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 by intersecting a type representing the shape of data with a predicate that further constrains it. 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 is 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. [9,4,2]. Mutual inductive datatypes in its simplest incarnation define two datatypes whose constructors may refer to the type of one another. The canonical example is the indexed datatypes Even and Odd.

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

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, below

the Ctx and Ty types (excluding a type representing terms) are defined.

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

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. Suppose we have a type A then 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 where all elements must be unique. While such a type can be defined via other methods (e.g. using quotients [7]), 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 [9].

This paper reports a novel result that induction-induction is a *derivable* concept within the dependently typed programming language Cedille. 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 induction-induction when defining data.

2 Background on Cedille

Cedille is a dependently typed programming language with a type theory based on the Calculus of Constructions with three extensions [10,11]. Many interesting encodings are possible with this theory including inductive data and simulated large eliminations as some examples [3,5].

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

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

Fig. 1: Implicit Functions

2.1 Erased Functions and Erasure

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 [8]. 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 more Agda-like syntax style than traditional Cedille 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.

2.2 Dependent Intersections

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

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

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

Fig. 2: Dependent Intersection

Inspired by Kopylov [6], 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 usage we are primarily interested in is to constrain some type via a predicate that matches its shape. Again, a more Agda-like syntax style is used with $(x : T_1) \cap (T_2)$ being equivalent syntax for $\iota\, x{:}T_1.\, T_2$.

2.3 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. We will not directly use

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

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

$$|\beta\{t'\}| \ = \ |t'| \quad |\rho\ t\ @\ x.T - t'| \ = \ |t'|$$

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

Fig. 3: Equality

the 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 final encoding, but the process to do so is standard for every other impredicative encoding carried out in Cedille.

3 Induction-Induction Encoding

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 as shown below

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

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

The same core idea works for reducing induction-induction to indexed inductive types. For example, the canonical example of Ctx and Ty is encoded first by

defining a mutual inductive type representing the shape of the type.

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

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.

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

Where elim b is a simulated large elimination with $\text{in}_1 : \text{Pre tt} \to \text{elim tt}$ and $\text{in}_2 : \text{Pre tt} \to \text{Pre ff} \to \text{elim ff}$ [5]. Then, the complete inductive-inductive types may be defined by the intersections.

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

Again, the expected induction principles are derivable in terms of Ctx and Ty.

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. 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)

4. Forsberg, F.N., Setzer, A.: A finite axiomatisation of inductive-inductive definitions. Logic, Construction, Computation **3**, 259–287 (2012)
5. 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
6. 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)
7. 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)
8. 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)
9. 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)
10. Stump, A.: The calculus of dependent lambda eliminations. Journal of Functional Programming **27**, e14 (2017)
11. 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