Impredicative Encodings of Inductive-Inductive Data in Cedille

Andrew Marmaduke, Larry Diehl, and Aaron Stump

The University of Iowa, Iowa City, Iowa, U.S.A. `{first}-{last}@uiowa.edu`

Abstract. Cedille is a dependently typed programming language known for expressive and efficient impredicative encodings. In this work, we show that encodings of induction-induction are also possible by employing a standard technique from other encodings in Cedille, where a type representing the shape of data is intersected with a predicate that further constrains Thus, just as with indexed inductive data, Cedille can encode a notion that is often axiomatically postulated or directly implemented in other dependent type theories without sacrificing efficiency.

Keywords: Impredicative Encoding · Induction-Induction · Cedille.

1 Introduction

Induction-induction is an extension of mutual inductive datatypes that further empowers a user to specify exactly the associated inhabitants. Denoted correct-by-construction, constructors are specified so that only the data of interest are expressible which prevents error handling or other boilerplate code for so-called "junk" data. These kinds of definitions where explored in detail by Forsberg et al. [20,11,10,2]. Mutual inductive datatypes in their simplest incarnation define two datatypes whose constructors may refer to the type of the other. The canonical example is the indexed datatypes Even and Odd.

$$\begin{aligned}
&\text{data Even} : \mathbb{N} \to \star \text{ where}\\
&\quad \text{ezer} : \text{Even } 0\\
&\quad \text{esuc} : (n : \mathbb{N}) \to \text{Odd } n \to \text{Even } (\text{suc } n)\\
&\text{data Odd} : \mathbb{N} \to \star \text{ where}\\
&\quad \text{osuc} : (n : \mathbb{N}) \to \text{Even } n \to \text{Odd } (\text{suc } n)
\end{aligned}$$

Induction-induction expands on this by allowing a type to be the *index* of the other. Thus, instead of two mutually defined types $A, B : \star$ there are two types $A : \star$ and $B : A \to \star$ mutually defined. Of course, the types can refer to one another in their constructors as before. The canonical example of induction-induction is a type representing the syntax of a dependent type theory. The Ctx and Ty types (excluding a type representing terms) are defined:

$$
\begin{aligned}
&\text{data Ctx} : \star \text{ where}\\
&\quad \text{nil} : \text{Ctx}\\
&\quad \text{cons} : (\Gamma : \text{Ctx}) \to \text{Ty } \Gamma \to \text{Ctx}\\
&\text{data Ty} : \text{Ctx} \to \star \text{ where}\\
&\quad \text{base} : (\Gamma : \text{Ctx}) \to \text{Ty } \Gamma\\
&\quad \text{arrow} : (\Gamma : \text{Ctx}) \to (A : \text{Ty } \Gamma) \to (B : \text{Ty } (\text{cons } \Gamma\ A)) \to \text{Ty } \Gamma
\end{aligned}
$$

Induction-induction is of particular interest when modeling programming language syntax. Indeed, a more general formulation of quotient inductive-inductive datatypes has been used to model dependent type theories with induction principles modulo definitional equality over syntax [1]. From the perspective of constructing Domain Specific Languages (DSLs) induction-induction is a desirable technique if available.

DSLs are not the only interesting data that can be modelled with induction-induction. A type A and a predicate $P : A \to \star$ may be mutually defined by induction-induction to enforce some desired property on the data of A. For example, a ListSet of natural numbers where all elements must be unique:

$$
\begin{aligned}
&\text{data ListSet} : \star \text{ where}\\
&\quad \text{nil} : \text{ListSet}\\
&\quad \text{cons} : (n : \mathbb{N}) \to (\ell : \text{ListSet}) \to \text{Unique } n\ \ell \to \text{ListSet}\\
&\text{data Unique} : \mathbb{N} \to \text{ListSet} \to \star \text{ where}\\
&\quad \text{triv} : (n : \mathbb{N}) \to \text{Unique } n \text{ nil}\\
&\quad \text{ucons} : (n\ m : \mathbb{N}) \to (\ell : \text{ListSet})\\
&\qquad \to m \neq n \to \text{Unique } m\ \ell \to \text{Unique } m\ (\text{cons } n\ \ell)
\end{aligned}
$$

While such a type can be defined via other methods (e.g. using quotients [18]), it is sometimes easier or more natural to define the property inductively. Additionally, the initial data without the constraining predicate may have no other use, thus a stronger guarantee is conveyed by demanding the data adheres to some predicate in its definition. Finally, there are some constructions in mathematical practice that have natural definitions via induction-induction in dependent type theory such as Conway's Surreal Numbers [20].

