \text{Nat} . \; (x \equiv x' . 1 \times Q \; x')
\end{align*}
\]

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 \( \star \rightarrow \star \).
2. A function \( \text{fmap} \) associated with \( F \):
   \[
   \text{fmap} \downarrow \forall X : \star. \forall Y : \star. (X \rightarrow Y) \rightarrow F X \rightarrow F Y
   \]
3. The identity law for \( \text{fmap} \):
   \[
   \text{Law1 } \downarrow \star = \forall X : \star. \forall Y : \star. (X \rightarrow Y) \rightarrow \text{fmap } \lambda x. f v \approx x.
   \]
4. The composition law for \( \text{fmap} \):
   \[
   \text{Law2 } \downarrow \star = \forall X : \star. \forall Y : \star. \forall Z : \star. (X \rightarrow Y) \rightarrow (Y \rightarrow Z) \rightarrow (X \rightarrow Z).
   \]
For every F-algebra \( f' : \text{AlgC} X' \) the function \( \text{foldC} f' \) is indeed a homomorphism:
   \[
   \text{foldC } \downarrow \forall X : \star. \text{AlgC} X \rightarrow \text{FixC} \rightarrow X = \lambda f. \text{fix alg}.
   \]

Lastly, the arrow of the initial F-algebra is a function \( \text{inC} \) from \( F \text{FixC} \) to \( \text{FixC} \), which denotes the collection of constructor functions for inductive datatype \( \text{FixC} \).

\[
\text{inC } \downarrow \text{AlgC} \text{FixC} = \lambda \text{fix alg. alg (fmap (foldC alg) fix)}.
\]
For every F-algebra \( f' : \text{AlgC} X' \) the function \( \text{foldC} f' \) is indeed a homomorphism:
   \[
   \text{HomC } \downarrow \forall X' : \star. \forall f' : \text{AlgC} X'. \forall \text{v} : \text{FixC}. \text{f} \downarrow (\text{fmap (foldC f')} \text{v}) \approx \text{foldC f'} \text{inC} v.
   \]

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

Categorically, one can prove Lambeck's lemma, which states that every initial F-algebra is an isomorphism. The lemma justifies that the carrier of the initial algebra (\( \text{FixC} \)) is a least fixed point of the functor. Unfortunately, absence of dependent elimination (induction rule) prevents us from proving that \( \text{inC} : \text{AlgC} \text{FixC} \) is initial and hence the proof of the Lambeck’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 \( \text{NatF} \):

\[
\text{NatF } \downarrow \star = \lambda X : \star. \text{Unit} + X.
\]

\[
\text{natFmap } \downarrow \forall X : \star. \forall Y : \star. (X \rightarrow Y) \rightarrow \text{NatF X} \rightarrow \text{NatF Y} = \lambda f. \lambda \text{nf. case nf (\text{Unit}} \text{unit}) \text{in2 (f x)).}
\]

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

\[
\text{NatC } \downarrow \star = \text{FixC}.
\]
To define the usual constructors of natural numbers we first create the values of type NatF Nat and then use function inC to \"inject\" them into NatC:

\[
\begin{align*}
\text{zeroC } &\text{ NatC } = \text{ inC (in1 unit)}.

\text{sucC } &\text{ Nat }\to\text{ Nat }= \lambda n. \text{ inC (in2 n)}.
\end{align*}
\]

### 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 \( F : C \to C \) is a pair \((X, \Phi)\) so that \( X \) is an object in \( C \) and \( \Phi : C(-, X) \to C(F -, X) \) is a natural transformation [23]. In Cedille, this translates into a polymorphic function:

\[
\begin{align*}
\text{AlgM } &\to \alpha \times. \text{ AlgM } X \to X.
\end{align*}
\]

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:

\[
\begin{align*}
\text{FixM } &\to \alpha = \forall x : \alpha. \text{ AlgM } X \to X.
\end{align*}
\]

As before, folding the value of FixM with an algebra \( \text{AlgM } X \) gives the homomorphism from \( \text{FixM} \) to \( X \):

\[
\begin{align*}
\text{foldM } &\to \forall x : \alpha. \text{ AlgM } X \to \text{ FixM } \to X
\end{align*}
\]

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

\[
\begin{align*}
\text{inM } &\to \alpha = \forall c. \forall v. \alpha. \text{ AlgM } \text{ FixM } = \lambda c. \lambda v. \lambda \text{ alg.} \text{ alg (foldM alg) (fmap c v).}
\end{align*}
\]

