

Elaborating Inductive Definitions and Course-of-Values Induction in Cedille

Christopher Jenkins

Computer Science
University of Iowa
Iowa City, Iowa, United States of
America
christopher-jenkins@uiowa.edu

Colin McDonald

Computer Science
University of Iowa
Iowa City, Iowa, United States
colinmcd0731@gmail.com

Aaron Stump

Professor
Computer Science
University of Iowa
Iowa City, Iowa, United States
aaron-stump@uiowa.edu

Abstract

In the Calculus of Dependent Lambda Eliminations (CDLE), a pure Curry-style type theory, it is possible to generically λ -encode inductive datatypes which support *course-of-values* (CoV) induction. We present a datatype subsystem for Cedille (an implementation of CDLE) that provides this feature to programmers through convenient notation for declaring datatypes and for defining functions over them by case analysis and fixpoint-style recursion guarded by a type-based termination checker. We demonstrate that this does not require extending CDLE by showing how datatypes and functions over them elaborate to λ -encodings, and proving that this elaboration is type- and value-preserving. This datatype subsystem and elaborator are implemented in Cedille, establishing for the first time a complete translation of inductive definitions to a small pure typed λ -calculus.

Keywords lambda-encodings, inductive datatypes, dependent types, pattern matching, elaboration, course-of-values

1 Introduction

Algebraic datatypes (ADTs) are a popular feature of functional programming languages that combine a concise scheme for declaring datatypes and their constructors with an intuitive mechanism for defining functions over them by pattern matching and recursion. Their popularity extends to implementations of type theories, wherein properties of data are proven using the same mechanisms as for defining functions over them. However, a wrinkle in bringing ADTs to proof assistants based on type theory is concern for the *de Bruijn criterion* [Geuvers 2009], i.e., that the assistant produce proof objects checkable by an implementation of a small kernel theory. Of particular concern are *termination checking* for recursive definitions and *positivity checking* for data declarations, since to maintain logical soundness implementations must usually ensure functions and proofs are well-founded ([Mendler 1991] showed that non-positive datatypes can be used to define looping terms, without any apparent recursion in the language).

The most common approaches to both positivity and termination checking are *syntactic*: for the former, this involves tracking, in the types of constructor arguments, the number of arrows of which a recursive occurrence of a datatype is to the left (c.f. [INRIA 2017, Section 4.5.2]); for the latter, recursive invocations are allowed only on subdata revealed by pattern matching within the function [Giménez 1995]. Any such syntactic criteria must usually be implemented in the kernel language also, increasing its complexity. This situation is especially unfortunate for termination checking, as simpler syntactic guards are brittle so it is tempting to make these more sophisticated to grow the set of accepted definitions.

Happily, positivity and termination checking have more *semantic* approaches: for the former, explicit evidence of positivity can be required to form an inductive type [Matthes 2002], or polarity annotations can be added to the language of kinds [Abel 2006]; for the latter, *type-based* termination checking augments the type system itself with some notion of the size of datatypes [Abel 2010; Barthe et al. 2004]. Such principled extensions have more modest impact on the complexity of the kernel language, though can require reworking of pre-existing meta-theoretic results.

In pure type systems, datatypes are defined with λ -encodings that combine case analysis and recursion into a single scheme that ensures termination. Cedille [Stump 2017, 2018] is a dependently typed programming language which overcomes some traditional shortcomings of λ -encodings in type theory (e.g., underivability of induction [Geuvers 2001]). Cedille's core theory, the *Calculus of Dependent Lambda Eliminations* (CDLE), is a compact pure Curry-style type theory with no primitive notion of inductive datatypes. Instead, and as shown by [Firsov et al. 2018a], it is possible to generically derive the induction principle for λ -encoded data using a Mendler-style of encoding that features constant-time predecessors and a linear-space representation. Furthermore, [Firsov et al. 2018b] show how to further augment this with *course-of-values* (CoV) induction, an expressive scheme wherein recursive calls are allowed on nested subdata at unbounded depth and whose well-foundedness is tricky to convey to syntactic termination checkers.

Contributions Most programmers (and type theorists!) do not wish to work directly with λ -encodings. Building off the work of [Firsov et al. 2018a,b], in this paper we add *language-level* support for inductive types in Cedille by presenting a datatype subsystem with convenient notation for declaring datatypes and functions defined over them using pattern matching and fixpoint-style recursion. In particular, we:

- design a *semantic* (type-based) termination checker based on *CoV pattern matching*, a novel feature allowing Cedille to accept recursive definitions expressed as CoV induction schemes (Section 2);
- show how datatype declarations and functions over them are elaborated to λ -encodings in Cedille (Sections 4 and 5); and
- prove that elaboration is type- and value-preserving, demonstrating that the above can be achieved without extension of CDLE (Sections 4.1, 5.1, and 5.2).

The datatype system and elaborator are implemented in Cedille (github.com/cedille/cedille). Our approach demonstrates that inductive definitions in constructive type theory can be soundly translated down to a very small pure type system. Indeed, there is already a translation from Cedille 1.0.0 (which does not have datatypes) to Cedille Core, a minimal specification of CDLE implemented in ~1K Haskell LoC. This paper and its proof appendix treats formally only the elaboration of non-indexed datatypes to Cedille 1.0.0.

The remainder of this paper is organized as follows: in Section 1.1 we review CDLE; in Section 1.2 we describe datatype system using standard examples; Section 2 explains CoV pattern matching; in Section 3 we describe the elaborator interface; in Section 4 we formally treat elaboration of datatype declarations; in Section 5 we explain elaboration of functions over data using CoV pattern-matching and recursion; and in Sections 6 and 7 we discuss related and future work.

1.1 Background: CDLE

We review the Calculus of Dependent Lambda Eliminations (CDLE), the type theory of Cedille. CDLE is an extension of the impredicative Curry-style (i.e., extrinsically typed) Calculus of Constructions (CC) that adds three new type constructs: equality of untyped terms ($\{t \simeq t'\}$); dependent intersections ($\iota x : T. T'$) of [Kopylov 2003]; and the implicit (erased) products ($\forall x : T. T'$) of [Miquel 2001]. The pure term language of CDLE is the untyped λ -calculus; to make type checking algorithmic, terms in Cedille are type annotated, and definitional equality of terms is modulo erasure of annotations. The typing and erasure rules for the fragment of CDLE relevant to this paper are given in Figure 1, with a full listing given in [Stump 2018] and this paper's proof appendix.

Equality $\{t_1 \simeq t_2\}$ is the type of proofs that the erasures of t_1 and t_2 (resp. $|t_1|$ and $|t_2|$) are equal. It is introduced with β (erasing to $\lambda x. x$) proving $\{t \simeq t\}$ for any untyped term

t . Combined with definitional equality, β can prove $\{t_1 \simeq t_2\}$ for any $\beta\eta$ -convertible t_1 and t_2 whose free variables are declared in the context. Equality proofs can be eliminated with φ , where the expression $\varphi t - t_1 \{t_2\}$ (erasing to $|t_2|$) casts t_2 to the type of t_1 when t proves t_1 and t_2 are equal.

Dependent intersection $\iota x : T. T'$ is the type of terms t which can be assigned both type T and $[t/x]T'$, and in the annotated language is introduced by $[t_1, t_2]$, where t_1 has type T , t_2 has type $[t_1/x]T'$, and $|t_1| =_{\beta\eta} |t_2|$. Dependent intersections are eliminated with projections $t.1$ and $t.2$, selecting resp. the view that term t has type T or $[t.1/x]T'$

Implicit product $\forall x : T. T'$ is the type of dependent functions with an erased argument of type T and a result of type T' . They are introduced with $\Lambda x : T. t$, provided x does not occur free in $|t|$, and they are eliminated with erased application $t_1 -t_2$. Erased arguments play no computational role and exist solely for the purposes of typing.

Figure 1 omits typing and erasure rules for the term and type constructs of CC. In terms, all type annotations and abstractions (also using Λ) are erased, and the argument of term to type applications (written $t \cdot S$) is erased. In types, \forall and λ resp. quantify and abstract over types, and type to type application is written $T \cdot S$. In code listings, we omit type arguments and annotations when Cedille can infer these.

1.2 Datatypes in Cedille

Declarations Figure 2a show definitions of well-known types using Cedille's datatype subsystem. The general scheme for declaring datatypes in Cedille should be straightforward to anyone familiar with GADTs in Haskell or with dependently typed languages like Agda, Coq, or Idris. We note some differences from the usual convention below.

- Occurrences of the inductive type being defined are not written applied to its parameters. For example, the constructor `nil` is written having type `List` rather than `List · A`; used outside of the datatype declaration, `nil` has the usual type $\forall A : \star. \text{List} \cdot A$.
- In constructor types, recursive occurrences of the datatype (such as `Nat` in `suc : Nat → Nat` must be positive, but *need not be* strictly positive ([Blanqui 2005] showed strict positivity is not needed for small datatypes).
- Declarations can only refer to the datatype itself and prior definitions. Inductive-recursive and inductive-inductive definitions are not part of this proposal.

