Efficient lambda encodings for Mendler-style coinductive types in Cedille

Christopher Jenkins Aaron Stump Larry Diehl
The University of Iowa, Iowa City, Iowa USA
{firstname-lastname}@uiowa.edu

In the calculus of dependent lambda eliminations (CDLE), it is possible to define inductive datatypes via lambda encodings that feature constant-time destructors and a course-of-values induction scheme. This paper begins to address the missing derivations for the dual, *coinductive* types. Our derivation utilizes new methods within CDLE, as there are seemingly fundamental difficulties in adapting previous known approaches for deriving inductive types. The lambda encodings we present implementing coinductive types feature constant-time constructors and a course-of-values corecursion scheme. Coinductive type families are also supported, enabling proofs for many standard coinductive properties such as stream bisimulation. All work is mechanically verified by the Cedille tool, an implementation of CDLE.

1 Introduction

Inductive (algebraic) datatypes, such as natural numbers and lists, serve the crucial role in functional languages of structuring the definitions of functions over (necessarily) finite objects. To each such datatype is associated an *eliminator*, a combinator for defining provably terminating functions over such objects, justifying structurally recursive definitions. Less familiar is the notion of coinductive (coalgebraic) datatypes, which serve to structure the definitions of functions generating (possibly) infinite objects. An example of a codatatype is the type of *streams*, which are infinite lists of some element type. Analogously, associated with each coinductive type is a *generator*, a combinator for giving definitions of such objects that are provably productive (that is, all finite observations made of these objects are defined).

Usually, inductive and coinductive types are built-in primitives to functional languages. Though they are sometimes declared via the same syntactic mechanisms (as in Haskell), in total functional languages and especially in implementations of dependent type theories they are better kept separate. As an example of such theory, the *calculus of constructions* [7] (CC) is extended by the *calculus of inductive constructions* [39] (CIC) with primitive inductive datatypes each with an associated induction principle; syntactically, they are given by a declaration listing the constructors generating elements of the datatype and used by pattern matching and recursion. CIC can be further extended with primitive coinductive types, and an elegant syntactic proposal for them comes from Abel et al. [1]: giving these by a declaration listing their *destructors* and generating them via copattern matching. The elegance of this proposal lies in its close fit with the semantic account of coinductive types developed by Hagino [14].

The *calculus of dependent lambda eliminations* [30, 32] (CDLE) is a compact Curry-style type theory with no primitive notion of (co)inductive datatypes. Instead, datatypes are encoded as lambda expressions. Historically, lambda encodings have languished due to several difficulties, which CDLE overcomes. Geuvers [12] showed the impossibility of deriving induction for them in second-order dependent type theory; Firsov and Stump [10] demonstrated how to generically derive the induction principle for them in CDLE. Parigot [27] showed that the Church-style of encoding has no better than linear-time data

Submitted to:
SOS 2007

accessors (such as predecessor for natural numbers); Firsov et al. [8] showed how to use the induction principle for a Mendler-style of encoding [24] to define *efficient* (constant-time) data accessors. Furthermore, Firsov et al. [9] show how to further augment this with *course-of-values* induction, an expressive scheme wherein the inductive hypothesis can be invoked on subdata at unbounded depth.

The present paper partially addresses the gap in the foregoing account of datatypes in CDLE by presenting a generic derivation of lambda-encoded *coinductive* data, giving analogous results for efficiency (for *constructors*) and expressivity (using *course-of-values coiteration*). What we do not show here, however, is a "true coinduction" principle in the sense of "bisimilarity implying equality" – indeed, a negative result concerning *Lambek's lemma* (a consequence of coinduction) indicates coinduction in this sense is not possible for our encoding. However, our derived codatatypes support *indexed coiteration*, which we argue suffices for implementing many standard examples of coinductive reasoning such as showing that two streams are bisimilar. Since efficiency for Mendler-encoded data came as a consequence of induction in [8], lacking true coinduction we cannot directly apply their method. Instead, we use (monotone) recursive types to achieve efficiency (a technique deployed for similar effect in e.g., [13, 22, 18, 19]), a feature which is itself derivable within CDLE.

Contributions Summarizing, in this paper we:

- generically derive Mendler-style lambda encodings for coinductive datatypes in CDLE using derivable monotone recursive types;
- show that codata so derived forms an adequate basis for functional programming due to the *expressive course-of-values corecursion* scheme it supports for generating codata and their *efficient codata constructors* (in the sense that the run-time overhead of every use of a constructor is constant in the number of observations made on the codata);
- show that our derived codata supports a proof principle in the form of *indexed course-of-values corecursion*, which suffices for proving many standard examples of coinductive properties such as relational properties over corresponding elements of two streams;
- and describe a negative result concerning *Lambek's lemma* that indicates the impossibility of coinduction in the sense of "bisimilarity implying equality" (with respect to CDLE's built-in equality) for this encoding; this in turn explains the use of monotone recursive types, as the lack of such a coinduction principle raises fundamental difficulties in adapting for this encoding previous approaches for deriving efficient encodings for *inductive* types in CDLE.

The remainder of this paper is organized as follows: in Section 2 we review CDLE and Mendler-style corecursion schemes; in Section 3 we detail the generic derivation of coinductive data, discussing its strengths and the above-mentioned negative result; in Section 4 we give concrete examples of programming with streams using expressive corecursion schemes supported by our derived codatatypes; in Section 5 we argue that the indexed coiteration scheme they support enables proofs of many properties of interest for coinductive types such as streams; in Section 6 we discuss related and future work; and we conclude in Section 7. All code found in listings can be found at `https://github.com/cedille/cedille-developments` and is checkable by the Cedille tool, and implementation of CDLE available at `https://github.com/cedille/cedille/releases`.

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

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

$$\begin{array}{rcl} |\beta| & = & \lambda\, x.x \\ |\rho\ t'\ -\ t| & = & |t| \\ |\delta\ T\ -\ t| & = & \lambda\, x.x \end{array} \qquad \begin{array}{rcl} |\Lambda x{:}T.t| & = & |t| \\ |t\ \text{-}t'| & = & |t| \end{array}$$

Figure 1: A fragment of CDLE (equality and implicit products)

2 Preliminaries

2.1 CDLE

We begin with a review of the CDLE, the type theory of Cedille. CDLE is an extension of impredicative Curry-style (i.e., extrinsically typed) CC that overcomes some traditional short-comings of lambda encodings in type theory (e.g., underivability of induction for them in second-order dependent type theory [12]) while maintaining a compact formal description: Cedille Core, a minimal specification of CDLE, can be implemented in ∼1K Haskell LoC [33]. CDLE accomplishes this by adding three new type constructs: equality of untyped terms ($\{t \simeq t'\}$); the dependent intersections ($\iota\, x{:}T.\, T'$) of Kopylov [16]; and the implicit (erased) products ($\forall x{:}T.\, T'$) of Miquel [25]. The pure term language of CDLE is the just the untyped λ-calculus; to make type checking algorithmic, terms in Cedille are given with typing annotations, and definitional equality of terms is modulo erasure of annotations. Figures 1 and 12 (Appendix A) gives the typing and erasure rules for the fragment of CDLE relevant to this paper. In particular, the details for dependent intersections are given in the appendix as they are not required understanding the main results of this paper – they are used in Appendix B to implement the derived type constructors of Section 2.2. A complete reference for the syntax, typing, and erasure, along with some meta-theoretic results, can be found in [32].