As in the case of \( \text{inC} \), the purpose of \( \text{inM} \) 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:

\[
\begin{align*}
\text{NatM } &\to \alpha = \text{ FixM.}
\end{align*}
\]

\[
\begin{align*}
\text{zeroM } &\to \text{ NatM } = \text{ inM (in1 x unit).}

\text{sucM } &\to \text{ NatM } \to \text{ NatM } = \lambda n. \text{ inM (in2 n).}
\end{align*}
\]

In the example above, the argument \( R \) is implicitly instantiated with \( \text{NatM} \), so that \( \text{inM (\lambda x. x)} : \text{NatF NatM } \to \text{NatM} \) 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:

\[
\begin{align*}
\text{evenC } &\to \text{ NatC } \to \text{ Bool } = \text{ foldC evenAlgC.}

\text{evenM } &\to \text{ NatM } \to \text{ Bool } = \text{ foldM evenAlgM.}
\end{align*}
\]

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

\[
\begin{align*}
\text{evenAlgC } &\to \text{ AlgC Bool } = \lambda \text{ fn.}

\text{case fn (\lambda _. true)} &\% \text{ zero case}
\text{ (\lambda b. not b).} &\% \text{ suc case}
\end{align*}
\]

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

\[
\begin{align*}
\text{evenAlgM } &\to \text{ AlgM Bool } = \lambda \text{ rec. } \lambda \text{ fr.}

\text{case fr (\lambda _. true)} &\% \text{ zero case}
\text{ (\lambda r. not (rec r)).} &\% \text{ suc case}
\end{align*}
\]

Delaware et al. explain that the explicit control over the recursive calls make the Mendler-style algebras behave reasonably in both \textit{lazy} and \textit{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:

\[
\begin{align*}
\text{ca2ma } &\to \forall x : \alpha. \text{ AlgC } X \to \text{ AlgM } X
\end{align*}
\]

\[
\begin{align*}
\text{ma2ca } &\to \forall x : \alpha. \text{ AlgM } X \to \text{ AlgC } X
\end{align*}
\]

\[
\begin{align*}
\text{ma2ca } &\to \forall x : \alpha. \text{ AlgM } X \to \text{ AlgC } X
\end{align*}
\]

Hence, \( \text{FixC} \) and \( \text{FixM} \) 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 \( \text{FixM} \) for which the induction principle is derivable. We define the subset as an intersection type of \( \text{FixM} \) with the \textit{"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 \( \text{FixM} \) as a “dependently-typed” version of \( \text{FixM} \). Recall, that \( \text{FixM} \) is defined in terms of Mendler-style algebra:

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

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

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:

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

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 \( \Pi \ r : R. \ Q (\text{cast } r) \), where \( \text{cast} \) converts polymorphic \( R \) to \( X \). For the inductive hypothesis to be strong enough, \( \text{cast} \) must not change the value it is being applied to.

\[
\text{PrfAlgM} \triangleq \Pi X : \star. (X \rightarrow \star) \rightarrow \text{AlgM} X \rightarrow \star = \lambda X : \star. \lambda Q : X \rightarrow \star. \lambda \text{alg} : \text{AlgM} X.
\forall R : \star. \Pi \text{cast} : R \rightarrow X.
\Pi _ : \forall r : R. \text{cast } r \Rightarrow r.
(\Pi r : R. Q (\text{cast } r)) \rightarrow
\Pi \text{fr} : F R. Q (\text{alg} \text{cast fr}).
\]

Given the inductive hypothesis for every \( R \), the proof algebra must conclude that \( \text{alg} \text{cast fr} \) satisfies \( Q \). Since \( \text{PrfAlgM} \) has more explicit parameters than \( \text{AlgM} \), the erasures of their values can never be the same (align)—this will prevent us from defining the inductive subset of \( \text{FixM} \) 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:

\[
\text{PrfAlgM}' \triangleq \Pi X : \star. (X \rightarrow \star) \rightarrow \text{AlgM} X \rightarrow \star = \lambda X : \star. \lambda Q : X \rightarrow \star. \lambda \text{alg} : \text{AlgM} X.
\forall R : \star. \forall \text{cast} : R \rightarrow X.
\forall _ : \forall r : R. \text{cast } r \Rightarrow r.
(\Pi r : R. Q (\text{cast } r)) \rightarrow
\Pi \text{fr} : F R. Q (\text{alg} \text{cast fr}).
\]