Functions To continue to familiarize the reader with Cedille's syntax, Figure 2b shows a few standard examples of functional and dependently typed programs. Function `pred` introduces operator μ' for *CoV pattern matching*, where it is used for standard pattern matching. Its operational semantics (Section 5.2) is the usual case branch selection. In `pred`, μ' is given scrutinee n of type `Nat` and a case tree with branches for each constructor of `Nat`.

$$\begin{array}{c}
\frac{FV(t) \subseteq \text{dom}(\Gamma) \quad \Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t_1 : T}{\Gamma \vdash \beta : \{t \simeq t\}} \quad \frac{\Gamma \vdash \varphi t - t_1 \{t_2\} : T}{\Gamma \vdash \varphi t - t_1 \{t_2\} : T} \\
\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : [t_1/x]T_2 \quad |t_1| = |t_2|}{\Gamma \vdash [t_1, t_2] : \iota x : T_1. T_2} \quad \frac{\Gamma \vdash t : \iota x : T_1. T_2}{\Gamma \vdash t.1 : T_1} \quad \frac{\Gamma \vdash t : \iota x : T_1. T_2}{\Gamma \vdash t.2 : [t.1/x]T_2} \\
\frac{\Gamma, x : T \vdash t' : T' \quad x \notin FV(|t'|)}{\Gamma \vdash \Lambda x : T. t' : \forall x : T. T'} \quad \frac{\Gamma \vdash t : \forall x : T'. T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t - t' : [t'/x]T} \\
\begin{array}{l}
|\beta| = \lambda x. x \\
|\varphi t - t_1 \{t_2\}| = |t_2| \\
|[t_1, t_2]| = |t_1| \\
|t.1| = |t| \\
|t.2| = |t| \\
|\Lambda x : T. t| = |t| \\
|t - t'| = |t|
\end{array}
\end{array}$$

Figure 1. Kinding, typing, and erasure for a fragment of CDLE

(a) Datatype declarations

```

data Bool: ★ = tt: Bool | ff: Bool.
data Nat: ★ = zero: Nat | suc: Nat → Nat.
data List (A: ★): ★
= nil: List | cons: A → List → List.

```

(b) Functions

```

pred: Nat → Nat = λ n. μ' n {zero → n | suc n' → n'}.
add: Nat → Nat → Nat
= λ m. λ n. μ addN. m {zero → n | suc m' → suc (addN m')}.

```

Figure 2. Example datatype declarations and functions

Function `add` introduces operator μ for *CoV induction* by combined pattern matching and recursion; the distinction between pattern matching by μ and μ' is made clear in Section 2. Its operational semantics is combined case branch selection and fixpoint unrolling. Here, μ is used for standard structurally recursive definitions: in `add` it is used to define function `addN` (so named because it adds n to its argument), and in the successor case `addN` is recursively invoked on the subdata m' revealed by the constructor pattern `suc m'`.

2 Course-of-Values Recursion

This section explains *course-of-values (CoV) pattern matching*, a feature that is the basis of Cedille's type-based termination checker. The example of division given in this section is similar to one appearing in [Firsov et al. 2018b]; whereas they use their generic development as a library to implement this, we use their development as a back-end, and the example illustrates CoV recursion in our surface language. Due to space restrictions we do not discuss here an example of CoV *induction* in the surface language, though this too is supported by the datatype subsystem (see Sections 3 and 5).

Termination checking In general purpose functional languages, programmers are free to define functions using powerful recursion schemes, including general recursion. Users of implementations of type theories are usually not afforded such freedom, as these implementations must usually ensure recursive definitions are well-founded or risk logical unsoundness. To that end, it is common to use a termination checker implementing a *syntactic guard*, enforcing that recursive calls are made only on terms revealed by case analysis on arguments of the function.

Unfortunately, syntactic termination checkers are usually unable to determine that complex recursion schemes are well-founded ([Barthe et al. 2004; Bove et al. 2016]). Consider

an intuitive definition of division by iterated subtraction. In a Haskell-like language, programmers write:

```

zero / d = 0
(suc n) / d = if (suc n < d)
  then zero else suc ((n - (d - 1)) / d)

```

This definition is guaranteed to terminate for all inputs, as the first argument to the recursive call, $n - (d - 1)$, is smaller than the original argument `suc n`. As innocuous as this definition may seem to functional programmers, it poses a difficulty for syntactic termination checkers, as $n - (d - 1)$ is not an expression produced by case analysis of n within the definition of division but an *arbitrary* predecessor produced by $d - 1$ iterations of case analysis. This is the *course-of-values* recursion scheme (categorically, *histomorphism*); it is guaranteed to be terminating, but this fact is difficult to communicate to syntactic termination checkers!

2.1 Course-of-values Pattern Matching

Cedille implements *type-based* termination checking that is powerful enough to accept functions defined using the CoV recursion scheme. At its heart is a feature we call *CoV pattern matching*, invoked by μ' , which can be used to define a version of division written close to the intuitive way, only requiring some typing annotations to guarantee termination.

Termination checking in Cedille works by replacing, in the types of subdata in pattern guards of inductive μ -expressions (but not μ'), the recursive occurrences of a datatype with an abstract (as in, universally quantified) type. This abstract type and not the usual datatype is the type of legal arguments for recursive calls. Crucially, CoV pattern matching with μ' *preserves* this type in the subdata revealed by case patterns, meaning users can write versions of e.g. predecessor and subtraction which can be used to compute values

which are then given to recursive calls of division; furthermore, they are easily *reused for ordinary numbers*. Figure 3 gives these and other auxiliary definitions.

Global declarations We first explain the types and definitions of `predCoV` and `minusCoV`. In `predCoV` we see the first use of predicate `Is/Nat`. Every datatype declaration in Cedille additionally introduces three global names derived from the datatype's name. For `Nat`, these are:

- `Is/Nat : ★ → ★`
A term of type `Is/Nat · N` is a witness that any term of type `N` may be treated as if it has type `Nat` for CoV pattern matching.
- `is/Nat : Is/Nat · Nat` is the trivial `Is/Nat` witness.
- `to/Nat : ∀ N : ★. ∀ is : Is/Nat · N. N → Nat`
`to/Nat` is a function that coerces a term of type `N` to `Nat`, given a witness `is` that `N` “is” `Nat`.

In `predCoV` the witness `is` of type `Is/Nat · N` is given explicitly to μ' with the notation $\mu' <is>$, allowing argument n (of type `N`) to be a legal scrutinee for `Nat` pattern matching. Reasoning parametrically, the only ways `predCoV` can produce an `N` output (i.e. preserve the abstract type of its argument) are by returning n itself or some subdata produced by CoV pattern matching on it – the predecessor n' also has type `N`. Thus, the type signature of `predCoV` has the following intuitive reading: it produces a number no larger than its argument, since a result like `suc (to/Nat -is n)` would be type-incorrect. Note though that this reading is informal and outside of the theory, whereas approaches based on sized types use explicit size indices to track structural decrease.

Code Reuse The reader may now wonder what the relation is between `predCoV` and the earlier `pred` of Figure 2b. The μ' -expression of `pred` with the witness given explicitly is:

$$\mu' <is/Nat> n \{ \text{zero} \rightarrow n \mid \text{suc } n' \rightarrow n' \}$$

In `pred`, the global witness `is/Nat` of type `Is/Nat · Nat` need not be passed explicitly, as it is inferable by the type `Nat` of the scrutinee n . Furthermore, `pred` and `predCoV` are definitionally equal, as these witnesses are erased from μ' -expressions (below `_` indicates an anonymous proof):

$$_ : \{ \text{pred} \approx \text{predCoV} \} = \beta.$$

This leads to a style of programming where, when possible, functions are defined over an abstract type `N` for which e.g. `Is/Nat · N` holds, and the usual versions of functions *reuse* these as a special case. This is how `minus` is defined – by specializing `minusCoV` with the trivial witness `is/Nat`.

The type signature of `minusCoV` similarly yields a reading that its result is no larger than its first argument. In the successor case, `predCoV` is given the (erased) witness `is`. That `minusCoV` preserves the type of its argument after n uses of `predCoV` is precisely what allows it to appear in an argument to recursive functions over `Nat`. Function `minus` is

used to define `lt`, the Boolean predicate deciding whether its first argument is less than its second; `ite` is the usual definition of a conditional expression by case analysis on `Bool`.

Division The last definition, `divide`, is as expected except for the successor case. Here, we make let bindings for pn' and $diff$, the syntax for which in Cedille is $[x = t] - t'$ analogous to `let x = t in t'`. Term pn' is the coercion to `Nat` of the predecessor of the dividend pn , using the as-yet unexplained `Is/Nat` witness `isType/divD`. Term $diff$ is the difference (computed using `minusCoV`) between pn and `pred d`. Note that $diff$ is guaranteed to be smaller than the original pattern `suc pn`. Finally, we test whether the dividend is less than the divisor: if so, return zero; if not, divide $diff$ by d and increment. The only parts of `divide` requiring further explanation are the witness `isType/divD` and the type of pn , which are the keys to CoV recursion in Cedille.

Local declarations Within the body of the μ -expression defining recursive function `divD` over scrutinee n of type `Nat`, the following names are automatically bound:

- `Type/divD : ★`, the type of recursive occurrences of `Nat` in the types of variables bound in constructor patterns (such as pn).
- `isType/divD : Is/Nat · Type/divD`, a witness that terms of type `Type/divD` may be used for CoV pattern matching.
- `divD : Type/divD → Nat`, the recursive function being defined, accepting only terms of the abstract type `Type/divD`. This restriction guarantees that `divD` is only called on expressions smaller than the previous argument to recursion.

The reader is now invited to revisit the definitions of Figure 2, keeping in mind that the μ -expression of `add`, for example, the subdata m' in pattern guard `suc m'` has an abstract type, and the recursively defined `addN` only accepts arguments of such a type. With this understood, so to is `divide`: predecessor pn has type `Type/divD`, witness `isType/divD` has type `Is/Nat · Type/divD` and so the local variable $diff$ has type `Type/divD` as required by `divD`.

3 Elaboration Interface

The generic library of [Firsov et al. 2018a,b] derives inductive datatypes using *Mendler-style F-algebras*, so we begin with a brief description of these. For a more thorough treatment of the expressive power of Mendler-style algebras, see [Ahn and Sheard 2011].