This paper reports a novel result that induction-induction is a *derivable* concept within the dependently typed programming language Cedille. Additionally, a generic encoding of induction-induction is formalized in the Cedille tool. In fact, all notions of data are derived by other type constructors in Cedille with induction-induction being the latest example. While other dependent type theories support induction-induction they do so by extending the core theory of datatypes. This is a valid approach, but it is the philosophy of Cedille that a smaller trusted computing base (i.e. a small core type checker) is a more desirable feature when designing a tool for dependent type theories. Moreover, other tools (as of 2022, Coq is one such example) do not permit inductive-inductive datatypes.

2 Background on Cedille

Cedille is a dependently typed programming language with a type theory based on the Calculus of Constructions with three extensions: erased functions, dependent intersections, and equality [21,22]. Many interesting encodings are possible with this theory including inductive data and simulated large eliminations as some examples [8,13].

2.1 Erased Functions and Erasure

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

$$|\Lambda\, x{:}T.\, t| \;=\; |t| \qquad\qquad |t \text{ -}t'| \;=\; |t|$$

Fig. 1: Erased Functions

Erased functions as shown in Figure 1 represent function spaces where the variable may not appear free in the erasure of the body. This type former is inspired by the implicit functions of Miquel [19]. The erasure of a term, $|t|$, is defined with each corresponding extension. Additionally, the definitional equality of the theory is extended to mean $|t_1| \equiv_{\beta\eta} |t_2|$ i.e. that two terms are definitionally equal if the $\beta\eta$-normal forms of their erasures are equivalent up-to renaming. We take the liberty of a syntax style resembling Agda and use $(x : T_1) \Rightarrow T_2$ to be an equivalent syntax for $\forall\, x{:}T_1.\, T_2$. Note that types in Cedille are always erased at the term level.

Erased functions allow for a fine-grained control over the relevant shape of a term which is critical when intersecting. Moreover, indices are almost always conceptually viewed as erased, but this fact is not usually expressible in a type theory. With erased functions, indices can always be marked as erased. Note that erased functions are not like implicit function spaces in other languages where a term is inferred via unification. Instead, an erased function is closer to the erased functions of Quantitative Type Theory [3].

2.2 Dependent Intersections

Inspired by Kopylov [17], dependent intersections, as shown in Figure 2, can be interpreted intuitively as a kind of refinement type. While the namesake makes sense, because the terms of an intersection must be definitionally equal, the

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : [t_1/x]T_2 \quad |t_1| = |t_2|}{\Gamma \vdash [t_1, t_2] : \iota\, x{:}T_1.\, T_2}$$

$$\frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.1 : T_1} \quad \frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.2 : [t.1/x]T_2}$$

$$|[t_1, t_2]| \;=\; |t_1| \quad |t.1| = |t| \quad |t.2| = |t|$$

Fig. 2: Dependent Intersection

usage we are primarily interested in is to constrain some type via a predicate that matches its shape. Note, this is a critical and powerful application which obviates many other practical concerns with refinement types. Again, a syntax style resembling Agda is used with $(x : T_1) \cap T_2$ being equivalent syntax for $\iota\, x : T_1.\, T_2$. While useful for presentation, these alternative syntaxes are not possible in the Cedille tool, thus an inspection of the formalization will require understanding the original syntactic forms.

2.3 Equality

$$\frac{FV(t) \subseteq dom(\Gamma)}{\Gamma \vdash \beta : \{t \simeq t\}} \quad \frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t' : [t_2/x]T}{\Gamma \vdash \rho\ t\ @\ x.T - t' : [t_1/x]T}$$

$$\frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t_1 : T}{\Gamma \vdash \varphi\ t - t_1\ \{t_2\} : T} \quad \frac{\Gamma \vdash t : \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\}}{\Gamma \vdash \delta - t : T}$$

$$|\beta| \;=\; \lambda\, x.\, x \quad |\rho\ t\ @\ x.T - t'| \;=\; |t'|$$

$$|\varphi\ t - t_1\ \{t_2\}| \;=\; |t_2| \quad |\delta - t| \;=\; \lambda\, x.\, x$$