Implicitly quantified \( \text{cast} \) might appear as a restriction on a derivation of \( Q (\text{alg} \text{cast fr}) \). 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 \( \text{FixM} \) we say that the value of \( x : \text{FixM} \) is inductive if a \( Q \)-proof algebra implies \( Q x \):

\[
\text{IsIndFixM} \triangleq \text{FixM} \rightarrow \star = \lambda x : \text{FixM}.
\forall Q : \text{FixM} \rightarrow \star.
\text{PrfAlgM}' \text{FixM} Q \text{inM} \rightarrow Q x.
\]

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

It is crucially important to maintain a similarity in the definition of \( \text{FixM} \) and the inductivity predicate \( \text{IsIndFixM} \).

\[
\text{FixM} = \text{AlgM} X \rightarrow X.
\text{IsIndFixM} x = \text{PrfAlgM}' \text{FixM} Q \text{inM} \rightarrow Q x.
\]

The analogy of definitions allows us to internalize the fact that induction can be reduced to iteration. Namely, that the inductive value \( x : \text{FixM} \) and the proof that \( x \) is inductive \( (\text{IsIndFixM} x) \) 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 \( \text{FixM} \) as a dependent intersection of \( \text{FixM} \) and predicate \( \text{IsIndFixM} \):

\[
\text{FixIndM} \triangleq \star = \lambda x : \text{FixM}. \text{IsIndFixM} x.
\]

Similarly to the function \( \text{inM} \), the function \( \text{inFixIndM} \) constructs the values of \( \text{FixIndM} \) from polymorphic \( R : \star \), function \( f : R \rightarrow \text{FixIndM} \), and value \( fr : R \). The implementation combines these arguments into value \( v : F \text{FixIndM} \) by mapping \( f \) over \( fr \):

\[
\text{inFixIndM} \triangleq \text{AlgM} \text{FixIndM} = \lambda f. \lambda fr. \text{let } v = f \text{map } fr \text{ in } [\text{tm1 } v, \text{tm2 } v \{\text{eqm } v\}].
\]

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

\[ \text{tm1} \triangleq F \text{FixIndM} \rightarrow \text{FixM} \]

\[ = \lambda v. \text{inM} (\lambda x. x) (\text{fmap} (\lambda x. x.1) v). \]

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

\[ \text{tm2} \triangleq \Pi v : F \text{FixIndM}. \text{IsIndFixM} (\text{tm1} v) \]

\[ = \lambda v. \lambda Q. \lambda q. q \text{FixIndM} \]

\[- (\lambda z. z.1) \]

\[- (\lambda r. r.2 \ q) (\text{fmap} \text{FixIndM} \text{FixIndM} (\lambda x. x.1) v). \]

(For better intuition the implicit arguments are shown.)

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

\[ \text{tm1} = \lambda v. \lambda q. q (\lambda r. (r q)) \]

\[ (\text{fmap} (\lambda x. x) (\text{fmap} (\lambda x. x.1) v)) \]

\[ \text{tm2} = \lambda v. \lambda q. q (\lambda r. (r q)) (\text{fmap} (\lambda x. x) v) \]

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

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

\[ \text{IsIndFixM} \triangleq F \text{FixIndM} \rightarrow \Pi x : \text{FixIndM}. \text{IsIndFixM} \]

\[ \forall Q : \text{FixIndM} \rightarrow \ast = \lambda x : \text{FixIndM}. \]

\[ \text{PrfAlgM'} \text{FixIndM} \rightarrow \Pi x : \text{FixIndM}. \text{IsIndFixIndM} x. \]

\[ \ast = \lambda x. \lambda Q. \lambda \text{algQ}. \text{prop2 x Q} \]

\[ (x.2 (\text{WithFixIndM Q}) \ (\text{convIH Q algQ})). \]