Equality $\{t_1 \simeq t_2\}$ is the type of proofs that the erasures of t_1 and t_2 ($|t_1|$ and $|t_2|$, resp.) are equal. It is introduced with β, which proves $\{t \simeq t\}$ for any t whose free variables are declared in the typing context. The term β erases to $\lambda\, x.x$, similar to the lambda encoding for Leibniz equality in CC. Combined with definitional equality, β proves $\{t_1 \simeq t_2\}$ for terms t_1 and t_2 whose erasures are $\beta\eta$-convertible.

We eliminate equality proofs with $\rho\ t'\ -\ t$, a substitution principle, and with $\delta\ T\ -\ t$, which provides the *principle of explosion* for a certain contradictory equation. For substitution with ρ, if the expected type of the expression $\rho\ t'\ -\ t$ is $T[t_1/x]$, and t' proves $\{t_1 \simeq t_2\}$, then t is checked against the type $T[t_2/x]$. The expression $\rho\ t'\ -\ t$ erases to $|t|$, making equality in Cedille *proof-irrelevant* (in contrast to theories like CIC and MLTT). For δ, if t proves that the lambda encoding of the Boolean for *true* is equal to the encoding for *false*, $\delta\ T\ -\ t$ has type T, for any type T. The expression $\delta\ T\ -\ t$ erases to $\lambda\, x.x$, so again the proof of equality t is computationally irrelevant. As a convenience, the Cedille tool implements for δ the *Böhm-out algorithm* [6] so that it may be used on proofs $\{t_1 \simeq t_2\}$ for any closed and normalizing terms t_1 and t_2 whose erasures are $\beta\eta$-inconvertible.

Implicit product $\forall x : T'.\, T$ is the type of dependent functions with an *erased* (computationally irrelevant) argument of type T' and a result of type T. They are introduces with an abstraction $\Lambda x : T'.\, t$ where we require that x does not occur free in the erasure of t, permitting the erasure of $\Lambda x : T'.\, t$ to be $|t|$. Because of this restriction on x, erased arguments play no computational role in the body t and thus exist solely for the purposes of typing.

Terms t having an implicit product type $\forall x : T'.\, T$ are eliminated with *erased application*: if t' has type T', then the erased application t -t' has type $T[t'/x]$ and erases to $|t|$. When x does not occur free in the type T', we write $T \Rightarrow T'$ for $\forall x : T.\, T'$, similar to $T \to T'$ for non-dependent non-implicit products.

Inherited type constructs from CC. Figure 1 omits typing and erasure rules for the term and type constructs of CC. In terms, all type abstractions $\Lambda X.\, t$ (so, Λ is used for introducing polymorphic terms as well as for functions with erased term arguments) erase to $|t|$, and all term-to-type applications $t \cdot S$ erase to $|t|$. In types, $\forall$ also quantifies over types, $\Pi x : T'.\, T$ is a (non-implicit) dependent product, and λ introduces a type-level function. In code listings, we omit type arguments and annotations when Cedille can infer these.

2.2 Derived constructs

Our derivation of coinductive datatypes makes use of several constructions which are themselves derived within CDLE. However, due to space restrictions we do not detail their definitions, instead choosing to present them axiomatically with type inference rules and (when appropriate) rules for definitional equality. These include familiar non-recursive datatypes – `Pair` for pairs and `Unit` for the singleton type (Figure 3) – all derivable following a similar approach as described by Stump [31]. Some of the derived type constructors are more exotic (Figure 2): `Cast` for zero-cost type coercions, `Mono` for internalized monotonicity witnesses of type schemes, and a recursive type former `Rec`. We describe this latter kind, whose complete derivation in Cedille is listed in Appendix B (and in the code repository for this paper), and are described at length by Jenkins and Stump in [15]. In order to support indexed coinductive types, the definitions in Figure 2 and through Section 3 make use of an indexing type $I : \star$.

`Cast`, zero-cost type coercions. For type families S and T of kind $I \to \star$, $\texttt{Cast} \cdot I \cdot S \cdot T$ is the type of generalized identity functions in CDLE; its formation, introduction, and elimination rules are given in Figure 2a. This family of types was first introduced by Firsov et al. in [8], wherein it is called `Id`, and only the non-indexed variant is given (for the related notion of zero-cost coercion in Haskell, see [5]). Since CDLE is Curry-style, an identity function from S to T might exists even if S and T are inconvertible types. Terms of type $\texttt{Cast} \cdot I \cdot S \cdot T$ are introduced with `intrCast` -f -p (that is, with both arguments erased), where f has type $\forall i : I.\, S\ i \to T\ i$ and p (of type $\forall i : I.\, \Pi x : S\ i.\, \{f\ x \simeq x\}$) is a proof that f behaves *extensionally* like the identity function (there is no need to write the equality as $\{f$ -$i\ x \simeq x\}$, as equality for terms is modulo erasure). Casts are eliminated with `elimCast`: if c has type $\texttt{Cast} \cdot I \cdot S \cdot T$ and i has type I, then `elimCast` -c -i is a function from $S\ i$ to $T\ i$, which (crucially) is definitionally equal to $\lambda x.\, x$ (indicated by the notation $=_\beta$ in the figure). Since `elimCast` takes its cast argument erased, `Cast` is also proof-irrelevant. Constructs `castRefl` and `castTrans` give us that casts form a preorder on type families (there can be at most one identity function between any two type families) and can be defined in terms of `intrCast` and `elimCast`. Both `castRefl` and `castTrans` are definitionally equal to $\lambda x.\, x$ as well.

(a) `Cast` (zero-cost coercions)

$$\frac{\Gamma \vdash S : I \to \star \quad \Gamma \vdash T : I \to \star}{\Gamma \vdash \mathtt{Cast} \cdot I \cdot S \cdot T : \star} \qquad \frac{\Gamma \vdash c : \mathtt{Cast} \cdot I \cdot S \cdot T \quad \Gamma \vdash i : I}{\Gamma \vdash \mathtt{elimCast}\ \text{-}c\ \text{-}i : S\ i \to T\ i}$$

$$\frac{\Gamma \vdash f : \forall i{:}I.A\ i \to B\ i \quad \Gamma \vdash p : \forall i{:}I.\Pi x{:}S\ i.\{f\ x \simeq x\}}{\Gamma \vdash \mathtt{intrCast}\ \text{-}f\ \text{-}p : \mathtt{Cast} \cdot I \cdot S \cdot T}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{Cast} \cdot I \cdot S \cdot T \quad \Gamma \vdash t_2 : \mathtt{Cast} \cdot I \cdot T \cdot U}{\Gamma \vdash \mathtt{castTrans}\ \text{-}t_1\ \text{-}\ t_2 : \mathtt{Cast} \cdot I \cdot S \cdot U} \qquad \frac{\Gamma \vdash S : I \to \star}{\Gamma \vdash \mathtt{castRefl} \cdot I \cdot S : \mathtt{Cast} \cdot I \cdot S \cdot S}$$

$$\mathtt{intrCast} =_\beta \lambda x.x, \quad \mathtt{elimCast} =_\beta \lambda x.x$$
$$\mathtt{castRefl} =_\beta \lambda x.x, \quad \mathtt{castTrans} =_\beta \lambda x.x$$