Fig. 3: Equality

The propositional equality of Cedille, as shown in Figure 3, is necessary for reasoning about the shape of terms and finalizing the development of an induction principle for the various possible encodings in Cedille. Cedille's equality is very different from other dependent type theories like Agda. Indeed, it is irrelevant, not inductive, and works over *untyped* terms. We will not directly use the equality type in this work as it is not necessary for the core idea. However, it is still necessary to complete the proof of induction for the various encodings, but the process to do so is standard.

$$\frac{\Gamma \vdash f : S \to T \quad \Gamma \vdash t : \Pi\, x{:}S.\, \{f\ x \simeq x\}}{\Gamma \vdash \mathsf{intrCast}\ \text{-}f\ \text{-}t : \mathsf{Cast}\ S\ T}$$

$$\frac{\Gamma \vdash t : \mathsf{Cast}\ S\ T}{\Gamma \vdash \mathsf{cast}\ \text{-}t : S \to T}$$

$$|\mathsf{intrCast}\ \text{-}f\ \text{-}t| = \lambda\, x.\, x \quad |\mathsf{cast}\ \text{-}t| = \lambda\, x.\, x$$

Fig. 4: Casts

2.4 Casts

A *cast* is a derived construct in Cedille that encodes an identity function between two, potentially definitionally distinct, types. Figure 4 presents a rule-based description of casts. Note that there are trivial casts from a dependent intersection to its first or second component. The φ constructor of the equality type is responsible for creating more interesting inhabitants of the Cast type. Casts are critical to deriving efficient inductive data. By *efficient,* we mean that the production of subdata is emulated with a constant number of β-reductions, and that folds over data is emulated in a proportional number of β-reductions relative to the size of the data. For example, the predecessor function for unary Natural numbers should be $\mathcal{O}(1)$ reductions and addition should be $\mathcal{O}(n)$ reductions in the first argument.

2.5 Indexed Inductive Data

Indexed-inductive datatypes are also a *derived* notion in Cedille. The Cedille tool supports special syntax with motive inference and other quality of life improvements to ease working with data. A complete derivation of indexed-inductive data is provided in the formalization [encoding]. Additionally, all formalized work presented in this paper uses this encoding. However, occasionally the convenient syntax is used instead, particularly when defining functors that are provided to generic encodings. This syntax allows the construction of data, e.g. of Natural numbers:

$$\begin{aligned} &\text{data Nat } : \star = \\ &\quad \mid \text{zero} : \text{Nat} \\ &\quad \mid \text{succ} : \text{Nat} \to \text{Nat} \end{aligned}$$

and an inductive eliminator, e.g. if $n : \text{Nat}$ then

$$\begin{aligned} &\mu\ ih.\ n\ @\ \lambda\, i.\, \text{Motive}\ i\ \{ \\ &\quad \mid \text{zero} \to \ldots \\ &\quad \mid \text{succ}\ x \to \ldots\ \} \end{aligned}$$

is an induction with ih binding the inductive hypothesis and producing a value of type Motive n with cases for each constructor as expected. Finally, there is a pattern matching form of the above induction principle $\mu'\ n$ which elides the inductive hypothesis. Further discussion of the core ideas will not use the above syntax and instead will take an informal approach, but this syntax is necessary for understanding the formalization.

3 Induction-Induction Encoding

3.1 The Core Idea

Impredicative encodings of inductive data follow from the observation that a simple view of the type in terms of System F and an induction principle stated relative to this simpler view yields the full inductive type when intersected. For example, consider a Church encoded Natural number where we first define the standard impredicative encoding in System F.

$$\mathrm{CNat} = (X : \star) \Rightarrow X \to (X \to X) \to X$$

Then, the inductive predicate we expect of natural numbers but stated relative to CNats:

$$\begin{aligned}\mathrm{CNatInd} = \lambda\, n.\, (P : \mathrm{CNat} \to \star) \Rightarrow P\ \mathrm{czero}\\ \to ((x : \mathrm{CNat}) \Rightarrow P\ x \to P\ (\mathrm{csucc}\ x)) \to P\ n\end{aligned}$$

Note that, critically, the subdata in the successor case of the induction predicative is quantified with an erased arrow. This allows the computational content of both types to match while simultaneously allowing for the expected induction principle to be stated. Now, the full inductive type is the intersection,

$$\mathrm{Nat} = (x : \mathrm{CNat}) \cap \mathrm{CNatInd}\ x$$

where the correct induction principle in terms of Nat is derivable.