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

Let us start the derivation by assuming the existence of a predicate \( Y : \text{FixM} \rightarrow \ast \) 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 \( \text{FixIndM} \) to a proof algebra of \( \text{FixM} \). In other words, we need a function from \( \text{PrfAlgM'} \text{FixIndM} Q \) to \( \text{PrfAlgM'} \text{FixM} Y \). For that purpose we also need an implication from \( Q x \) to \( Y x \). The most important part of the derivation is to show how to convert a predicate on \( \text{FixIndM} \) to a predicate on \( \text{FixM} \) satisfying the above properties:

\[ \text{WithWitness} \triangleq \Pi X : \ast. \Pi Y : \ast. \]

\[ (X \rightarrow \ast) \rightarrow (X \rightarrow Y) \rightarrow Y \rightarrow \ast \]

\[ = \lambda X : \ast. \lambda Y : \ast. \lambda Q : X \rightarrow \ast. \]

\[ \lambda \text{cast} : X \rightarrow Y. \lambda y : Y. \]

\[ \Sigma x : X. (y = \text{cast} x) \times Q x. \]

\[ \text{WithFixIndM} \triangleq (\text{FixIndM} \rightarrow \ast) \rightarrow \text{FixM} \rightarrow \ast \]

\[ = \lambda Q : \text{FixIndM} \rightarrow \ast. \]

\[ \text{WithWitness FixIndM FixM} (\lambda x. x.1). \]

The predicate \( \text{WithWitness} X Y Q \) cast is satisfied by value \( y : Y \) iff there exists a value \( x : X \) so that \( Q x \) holds and \( y = \text{cast} x \). Therefore, the predicate \( \text{WithFixIndM Q} \) is satisfied by value \( y : \text{FixM} \) iff there exists a value \( x : \text{FixIndM} \) 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 \( \text{WithFixIndM Q} \rightarrow \ast \) does.

\[ \text{prop1} \triangleq \Pi x : \text{FixIndM}. \forall Q : \text{FixIndM} \rightarrow \ast. \]

\[ Q x \rightarrow \text{WithFixIndM Q} \ x.1 = \ast. \]

\[ \text{prop2} \triangleq \Pi x : \text{FixIndM}. \forall Q : \text{FixIndM} \rightarrow \ast. \]

\[ \text{WithFixIndM Q} x.1 \rightarrow Q x = \ast. \]

\[ \text{convIH} \triangleq \forall Q : \text{FixIndM} \rightarrow \ast. \]

\[ \text{PrfAlgM'} \text{FixIndM} Q \ \text{inFixIndM} \rightarrow \text{PrfAlgM'} \text{FixM} (\text{WithFixIndM Q}) \ \text{inM} = \ast. \]

(\text{convIH} \text{is implemented in terms of} \text{prop1 and prop2}.)

This is enough to show that all \( \text{FixIndM} \) are inductive:

\[ \text{allIndFixIndM} \triangleq \Pi x : \text{FixIndM}. \text{IsIndFixIndM} x. \]

\[ = \lambda x. \lambda Q. \lambda \text{algQ}. \text{prop2 x Q} \]

\[ (x.2 (\text{WithFixIndM Q}) (\text{convIH Q algQ})). \]

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

\[ \text{inductionM} \triangleq \forall Q : \text{FixIndM} \rightarrow \ast. \]

\[ \text{PrfAlgM'} \text{FixIndM} Q \ \text{inFixIndM} \rightarrow \Pi x : \text{FixIndM}. \ Q x \]

\[ = \lambda \text{algQ}. \lambda x. \text{allIndFixIndM} x. \ \text{algQ}. \]

Recall that we designed \( \text{PrfAlgM'} \) to align with \( \text{AlgM} \). Since \( \text{PrfAlgM} \) has more explicit parameters, it is more convenient for the user to define. In the context of the induction rule the original \( \text{PrfAlgM} \) is equivalent to \( \text{PrfAlgM'} \). The central idea is that proof algebra \( \text{PrfAlgM'} \) for lifted predicate \( \text{WithWitness} X Y Q \) is equivalent to \( \text{PrfAlgM} \) 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 \( \text{PrfAlgM'} \):

\[ \text{inductionM} \triangleq \forall Q : \text{FixIndM} \rightarrow \ast. \]

\[ \text{PrfAlgM} \text{FixIndM} Q \ \text{inFixIndM} \rightarrow \Pi x : \text{FixIndM}. \ Q x = \ast. \]

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 \( \text{FixC} \) for which the induction principle is derivable. We define this subset as an intersection type of \( \text{FixC} \) with the "inductivity" predicate \( \text{IsIndFixC} \). As before, the erasures of \( x : \text{FixC} \) and \( \text{IsIndFixC} x \) must align. We define \( \text{IsIndFixC} \) by following the definition of \( \text{FixC} \):
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 \( \Sigma X Q \). 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 \( x : F (\Sigma X Q) \) to \( Q (\text{alg} (\text{fmap} \; \pi_1 \; x)) \).