(b) `Mono` (internalized monotonicity predicate)

$$\frac{\Gamma \vdash I : \star \quad \Gamma \vdash F : (I \to \star) \to I \to \star}{\Gamma \vdash \mathtt{Mono} \cdot I \cdot F : \star} \qquad \frac{\Gamma \vdash m : \mathtt{Mono} \cdot I \cdot F \quad \Gamma \vdash c : \mathtt{Cast} \cdot I \cdot S \cdot T \quad \Gamma \vdash i : I}{\Gamma \vdash \mathtt{elimMono}\ \text{-}m\ \text{-}c\ \text{-}i : F \cdot S\ i \to F \cdot T\ i}$$

$$\frac{\Gamma \vdash f : \forall X{:}I \to \star.\forall Y{:}I \to \star.\mathtt{Cast} \cdot I \cdot X \cdot Y \Rightarrow \mathtt{Cast} \cdot I \cdot (F \cdot X) \cdot (F \cdot Y)}{\Gamma \vdash \mathtt{intrMono}\ \text{-}f : \mathtt{Mono} \cdot I \cdot F}$$

$$\mathtt{intrMono} =_\beta \lambda x.x, \quad \mathtt{elimMono} =_\beta \lambda x.x$$

(c) `Rec` (recursive type former)

$$\frac{\Gamma \vdash I : \star \quad \Gamma \vdash F : (I \to \star) \to I \to \star}{\Gamma \vdash \mathtt{Rec} \cdot I \cdot F : I \to \star} \qquad \frac{\Gamma \vdash m : \mathtt{Mono} \cdot I \quad \Gamma \vdash i : I}{\Gamma \vdash \mathtt{unroll}\ \text{-}m\ \text{-}i : \mathtt{Rec} \cdot F\ i \to F \cdot (\mathtt{Rec} \cdot F)\ i}$$

$$\frac{\Gamma \vdash m : \mathtt{Mono} \cdot I \cdot F \quad \Gamma \vdash i : I}{\Gamma \vdash \mathtt{roll}\ \text{-}m\ \text{-}i : F \cdot (\mathtt{Rec} \cdot F)\ i \to \mathtt{Rec} \cdot F\ i}$$

$$\mathtt{roll} =_\beta \lambda x.x, \quad \mathtt{unroll} =_\beta \lambda x.x$$

Figure 2: Derived type constructors for monotone recursive types

$$\frac{\Gamma \vdash A : \star \quad \Gamma \vdash B : \star}{\Gamma \vdash \texttt{Pair} \cdot A \cdot B : \star} \qquad \frac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : B}{\Gamma \vdash \texttt{intrPair}\ t_1\ t_2 : \texttt{Pair} \cdot A \cdot B}$$

$$\frac{\Gamma \vdash p : \texttt{Pair} \cdot A \cdot B}{\Gamma \vdash \texttt{fst}\ p : A} \qquad \frac{\Gamma \vdash p : \texttt{Pair} \cdot A \cdot B}{\Gamma \vdash \texttt{snd}\ p : B}$$

$$\frac{}{\Gamma \vdash \texttt{Unit} : \star} \qquad \frac{}{\Gamma \vdash \texttt{unit} : \texttt{Unit}}$$

Figure 3: Derived datatypes: pairs and the unitary type

`Mono`, internalized positivity. Given $F : (I \to \star) \to I \to \star$, $\texttt{Mono} \cdot I \cdot F$ (Figure 2b) is the type of proofs that the type scheme F is *monotonic*. Monotonicity for type schemes (as opposed to *syntactic* positivity) as presented here resemble the work of Matthes [21, 19], in which monotonicity is given by "terms of functorial strength". Terms of type $\texttt{Mono} \cdot I \cdot F$ are introduced by `intrMono`, which takes as an argument some f which (similar to the more familiar morphism mapping of a functor in category theory), for all type families X and Y of kind $I \to \star$, transforms an (erased) cast from X to Y to a cast from $F \cdot X$ to $F \cdot Y$. A witness of positivity m of type $\texttt{Mono} \cdot I \cdot F$ can be eliminated with `elimMono` *-m -c -i* to a function of type $F \cdot S\ i \to F \cdot T\ i$, where c has type $\texttt{Cast} \cdot I \cdot S \cdot T$ (for some type families S and T) and i has type I. The entire expression is definitionally equal to $\lambda\, x.x$ (notice that all arguments to `elimMono` are erased).

`Rec`, a recursive type former. Given $F : (I \to \star) \to I \to \star$, $\texttt{Rec} \cdot I \cdot F$ (Figure 2c) is a fixedpoint of F (strictly speaking, it is the *least* fixedpoint in the pre-order on type schemes induced by `Cast`). It is well-known that unrestricted formation and use of recursive types is unsound when interpreting type theories under the Curry-Howard isomorphism, and so accordingly the introduction (`roll`) and elimination (`unroll`) of terms with recursive types requires evidence that the type scheme F whose fixedpoint was taken is monotonic. If m has type $\texttt{Mono} \cdot I \cdot F$ and i has type I, then `roll` *-m -i* is a function taking some term of type $F \cdot (\texttt{Rec} \cdot F)\ i$ and producing a term of type $\texttt{Rec} \cdot F\ i$. The situation is symmetric for `unroll`.

In type theories with primitive iso-recursive types, `roll` and `unroll` must form an isomorphism. Usually, definitional equality is extended with the β-law for `Rec`, meaning $\texttt{unroll}\ (\texttt{roll}\ t)$ reduces to t, with the η-law $\texttt{roll}\ (\texttt{unroll}\ t) = t$ possibly holding only meta-theoretically. The derived recursive types of CDLE differ from this in that both `roll` and `unroll` are each themselves definitionally equivalent to $\lambda\, x.x$; in this respect, recursive types in CDLE are akin to *equi-recursive* types as the terms `roll` *-m -i t* and t are definitionally equivalent (for t of type $F \cdot (\texttt{Rec} \cdot F)\ i$), as well as the terms `unroll` *-m -i* t' and t' (for t' of type $\texttt{Rec} \cdot F\ i$).

2.3 Coalgebras and coiteration schemes

In category theory, coinductive datatypes are understood as *final F-coalgebras* [14]. Our generic derivation rests upon this understanding, but in order to support efficient constructors and course-of-values corecursion we find it convenient to have our semantic account begin with *Mendler-style F-coalgebras*. The dual notion, the Mendler-style F-algebra, is discussed in more depth by Uustalu and Vene [35] and by Vene [37]; the following categorical account is a straightforward adaptation of this description for F-coalgebras.

Definition 2.1 (Ordinary F-coalgebras)**.** Assuming $F : \mathscr{C} \to \mathscr{C}$ is a functor, the usual definition of an F-coalgebra is a pair (X, ϕ) where X is an object (e.g., a type) called the *carrier* of the algebra and $\phi : X \to F\ X$ is a morphism (e.g., a function) from X to $F\ X$ called the *action* of the coalgebra.