The same core idea works for reducing inductive-inductive data to indexed inductive data. For example, the canonical example of Ctx and Ty is encoded first by defining a mutual inductive type representing the shape of the type.

$$\begin{aligned}&\mathrm{data\ Pre} : \mathbb{B} \to \star\ \mathrm{where}\\ &\quad \mathrm{pnil} : \mathrm{Pre\ tt}\\ &\quad \mathrm{pcons} : \mathrm{Pre\ tt} \to \mathrm{Pre\ ff} \to \mathrm{Pre\ tt}\\ &\quad \mathrm{pbase} : \mathrm{Pre\ tt} \to \mathrm{Pre\ ff}\\ &\quad \mathrm{parrow} : \mathrm{Pre\ tt} \to \mathrm{Pre\ ff} \to \mathrm{Pre\ ff} \to \mathrm{Pre\ ff}\end{aligned}$$

Now Pre tt is the PreCtx and Pre ff is the PreTy, the initial shapes of both types. Second, we construct a predicate over Pre types capturing induction relative to a Pre value.

$$
\begin{array}{l}
\text{data Ind} : (b : \mathbb{B}) \to \text{elim } b \to \star \text{ where} \\
\quad \text{gnil} : \text{Ind tt } (\text{in}_1 \text{ pnil}) \\
\quad \text{gcons} : (c : \text{PreCtx}) \Rightarrow \text{Ind tt } (\text{in}_1\ c) \\
\qquad \to (t : \text{PreTy}) \Rightarrow \text{Ind ff } (\text{in}_2\ c\ t) \\
\qquad \to \text{Ind tt } (\text{in}_1\ (\text{pcons } c\ t)) \\
\quad \text{gbase} : (c : \text{PreCtx}) \Rightarrow \text{Ind tt } (\text{in}_1\ c) \to \text{Ind ff } (\text{in}_2\ (\text{pbase } c)) \\
\quad \text{garrow} : (c : \text{PreCtx}) \Rightarrow \text{Ind tt } (\text{in}_1\ c) \\
\qquad \to (a : \text{PreTy}) \Rightarrow \text{Ind ff } (\text{in}_2\ c\ a) \\
\qquad \to (b : \text{PreTy}) \Rightarrow \text{Ind ff } (\text{in}_2\ (\text{pcons } c\ a)\ b) \\
\qquad \to \text{Ind ff } (\text{in}_2\ c\ (\text{parrow } c\ a\ b))
\end{array}
$$

Where elim b is a simulated large elimination [13] with constructors

$$
\begin{array}{l}
\text{in}_1 : \text{Pre tt} \to \text{elim tt} \\
\text{in}_2 : \text{Pre tt} \to \text{Pre ff} \to \text{elim ff}
\end{array}
$$

Notice that the type Ind, when erased arrows and the elim b dependencie are removed, is *exactly* the Pre type. Moreover, in Cedille when two inductive types are defined with the same number of constructors, as long as the relevant types of those constructors agree, then the constructors themselves are equal. Thus, we have that pnil is definitionally equal to gnil, and likewise for the three remaining constructors. Conceptually, we will see that we can also view Ind as capturing the canonical (or normal) elements of Pre that are inductive.

Now, the complete inductive-inductive types may be defined by the intersections:

$$
\begin{array}{l}
\text{Ctx} = (x : \text{PreCtx}) \cap \text{Ind tt } (\text{in}_1\ x) \\
\text{Ty } c = (x : \text{PreTy}) \cap \text{Ind ff } (\text{in}_2\ c.1\ x)
\end{array}
$$

Again, the expected induction principles are derivable in terms of Ctx and Ty. Note that the technique of encoding mutual inductive types via indexed inductive types is a standard trick [16]. Moreover, the efficiency of the encoding is dependent entirely on the efficiency of the underlying indexed-inductive data encoding.