Mendler-style F-algebras It is well understood that an inductive datatype D can be represented categorically as the carrier of the initial algebra for (i.e., the least fixed-point of) its signature functor F [Malcolm 1990], with the definition of a conventional F -algebra in type theory as a pair (X, ϕ) , where X is a type (called the *carrier*) and ϕ is a function of

predCoV: $\forall N: \star. \forall is: Is/Nat \cdot N. N \rightarrow N = \Lambda N. \Lambda is. \lambda n. \mu' \langle is \rangle n \{zero \rightarrow n \mid suc\ n' \rightarrow n'\}$.

minusCoV: $\forall N: \star. \forall is: Is/Nat \cdot N. N \rightarrow Nat \rightarrow N$
 $= \Lambda N. \Lambda is. \lambda m. \lambda n. \mu\ mMinus. n \{$
 $\mid zero \rightarrow m$
 $\mid suc\ n' \rightarrow predCoV\ -is\ (mMinus\ n') \}$.
 $minus = minusCoV\ -is/Nat$.

lt: $Nat \rightarrow Nat \rightarrow Bool = \lambda m. \lambda n. \mu' (minus (suc\ m)\ n) \{zero \rightarrow tt \mid suc\ r \rightarrow ff\}$.

ite: $\forall X: \star. Bool \rightarrow X \rightarrow X \rightarrow X = \Lambda X. \lambda b. \lambda t. \lambda f. \mu' b \{tt \rightarrow t \mid ff \rightarrow f\}$.

divide: $Nat \rightarrow Nat \rightarrow Nat = \lambda n. \lambda d. \mu\ divD. n \{$
 $\mid zero \rightarrow zero$
 $\mid suc\ pn \rightarrow [pn' = to/Nat\ -isType/divD\ pn] - [diff = minusCoV\ -isType/divD\ pn\ (pred\ d)] -$
 $ite\ (lt\ (suc\ pn')\ d)\ zero\ (suc\ (divD\ diff)) \}$.

Figure 3. Division using course-of-values recursion

(a) Casts, positivity, and type fixpoints

$$\frac{\Gamma \vdash A: \star \quad \Gamma \vdash B: \star}{\Gamma \vdash Cast \cdot A \cdot B: \star} \quad \frac{\Gamma \vdash f: A \rightarrow B \quad \Gamma \vdash p: \Pi x:A. \{f\ x \approx x\}}{\Gamma \vdash intrCast\ f\ p: Cast \cdot A \cdot B} \quad \frac{\Gamma \vdash c: Cast \cdot A \cdot B}{\Gamma \vdash elimCast\ -c \cong \lambda x. x: A \rightarrow B}$$

$$\frac{\Gamma \vdash F: \star \rightarrow \star}{\Gamma \vdash Mono \cdot F: \star} \quad \frac{\Gamma \vdash f: \forall X: \star. \forall Y: \star. Cast \cdot X \cdot Y \rightarrow Cast \cdot (F \cdot X) \cdot (F \cdot Y)}{\Gamma \vdash intrMono\ f: Mono \cdot F}$$

$$\frac{\Gamma \vdash im: Mono \cdot F \quad \Gamma \vdash c: Cast \cdot A \cdot B}{\Gamma \vdash elimMono\ -im\ -c \cong \lambda x. x: F \cdot A \rightarrow F \cdot B} \quad \frac{\Gamma \vdash F: \star \rightarrow \star \quad \Gamma \vdash im: Mono \cdot F}{\Gamma \vdash Fix \cdot F\ im: \star}$$

$$\frac{\Gamma \vdash F: \star \rightarrow \star \quad \Gamma \vdash im: Mono \cdot F \quad \Gamma \vdash xs: F \cdot (Fix \cdot F\ im)}{\Gamma \vdash in\ xs: Fix \cdot F\ im} \quad \frac{\Gamma \vdash F: \star \rightarrow \star \quad \Gamma \vdash im: Mono \cdot F \quad \Gamma \vdash x: Fix \cdot F\ im}{\Gamma \vdash out\ x: F \cdot (Fix \cdot F\ im)}$$

(b) Generic induction principle for λ -encoded data

module GenericCoV (F: $\star \rightarrow \star$) {im: Mono \cdot F}

D: $\star = Fix \cdot F\ im$.

PrfAlg: (D $\rightarrow \star$) $\rightarrow \star = \lambda P: D \rightarrow \star. \forall R: \star. \forall c: Cast \cdot R \cdot D. \Pi o: R \rightarrow F \cdot R. \forall oeq: \{o \approx out\}.$
 $(\Pi x: R. P (elimCast\ -c\ x)) \rightarrow \Pi xs: F \cdot R. P (in (elimMono\ -im\ -c\ xs))$.

induction: $\forall P: D \rightarrow \star. PrfAlg \cdot P \rightarrow \Pi x: D. P\ x = \langle \dots \rangle$

Figure 4. Generic library

type $F \cdot X \rightarrow X$. A Mendler-style F -algebra, which can also be used to define D [Mendler 1991; Uustalu and Vene 1999], is a pair (X, Φ) , where X is still a type but now function Φ has type $\forall R: \star. (R \rightarrow X) \rightarrow F \cdot R \rightarrow X$, with the $R \rightarrow X$ argument used to make recursive calls on subdata of the quantified type R . A Mendler-style *CoV algebra* additionally equips Φ with an abstract destructor (i.e., fixpoint unrolling function) via an argument of type $R \rightarrow F \cdot R$, allowing for further case analysis on subdata at the quantified type R .

3.1 Generic library

Briefly, we describe the definitions of the generic library of [Firsov et al. 2018b] utilized for datatype elaboration, given in Figures 4a and 4b. To improve readability, we informally present some of these definitions as type inference rules rather than verbatim Cedille code. All such definitions are definable in Cedille 1.0.0, which lacks datatypes.

Type coercions For any types A and B , $Cast \cdot A \cdot B$ is the type of generalized identity functions in CDLE, first described by [Firsov et al. 2018a]. Since CDLE is Curry-style, such a

function might exist even if A and B are inconvertible. It is introduced with `intrCast f p` assuming $f : A \rightarrow B$ and p is a proof that f behaves *extensionally* like the identity function; if $c : \text{Cast} \cdot A \cdot B$, then c can be eliminated with `elimCast -c` which has type $A \rightarrow B$ and which, crucially, is *intensionally* equal (indicated by notation \cong in the figure) to $\lambda x. x$.

Positive type schemes Given $F : \star \rightarrow \star$, $\text{Mono} \cdot F$ is the type of proofs that F is positive (or *monotonic*). In fact, datatype elaboration produces just such a proof when checking positivity of a datatype declaration (Section 4.2). It is introduced by `intrMono`, which takes as argument some f of similar type to the usual lifting of a function over a functor, but restricted to `Casts`; if $im : \text{Mono} \cdot F$ and $c : \text{Cast} \cdot A \cdot B$, then im is eliminated with `elimMono -im -c` which has type $F \cdot A \rightarrow F \cdot B$ and which is equal to $\lambda x. x$.

Type fixpoints The type $\text{Fix} \cdot F \cdot im$ is the least fixpoint of a type scheme F whose positivity is proven by $im : \text{Mono} \cdot F$. Functions `in` and `out` are the expected *rolling* and *unrolling* functions representing resp. a generic collection of constructors and destructors for a datatype with signature F .

Induction In Figure 4b we give the type signature for the induction principle of the generic library (notice module parameters F and im). The type D (the datatype whose signature is F) is simply an abbreviation for $\text{Fix} \cdot F \cdot im$. The type family PrfAlg is a dependent version of the Mendler-style CoV algebra. Its additional (erased) arguments are c , an type coercion from R to D , and oeq , a proof that the abstract destructor o is equal to `out`. Argument c is required to be even able to state the codomain of PrfAlg , which is the type of proofs that P holds of the `in` of xs , after coercing this using `elimMono` and c to type $F \cdot D$. Finally, `induction` is the generic induction principle for D .

We conclude by stating the requirements that are needed to show *value-preservation* (Theorem 5.2) and the *termination guarantee* (Theorem 5.3) that any λ -encoding implementing this interface must satisfy.

Requirement 1. *Definitions `in` and `out` are mutual inverses. Furthermore, there is a constant bound such that for all $xs : F \cdot D$, expression `out (in xs)` β -reduces to xs in a number of steps within that bound.*

Requirement 2. *For every untyped λ -expression a , there exists some term t such that `induction a` $\rightsquigarrow^* t$ and that `induction a (in xs)` $\rightsquigarrow^* a \text{ out } t \text{ xs}$, for all terms xs .*

Requirement 3. *For closed definitions of $F : \star \rightarrow \star$ and $im : \text{Mono} \cdot F$, there exists some closed term t' of type $\text{Fix} \cdot F \cdot im \rightarrow \prod x : T_1. T_2$ (for some T_1 and T_2) that erases to $\lambda x. x$.*

The first part of Requirement 1 is known as *Lambek's lemma* [Lambek 1968]. Requirement 2 expresses the *cancellation law* for the initial Mendler-style CoV F -algebra (phrased

differently: induction computes as a course-of-values re-cursor for data). Requirement 3 simply states that the elaborations of datatypes *must be functional*. All three requirements are satisfied by the library provided by [Firsov et al. 2018b].

3.2 Implementing CoV Pattern Matching

There is a discrepancy between the facilities of the generic library given in Figure 4 and the features of the surface language. A recursive function over datatype D with signature F defined using `induction` has available as assumptions $o : R \rightarrow F \cdot R$ (where R is a type variable that has been quantified over) and $oeq : \{o \simeq \text{out}\}$. However, it is undesirable to expose in the surface language details such as the signature F or the generic destructor `out`; we expect to be able to work with case trees and that there is shared syntax for pattern matching over data with both the concrete and abstract type.

This discrepancy is bridged by a small translation layer sketched in Figure 5, which serves as the interface for datatype elaboration. Term definitions are omitted (indicated by $\langle . \dots \rangle$; we briefly summarize them.

"Is" witnesses For any type R , $\text{IsD} \cdot R$ is the type of triples consisting of an type coercion of type $\text{Cast} \cdot R \cdot D$, a generic destructor $o : R \rightarrow F \cdot R$, and proof $\{o \simeq \text{out}\}$. It simply packages together some of the assumptions available in any proof by `induction`. Term `isD` is the trivial witness of $\text{isD} \cdot D$. Function `toD` (which is definitionally equal to $\lambda x. x$) takes evidence of $\text{IsD} \cdot R$ (for some R) and uses this to produce a type coercion from R to D . These definitions correspond to Is/D , is/D , and to/D (for a given datatype D) in the surface language. Function `toFD` is not exported to the surface language and uses `elimMono` to cast $F \cdot R$ to $F \cdot D$, given some term of type $\text{IsD} \cdot R$.