Translated to type theory, the actions of F-coalgebras are expressed by the family of types

$$\lambda\, X{:}\star.\, X \to F \cdot X$$

Definition 2.2. (Mendler-style F-coalgebras) A *Mendler-style F-coalgebra* is a pair (X, Φ) where the carrier X is an object and the action Φ is a natural transformation (e.g., a polymorphic function) that, for every object R in $\mathscr{C}$, maps elements of $\mathscr{C}(X, R)$ (the set of morphisms from X to R) to elements of $\mathscr{C}(X, F\ R)$.

Translated to type theory, the actions of Mendler F-coalgebras are expressed by the family of types:

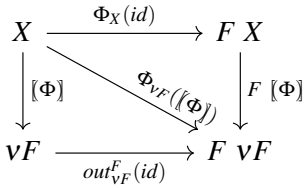
$$\texttt{CoAlgM} = \lambda\, X{:}\star.\, \forall R{:}\star.\, (X \to R) \to X \to F \cdot R$$

In functional programming, the operational reading of the type $X \to F \cdot X$ of an action of an ordinary F-coalgebra (X, ϕ) is that it is a function taking some "seed value" of type X and producing some "structure" of type $F \cdot X$. For an action of a Mendler F-coalgebra polymorphic in R, it instead produces a structure of type $F \cdot R$ given both a seed value of type X and a function for transforming such seed values to values of type R.

Definition 2.3 (Final Mendler F-coalgebras)**.** A *final* Mendler-style F-coalgebra $(\nu F, out^F)$ is one such that for every other Mendler-style F-coalgebra (X, Φ), there exists a unique morphism $[\![\Phi]\!] : X \to \nu F$ (called the *Mendler anamorphism* of Φ) such that

$$out^F_{\nu F}(id) \circ [\![\Phi]\!] = \Phi_{\nu F}([\![\Phi]\!]) = (F\ [\![\Phi]\!]) \circ \Phi_X(id)$$

where subscripts on natural transformations indicating component selection (e.g., polymorphic instantiation) and $F\ [\![\Phi]\!] : F\ X \to F\, \nu F$ denotes the functorial lifting of $[\![\Phi]\!]$ to a morphism from $F\ X$ to $F\ \nu F$. This condition can be alternatively be stated as saying that, for any Mendler F-coalgebras (X, Φ), $[\![\Phi]\!]$ is the unique morphism making the diagram below commute:



In type theory, the carrier νF of the final Mendler F-coalgebra can be given as:

$$\texttt{Nu} = \exists X{:}\star.\, X \times \texttt{CoAlgM} \cdot X$$

or encoded as $\forall Y{:}\star.\, (\forall X{:}\star.\, X \to \texttt{CoAlgM} \cdot X \to Y) \to Y$ if existentials and products are unavailable.

Intuitively, $out^F_{\nu F}$ is the generic destructor for the coinductive datatype νF with pattern functor F, and for any Mendler F-coalgebra (X, Φ), $[\![\Phi]\!]$ is a generator for νF. The equations express how to *compute* the observations $out^F_{\nu F}(id)$ for codata generated with $[\![\Phi]\!]$: these are given by simply calling Φ with the generator $[\![\Phi]\!]$, or equivalently by first calling Φ with a trivial generator id_X, then mapping over the resulting $F\ X$ structure with the generator $[\![\Phi]\!]$.

Mendler-style coiteration The Mendler-style anamorphism translates to a *coiteration* scheme *coit* for codata, whose typing and computation rules are given below (with $out : \nu F \to F\ \nu F$ below corresponds to $out^F_{\nu F}(id)$ above).

$$\frac{a : \forall R{:}\star.\,(X \to R) \to X \to F \cdot R \quad x : X}{coit\ a : X \to \nu F} \qquad out\ (coit\ a\ x) \twoheadrightarrow a\ (coit\ a)\ x$$

Read operationally, the type of a suggests that it will, from a seed value of type X and function for making coiterative calls $X \to R$ (where R is universally quantified over), construct an F-collection of R subdata, corresponding to one additional observation that can later be made on the codata being generated.

Mendler-style corecursion The Mendler-style *corecursion* scheme (categorically, the *apomorphism*) can be described by equipping the coalgebra action a with an additional argument of type $\nu F \to R$, supporting an alternative method for constructing codata – a way of "short-cutting" coiteration by injecting codata directly into the abstracted type R rather than generating it from values of type X. We expect that the type argument of a will always be instantiated to νF, and thus that $\lambda\, x.x$ can be given for this additional argument:

$$\frac{a : \forall R{:}\star.\,(\nu F \to R) \to (X \to R) \to X \to F \cdot R}{corec\ a : X \to \nu F} \qquad out\ (corec\ a\ x)\ \twoheadrightarrow a\ (\lambda\, x.x)\ (corec\ a)\ x$$

Mendler-style course-of-values coiteration The Mendler-style *course-of-values coiteration* scheme (categorically, the *futumorphism*) can be given by equipping a with an argument of type $F \cdot R \to R$, another alternative to coiteration for constructing codata that takes an F-collection of R subdata. This additional argument enables a to construct an arbitrary number of observations for the codata being generated *in addition to* those produced by coiteration; put another way, the $F \cdot R$ result that a produces may have some R sub-component built using this additional argument from another $F \cdot R$ expression, which itself would be later accessed by making (at least) two observations on the generated codata. We expect that $out^{-1} : F \cdot \nu F \to \nu F$ (that is, the generic codata constructor) can be given for this argument:

$$\frac{a : \forall R{:}\star.\,(F \cdot R \to R) \to (X \to R) \to X \to F \cdot R}{cov\ a : X \to \nu F} \qquad out\ (cov\ a\ x)\ \twoheadrightarrow a\ out^{-1}\ (cov\ a)\ x$$

A categorical account of the corecursion scheme for ordinary F-coalgebras is given by Geuvers [11], and an account of the recursion scheme and course-of-values iteration scheme (duals to the corecursion scheme and course-of-values coiteration scheme) for Mendler-style F-algebras is given by Vene [37]. For understanding the derivation of coinductive data in the next section, which supports a combination of the corecursion and course-of-values coiteration schemes which we call *course-of-values corecursion*, the above type-theoretic account suffices.

3 Generic lambda encoding for codata

Figures 4, 5, and 6 give the complete derivation of coinductive data in Cedille. This derivation is *generic*, in that it works for any monotone type scheme F of kind $(I \to \star) \to I \to \star$ (where monotonicity is expressed as evidence *cm* of type $\texttt{Mono} \cdot I \cdot F$). I, F, and *cm* are all module parameters to the derivation, and the curly braces around *cm* indicate that it is an *erased* module parameter (as suggested by the type inference and definitional equal rules in Figure 2b, the type of monotonicity witnesses is *proof-irrelevant*). We walk through code listings of these figures in detail.

```
import utils.

module nu (I: ⋆) (F: (I → ⋆) → I → ⋆) {cm: Mono ·I ·F}.

CoAlgM : (I → ⋆) → (I → ⋆) → ⋆
= λ X: I → ⋆. λ C: I → ⋆.
  ∀ R: I → ⋆. Cast ·I ·C ·R ⇒ (∀ i: I. F ·R i → R i) →
  Π ch: (∀ i: I. X i → R i). ∀ i: I. X i → F ·R i.

NuF : (I → ⋆) → I → ⋆
= λ C: I → ⋆. λ i: I. ∀ Y: I → ⋆. (∀ X: I → ⋆. X i → CoAlgM ·C ·X → Y i) → Y i.

Nu : I → ⋆ = Rec ·I ·NuF.
```