3.2 Generic Encoding: First Variant

To obtain a generic version of the core idea we first must describe a functor representation of the required data. First, the ShapeF functor must encode the shape of all possible constructors. Let n be the number of datatypes under definition, then

$$(\text{Fin } n \to \star) \to \text{Fin } n \to \star$$

is the type of the ShapeF functor. Of course, in order for this to really be a functor it must be monotonic. A functor F with index I is *monotonic* if it preserves casts on indexed data, or concretely if it satisfies the following property:

$$\begin{aligned}&(A : I \to \star) \to (B : I \to \star)\\&\quad \to ((i : I) \to \text{Cast } (A\ i)\ (B\ i))\\&\quad \to (i : I) \to \text{Cast } (F\ A\ i)\ (F\ B\ i)\end{aligned}$$

Second, we must constrain the shape with what we will suggestively call the NormalF functor. Let Shape be the inductive type corresponding to a monotonic ShapeF functor. Let Idx be the type $(i : \text{Fin } n) \times (\text{Tuple } n \text{ Shape } (\text{fsucc -}n\ i))$ where $(a : A) \times P\ a$ is a derived sigma type and Tuple n Shape (fsucc -n i) is a derived simulated large elimination with the computation rules:

$$\text{tupleP} : \text{Tuple } n \text{ Shape } (\text{fsucc -}n\ i) \to (\text{Shape } i) \times (\text{Tuple } n \text{ Shape } i)$$

$$\text{tupleS} : (\text{Shape } i) \times (\text{Tuple } n \text{ Shape } i) \to \text{Tuple } n \text{ Shape } (\text{fsucc -}n\ i)$$

$$\text{tupleZ} : \text{Tuple } n \text{ Shape } 0 \to \text{Unit}$$

The formalization of Tuple is available in [lib/tuple.ced]. Now, the type of NormalF is

$$\text{NormalF} : (\text{Idx} \to \star) \to \text{Idx} \to \star$$

and it must additionally be monotonic relative to Idx.

Finally, we must know that the computational shape of NormalF data is the same as ShapeF data. That is, we require a cast:

$$(i : \text{Idx}) \to \text{Cast } (\text{NormalF } i)\ (\text{ShapeF } (\text{fst } i))$$

With all of this input data supplied the core idea may be carried out. Let QIdx be the type $(i : \text{Fin } n) \times (\text{Tuple } n \text{ Shape } i)$, Shape be the inductive type constructed from ShapeF, and Normal the inductive type constructed from NormalF. Then the *quotient* is

$$\text{Quotient } i = (s : \text{Shape}) \cap (\text{Normal } (\text{pair } (\text{fst } i)\ (\text{tupleS } (\text{pair } s\ i))))$$

Notice that QIdx contains the indexes of all previously defined types in the chain, i.e. the types with fewer indices. The final Shape missing from QIdx is the very Shape to be constrained by the corresponding Normal. We have called this type a quotient because this construction, upon further reflection, is essentially a generic quotient construction. The complete formalization of this encoding is available in [indind.ced]. However, it has two flaws:

1. the input data is onerous, requiring specifying the constructors twice and proving an unnecessary cast;
2. and the indices of Quotient are Shapes instead of Quotients.

If we want to elaborate a custom syntactic representation of induction-induction to this encoding then these problems are recoverable, but implementing this custom syntax is time-consuming. Moreover, Cedille's theory has a wealth of potential inductive encodings whose limits are not yet realized. Instead, we discuss a way of fixing the above hiccups internally.

3.3 Generic Encoding: Second Variant

Luckily, we do not need to start over. We will define a smaller set of inputs and use those inputs to construct the necessary data for the quotient construction. Then, using the resulting types and induction principles from the quotient encoding, build the expected types and induction principles for the smaller input.

For this variant we focus on constructing only two types as opposed to any arbitrary n types. Consider a functor

$$F : (X : \star) \to (Y : X \to \star) \to \star$$

where X is conceptually the abstract representation of the first inductive type and Y is the abstract representation of the second inductive type. Then, a functor for the second type

$$G : (X : \star) \to (Y : X \to \star) \to (alg : F\ X\ Y \to X) \to X \to \star$$

has almost the same signature but crucially must be given an algebra to describe how to construct abstract Xs from $F\ X\ Y$ data. For example, CtxF and TyF functors are defined in this style as:

$$\begin{aligned}
&\text{data CtxF } (X : \star)\ (Y : X \to \star)\ :\ \star \text{ where}\\
&\quad \text{nilF} : \text{CtxF}\\
&\quad \text{consF} : (g : X) \to Y\ g \to \text{CtxF}
\end{aligned}$$

$$\begin{aligned}
&\text{data TyF } (X : \star)\ (Y : X \to \star)\ (alg : \text{CtxF } X\ Y \to X)\ :\ X \to \star \text{ where}\\
&\quad \text{baseF} : (g : X) \to \text{TyF } g\\
&\quad \text{arrowF} : (g : X) \to (a : Y\ g) \to (b : Y\ (alg\ (\text{consF } g\ a))) \to \text{TyF } g
\end{aligned}$$