Proofs by cases Type $\text{ByCases} \cdot P \cdot R$ is the type of proofs that P holds by case analysis, where $P : D \rightarrow \star$. Thus, the type of `mu'` says that for any term x of type R where $\text{IsD} \cdot R$ holds, to show P holds of x (after casting x to D), it suffices to give a proof by case analysis on R . Its definition uses the abstract destructor o (given by $is : \text{IsD} \cdot R$) on x . Definition `mu'` corresponds directly to μ' in the surface language.

Proofs by induction $\text{ByInd} \cdot P$ is the generic type of proofs that property P holds by induction. It is defined using `ByCases` and additionally equipped with an inductive hypothesis and evidence of $\text{IsD} \cdot R$ for the quantified type R . Thus, the type of `mu` says that P holds for any $x : D$ when, under the assumption that P holds for every x' of type R (for some R for which $\text{IsD} \cdot R$ holds), P holds for the `in` of xs for all $xs : F \cdot R$; use of the abstract type R breaks circularity. Definition `mu` corresponds to μ in the surface and uses `induction`, repackaging the assumptions available to the `PrfAlg` argument for use by its argument of type $\text{ByInd} \cdot P$.

```

module DataInterface (F: ★ → ★) {im: Mono · F}.
import GenericCoV ·F -im.

IsD: ★ → ★ = <..>
isD: IsD ·D = <..>
toD: ∀ R: ★. ∀ _: IsD ·R. R → D = <..>
toFD: ∀ R: ★. ∀ _: IsD ·R. F ·R → F ·D = <..>

ByCases: (D → ★) → Π R: ★. IsD ·R → ★ = λ P: D → ★. λ R: ★. λ is: IsD ·R. Π xs: F ·R. P (in (toFD -is xs)).
mu': ∀ R: ★. ∀ is: IsD ·R. Π x: R. ∀ P: D → ★. ByCases ·P ·R is → P (toD -is x) = <..>

ByInd: (D → ★) → ★ = λ P: D → ★. ∀ R: ★. ∀ is: IsD ·R. (Π x: R. P (toD -is x)) → ByCases ·P ·R is.
mu: Π x: D. ∀ P: D → ★. ByInd ·P → P x = <..>

```

Figure 5. Interface for datatype elaboration

4 Elaboration of Datatype Declarations

Notation In this section we give a formal description of the elaboration of non-indexed datatypes in Cedille to λ -encodings in Cedille 1.0.0 (which lacks datatypes); this is also the scope of the accompanying proof appendix. A declaration of datatype D of kind \star is written $\text{Ind}[D, R, \Delta]$, where

- R is a fresh type-variable of kind \star whose scope is Δ
- Δ is an association of the constructors of D with their type signatures such that all occurrences of D in the types of constructor arguments in the surface language have been replaced by R

For example, the declaration of Nat in Figure 2 translates to

$$\text{Ind}[\text{Nat}, R, \begin{array}{l} \text{zero} : \text{Nat} \\ \text{suc} : R \rightarrow \text{Nat} \end{array}]$$

We write $\Gamma \vdash \text{Ind}[D, R, \Delta]$ *wf* to indicate that, for every i ranging from 1 to the number of constructors in Δ (written $i = 1..#\Delta$), the i th constructor c_i in Δ has a type of the form $\prod_{\bar{v}} \overline{a_i} : A_i \cdot D$ (indicating the mixed-erasure quantification over the dependent telescope of terms and types $\overline{a_i} : A_i$) that is well-kinded in context Γ extended by variables $R : \star$ and $D : \star$, and furthermore there are no occurrences of D in the classifiers of the telescope $\overline{a_i} : A_i$. Notations $\overline{\lambda} \overline{a_i} . t$ and $t \overline{a_i}$ indicates resp. the term-level abstraction and application over this telescope that respects the erasures and classifiers over which the variables were quantified. This convention generalizes to the sequence of term and type expressions $\overline{s_i}$, as in $P(c_i \overline{s_i})$, when indicated that $\overline{s_i}$ are produced from type and kind coercions of $\overline{a_i}$.

By convention, judgments with the hooked arrow $\Gamma \vdash t : T \hookrightarrow t' : T'$ are elaboration rules, written $\Gamma \vdash t : T \hookrightarrow _$ when we need only that t is well-typed. Judgments without hooked arrows $\Gamma \vdash t : T$ and $\Gamma \vdash T : K$ indicate typing and kinding in Cedille 1.0.0. Some inference rules have premises of the form $(\Gamma \vdash \prod_{\bar{v}} \overline{a_i} : A_i \cdot T : \star \hookrightarrow _)$ $_{i=1..#\Delta}$, accompanied by a premise $(c_i : \prod_{\bar{v}} \overline{a_i} : A_i \cdot D \in \Delta)_{i=1..#\Delta}$; the first indicates

a family of derivations of the parenthesized judgment indexed by the i th constructor of Δ and its constructor argument telescope $\overline{a_i} : A_i$, and the second merely names these telescopes explicitly and exhaustively. Γ^G indicates a typing context consisting of the definitions in Figures 4 and 5.

Italics indicates meta-variables, *teletype* font indicates code literals (except in meta-variables denoting generated names like Is/D), and ^{superscript} denotes labels for meta-variables. We use the following labeling convention for expressions elaborated from datatypes and their constructors: F for the usual impredicative encoding of a datatype's signature type scheme; FI for the datatype signature formed by dependent intersection and supporting a proof principle; and FIX for the least fixpoint of the datatype's "proof signature."

4.1 Datatype and Constructor Elaboration

Figure 6 shows elaboration of a datatype D and its constructors. To improve readability we give a *set of judgments*, each formed from a *single rule* performing one task: [F] and [cF] elaborate resp. the usual impredicative encoding of a datatype's signature type scheme and its constructors; [FI] and [cFI] the inductive signature and its constructors; [FIX] and [cFIX] the least fixpoint of the inductive signature and its constructors; and [DATA] of the form $\Gamma \vdash \text{Ind}[D, R, \Delta] \dashv \Gamma, \text{IndEl}[D, R, \Delta, \Theta, \mathcal{E}]$ adds the datatype, constructors, globals (Θ), and elaborations (\mathcal{E}) to the context.

In rule [F], the first premise serves to name the family of constructor argument telescopes $(\overline{a_i} : A_i)_{i=1..#\Delta}$, and the second premise elaborates the family of types $\prod_{\bar{v}} \overline{a_i} : A_i \cdot X$, where X is fresh wrt Γ . The body of the elaborated type scheme is a function type quantifying over X and abstract constructors x_i for $i = 1..#\Delta$ (themselves functions quantifying over the appropriate elaborated constructor argument types) with codomain X . In rule [cF] we elaborate the j th constructor for this signature type scheme, abstracting over the recursive-occurrence type R , the j th sequence of arguments $\overline{a_j}$, and abstract constructors x_i to produce $x_j \overline{a_j}$. Concretely, the elaborations for Nat by these two rules are:

$$\text{Nat}^F : \star \rightarrow \star = \lambda R : \star. \forall X : \star. \Pi z : X. \Pi s : R \rightarrow X. X.$$

$$\begin{array}{c}
\frac{(c_i : \prod_{\forall} \overline{a_i} : A_i. D \in \Delta)_{i=1..#\Delta} \quad (\Gamma, R : \star, X : \star \vdash \prod_{\forall} \overline{a_i} : A_i. X : \star \hookrightarrow \prod_{\forall} \overline{a_i} : A'_i. X)_{i=1..#\Delta}}{\Gamma \vdash \text{Ind}[D, R, \Delta] \xrightarrow{F} \lambda R. \forall X : \star. (\prod x_i : \prod_{\forall} \overline{a_i} : A'_i. X)_{i=1..#\Delta} \cdot X} \text{ [F]} \\
\\
\frac{c_j : \prod_{\forall} \overline{a_j} : A_j. D \in \Delta}{\Gamma \vdash (\text{Ind}[D, R, \Delta], j) \xrightarrow{cF} \Lambda R. \lambda \overline{a_j}. \Lambda X. \lambda x_{i=1..#\Delta}. x_j \overline{a_j}} \text{ [cF]} \\
\\
\frac{\Gamma \vdash \text{Ind}[D, R, \Delta] \xrightarrow{F} D^F \quad (\Gamma \vdash (\text{Ind}[D, R, \Delta], i) \xrightarrow{cF} c_i^F)_{i=1..#\Delta} \quad (c_i : \prod_{\forall} \overline{a_i} : A_i. D \in \Delta)_{i=1..#\Delta} \quad (\Gamma, R : \star, X : \star \vdash \prod_{\forall} \overline{a_i} : A_i. X : \star \hookrightarrow \prod_{\forall} \overline{a_i} : A'_i. X)_{i=1..#\Delta}}{\Gamma \vdash \text{Ind}[D, R, \Delta] \xrightarrow{FI} \lambda R. \iota x : D^F \cdot R. \forall X : D^F \cdot R \rightarrow \star. (\prod x_i : \prod_{\forall} \overline{a_i} : A'_i. X (c_i^F \overline{a_i}))_{i=1..#\Delta} \cdot X x} \text{ [FI]} \\
\\
\frac{\Gamma \vdash (\text{Ind}[D, R, \Delta], j) \xrightarrow{cF} c_j^F \quad c_j : \prod_{\forall} \overline{a_j} : A_j. D \in \Delta}{\Gamma \vdash (\text{Ind}[D, R, \Delta], j) \xrightarrow{cFI} \Lambda R. \lambda \overline{a_j}. [c_j^F \overline{a_j}, \Lambda X. \lambda x_{i=1..#\Delta}. x_j \overline{a_j}]} \text{ [cFI]} \quad \frac{\Gamma \vdash \text{Ind}[D, R, \Delta] \xrightarrow{FI} D^{FI} \quad \Gamma \vdash D^{FI} \xrightarrow{+} pos}{\Gamma \vdash \text{Ind}[D, R, \Delta] \xrightarrow{FIX} \text{Fix} \cdot D^{FI} pos} \text{ [FIX]} \\
\\
\frac{\Gamma \vdash \text{Ind}[D, R, \Delta] \xrightarrow{FIX} \text{Fix} \cdot D^{FI} pos \quad \Gamma \vdash (\text{Ind}[D, R, \Delta], j) \xrightarrow{cFI} c_j^FI \quad c_j : \prod_{\forall} \overline{a_j} : A_j. D \in \Delta}{\Gamma \vdash (\text{Ind}[D, R, \Delta], j) \xrightarrow{cFIX} \lambda \overline{a_j}. \text{in} \cdot D^{FI} \text{-pos} (c_j^FI \cdot (\text{Fix} \cdot D^{FI} pos) \overline{a_j})} \text{ [cFIX]} \\
\\
\frac{\Gamma \vdash \text{Ind}[D, R, \Delta] wf \quad \Gamma \vdash \text{Ind}[D, R, \Delta] \xrightarrow{FIX} \text{Fix} \cdot D^{FI} pos \quad (\Gamma \vdash (\text{Ind}[D, R, \Delta], i) \xrightarrow{cFIX} c_i^{FIX})_{i=1..#\Delta} \quad \Theta = (\text{Is}/D : \star \rightarrow \star, \text{is}/D : \text{Is}/D \cdot D, \text{to}/D = \lambda x. x : \forall R : \star. \forall \text{is} : \text{Is}/D \cdot R. R \rightarrow D) \quad \mathcal{E} = \left\{ \begin{array}{l} D \mapsto \text{Fix} \cdot D^{FI} pos, \quad (c_i \mapsto c_i^{FIX})_{i=1..#\Delta}, \\ \text{Is}/D \mapsto \text{Is}D \cdot D^{FI} pos, \quad \text{is}/D \mapsto \text{is}D \cdot D^{FI} \text{-pos}, \quad \text{to}/D \mapsto \text{to}D \cdot D^{FI} \text{-pos} \end{array} \right\}}{\Gamma \vdash \text{Ind}[D, R, \Delta] \dashv \Gamma, \text{IndEl}[D, R, \Delta, \Theta, \mathcal{E}]} \text{ [DATA]}
\end{array}$$