Figure 4: Generic lambda encoding for codata (part 1)

```
monoCoAlgM : ∀ X: I → ⋆. Mono ·I ·(λ C: I → ⋆. λ i: I. CoAlgM ·C ·X) = Λ X.
  intrMono -(Λ C1. Λ C2. Λ c.
    intrCast
      -(Λ i. λ coa. Λ R. Λ c'. coa -(castTrans -c -c'))
      -(Λ i. λ coa. β)) .

monoNuF : Mono ·I ·NuF = <..>

nuRoll   : ∀ i: I. NuF ·Nu i → Nu i = Λ i. roll   -monoNuF -i .
nuUnroll : ∀ i: I. Nu i → NuF ·Nu i = Λ i. unroll -monoNuF -i .
```

Figure 5: Generic lambda encoding for codata (part 2)

`CoAlgM`**.** In Figure 4, our variant Mendler-style F-coalgebra, `CoAlgM`, generalizes the description given in Section 2.3 to type families `X: I → ⋆`. `CoAlgM` takes an additional type family argument `C`, and describes a family of polymorphic functions whose arguments facilitate a combined course-of-values corecursion scheme for codata. The variable `C` stands in for occurrences of the generic codatatype `Nu` – which itself recursively defined in terms of `CoAlgM`. In the body of the definition, the first additional argument is a *cast* from `C` to the quantified `R`, enabling *corecursion* by allowing us to produce (for any `i` of type `I`) some term of type `R i` from some pre-existing codata of type `C i`, rather than via `ch`. The second argument is an *abstract constructor*: given some collection of subdata of type `F ·R i` (for any `i`), it builds a value of type `R i`. This additional argument enables *course-of-values coiteration*: for each single step of codata generation, an *arbitrary* number of observations that will be made of the codata can be constructed. The remaining arguments are the same as for ordinary Mendler-style F-coalgebras: `ch` is the handle for making coiterative calls, and the argument of type `X i` is the value from which we are to coiteratively construct the result of type `F ·R i`.

`NuF` **and** `Nu`**.** Type family `NuF` is defined using the standard type for lambda encodings of existentials and products, and its body is more naturally read as `∃ X: I → ⋆. X i × CoAlgM ·C ·X`. This is similar to the standard definition of the greatest fixpoint of `F` (Section 2.3, see also [38]). `NuF` is parameterized by a type family `C: I → ⋆` standing in for recursive reference to a fixpoint of itself. That fixpoint is `Nu`, defined as `Rec ·I ·NuF`.

```
unfoldM : ∀ X: I → ⋆. CoAlgM ·Nu ·X → ∀ i: I. X i → Nu i
= Λ X. λ coa. Λ i. λ x. nuRoll -i (Λ Y. λ f. f x coa) .

inM : ∀ i: I. F ·Nu i → Nu i
= unfoldM ·(F ·Nu) (Λ R. Λ c. λ v. λ ch. Λ i. λ x. elimMono -cm -c -i x) .

outM : ∀ i: I. Nu i → F ·Nu i
= Λ i. λ co.
  nuUnroll -i co ·(F ·Nu)
    (Λ X. λ x. λ coa. coa -(castRefl ·I ·Nu) inM (unfoldM coa) -i x) .
```

Figure 6: Generic lambda encoding for codata (part 3)

monoCoAlgM, monoNuF, nuRoll, and nuUnroll As discussed earlier, recursive types must be restricted in some fashion to positive (monotonic) type schemes only. The term monoNuF in Figure 5 (definition omitted, indicated by <..>) proves that the type scheme NuF is positive; it uses evidence monoCoAlgM that CoAlgM is positive in its first argument. To show that a type scheme is positive (Mono, Section 2.2), we use intrMono and may then assume two type families C1 and C2 and a cast c between them. Aside from monoCoAlgM, in code listings we omit monotonicity proofs as once the general principle is understood these can be quite tedious to work through.

For monoCoAlgM, the goal is to show there is a cast from CoAlgM ·C1 ·X to CoAlgM ·C2 ·X (for any X of kind I → ⋆). This is done with intrCast (also in Section 2.2), whose first argument is simply a function of this type and whose second is a proof that this function is extensionally equal to $\lambda x.x$. For the functional argument, it is simply a matter of deriving a Cast ·I ·C1 ·R (needed for the assumed coa of type CoAlgM ·C1 ·R) from the given c' of type Cast ·C2 ·R and c of type Cast ·C1 ·C2, which we have by castTrans, the composition operator for casts. This new cast is given to coa as an erased argument, so the function simply erases to λ coa. coa, and therefore the proof that this is extensionally equal to $\lambda x.x$ is trivial (λ coa. β), since it is *intensionally* equal to it.

With these proofs that NuF is a positive type scheme, we can define the rolling and unrolling operations nuRoll and nuUnroll for the recursive type family Nu.

unfoldM and inM. We now discuss the definitions given in Figure 6. Function unfoldM is the generator for our Mendler-style codata. For any type family X of kind I → ⋆, given some coa of type CoAlgM ·Nu ·X and some x of type X i (for arbitrary i of type I), we generate a term of type Nu i by first invoking nuRoll, which obligates us to produce some argument of type NuF ·Nu i. Unfolding the definition of NuF, we provide for this an encoded existential: a function polymorphic over a type family Y of kind I → ⋆ taking another function f of type ∀ X: I → ⋆. X i → CoAlgM ·Nu ·X → Y i and applying f to the given x and coa.

Our generic constructor inM is defined in terms of unfoldM. The coalgebra given to unfoldM ignores its argument ch and instead casts (using cm, the module parameter that proves monotonicity of F, and c, the coalgebra's assumed Cast ·I ·Nu ·R) the assumed x of type F ·(Nu ·F) to the type F ·R. Avoidance of ch, the coalgebra's handle for coiterative calls, ensures that inM is *efficient* – future observations of the codata constructed with inM will not needlessly rebuild the sub-components of the codata with which it was constructed. This discussion is made more concrete in Section 3.1.

outM. Finally, outM is our generic destructor for codata. Given some co of type Nu i, we use the unrolling operation for recursive type Nu on co to produce a term of type NuF ·Nu i. Unfolding the definition of NuF, we provide the resulting expression a function which assumes a type family X of kind I → ⋆, a term x of type X i, and a coalgebra coa of type CoAlgM ·(Nu ·F) ·X. In the body of this given function, we eliminate coa by instantiating its type argument to Nu ·F and giving it a cast from Nu ·F to Nu ·F (using castRefl), the constructor inM, a handle for making coiterative calls unfoldM coa, and the assumed x. We can show in Cedille that the computation rules expected of our derived codatatype Nu and generator unfoldM (Section 2.3) hold by β-equivalence (the cast argument, corresponding to $\lambda x.x$ in the computation rule for the corecursion scheme, is erased; inM corresponds to out^{-1}):

```
reduce : ∀ X: I → ⋆. Π coa: CoAlgM ·Nu ·X. ∀ i: I. Π x: X i.
  { outM (unfoldM coa x) ≃ coa inM (unfoldM coa) x }
= Λ X. λ coa. Λ i. λ x. β .
```