The function \( \text{IsIndFixC} \) is inductive if given a \( \text{FixIndC} \) erasure \( \text{InFixIndC} \) \( \iff \) \( \text{FixIndC} \) \( \iff \) \( \text{IsIndFixC} \) \( \iff \) \( \text{FixIndC} \) \( \iff \) \( \text{IsIndFixC} \) \( \iff \) \( \text{FixIndC} \).

The inductive subset of \( \text{FixC} \) is inductive if given a \( \text{FixIndC} \) \( \iff \) \( \text{FixIndC} \) \( \iff \) \( \text{IsIndFixC} \) \( \iff \) \( \text{FixIndC} \) \( \iff \) \( \text{IsIndFixC} \) \( \iff \) \( \text{FixIndC} \).

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 \( \Sigma X Q \).

To force the equality between the erasures of \( \text{tc1} \) and \( \text{tc2} \) we must adjust the algebras. To achieve that we wrap the recursive call into unary product and use a “weak” sigma type for the inductive hypothesis in the proof algebra. The definition of unary product is simple:

\[
\begin{align*}
\text{unary} \mapsto & \ * \mapsto (*) = \lambda \; A : * \ . \ V \ ; \ X : * \ . \ (A \to X) \to X . \\
\end{align*}
\]

The weak sigma type represents the “dependent” version of unary product. In other words, one can think of \( \text{wsigma} \) as usual sigma type but with the first projection being implicit (erased).

\[
\begin{align*}
\text{wsigma} \mapsto & \ V \ ; \ X : * \ . \ V \ ; \ Y : X \to * . \\
\end{align*}
\]

Observe, that erasure of \( \text{wsigma} \) is equal to \( \lambda \ a : * \ . \ \lambda \ c : c \ a \) which is the same as the erasure of unary. Hence, if we wrap the recursive call into unary product unary \( \text{foldC} \) \( k \) and wrap the inductive hypothesis into weak sigma type \( \text{wsigma} \) \( q.1 \) \( q.2 \) \( k \) then the erasures will be equal to \( \lambda \ c : c \ (q \ k) \) 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 \( \text{wsigma} \) \( A \) \( B \). We can implement both projections for the special case \( \text{wsigma} \) \( A \) \( \text{WWId} \) \( A \) \( B \), where \( \text{WWId} \) lifts the predicate \( B \) to the logically equivalent one that also stores the witness \( A \):

\[
\begin{align*}
\text{WWId} \mapsto & \ V \ ; \ X : * . \ V \ ; \ Y : X \to * . \\
\Sigma \ ; \ X \ ; \ (\text{WWId} \ ; \ Y) \to X = \ <..> . \\
\end{align*}
\]

Now, to guarantee the alignment of algebras we can redefine Church \( F \)-algebra in terms of Unary and proof algebra in terms of \( \text{WWId} \) \( X \) \( \text{WWId} \) \( X \).

\[
\begin{align*}
\text{AlgC} & \mapsto * \mapsto (*) = \lambda \ ; X : * . \ F \ (\text{Unary} \ ; X) \to X . \\
\end{align*}
\]

\[
\begin{align*}
\text{PrfAlgC} & \mapsto \ V \ ; \ X : * . \ (X \to *) \to \text{AlgC} \ ; X \to * . \\
\text{PrfAlgC} & \mapsto \ V \ ; \ X : * . \ (X \to *) \to \text{AlgC} \ ; X \to * . \\
\end{align*}
\]

\[
\begin{align*}
\text{WWId} \ ; \ X \ ; \ (\text{WWId} \ ; \ X \ ; \ (\text{WWId} \ ; \ X \ ; \ Q)) . \\
\end{align*}
\]

(The predicate \( \text{IsIndFixC} \) must be adjusted to \( \text{PrfAlgC} \)’). By using the adjusted definitions of algebras we developed functions \( \text{tc1}' \) and \( \text{tc2}' \) so that their erasures are equal.
We discovered that it was simpler to derive the generic in-
with the respective F-algebras.