Figure 6. Elaboration of datatype declarations

$\text{zero}^F : \forall R : \star. \text{Nat}^F \cdot R = \Lambda R. \Lambda X. \lambda z. \lambda s. z.$
 $\text{suc}^F : \forall R : \star. R \rightarrow \text{Nat}^F \cdot R = \Lambda R. \lambda n. \Lambda X. \lambda z. \lambda s. s n.$

The next two rules, [FI] and [cFI], show elaboration to resp. the inductive signature type scheme and its constructors. The type scheme elaborated by [FI] returns from a type argument R the *dependent intersection* of $x : D^F \cdot R$ (where D^F is produced by rule [F]) and a proof that, for any property $X : D^F \cdot R \rightarrow \star$, $X x$ holds if X holds for the constructors of $D^F \cdot R$ applied to their arguments ($X (c_i^F \overline{a_i})$ in the rule). [cFI] elaborates the j th constructor of the inductive signature D^{FI} , whose first component $c_j^F \overline{a_j}$ is the j th constructor of $D^F \cdot R$ applied to its arguments and whose second component is a proof (by using the appropriate assumption x_j) that $X (c_j^F \overline{a_j})$ holds. The two components are indeed convertible (modulo erasure), satisfying the requirements for introducing a dependent intersection. Concretely, the elaborations for Nat by these rules are:

$\text{Nat}^{FI} : \star \rightarrow \star = \lambda R : \star.$
 $\iota x : \text{Nat}^F \cdot R. \forall X : \text{Nat}^F \cdot R \rightarrow \star.$
 $\Pi z : X \text{ zero}^F. \Pi s : (\Pi r : R. X (\text{suc}^F r)). X x.$

$\text{zero}^{FI} : \forall R : \star. \text{Nat}^{FI} \cdot R$
 $= \Lambda R. [\text{zero}^F \cdot R, \Lambda X. \lambda z. \lambda s. z].$

$\text{suc}^{FI} : \forall R : \star. R \rightarrow \text{Nat}^{FI} \cdot R$
 $= \Lambda R. \lambda n. [\text{suc}^F n, \Lambda X. \lambda z. \lambda s. s n].$

Rules [FIX] and [cFIX] tie the recursive knot using the generic interface of Figure 5: datatype D elaborates to $\text{Fix} \cdot D^{FI} pos$, where D^{FI} is produced by [FI] and pos is a term of type $\text{Mono} \cdot D^{FI}$ (i.e., a proof that D^{FI} is covariant) whose production is described in Section 4.2. Rule [cFIX] elaborate datatype constructors to the in of the constructors of D^{FI} applied to their arguments and instantiated to type $\text{Fix} \cdot D^{FI} pos$. Finally, rule [DATA] associates the datatype declaration with its elaboration in the typing context, with Θ binding the globals Is/D , is/D , and to/D and \mathcal{E} associating datatype D , its constructors, and its globals with their elaborations. Note that to/D in Θ is *defined* to be $\lambda x. x$ (not just declared to have a type) for purposes of definitional equality.

Soundness Properties The elaborations of datatype declarations enjoys the following soundness property.

Theorem 4.1 (Elaboration of declarations). *Assuming*

- $\Gamma \vdash \text{Ind}[D, R, \Delta] \dashv \Gamma, \text{IndEl}[D, R, \Delta, \Theta, \mathcal{E}]$, and
- $(c_i : \prod_{\forall} \overline{a_i} : A_i. D \in \Delta)_{i=1..#\Delta}$

(we have elaborated a well-formed datatype with constructors of a certain shape)

- $\vdash \Gamma \hookrightarrow \Gamma'$ (the typing context elaborates, Figure 9)
- $(\Gamma, X: \star, R: \star \vdash \prod_{i=1..#\Delta} \overline{a_i: A_i}. X : \star \hookrightarrow \prod_{i=1..#\Delta} \overline{a_i: A_i'}. X \text{ implies } \Gamma^G, \Gamma', R: \star, X: \star \vdash \prod_{i=1..#\Delta} \overline{a_i: A_i'}. X : \star)$ (the elaborated constructor argument types are well-kinded)

we have that

- $\Gamma^G, \Gamma' \vdash \mathcal{E}(D) : \star \rightarrow \star$
 $(\Gamma^G, \Gamma' \vdash \mathcal{E}(c_i) : \prod_{i=1..#\Delta} \overline{a_i: [\mathcal{E}(D)/R]A_i'}. \mathcal{E}(D))_{i=1..#\Delta}$
 (the elaborated datatype and constructors have the expected kind and type)
- $\Gamma^G, \Gamma' \vdash \mathcal{E}(\text{Is}/D) : \star \rightarrow \star$,
 $\Gamma^G, \Gamma' \vdash \mathcal{E}(\text{is}/D) : \mathcal{E}(\text{Is}/D) \cdot \mathcal{E}(D)$
 (the elaborations of Is/D and is/D have their expected kinds and types)
- $\Gamma^G, \Gamma' \vdash \mathcal{E}(\text{to}/D) : \forall R: \star. \forall \text{is}: \mathcal{E}(\text{Is}/D) \cdot R. R \rightarrow \mathcal{E}(D)$,
 with $|\mathcal{E}(\text{to}/D)| =_{\beta\eta} \lambda x. x$
 (the elaboration of to/D has its expected type and convertibility)

4.2 Positivity Checker

Figure 7 lists the judgments used for checking datatype positivity, the single rule defining its primary judgment $\Gamma \vdash \lambda R: \star. T \hookrightarrow \text{pos}$ that proves $\lambda R: \star. T$ (of kind $\star \rightarrow \star$) is positive, and some representative rules for the subtyping judgment; the complete set of rules is given in the proof appendix. These rules are *evidence-producing*, as the elaborator interface of Section 3 (specifically rule [FIX]) requires explicit proof of positivity in the form of Mono . This proof is generated by invoking the subtyping judgment, with an intuitive reading that $\lambda R: \star. T$ is positive if for any R_1 and R_2 where $R_1 \leq R_2$ we have $[R_1/R]T \leq [R_2/R]T$ (with \leq suggesting a form of subtyping whose semantics is Cast).

In the subtyping judgment $\Gamma; s \vdash S \leq T \hookrightarrow s'$, input s witnesses a base subtyping assumption (demonstrated in Figure 7b, top left), and output s' is a coercion derived from it, definitionally equal to $\lambda x. x$. In the positivity rule, this input is $\text{elimCast-}z$ for an arbitrary z of type $\text{Cast} \cdot R_1 \cdot R_2$, and the output has type $[R_1/R]T \rightarrow [R_2/R]T$. To illustrate the machinery of the subtyping judgment, consider the rule for Π -types (top right): to show $\Pi x: S_1. T_1 \leq \Pi x: S_2. T_2$ with base assumption s' , first produce for the domain a coercion s proving $S_2 \leq S_1$ (note the contravariance), then produce for the codomain a coercion t proving for all $y: S_2$, $[(s y)/x]T_1 \leq [y/x]T_2$; the coercion in the conclusion clearly $\beta\eta$ -reduces to $\lambda f. f$ since coercions s and t do.

A similar reading as for subtyping holds for the subkinding judgment $\Gamma; s \vdash K_1 \leq K_2 \hookrightarrow S$, though the shape of kind coercion S need not be specified (all types are erased in terms). The telescope coercion judgment breaks the pattern by producing a coerced sequence of terms and types \overline{s} (and not the coercions) from a telescope $\overline{a: A_1}$. These are equal

(modulo erasure) to \overline{a} and typeable with telescope $\overline{a: A_2}$ (Figure 9); they are used in Figure 11 to state the expected type of each case body for μ - and μ' -expressions.

This description of our positivity checker is made precise by the following soundness properties:

Theorem 4.2 (Positivity checker).