3.1 *Lambek's lemma*

In following sections, we shall demonstrate that our generic coinductive datatype is adequate for both expressive functional programming and for giving proofs of many standard coinductive properties such as bisimulation. Our justification for the first claim rests both on the *expressivity* allowed to programmers in defining functions producing codata (using a course-of-values corecursion scheme) and the *efficiency* of the functions so defined. This efficiency is due to our constructor inM being an operation incurring run-time overhead that is constant in the number of observations (destructions) made on the codata. In Cedille, we can show directly that destructing an arbitrary constructed value produces the original collection of subdata from which the value was constructed in a constant number of β-reductions.

```
lambek1 : ∀ i: I. Π xs: F ·Nu i. {outM (inM xs) ≃ xs} = Λ i. λ xs. β.
```

The name of this proof comes from *Lambek's lemma* [17], which proves that (the actions of) final coalgebras are isomorphisms. Specifically, for any functor F, Lambek's lemma for final (ordinary) F-coalgebra $(\nu F, out^F)$ states that there exists an inverse $in^F : F\ \nu F \to \nu F$ such that $out^F \circ in^F = id_{F\ \nu F}$ and $in^F \circ out^F = id_{\nu F}$. Lambek's lemma is a consequence of the uniqueness of the anamorphism, which is itself a category-theoretic expression of coinduction: the usual definition of in^F is given by the generation of codata using the anamorphism of $F\ out^F : F\ \nu F \to F(F\ \nu F)$ to coiteratively rebuild its observations. Our derivation of Mendler-style coinductive data restricts F in such a way that functorial lifting is only defined for casts (F is assumed to be Mono, not a functor), and our alternative definition for in^F (inM) avoids this needless rebuilding, and so one direction of Lambek's lemma holds trivially.

Proving the Lambek's lemma in the form $out^F \circ in^F = id_{F\ \nu F}$ would require *functional extensionality*, but this in itself does not necessitate that the equality implied to hold for corresponding elements in images of $out^F \circ in^F$ and $id_{F\ \nu F}$ must be extensional. Indeed, for derivations of *inductive* datatypes in Cedille both directions of Lambek's lemma need only *intensional* equality. We now show that proof of the other direction of Lambek's lemma for our derived coinductive data appears to truly require an *extensional* equality type by giving a counter-example in Figure 7 with respect to Cedille's currently intensional equality type.

TF, tcoa, and t. Type scheme TF is the signature for T, and simply maps any type family X of kind Unit → ⋆ to itself (recall that Unit is the unitary type, Figure 3); our generic development requires that every signature be indexed, so we provide Unit as a "dummy"). TF is obviously monotonic, so

```
import utils.
import nu.

module lambek.

TF : (Unit → ⋆) → (Unit → ⋆) = λ X: Unit → ⋆. X.
monoTF : Mono ·Unit ·TF = <..>

T : Unit → ⋆ = Nu ·Unit ·TF monoTF.
TCoAlgM : ⋆ → ⋆ = λ X: ⋆. CoAlgM ·Unit ·TF monoTF ·T ·(λ _: Unit. X).

tcoa : TCoAlgM ·Unit = Λ R. Λ c. λ v. λ ch. Λ i. λ x. ch -i x.
t : T unit = unfoldM -monoTF ·(λ _: Unit. Unit) tcoa -unit unit .

noLambek2 : {inM (outM t) ≃ t} → ∀ X: ⋆. X = λ eq. Λ X. δ X - eq.
```

Figure 7: Violation of *Lambek's lemma*

the definition of the proof of this fact `monoTF` is omitted (indicated by `<..>`). This given, we define coinductive datatype `T` as the greatest fixpoint of `TF`, and for convenience define the type family `TCoAlgM` of `TF`-coalgebras. Term `t` of type `T unit` is generated via `unfoldM` using `unit` as the "seed" and a `TF`-coalgebra `tcoa` which simply makes a corecursive call with `ch` of type `∀ i: Unit. Unit → R i` to produce a result of type `TF ·R i` (convertible with the return type `R i` of the expression `ch -i x`).

`noLambek2`**.** To see why the final proof `noLambek2` holds, it is useful to rewrite the two sides of the assumed equation `eq` to β-equivalent expressions:

```
{ λ g. g (λ f. f unit tcoa) (λ v. λ ch. λ x. x) ≃ λ f. f unit tcoa }
```

The right-hand side of the equation (corresponding to `t`) is a lambda encoding of a pair consisting of a seed value `unit` and generator `tcoa`. However, the pair on the left-hand side has a seed value which itself is the very same pair on the right-hand side, and a generator which simply returns the seed value as-is. Though we understand that these two expressions are *extensionally* equal (that is, treated as black-boxed terms of type `T unit` they produce the same observations), in Cedille the current built-in equality type is *intensional*. Indeed, δ (Section 2.1) makes it *anti-extensional*, though we know of no fundamental reason why CDLE could not be extended with a more general extensional equality type. Additionally (and as suggested by a reviewer for the draft version of this paper), we conjecture that a reformulation of Lambek's lemma in terms of the equivalence relation for existential types given by Reynold's relational parametricity (such as undertaken by Pitts in [28]) should be provable. In CDLE, dependent intersections can be used to equip lambda encodings with a proof principle for parametricity in precisely the same way used to equip encodings of inductive types with an induction principle [31]; we leave this as future work.

As we will see in Section 5, the foregoing negative result does not impact the ability to give proofs for many standard coinductive properties for this encoding of codata, such as showing stream bisimilarity or other relational properties; for many use cases, indexed coiteration suffices. What this result indicates is currently impossible for our encoding is proving "true coinduction" in the sense of bisimilarity implying equality (since if this we had this, we would be able to prove Lambek's lemma).

```
import utils.
module examples/streamf (A: ⋆).

StreamF : (Unit → ⋆) → Unit → ⋆ = λ R: Unit → ⋆. λ u: Unit. Pair ·A ·(R u).

monoStreamF : Mono ·Unit ·StreamF = <..>

import nu/nu ·Unit ·StreamF -monoStreamF.

Stream : ⋆ = Nu unit.
StreamCoAlg : ⋆ → ⋆ = λ X: ⋆. CoAlgM ·(λ _: Unit. Stream) ·(λ _: Unit. X).

head : Stream → A = λ xs. fst (outM -unit xs) .
tail : Stream → Stream = λ xs. snd (outM -unit xs) .

unfoldStream : ∀ X: ⋆. StreamCoAlg ·X → X → Stream = <..>
```

Figure 8: Definition of streams

4 Functional programming with streams

In this section, we give a few examples of programming with lambda-encoded codata (specifically, streams), emphasizing the expressivity the course-of-values corecursion scheme available to programmers. After defining the stream codatatype by instantiating the parameters of our generic development (Figure 8), we show how to define: the mapping of a function over the elements of a stream, an example of *coiteration*; the mapping of a function over just the head element of a stream, an example of *corecursion*; and the pairwise exchange of elements of a stream, an example of *course-of-values coiteration*. These example functions, given in Figure 9, also appear in Vene's thesis [37], though there they are given using the traditional account of coinductive types as final F-algebras (not final Mendler-style ones).