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 previ-
ous section we derive the induction principle for the lifted
predicates WWId FixIndC Q:

\[ \text{inductionC} ◁ ∀ Q : \text{FixIndC} → ∗. \]

\[ \text{PrfAlgC} \text{FixIndC} Q \text{inFixIndC} ◁ \Pi x : \text{FixIndC}. \text{WWId FixIndC} Q x = <..>. \]

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:

\[ \text{inductionC} ◁ ∀ Q : \text{FixIndC} → ∗. \]

\[ \text{PrfAlgC} \text{FixIndC} Q \text{inFixIndC} ◁ \Pi x : \text{FixIndC}. Q x = <..>. \]

4.3 Discussion

We discovered that it was simpler to derive the generic in-
duction 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 pro-
perly. 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 ex-
licit recursive calls by providing the function R → X and
elements of F R, where R is a polymorphic type. Analog-
ously, 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.

\[ \text{foldIndM} ◁ ∀ X : ∗. \text{AlgM} X → \text{FixIndM} X = \lambda \text{alg}. \lambda \text{fix}. \text{foldM} \text{alg} \text{fix}.1. \]

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

\[ \text{universalM} ◁ ∀ X : ∗. \Pi h : \text{FixIndM} X. \]

\[ \Pi \text{algM} : \text{AlgM} X. \]

\[ \Pi y : F \text{FixIndM}. \]

\[ h (\text{inFixIndM} (λ x. x) y) \cong \text{algM} h y → \]

\[ \Pi x : \text{FixIndM}. h x \cong \text{foldIndM} \text{algM} x = <..>. \]

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: \text{algM} X (λ x. x) (fmap h e). The second
possibility is to specify R to FixIndM and then construct the
same value differently—\text{algM} FixIndM h e. In a categorical
setting the equality of both values follows from naturality
conditions on \text{algM} [23]. In Cedille, we cannot prove that all
Mendler-style F-algebras are natural. Instead, we define a
predicate:

\[ \text{Natural} ◁ \Pi X : ∗. \text{AlgM} X → ∗ = \]

\[ \lambda X : ∗. \lambda \text{algM} : \text{AlgM} X. \]

\[ \forall R : ∗. ∀ F : R → X. ∀ fr : F R. \]

\[ \text{algM} f \text{fr} \cong \text{algM} (λ x. x) (\text{fmap} f \text{fr}). \]

(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:

\[ \text{universalM} ◁ ∀ X : ∗. \Pi h : \text{FixIndM} X. \]

\[ \Pi \text{algM} : \text{AlgM} X. \text{Natural X} \text{algM} → \]

\[ \Pi y : F \text{FixIndM}. \]

\[ h (\text{inFixIndM} (λ x. x) y) \cong \text{algM} h y → \]

\[ \Pi x : \text{FixIndM}. h x \cong \text{foldIndM} \text{algM} x = <..>. \]

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 re-
flection, cancellation, and fusion laws.

The reflection property states that folding the value with
its constructors does not change it:
We show that it is a pre-inverse:

\[ \text{foldIndM} \circ \text{inFixIndM} x \simeq x = \langle .. \rangle. \]

Reflection is a direct consequence of previously proved initiality. Since \( \text{inFixIndM} \) is natural and \( \text{foldIndM} \circ \text{inFixIndM} \) is an F-algebra homomorphism from \( \text{FixIndM} \) to \( \text{FixIndM} \) 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 \( \text{FixIndM} \) with given F-algebra.

\[ \text{cancellationM} \triangleleft \forall X : \ast. \forall Y : \ast. \exists f : X \rightarrow Y. \exists \text{alg1} : \text{AlgM} X. \text{Natural} X \text{alg1} \rightarrow \Pi x : \text{FixIndM}. \text{algM} (\text{inFixIndM} (\lambda x. x) x) = \text{algM} \circ (\text{foldIndM} \circ \text{algM}) x = \langle .. \rangle. \]

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.

\[ \text{fusionM} \triangleleft \forall X : \ast. \forall Y : \ast. \forall f : X \rightarrow Y. \Pi \text{alg1} : \text{AlgM} X. \text{Natural} X \text{alg1} \rightarrow \Pi \text{alg2} : \text{AlgM} Y. \text{Natural} Y \text{alg2} \rightarrow \Pi f : X \rightarrow F X. \text{f} (\text{alg1} \circ (\lambda x. x) \text{fx}) \simeq \text{alg2} \circ f \text{fx} \rightarrow \Pi x : \text{FixIndM}. \text{f} (\text{foldIndM} \circ \text{alg1} x) \simeq \text{foldIndM} \circ \text{alg2} x = \langle .. \rangle. \]