1. If $\Gamma; s \vdash S \leq T \hookrightarrow s'$ then $\Gamma \vdash s' : S \rightarrow T \hookrightarrow _$ and $|s'| =_{\beta\eta} \lambda x. x$
2. If $\Gamma; s \vdash K_1 \leq K_2 \hookrightarrow S$ then $\Gamma \vdash S : K_1 \rightarrow K_2 \hookrightarrow _$
3. If $\Gamma \vdash F \xrightarrow{+} \text{pos}$ then $\Gamma, \Gamma^G \vdash \text{pos} : \text{Mono} \cdot F \hookrightarrow _$
4. If $\Gamma; s' \vdash \overline{a: A} \leq \overline{a: B} \hookrightarrow \overline{s}$ then $\Gamma; (\overline{a: A}) \vdash \overline{s} : (\overline{a: B}) \hookrightarrow _$ and $|\overline{s}| \cong |\overline{a}|$

These properties are self-explanatory, except for (4) which makes use of a new judgment form $\Gamma; (\overline{a: A}) \vdash \overline{s} : (\overline{a: B}) \hookrightarrow _$ (Figure 9). This judgment is read “under Γ and a telescope $\overline{a: A}$, the sequence \overline{s} is classified by the telescope $\overline{a: B}$ ”, and is defined by progressively extending the context by each variable in $\overline{a: A}$, typing (kinding) each term (type) in \overline{s} according to each classifier in $\overline{a: B}$, and substituting this term (type) into the remainder of the telescope $\overline{a: B}$.

5 Elaboration of Datatype Functions

This section details the typing, operational semantics, and elaboration of μ - and μ' -expressions. Due to space restrictions we save for the separate proof appendix the complete listing of elaboration rules, as elaborating the rest of Cedille is straightforward: all occurrences of datatypes, their constructors, and exported global definitions are replaced with the elaborations mapped by \mathcal{E} (Figure 6), auxiliary rules for elaborating the context and type-coerced constructor arguments, and congruence rules for elaborating non-datatype term, type, and kind constructs. The main judgments comprising elaboration are given in Figure 9. The elaboration rules are made syntax-directed with a *bidirectional* reading [Pierce and Turner 2000] wherein types for elimination forms (such as μ and μ') are *checked* and those for introduction forms (such as constructors, not shown) are *synthesized*.

5.1 Type inference rules

Property lifting The elaboration rules in Figure 6 are able to satisfy the elaborator interface of Figure 5 by producing from a well-formed declaration of positive datatype D a signature functor D^{FI} and positivity proof pos . Even so, it is not yet obvious that the appropriate arguments to functions mu' and mu can be given when elaborating μ' - and μ -expressions. In particular, both mu' and mu require proofs ByCases , which in general requires a (non-recursive) dependent eliminator for $D^{\text{FI}} \cdot D^{\text{FIX}}$. The careful reader will have noted in the preceding section that type scheme D^{FI} produced by rule [FI] does support proofs by cases – but only for properties stated over the type scheme D^{F} produced by [F].

(a) Main judgments and positivity rule

$$\begin{array}{c}
\boxed{\Gamma \vdash F \hookrightarrow^+ \text{pos}} \quad \boxed{\Gamma; s \vdash S \leq T \hookrightarrow s'} \quad \boxed{\Gamma; s \vdash K_1 \leq K_2 \hookrightarrow S} \quad \boxed{\Gamma; s \vdash (\overline{a:A_1}) \leq (\overline{a:A_2}) \hookrightarrow \overline{s}} \\
\text{Positivity} \quad \text{Subtyping} \quad \text{Subkinding} \quad \text{Telescope coercion} \\
\hline
\Gamma, R: \star \vdash T : \star \hookrightarrow _ \quad \Gamma, R_1: \star, R_2: \star, z: \text{Cast} \cdot R_1 \cdot R_2; \text{elimCast} \cdot z \vdash [R_1/R]T \leq [R_2/R]T \hookrightarrow s' \\
\hline
\Gamma \vdash \lambda R: \star. T \hookrightarrow^+ \text{intrMono} (\Lambda R_1. \Lambda R_2. \lambda z. \text{intrCast} s' (\lambda _ . \beta))
\end{array}$$

(b) Subtyping rules (incomplete listing)

$$\frac{\Gamma \vdash s : S \rightarrow T \hookrightarrow _ \quad |s| \cong \lambda x. x}{\Gamma; s \vdash S \leq T \hookrightarrow s} \quad \frac{\Gamma; s' \vdash S_2 \leq S_1 \hookrightarrow s \quad \Gamma, y: S_2; s' \vdash [(s y)/x]T_1 \leq [y/x]T_2 \hookrightarrow t}{\Gamma; s' \vdash \Pi x: S_1. T_1 \leq \Pi x: S_2. T_2 \hookrightarrow \lambda f. \lambda y. t (f (s y))}$$

Figure 7. Positivity checker

```

import DataInterface · DFI · pos.
LiftD : Π P: DFIX → ★. Π R: ★. Π is: IsD · R. DF · R → ★
= λ P: DFIX → ★. λ R: ★. λ is: IsD · R. λ x: DF · R.
  ∀ m: DFI · R. ∀ eq: {m ≈ x}.
  P (in (φ eq - (toFD -is m) {x})).

```

Figure 8. Lifting of properties of D^{FIX} to D^{F}

The solution to this mismatch is given in Figure 8, listing the type-level function Lift_D lifting properties P of D^{FIX} to a property of D^{F} . Given such P , a type R , a witness is of type $\text{IsD} \cdot R$, and x of type $D^{\text{F}} \cdot R$, $\text{Lift}_D \cdot P \cdot R \text{ is } x$ is a proof that P holds for the in of x , where the φ -expression casts x to the type $D^{\text{FI}} \cdot D^{\text{FIX}}$ of the expression $\text{toFD} \cdot \text{pos} \cdot \text{is } m$, for any m equal (by eq) to x . Recall that toFD erases to $\lambda x. x$; thus the expression $\text{toFD} \cdot \text{pos} \cdot \text{is } m$ is convertible with m . Because of this, eq really does prove these two expressions are equal.

Case branches To aid in reading the elaboration rules for μ and μ' -expressions, we separate into a single judgment the book-keeping common to both for elaborating constructor case branches. The single rule [CASES] forming this judgment is given in Figure 10. It should be read as taking as input a case tree $\{c_i \overline{a}_i \rightarrow t_i\}_{i=1..#\Delta}$, a type family P , and a witness is , and producing type-coerced constructor arguments \overline{s}_i , a collection $\{\lambda \overline{a}_i. t'_i\}_{i=1..#\Delta}$ of elaborated case bodies, and an elaborated witness is' . In its premises, we check that the given witness is has type $\text{Is}/D \cdot T$ (where Is/D is associated with some declared datatype D) and elaborate it, then check that constructors of the case tree cover exhaustively the constructors of D (and are given the correct number of and erasures for the pattern-bound variables). We produce \overline{s}_i via telescope coercion of \overline{a}_i (Figure 7), using the coercion $\text{to}/D \cdot \text{is}$ to cast recursive occurrences of T (given by the occurrences R) to D in the types of pattern-bound variables given to c_i in the case tree. In the last premise, we elaborate

each case branch at its expected type – a mixed-erasure abstraction over the constructor arguments \overline{a}_i with codomain $P (c_i \overline{s}_i)$.

Elaboration of μ - and μ' -expressions With Lift_D and rule [CASES] we are now able to explain how μ - and μ' -expressions are elaborated, shown in Figure 11.

In the premises of rule [MU], we begin by requiring that the kind of the motive P is $D \rightarrow \star$, and the type of the scrutinee t is some concrete datatype D , elaborating them to resp. P' and t' . We then declare an extended typing context Γ' , formed by Γ and the μ -locals Type/ih , isType/ih , and ih , which directly correspond to the assumptions available for any proof $\text{ByInd} \cdot P'$. We then elaborate the case branches with type Type/ih and witness isType/ih , producing the collection of elaborated case bodies $\{\lambda \overline{a}_i. t'_i\}_{i=1..#\Delta}$ (since witness isType/ih is a variable, it will elaborate to itself).

In the conclusion of [MU], to elaborate the entire μ -expression we use the generic function mu of Figure 5, instantiating it with the datatype's elaborated (inductive) signature functor D^{FI} and proof pos it is positive. We also give mu the elaborated scrutinee t' and motive P' . The final argument to mu is a proof of type $\text{ByInd} \cdot P$. Within the body of this λ -expression, we invoke $x.2$ with a lifted motive, giving it the elaborated case branches extended by assumptions m and eq introduced by lifting. Under this context, the elaborated case bodies t'_i are expected to have a type produced by lifting P' , and (by a forward reference to Theorem 5.1) they indeed have types convertible with this expected type. The final arguments required for the lifted elimination is some $D^{\text{FI}} \cdot \text{Type}/ih$ (given by x) and proof it is equal to x (proved by β).