Note that the arrowF constructor is only possible with the additional alg parameter. These new functors require their own conditions of monotonicity, but the core idea is the same: the functor must preserve casts. In order to specify this requirement for the G functor there must be two algebras. However, the algebras should both correspond to the constructor for CtxF! Therefore, the additional restraint is imposed that the algebras are definitionally equal. The formalized monotonicity conditions are available in [indind2/mono.ced]. The two functors, F and G, with proofs that they are monotonic is all that is needed to derive the corresponding inductive types.

With this input data we proceed by defining a ShapeF and NormalF to instantiate the quotient encoding. To define ShapeF, let $R : \text{Fin } 2 \to \star$ be the abstract type, then we have two constructors:

$$\text{ShapeFinF} : F\ (R\ 0)\ (\lambda\ \_.\ R\ 1) \to \text{ShapeF}\ 0$$

$$\begin{aligned}&\text{ShapeFinG} : \forall\ A\ mA\ inj \Rightarrow (r : \text{IndA}\ 0) \Rightarrow\\ &\quad G\ (\text{IndA}\ 0)\ (\lambda\, i.\, \text{inA}\ (inj\ i))\ r\\ &\quad \to \text{ShapeF}\ 1\end{aligned}$$

Note a peculiarity in the definition of ShapeFinG. We must postulate a functor A of the same type as ShapeF that is monotonic (mA) and has an injection (inj). With this data, an inductive type (IndA) is instantiated with a corresponding constructor (inA). This abstracts the inductive type Shape that can not yet be spoken about, but is otherwise needed to enable typing G data. Proving monotonicity for this functor is straightforward. The full formalization is available in [indind2/shape.ced].

Next we define NormalF, recall that the index is defined in terms of Tuple, although the size of the Tuple is restricted to two types. To simplify the presentation we assume the functions

$$\text{in}_1 : \text{Shape}\ 0 \to \text{Idx}$$

and

$$\text{in}_2 : \text{Shape}\ 1 \to \text{Shape}\ 0 \to \text{Idx}$$

have been defined. Also, we elide casts to reduce the noise in the presentation. Let $R : \text{Idx} \to \star$ be the abstract type.

$$\begin{aligned}&\text{let } A\ 0 = (s : \text{Shape}\ 0) \cap R\ (\text{in}_1 c)\\ &\text{let } A\ 1 = \lambda\, x{:}A\ 0.\, (s : \text{Shape}\ 1) \cap R\ (\text{in}_2\ s\ x.1)\\ &\text{NormalFinF} : (xs : F\ (A\ 0)\ (A\ 1))\\ &\quad \text{NormalF}\ (\text{in}_1\ (\text{inShape -0}\ (\text{ShapeFinF xs})))\end{aligned}$$

The constructor NormalFinG follows the same pattern as ShapeFinG by abstracting an arbitrary functor A that stands in for a Normal as opposed to a Shape. The only additional requirement is that the inductive type IndA must be castable to the abstract type R. We elide its concrete definition because it is technical but not conceptually harder than ShapeFinG and thus not illuminating. The full formalization is available in [indind2/constraint.ced].

The definitions are carefully chosen so that a NormalF casts into a ShapeF for compatible indices which completes all required input to instantiate the quotient generic encoding. Now, after instantiation, the types TypeF : $\star$ and TypeG : TypeF $\to \star$ are definable. Notice that TypeG has the correct index. Moreover, the constructors:

$$\text{inF} : F\ \text{TypeF}\ \text{TypeG} \to \text{TypeF}$$

$$\text{inG} : (i : \text{TypeF}) \Rightarrow G\ \text{TypeF}\ \text{TypeG}\ \text{inF}\ i \to \text{TypeG}\ i$$

and the associated induction principles:

$$\begin{aligned}\text{inductF} : (P : \text{TypeF} \to \star) \to (Q : (i : \text{TypeF}) \to \text{TypeG}\ i \to \star)\\ \to \text{PrfAlgF}\ P\ Q \to \text{PrfAlgG}\ P\ Q \to (x : \text{TypeF}) \to P\ x\end{aligned}$$

$$\begin{aligned}\text{inductG} : (P : \text{TypeF} \to \star) \to (Q : (i : \text{TypeF}) \to \text{TypeG}\ i \to \star)\\ \to \text{PrfAlgF}\ P\ Q \to \text{PrfAlgG}\ P\ Q \to (i : \text{TypeF}) \Rightarrow (x : \text{TypeG}\ i) \to Q\ i\ x\end{aligned}$$

are all derived. We direct the reader to the formalization for the definition of these functions and the definition of the associated proof algebras [indind2/ind.ced]. Note that the proof algebras follow the same pattern as the efficient Mendler-style proof algebras of previous Cedille encodings [9]. Indeed, these inductive encodings *are* Mendler-style inductive types as they reduce, now through an additional layer, to Mendler-style indexed-inductive data. An example of this encoding applied to CtxF and TyF where more standard induction principles are defined is also available [example.ced].

This second variant has the correct indices and has a smaller input burden. Of course, an elaborated definition would be able to impose a syntactic restriction to automatically derive monotonicity, but that is a price that must be paid for the additional flexibility of a semantic criterion of monotonicity.

4 Related and Future Work

Inductive inductive definitions where studied extensively by Forsberg et al [20,11]. Forsberg's thesis was used extensively and heavily inspired this work [10]. In particular, Forsberg describes an axiomatic description of Inductive-inductive types and shows how to model them with indexed inductive types when using equality reflection (i.e. uniqueness or identity proofs and function extensionality). Moreover, his work is relative to a predicative theory and is presented via a theory of containers where inductive types are presented by a type of codes. Kaposi et al. expand on this reduction showing that inductive data are sufficient for finitary inductive-inductive types in Extensional Type Theory [15]. The first variant of the generic encoding we present differs from Forsberg's in that it does not use codes for types and instead impredicativity and does not require an equality reflection rule or function extensionality. Like Forsberg's translation (but not like his axiomatic description), our encoding is limited to "simple" inductive motives. However, we conjecture that general inductive motives are also possible by postulating a second set of general proof algebras and using the simple eliminator to construct the correct motives in the general case. Critically, if the eliminators are definitionally equal then when we move from the first variant to the second variant the difference will disappear.

Quotient inductive-inductive types are an extension that have been demonstrated as a powerful technique for internalizing the definition of dependent type theories [1,4,14]. The first generic variant is suggestively described as a *quotient*. Indeed, many constructions in Cedille via dependent intersection might fruitfully be cast in a framework of quotients. However, these kinds of quotients are through a normalization argument only and thus quotients such as multisets would not be possible. It is an open question if quotient inductive-inductive definitions could be generically encoded in Cedille using similar techniques, but we conjecture that, under circumstances where the equivalence relation has decidable canonical elements, that it is possible.

Induction-recursion is another powerful technique for defining universes of type codes [5,6,7]. Cedille does not possess large eliminations natively because it does not possess inductive data natively. Thus, computing types by recursion is relegated to simulated computation rules or type inclusions. However, these simulated large eliminations are encoded themselves via inductive definitions. It stands to reason that if a simulated large elimination is an inductive type already, that an inductive-inductive definition could yield a simulated inductive-recursive definition. Note that this is different from *small* induction-recursion where the recursive function computes a term instead of a type [12].

5 Conclusion

In this work we have shown how to encode inductive-inductive data in Cedille, a dependently type programming language based on the Calculus of Constructions with three extensions. Our first generic construction follows a construction of Forsberg in reducing inductive-inductive data to indexed inductive data where we define first the shape of the constructors and then a predicate to quotient normal forms. Second, we demonstrate another framework that layers on the first and fixes redundancy and indices problems in the prior reduction. This is another chapter in the story of encoding efficient inductive data in Cedille and it is the author's opinion that there is still a lot of potential left to be uncovered.