### 5.3 Lambek’s lemma

The Lambek’s lemma says that if \( \text{in} : F (\text{Fix} F) \rightarrow \text{Fix} F \) forms an initial F-algebra then \( \text{in} \) is an isomorphism with inverse being \( \text{fold} \circ (\text{fmap} \circ \text{in}) \) [11]. In this section we formalize the Lambek’s lemma for Mendler-style types. In particular we show that \( \text{FixIndM} \) is isomorphic to \( \text{F FixIndM} \) (the same holds for \( \text{FixIndC} \)). 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:

\[ \text{inFixIndM'} \triangleleft F \text{FixIndM} \rightarrow \text{FixIndM} = \text{ma2ca} \circ \text{inFixIndM}. \]

As mentioned previously, the categorical model of inductive types gives the exact recipe on how to implement the inverse of \( \text{inFixIndM'} \), namely:

\[ \text{outFixIndM} \triangleleft \text{FixIndM} \rightarrow F \text{FixIndM} = \text{foldIndM} \circ (\text{ca2ma} \circ (\text{fmap} \circ \text{inFixIndM'})). \]

We show that it is a pre-inverse:

\[ \text{inoutM} \triangleleft \Pi x : \text{FixIndM}. \text{inFixIndM'} (\text{outFixIndM} x) \simeq x = \langle .. \rangle. \]

Definitionally, \( \text{inFixIndM'} \circ (\text{outFixIndM} x) \) is equal to \( \text{inFixIndM'} \circ (\text{foldIndM} \circ (\text{ca2ma} \circ (\text{fmap} \circ \text{inFixIndM'})) x) \), therefore, by fusion law it is equal to \( \text{foldIndM} \circ \text{inFixIndM} x \) which by reflection law is \( x \).

The function \( \text{outFixIndM} \) is also a post-inverse:

\[ \text{outinM} \triangleleft \Pi x : F \text{FixIndM}. \text{outFixIndM} (\text{inFixIndM'} x) \simeq x = \langle .. \rangle. \]

Since, \( \text{FixIndM} \) is isomorphic to \( F \text{FixIndM} \) then we can justify 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 \( \text{ca2ma} \) and \( \text{ma2ca} \)). Hence, to convert between encodings of fixed points we must fold the original value with the constructors (initial algebras) of the target encoding:

\[ \text{c2m} \triangleleft F \text{FixIndC} \rightarrow \text{FixIndM} = \text{foldIndC} \circ (\text{ma2ca} \circ \text{inFixIndM}). \]

\[ \text{m2c} \triangleleft F \text{FixIndM} \rightarrow \text{FixIndC} = \text{foldIndC} \circ (\text{ca2ma} \circ \text{inFixIndM}). \]

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

\[ \text{isoM} \triangleleft \Pi x : \text{FixIndM}. \text{c2m} \circ (\text{m2c} x) = x = \langle .. \rangle. \]

The same reasoning applies for the opposite direction:

\[ \text{isoC} \triangleleft \Pi x : \text{FixIndC}. \text{m2c} \circ (\text{c2m} x) = x = \langle .. \rangle. \]

### 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 \( \text{cNat} \).
2. Implementing constructors \( \text{cS} \) and \( \text{cS} \) for \( \text{cNat} \).
3. Defining the inductivity predicate \( \text{Inductive} \) in terms of constructor functions \( \text{cS} \) and \( \text{cS} \).
4. Defining the inductive subset of \( \text{cNat} \) as the intersection type of \( \text{cNat} \) and \( \text{Inductive} \).
5. Implementing the constructors \( \text{cS} \) and \( \text{cS} \) for \( \text{Nat} \).
6. Stating and deriving induction for \( \text{Nat} \).