Rule [MU'] is similar to [MU], so we describe only the significant differences. Operator μ' is given a scrutinee t of type T , and expects a witness is proving that $\text{Is}/D \cdot T$ for a suitable datatype D ; this is checked by using the auxiliary judgment for elaborating case branches. In the conclusion,

$$\begin{array}{ccccccc}
\boxed{\Gamma \vdash t : T \hookrightarrow t'} & \boxed{\Gamma \vdash T : K \hookrightarrow T'} & \boxed{\Gamma \vdash K \hookrightarrow K'} & \boxed{\Gamma \vdash t \hookrightarrow p} & \boxed{\vdash \Gamma \hookrightarrow \Gamma'} & \boxed{\Gamma; (\overline{a:A}) \vdash \overline{s} : (\overline{a:B}) \hookrightarrow \overline{s'}} \\
\text{Terms} & \text{Types} & \text{Kinds} & \text{Pure terms} & \text{Contexts} & \text{Telescope coercions}
\end{array}$$

Figure 9. Elaboration judgments

$$\frac{\Gamma \vdash is : Is/D \cdot T \hookrightarrow is' \quad \text{IndEl}[D, R, \Delta, \Theta, \mathcal{E}] \in \Gamma \quad Is/D \in \Theta, to/D \in \Theta \quad (c_i \cdot \prod_{i=1..#\Delta} \overline{a_i} : \overline{A_i} \cdot D \in \Delta)_{i=1..#\Delta} \quad (\Gamma; to/D - is \vdash \overline{a_i} : [T/R]A_i \leq \overline{a_i} : [D/R]A_i \hookrightarrow \overline{s_i})_{i=1..#\Delta} \quad (\Gamma \vdash \overline{\lambda} \overline{a_i} \cdot t_i : \prod_{i=1..#\Delta} \overline{a_i} : [T/R]A_i \cdot P(c_i \overline{s_i}) \hookrightarrow \overline{\lambda} \overline{a_i} \cdot t'_i)_{i=1..#\Delta}}{\Gamma \vdash \{c_i \overline{a_i} \rightarrow t_i\}_{i=1..#\Delta} : \text{Cases}(\{P(c_i \overline{s_i})\}_{i=1..#\Delta}, is) \hookrightarrow (\overline{\lambda} \overline{a_i} \cdot t'_i)_{i=1..#\Delta}, is')} \text{ [CASES]}$$

Figure 10. Elaboration of case branches

$$\frac{\text{IndEl}[D, R, \Delta, \Theta, \mathcal{E}] \in \Gamma, \mathcal{E}(D) = \text{Fix} \cdot D^{\text{Fl}} \text{ pos}, to/D \in \Theta \quad \Gamma \vdash P : D \rightarrow \star \hookrightarrow P' \quad \Gamma \vdash t : D \hookrightarrow t' \quad \Gamma' = \Gamma, \text{Type}/ih : \star, is\text{Type}/ih : Is/D \cdot \text{Type}/ih, ih : \prod y : \text{Type}/ih. P(to/D - is\text{Type}/ih y) \quad \Gamma' \vdash \{c_i \overline{a_i} \rightarrow t_i\} : \text{Cases}(\{P(c_i \overline{s_i})\}, is\text{Type}/ih) \hookrightarrow (\overline{\lambda} \overline{a_i} \cdot t'_i)_{i=1..#\Delta}, is\text{Type}/ih}}{\Gamma \vdash \mu \text{ ih. } t @P \{c_i \overline{a_i} \rightarrow t_i\}_{i=1..#\Delta} : P t \hookrightarrow \mu \cdot D^{\text{Fl}} \text{-pos } t' \cdot P' \quad (\wedge \text{Type}/ih. \wedge is\text{Type}/ih. \lambda \text{ ih. } \lambda x. x.2 \cdot (\text{Lift}_D \cdot P' \cdot \text{Type}/ih is\text{Type}/ih) (\overline{\lambda} \overline{a_i} \cdot \wedge m. \wedge eq. t'_i)_{i=1..#\Delta} -x -\beta)} \text{ [Mu]}$$

$$\frac{\text{IndEl}[D, R, \Delta, \Theta, \mathcal{E}] \in \Gamma, \mathcal{E}(D) = \text{Fix} \cdot D^{\text{Fl}} \text{ pos}, to/D \in \Theta \quad \Gamma \vdash t : T \hookrightarrow t' \quad \Gamma \vdash T : \star \hookrightarrow T' \quad \Gamma \vdash P : D \rightarrow \star \hookrightarrow P' \quad \Gamma \vdash \{c_i \overline{a_i} \rightarrow t_i\} : \text{Cases}(\{P(c_i \overline{s_i})\}, is) \hookrightarrow (\overline{\lambda} \overline{a_i} \cdot t'_i)_{i=1..#\Delta}, is'}{\mu' <is> t @P \{c_i \overline{a_i} \rightarrow t_i\}_{i=1..#\Delta} : P (to/D - is t) \hookrightarrow \mu' \cdot D^{\text{Fl}} \text{-pos -is } t' \cdot P' \quad (\lambda x. x.2 \cdot (\text{Lift}_D \cdot P' \cdot T' is') (\overline{\lambda} \overline{a_i} \cdot \wedge m. \wedge eq. t'_i)_{i=1..#\Delta} -x -\beta)} \text{ [Mu']}$$

Figure 11. $\boxed{\Gamma \vdash t : T \hookrightarrow t'}$ Elaboration of terms (shown: μ, μ')

we elaborate the μ' -expression using μ' , whose last argument must be a proof of type $\text{ByCases} \cdot P' \cdot T \text{ is}$, similarly given by property lifting.

Soundness Properties The elaborations of terms (types) from the surface language have their elaborated types (kinds) in the internal language:

Theorem 5.1 (Type-preservation). *If $\vdash \Gamma \hookrightarrow \Gamma'$ then:*

- If $\Gamma \vdash K \hookrightarrow K'$ then $\Gamma^G, \Gamma' \vdash K'$
- If $\Gamma \vdash T : K \hookrightarrow T'$ then for some $K', \Gamma \vdash K \hookrightarrow K'$ and $\Gamma^G, \Gamma' \vdash T' : K'$
- If $\Gamma \vdash t : T \hookrightarrow t'$ then for some $T', \Gamma \vdash T : \star \hookrightarrow T'$ and $\Gamma \vdash t' : T'$

5.2 Operational Semantics

Part of the unwieldiness of working directly with λ -encodings is their size. Our datatype subsystem for Cedille addresses this by treating datatypes and their constructors opaquely, giving μ - and μ' -expressions a primitive operational semantics shown in Figure 12. Cedille's operational semantics is defined for untyped terms, i.e., for terms after the erasure of annotations. To erase both μ - and μ' -expressions (also Figure 12) we erase the scrutinee, the motive, any type or

erased term arguments bound by constructor patterns (indicated by $|\overline{a_i}|$), and use the erasures of the branch bodies; in μ' -expressions we also erase the witness is .

Soundness Properties To show that our extension of the operational semantics is sound with respect to that of the target language, we must introduce an auxiliary judgment for *elaboration of pure (post-erasure) terms* whose rules mirror those for elaborating annotated terms – the rules for this judgment for μ and μ' are listed in Figure 13.

Theorem 5.2 (Value Preservation for μ and μ'). *The elaborations of μ - and μ' -expressions and the elaborations of the terms they single-step are joinable:*

- If $\Gamma \vdash \mu \text{ ih. } (c_j \overline{s}) \{c_i \overline{a} \rightarrow t_i\}_{i=1..n} \hookrightarrow e_1$, and $\mu \text{ ih. } (c_j \overline{s}) \{c_i \overline{a} \rightarrow t_i\}_{i=1..n} \rightsquigarrow t$, and $\Gamma \vdash t \hookrightarrow e_2$, then there exists some e_3 such that $e_2 \rightsquigarrow^* e_3$ and $e_1 \rightsquigarrow^* e_3$
- If $\Gamma \vdash \mu' (c_j \overline{s}) \{c_i \overline{a_i} \rightarrow t_i\}_{i=1..n} \hookrightarrow e_1$, and $\mu' (c_j \overline{s}) \{c_i \overline{a_i} \rightarrow t_i\}_{i=1..n} \rightsquigarrow t$, and $\Gamma \vdash t \hookrightarrow e_2$, then there exists some e_3 such that $e_1 \rightsquigarrow^* e_3$ and $e_2 \rightsquigarrow^* e_3$

Finally, Theorem 5.3 states the termination guarantee of our datatype subsystem.

Theorem 5.3 (Call-by-name Normalization).

If $\Gamma \vdash t : D \hookrightarrow t'$ and $\text{IndEl}[D, R, \Delta, \Theta, \mathcal{E}] \in \Gamma$, and if t is a closed term, then $|t'|$ is call-by-name normalizing.

$$\begin{array}{l}
|\mu' <is> t @P \{ c_i \bar{a}_i \rightarrow t_i \}_{i=1..n} | = \mu' |t| \{ c_i |\bar{a}_i| \rightarrow |t_i| \}_{i=1..n} \\
|\mu \text{ ih. } t @P \{ c_i \bar{a}_i \rightarrow t_i \}_{i=1..n} | = \mu \text{ ih. } |t| \{ c_i |\bar{a}_i| \rightarrow |t_i| \}_{i=1..n} \\
\hline
\frac{1 \leq j \leq n \quad \# \bar{s} = \# \bar{a}_j}{\mu' (c_j \bar{s}) \{ c_i \bar{a}_i \rightarrow t_i \}_{i=1..n} \rightsquigarrow [s/a_j] t_j} \quad \frac{1 \leq j \leq n \quad \# \bar{s} = \# \bar{a}_j \quad r = \lambda x. \mu \text{ ih. } x \{ c_i \bar{a}_i \rightarrow t_i \}_{i=1..n}}{\mu \text{ ih. } (c_j \bar{s}) \{ c_i \bar{a}_i \rightarrow t_i \}_{i=1..n} \rightsquigarrow [s/a_j][r/ih] t_j}
\end{array}$$

Figure 12. Erasure and reduction for μ and μ'

$$\frac{\Gamma \vdash t \hookrightarrow t' \quad (\Gamma \vdash t_i \hookrightarrow t'_i)_{i=1..n}}{\Gamma \vdash \mu \text{ ih. } t \{ c_i \bar{a}_i \rightarrow t_i \}_{i=1..n} \hookrightarrow |\mu| t' (\lambda \text{ ih. } \lambda x. x (\lambda \bar{a}_i. t'_i)_{i=1..n})} \quad \frac{\Gamma \vdash t \hookrightarrow t' \quad (\Gamma \vdash t_i \hookrightarrow t'_i)_{i=1..n}}{\Gamma \vdash \mu' t \{ c_i \bar{a}_i \rightarrow t_i \}_{i=1..n} \hookrightarrow |\mu'| t' (\lambda x. x (\lambda \bar{a}_i. t'_i)_{i=1..n})}$$

Figure 13. $\boxed{\Gamma \vdash t \hookrightarrow t'}$ Elaboration of pure (post-erasure) terms (shown: μ and μ')

6 Related Work

λ -encodings in CDLE This work builds upon [Firsov et al. 2018a,b] which generically derives induction (and CoV induction) in CDLE for λ -encoded datatypes arising as the least fixed point of a class of type schemes generalizing co-variant functors. Our elaborator interface was derived from these developments: we repackaged the facilities of the generic library to implement CoV pattern matching in the surface language without revealing implementation details.