References

1. Altenkirch, T., Capriotti, P., Dijkstra, G., Kraus, N., Nordvall Forsberg, F.: Quotient inductive-inductive types. In: International Conference on Foundations of Software Science and Computation Structures. pp. 293–310. Springer, Cham (2018)
2. Altenkirch, T., Morris, P., Nordvall Forsberg, F., Setzer, A.: A categorical semantics for inductive-inductive definitions. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) Algebra and Coalgebra in Computer Science. pp. 70–84. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
3. Atkey, R.: Syntax and semantics of quantitative type theory. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 56–65 (2018)
4. Dijkstra, G.: Quotient inductive-inductive definitions. Ph.D. thesis, University of Nottingham (2017)
5. Dybjer, P., Setzer, A.: A finite axiomatization of inductive-recursive definitions. In: Typed Lambda Calculi and Applications: 4th International Conference, TLCA'99 L'Aquila, Italy, April 7–9, 1999 Proceedings 4. pp. 129–146. Springer (1999)

6. Dybjer, P., Setzer, A.: Induction–recursion and initial algebras. Annals of Pure and Applied Logic **124**(1-3), 1–47 (2003)
7. Dybjer, P., Setzer, A.: Indexed induction–recursion. The Journal of Logic and Algebraic Programming **66**(1), 1–49 (2006)
8. Firsov, D., Blair, R., Stump, A.: Efficient Mendler-style lambda-encodings in Cedille. In: International Conference on Interactive Theorem Proving. pp. 235–252. Springer (2018)
9. Firsov, D., Blair, R., Stump, A.: Efficient mendler-style lambda-encodings in cedille. In: Interactive Theorem Proving: 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings 9. pp. 235–252. Springer (2018)
10. Forsberg, F.N.: Inductive-inductive definitions. Ph.D. thesis, Swansea University (2013), `http://login.proxy.lib.uiowa.edu/login?url=https://www.proquest.com/dissertations-theses/inductive-definitions/docview/2041902169/se-2`, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2022-10-21
11. Forsberg, F.N., Setzer, A.: A finite axiomatisation of inductive-inductive definitions. Logic, Construction, Computation **3**, 259–287 (2012)
12. Hancock, P., McBride, C., Ghani, N., Malatesta, L., Altenkirch, T.: Small induction recursion. In: Typed Lambda Calculi and Applications: 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings 11. pp. 156–172. Springer (2013)
13. Jenkins, C., Marmaduke, A., Stump, A.: Simulating Large Eliminations in Cedille. In: Basold, H., Cockx, J., Ghilezan, S. (eds.) 27th International Conference on Types for Proofs and Programs (TYPES 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 239, pp. 9:1–9:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). `https://doi.org/10.4230/LIPIcs.TYPES.2021.9`, `https://drops.dagstuhl.de/opus/volltexte/2022/16778`
14. Kaposi, A., Kovács, A., Altenkirch, T.: Constructing quotient inductive-inductive types. Proceedings of the ACM on Programming Languages **3**(POPL), 1–24 (2019)
15. Kaposi, A., Kovács, A., Lafont, A.: For finitary induction-induction, induction is enough. In: TYPES 2019: 25th International Conference on Types for Proofs and Programs. vol. 175, pp. 6–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2019)
16. Kaposi, A., von Raumer, J.: A syntax for mutual inductive families (2020)
17. Kopylov, A.: Dependent intersection: A new way of defining records in type theory. In: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science. pp. 86–. LICS '03, IEEE Computer Society, Washington, DC, USA (2003)
18. Marmaduke, A., Jenkins, C., Stump, A.: Quotients by idempotent functions in cedille. In: Bowman, W.J., Garcia, R. (eds.) Trends in Functional Programming. pp. 1–20. Springer International Publishing, Cham (2020)
19. Miquel, A.: The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In: Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications. pp. 344–359. TLCA'01, Springer-Verlag, Berlin, Heidelberg (2001)
20. Nordvall Forsberg, F., Setzer, A.: Inductive-inductive definitions. In: Dawar, A., Veith, H. (eds.) Computer Science Logic. pp. 454–468. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
21. Stump, A.: The calculus of dependent lambda eliminations. Journal of Functional Programming **27**, e14 (2017)

22. Stump, A.: From realizability to induction via dependent intersection. Ann. Pure Appl. Logic **169**(7), 637–655 (2018). `https://doi.org/10.1016/j.apal.2018.03.002`, `https://doi.org/10.1016/j.apal.2018.03.002`