Definition of streams. In Figure 8, `StreamF` is the signature for streams of elements of type `A` (where `A` is a module parameter; it is implicitly quantified over in all definitions in the figure), defined in terms of `Pair` (Figure 3); the definition of `StreamF` is more recognizable in the form $R \mapsto A \times R$. Term `monoStreamF` is a witness to the fact that `StreamF` is positive (definition omitted, indicated by `<..>`).

The import of module `nu/nu` instantiates that module's parameters with index type `Unit`, signature functor `StreamF`, and positivity proof `monoStreamF`. Type `Stream` is the greatest fixpoint of `StreamF` with a fixed index of `unit`. Destructors `head` and `tail` are defined in terms of the generic destructor `out` and pair projections `fst` and `snd`. The generator `unfoldStream` has a first argument of type `StreamCoAlg ·X`, which specializes `CoAlgM` such that the type families for the codatatype and carrier ignore their `Unit` indices. The body of the definition of `unfoldStream` is omitted, as it is somewhat cluttered by type annotations to replace the assumed `i: Unit` with the constant `unit` in the type of the coalgebra given to `unfoldM`. We define this function so that the following examples may use it, and not be themselves so cluttered.

`map`. Our first function `map` is an example of *coiteration* over streams. First, note that this and following functions occur in a different module than the definition of `Stream`, so now the element type must be given explicitly both for `Stream` and `unfoldStream`. With a function `f` of type `A → B`, in the body

```
map : ∀ A: ⋆. ∀ B: ⋆. (A → B) → Stream ·A → Stream ·B
= Λ A. Λ B. λ f.
  unfoldStream ·B ·(Stream ·A)
    (Λ R. Λ c. λ v. λ map. Λ i. λ xs.
     intrPair (f (head xs)) (map -i (tail xs))) .

mapHd : ∀ A: ⋆. (A → A) → Stream ·A → Stream ·A = Λ A. λ f.
  unfoldStream ·A ·(Stream ·A)
    (Λ R. Λ c. λ v. λ mapHd. Λ i. λ xs.
     intrPair (f (head xs)) (elimCast -c -i (tail xs))) .

exch : ∀ A: ⋆. Stream ·A → Stream ·A = Λ A.
  unfoldStream ·A ·(Stream ·A)
    (Λ R. Λ c. λ v. λ exch. Λ i. λ xs.
       [hd1 : A = head (tail xs)]
     - [hd2 : A = head xs]
     - [tl2 : R i = exch -i (tail (tail xs))]
     - intrPair hd1 (v -i (intrPair hd2 tl))) .
```

Figure 9: Programming with the coiteration, corecursion, and course-of-values coiteration schemes

of `map` we generate using `unfoldStream` a stream with elements of type `B` from: a coalgebra wherein `map` (of type `∀ i: Unit. Stream ·A → R i`) can be used for corecursive calls; and a seed value `xs` of type `Stream ·A`. In the body of the given coalgebra, we construct a `StreamF ·A` by supplying for the head `f (head xs)` and for the tail `map -i (tail xs)` (of type `R i`).

In a high-level surface language supporting *copatterns* [1], the definition of `map` might look like:

```
head (map f xs) = f (head xs)
tail (map f xs) = map f (tail xs)
```

`mapHd`. Function `mapHd` is an example of *corecursion*. Recall that the corecursion scheme supports an alternative for generating codata: it may be given *directly* in a coalgebra, rather than being generated from coiteration. This mirrors the situation for *iteration* (where predecessors must be iteratively rebuilt) and *recursion* (where they are available "for free") for inductive types. In the definition, with function `f` of type `A → A`, we produce the desired stream by providing for its head `f (head xs)` and for its tail `tail xs`, after coercing its type to the desired `R i` using the given cast `c` from `Stream ·A` to `R i`.

Using a high-level surface language, the definition of `mapHd` might look like:

```
head (mapHd f xs) = f (head xs)
tail (mapHd f xs) = tail xs
```

For comparison, the standard definition of corecursion from coiteration requires "cotupling" (using a sum type as the generating value of the codata). In the same high-level pseudo-code, `mapHd` defined in this way would be more tedious and error-prone, and result in run-time overhead linear in the number of observations made on the codata:

```
mapHd' f xs = go (in1 xs) where
  go : Sum ·(Stream ·A) ·(Stream ·A) → Stream ·A
  head (go (in1 ys)) = f (head ys)
  head (go (in2 ys)) = head ys
  tail (go (in1 ys)) = go (in2 (tail ys))
  tail (go (in2 ys)) = go (in2 (tail ys))
```

exch. Our final example exch, a function which swaps the positions of every two elements of a stream, demonstrates the *course-of-values coiteration* scheme, wherein each step of generating codata may describe an arbitrary number of the observations made of that codata (pairwise exchange only requires the construction of one additional observation at each step). In the coalgebra given in the definition of exch, we make three local definitions (using Cedille's syntax $[x : T = t_1] - t_2$, to be read let $x : T = t_1$ in t_2; we use an additional space to distinguish hyphens used for local definitions from hyphens used in erased applications). These local definitions are: hd1, the second element of the stream xs; hd2, the first element; and tl2, the exchange of elements of the second tail of xs. These definitions are then incorporated into the produced stream, where in the tail we use the abstract constructor v of type ∀ i: Unit. StreamF ·A ·R i → R i.

In a higher-level language, exch can be written using *nested copatterns* (because each step of generation produces a static number of future observations; course-of-values coiteration permits this number to be dynamically computed at each step):

```
head (exch xs)        = head (tail xs)
head (tail (exch xs)) = head xs
tail (tail (exch xs)) = exch (tail (tail xs))
```

5 Coinductive proofs of properties of streams

In this section, we argue that the indexed coiteration scheme enjoyed by our derived codatatypes is sufficient for giving proofs of many standard coinductive properties. We use as an example generalized relations on streams: the lifting of some relation Rel: A → A → ⋆ (over an element type A: ⋆) to a relation StreamRel over streams in which corresponding elements are related; instantiation of Rel to an equivalence relation produces the relation of stream bisimilarity (up to equivalence of elements). The definition of StreamRel is given in Figure 10 (note again the use of module parameters, in particular the import of streamf allowing us to refer by Stream to streams of elements of type A). For proofs, we show in Figure 11 that if Rel is reflexive, symmetric, or transitive, then so too is the relation StreamRel.