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 \( \text{NatF} \):

\[ \text{NatF} \triangleleft \ast \rightarrow \ast = \lambda x : \ast. \text{Unit} + X. \]
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 \(NatF\):

\[
\text{Nat} \circ \ast = \text{FixIndC}.
\]

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

\[
\text{inductionNatGen} \circ V : \text{Nat} \rightarrow \ast.
\]

\[
(\Pi \text{ih} : \text{Nat}F (\Sigma \text{Nat} Q). \\
\text{Q} (\text{inFixIndC} (\text{fmap} \pi_1 \text{ih}))) \rightarrow \\
\Pi x : \text{Nat}. \text{Q} x = \text{inductionC}.
\]

After defining usual constructors for \(Nat\) we can derive the equivalent “flat” version of induction rule:

\[
\text{inductionNat} \circ V : \text{Nat} \rightarrow \ast.
\]

\[
\text{Q} \text{zero} \rightarrow (\Pi n : \text{Nat}. \text{Q} n \rightarrow \text{Q} (\text{succ} n)) \rightarrow \\
\Pi x : \text{Nat}. \text{Q} x = \lambda qz. \lambda qs. \lambda x. \\
\text{inductionNatGen} (\lambda \text{ih}. \text{case} \text{ih} \\
\lambda u'. \lambda x. (\eta\text{-unit} u') - qz) \% \text{zero case} \\
(\lambda b. \lambda x. (\pi_1 b) (\pi_2 b))) \% \text{suc case} \\
x.
\]

In the zero case we use the fact \((\eta\text{-unit})\) that type \(\text{Unit}\) has

\[
\text{the unique inhabitant unit, so} \ \eta (\text{eta-unit u'}) \text{rewrites} \\
\text{the goal by equation} u' \equiv \text{unit}.
\]

### 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 \(ListF\):

\[
\text{ListF} \circ \ast \rightarrow \ast \rightarrow \ast = \lambda A : \ast. \lambda x : \ast. \\
\text{Unit} + (A \times X).
\]

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

\[
\text{List} \circ \ast \rightarrow \ast = \lambda A : \ast. \text{FixIndM} (\text{ListF} A).
\]

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

\[
\text{inductionListGen} \circ V : A : \ast.
\]

\[
(V R : \ast. \Pi \text{cast} : R \rightarrow \text{List} A. \\
\Pi r : V R : R. \text{cast} r \equiv r. \\
(\Pi r : R. \text{Q} (\text{cast} r)) \rightarrow \\
\Pi fr : \text{ListF} A R. \text{Q} (\text{inFixIndM} \text{cast fr}) \rightarrow \\
\Pi x : \text{List} A. \text{Q} x = <..>.
\]

We define constructors and the flat version of induction rule:

\[
\text{cons} \circ V : A : \ast. A \rightarrow \text{List} A \rightarrow \text{List} A \\
= \lambda x. \lambda xs. \text{inFixIndM} (\lambda x. x) (\lambda2 \text{pair} x xs).
\]

\[
\text{inductionListM} \circ V : A : \ast. V Q : \text{List} A \rightarrow \ast. \\
\text{Q} \text{nil} \rightarrow \\
(\Pi x : A. \Pi xs : \text{List} A. \text{Q} xs \rightarrow \text{Q} (\text{cons} x xs)) \rightarrow \\
\Pi xs : \text{List} A. \text{Q} xs = \lambda qnil. \lambda qcons. \\
\text{case fr} \\
(\lambda u'. \lambda (\eta\text{-unit} u') - qnil) \% \text{nil case} \\
(\lambda p. (\text{qcons} (\pi_1 p) \% \text{cons case} \\
\text{cast} (\pi_2 p)) (\text{ih} (\pi_2 p))).
\]

It is worth noting that in the “cons” case the inductive hypothesis \(\text{Q} (\text{cast} (\pi_2 p))\) is produced explicitly by invoking function \(\text{ih} : \Pi r : R \rightarrow \text{Q} (\text{cast} r)\).

### 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 \(\text{Fix} F + : F \rightarrow X\) is also a functor. This allows to derive function \(\text{Fix} F \rightarrow X\) from independently defined functions \(\text{Fix} F \rightarrow X\) and \(\text{Fix} F' \rightarrow X\).

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 \(v : \text{Fix} F\) is universal if \(v \equiv \text{fold alg}\) for any algebra \(\text{alg}\) 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 \(\text{Fix} (F + : F')\) from properties of \(\text{Fix} F\) and \(\text{Fix} F'\). 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
We observed that Mendler-style representation of recursive 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 weights 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 $S : \star \to \star$ ($S$ can be mixed-variant—an argument can appear on covariant and contravariant positions) into an isomorphic scheme $S^e : \star \to \star$ 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