TCBs in ITPs Many interactive theorem provers (ITPs) have large trusted computing bases (TCBs). For example, Coq's kernel is $\sim 30\text{K}$ OCaml LoC, and some provers like Agda [Norell 2007] ($\sim 100\text{K}$ Haskell LoC) have no kernel. But, there is much interest in verifying provers themselves [Davis and Ableson 2015; Harrison 2006] and thus practical interest in keeping their kernels small [Appel 2001].

[Dagand and McBride 2012] shares with us this goal, describing the elaboration a language with inductive definitions, pattern matching, and recursion to a simpler core theory. They show how to translate datatype declaration to Martin-Löf type theory extended with a universe of positive inductive types and description labels. In comparison, our core theory has no inductive primitives and elaboration produces explicit proofs of positivity rather than elaborating to types that are positive by construction.

[Goguen et al. 2006] show how *dependent pattern matching* ([Coquand 1992]) can be elaborated to a use of a datatype's dependent eliminator. CoV pattern matching in this paper is in many respects less sophisticated than dependent pattern matching; however, an interesting point of comparison is the treatment of CoV induction. [Goguen et al. 2006] accomplish this by providing as the inductive hypothesis $\text{Below}_D P x$, a large tuple containing proofs that P holds for all subdata of x . Functions analyzing a static number of cases analyzed (e.g. fib) may easily make use of this, but accessing a proof for dynamically computed subdata (e.g. the result of `minusCoV` in `divide`) requires an inductive proof of a lemma such as $\text{Below}_{\text{Nat}} P (\text{suc } n) \rightarrow P (\text{minus } n m)$

(for any m), not required in our work (nor of [Firsov et al. 2018b]).

Semantic Termination Checking [Abel 2010] extends type theory with *sized types*, allowing datatypes to be annotated with size information and the type system guaranteeing that recursive calls are made on arguments of decreasing size. Sized types require defining alternative, size-indexed versions of datatypes and extension of the underlying theory, whereas in Cedille every standard datatype declaration is defined with the usual notation and automatically supports CoV induction. On the other hand, sized types allow for even more powerful forms of recursive definitions. In particular, the usual implementation of merge-sort, which is definable using sized types, is not straight-forwardly expressible as CoV recursion as it involves recursion on terms that are not subdata of the original list.

The Nax language, described by [Ahn 2014], takes an approach to termination checking similar to ours. In Nax, recursive functions are defined in terms of Mendler-style recursion schemes, including CoV recursion and Mendler-style induction, whereas in Cedille the μ -operator of Cedille provides users the ability to write definitions using CoV induction. On the other hand, Nax soundly permits datatype definitions with *negative* recursive occurrences, possible because Nax restricts the *usage* of negative datatypes, whereas we opt for the more traditional approach of restricting the rules for the *formation* of datatypes.

7 Conclusion and Future Work

We have presented a datatype subsystem for Cedille that enjoys both the expected conveniences (compact notation for datatype declarations, case analysis, and fixpoint-style recursive definitions) and the desirable feature of CoV induction derived of λ -encodings in CDLE. We further showed that this subsystem does not require extending CDLE by presenting inference rules for the additional language constructs that elaborate to expressions in Cedille 1.0.0 (which

lacks a datatype subsystem), and showing important soundness properties of the types and operational semantics of elaborated terms with respect to the surface language.

One immediate usability concern is the proliferation of explicit type coercions in the case branches of μ - and μ' -expressions. We already automatically infer the necessary type coercions for constructor arguments in the *expected type* of case branches using the subtyping judgment in Figure 7; this can be further integrated into the type system so that type coercions in the *bodies* of case branches need not be explicitly coerced by the programmer, either.

Another direction is extending our datatype subsystem to support *zero-cost reuse* for programs and data, derived generically in CDLE by [Diehl et al. 2018]. One modest step would be to extend definitional equality in the surface language so that constructors of different datatypes are considered equal when their elaborated λ -expressions are, allowing users to derive reuse *manually* for datatypes and functions. More ambitiously, a higher level syntax (such as *ornaments* [Dagand and McBride 2014; McBride 2010]) would allow programmers to define one type (like `Vec`) in terms of another (like `List`) by describing the function or relation on terms of the latter to the indices of the former. Such definitions could then be elaborated using generic zero-cost reuse combinators.

Acknowledgments

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

References

- Andreas Abel. 2006. Polarized Subtyping for Sized Types. In *Computer Science – Theory and Applications*, Dima Grigoriev, John Harrison, and Edward A. Hirsch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 381–392.
- Andreas Abel. 2010. MiniAgda: Integrating Sized and Dependent Types. *Electronic Proceedings in Theoretical Computer Science* 43 (Dec 2010), 14–28. <https://doi.org/10.4204/eptcs.43.2>
- Ki Yung Ahn. 2014. The Nax Language: Unifying Functional Programming and Logical Reasoning in a Language based on Mendler-style Recursion Schemes and Term-indexed Types. (2014).
- Ki Yung Ahn and Tim Sheard. 2011. A Hierarchy of Mendler Style Recursion Combinators: Taming Inductive Datatypes with Negative Occurrences. *SIGPLAN Not.* 46, 9 (Sept. 2011), 234–246. <https://doi.org/10.1145/2034574.2034807>
- Andrew W Appel. 2001. Foundational proof-carrying code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 247–256.
- Gilles Barthe, Maria Joao Frade, Eduardo Giménez, Luis Pinto, and Tarmo Uustalu. 2004. Type-based termination of recursive definitions. *Mathematical structures in computer science* 14, 1 (2004), 97–141.
- Frédéric Blanqui. 2005. Inductive types in the Calculus of Algebraic Constructions. *Fundamenta Informaticae* 65, 1-2 (2005), 61–86.
- Ana Bove, Alexander Krauss, and Matthieu Sozeau. 2016. Partiality and recursion in interactive theorem provers—an overview. *Mathematical Structures in Computer Science* 26, 1 (2016), 38–88.
- Thierry Coquand. 1992. Pattern matching with dependent types. In *Informal proceedings of Logical Frameworks*, Vol. 92. 66–79.
- Pierre-Evariste Dagand and Conor McBride. 2012. Elaborating Inductive Definitions. arXiv:cs.PL/1210.6390
- Pierre-Évariste Dagand and Conor McBride. 2014. Transporting functions across ornaments. *Journal of functional programming* 24, 2-3 (2014), 316–383.
- Jared Davis and Magnus O Myreen. 2015. The reflective Milawa theorem prover is sound (down to the machine code that runs it). *Journal of Automated Reasoning* 55, 2 (2015), 117–183.
- Larry Diehl, Denis Firsov, and Aaron Stump. 2018. Generic Zero-cost Reuse for Dependent Types. *Proc. ACM Program. Lang.* 2, ICFP, Article 104 (July 2018), 30 pages. <https://doi.org/10.1145/3236799>
- Denis Firsov, Richard Blair, and Aaron Stump. 2018a. Efficient Mendler-Style Lambda-Encodings in Cedille. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, 235–252.
- Denis Firsov, Larry Diehl, Christopher Jenkins, and Aaron Stump. 2018b. Course-of-Value Induction in Cedille. arXiv:cs.LO/1811.11961
- Herman Geuvers. 2001. Induction Is Not Derivable in Second Order Dependent Type Theory. In *Typed Lambda Calculi and Applications*, Samson Abramsky (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 166–181.
- Herman Geuvers. 2009. Proof assistants: History, ideas and future. *Sadhana* 34, Part 1 (2009), 3–25.
- Eduarde Giménez. 1995. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs*, Peter Dybjer, Bengt Nordström, and Jan Smith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 39–59.
- Healfdene Goguen, Conor McBride, and James McKinna. 2006. *Eliminating Dependent Pattern Matching*. Springer Berlin Heidelberg, Berlin, Heidelberg, 521–540. https://doi.org/10.1007/11780274_27
- John Harrison. 2006. Towards self-verification of HOL Light. In *International Joint Conference on Automated Reasoning*. Springer, 177–191.
- INRIA. 2017. The Coq Proof Assistant, version 8.7.0. <https://doi.org/10.5281/zenodo.1028037>
- Alexei Kopylov. 2003. Dependent Intersection: A New Way of Defining Records in Type Theory. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS '03)*. IEEE Computer Society, Washington, DC, USA, 86–. <http://dl.acm.org/citation.cfm?id=788023.789073>
- Joachim Lambek. 1968. A Fixpoint Theorem for Complete Categories. *Mathematische Zeitschrift* 103, 2 (1968), 151–161. <https://doi.org/10.1007/bf01110627>
- Grant Malcolm. 1990. Data structures and program transformation. *Science of computer programming* 14, 2-3 (1990), 255–279.
- Ralph Matthes. 2002. Tarski's Fixed-Point Theorem And Lambda Calculi With Monotone Inductive Types. *Synthese* 133, 1 (01 Oct 2002), 107–129. <https://doi.org/10.1023/A:1020831825964>
- Conor McBride. 2010. Ornamental algebras, algebraic ornaments. *Journal of functional programming* (2010).
- Nax Paul Mendler. 1991. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic* 51, 1 (1991), 159–172. [https://doi.org/10.1016/0168-0072\(91\)90069-X](https://doi.org/10.1016/0168-0072(91)90069-X)
- Alexandre Miquel. 2001. 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 (TLCA'01)*. Springer-Verlag, Berlin, Heidelberg, 344–359. <http://dl.acm.org/citation.cfm?id=1754621.1754650>
- Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

- Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (jan 2000), 1–44. <https://doi.org/10.1145/345099.345100>
- Aaron Stump. 2017. The calculus of dependent lambda eliminations. *J. Funct. Program.* 27 (2017), e14.
- Aaron Stump. 2018. Syntax and Semantics of Cedille. arXiv:[cs.PL/1806.04709](https://arxiv.org/abs/cs.PL/1806.04709)
- Tarmo Uustalu and Varmo Vene. 1999. Mendler-style inductive types, categorically. *Nord. J. Comput.* 6, 3 (1999), 343.