StreamRelF and StreamRel. The signature for generalized stream relations, StreamRelF, takes a type family R: Pair ·Stream ·Stream → ⋆ (standing in for recursive occurrences of the greatest fixpoint of StreamRelF itself) and a pair p: Pair ·Stream ·Stream of the two streams to be related, and in the body is defined as the type of pairs of proofs that the first elements of the two streams are related by Rel and proofs that the tails of the streams are related by R. This type scheme is positive in R, as proven by monoStreamRelF (definition omitted). We instantiate the generic codata derivation with these definitions and define StreamRel as the greatest fixpoint of StreamRelF. Destructors headRel and tailRel are given (their definitions are similar to head and tail for streams and so are omitted), as well as the simplified generator unfoldStreamRel: the properties we shall prove using StreamRel require only indexed coiteration, so unfoldStreamRel forgoes the facilities for more advanced generation schemes (namely, indexed course-of-values coiteration) in order to simplify their proofs.

strRefl. To prove reflexivity of StreamRel assuming reflexivity of Rel, we use for the generator the type family of proofs that two streams are equal. In the body of the coalgebra given to unfoldStreamRel we have the following obligations: for the head we must prove Rel (head xs) (head ys), given by rewriting (with ρ, Figure 1) by the assumed g: {xs ≃ ys} and using the assumption that Rel is

```
import utils.

module examples/streamrelf (A: ⋆) (Rel: A → A → ⋆).
import streamf ·A.

StreamRelF : (Pair ·Stream ·Stream → ⋆) → Pair ·Stream ·Stream → ⋆
= λ R: Pair ·Stream ·Stream → ⋆. λ p: Pair ·Stream ·Stream.
  [xs = fst p] - [ys = snd p] -
  Pair ·(Rel (head xs) (head ys)) ·(R (intrPair (tail xs) (tail ys))) .

monoStreamRelF : Mono ·(Pair ·Stream ·Stream) ·StreamRelF = <..>

import nu/nu ·(Pair ·Stream ·Stream) ·StreamRelF -monoStreamRelF.

StreamRel : Stream → Stream → ⋆ = λ xs: Stream. λ ys: Stream. Nu (intrPair xs ys).

headRel : ∀ xs: Stream. ∀ ys: Stream.
          StreamRel xs ys → Rel (head xs) (head ys) = <..>
tailRel : ∀ xs: Stream. ∀ ys: Stream.
          StreamRel xs ys → StreamRel (tail xs) (tail ys) = <..>

unfoldStreamRel : ∀ X: Stream → Stream → ⋆.
  (∀ R: Stream → Stream → ⋆.
     (∀ xs: Stream. ∀ ys: Stream. X xs ys → R xs ys) →
     ∀ xs: Stream. ∀ ys: Stream. X xs ys →
     StreamRelF ·(λ p: Pair ·Stream ·Stream. R (fst p) (snd p)) (intrPair xs ys)) →
  ∀ xs: Stream. ∀ ys: Stream. X xs ys → StreamRel xs ys = <..>
```

Figure 10: Definition of generalized relations between streams

```
Reflexive : Π A: ⋆. (A → A → ⋆) → ⋆
= λ A: ⋆. λ Rel: A → A → ⋆. ∀ x: A. Rel x x.

Symmetric : Π A: ⋆. (A → A → ⋆) → ⋆
= λ A: ⋆. λ Rel: A → A → ⋆. ∀ x: A. ∀ y: A. Rel x y → Rel y x .

Transitive : Π A: ⋆. (A → A → ⋆) → ⋆
= λ A: ⋆. λ Rel: A → A → ⋆. ∀ x: A. ∀ y: A. ∀ z: A. Rel x y → Rel y z → Rel x z .

strRefl : Reflexive ·A ·Rel → Reflexive ·Stream ·StreamRel
= λ refl. Λ xs.
  unfoldStreamRel ·(λ xs: Stream. λ ys: Stream. {xs ≃ ys})
    (Λ R. λ ch. Λ xs. Λ ys. λ g.
     intrPair (ρ g - (refl -(head ys))) (ch -(tail xs) -(tail ys) (ρ g - β)))
    -xs -xs β.

strSym : Symmetric ·A ·Rel → Symmetric ·Stream ·StreamRel
= λ sym. Λ xs. Λ ys. λ rel.
  unfoldStreamRel ·(λ ys: Stream. λ xs: Stream. StreamRel xs ys)
    (Λ R. λ ch. Λ ys. Λ xs. λ g.
     intrPair
       (sym -(head xs) -(head ys) (headRel -xs -ys g))
       (ch -(tail ys) -(tail xs) (tailRel -xs -ys g)))
    -ys -xs rel .

strTra : Transitive ·A ·Rel → Transitive ·Stream ·StreamRel
= λ tra. Λ xs. Λ ys. Λ zs. λ rel1. λ rel2.
  [X : Stream → Stream → ⋆
   = λ xs: Stream. λ zs: Stream.
     ∀ Y: ⋆. (∀ ys: Stream. StreamRel xs ys → StreamRel ys zs → Y) → Y]
- unfoldStreamRel ·X
    (Λ R. λ ch. Λ xs. Λ zs. λ g.
     g (Λ ys. λ rel1. λ rel2.
          [hd : Rel (head xs) (head zs)
           = tra -(head xs) -(head ys) -(head zs)
                 (headRel -xs -ys rel1) (headRel -ys -zs rel2)]
        - [tl : R   (tail xs) (tail zs)
           = ch -(tail xs) -(tail zs)
                (Λ Y. λ f. f -(tail ys) (tailRel -xs -ys rel1) (tailRel -ys -zs rel2))]
        - intrPair hd tl))
    -xs -zs (Λ Y. λ f. f -ys rel1 rel2) .
```

Figure 11: Coinductive proofs of reflexivity, symmetry, and transitivity for `StreamRel`

reflexive; for the tail we must prove `R (tail xs) (tail ys)`, given by providing the coinductive hypothesis `ch` with a proof that `{tail xs ≃ tail ys}`.

`strSym`. To prove symmetry of `StreamRel` assuming symmetry of `Rel`, we use for the generator the type family, over streams `ys` and `xs`, of proofs that `xs` is related to `ys`. In the body of the coalgebra given to `unfoldStreamRel` we have the following obligations: for the head we must prove `Rel (head ys) (head xs)`, given by invoking the proof `sym` that `Rel` is symmetric on a proof of `Rel (head xs) (head ys)` which we extract from the head of the assumption `g: StreamRel xs ys`; for the tail we must prove `R (tail ys) (tail xs)`, given by providing the coinductive hypothesis `ch` with a proof of `StreamRel (tail xs) (tail ys)`, extracted from the tail of `g`.

`strTra`. To prove transitivity of `StreamRel` assuming transitivity of `Rel`, we use for the generator the type family (locally named `X`) over streams `xs` and `zs` of proofs that *there exists a `ys: Stream` such that `StreamRel xs ys` and `StreamRel ys zs`*. In the body of the coalgebra given to `unfoldStreamRel`, we first unpack the existential `g` to access these assumptions directly. For the head (locally named `hd`) we must prove `Rel (head xs) (head zs)`, given by invoking the proof `tra` that `Rel` is transitive on proofs that `Rel (head xs) (head ys)` and that `Rel (head ys) (head zs)`. For the tail (locally named `tl`) we must prove `R (tail xs) (tail zs)`, given by invoking the coinductive hypothesis `ch` on a proof that their exists some stream `ys'` such that `StreamRel (tail xs) ys'` and `StreamRel ys' (tail zs)`; `tail ys` is such a stream, with the required proofs extracted from the tails of the proofs that `StreamRel xs ys` and `StreamRel ys zs`.

6 Related & Future Work

Previous work on lambda encodings of datatypes in Cedille have focused on inductive datatypes. In [10], Firsov and Stump generically derive the induction principle for Church- and Mendler-style lambda encodings; this work was extended by Firsov et al. [8] to equip datatypes encoded in the Mendler-style with *constant-time* destructors, and by Firsov et al. [9] to support *course-of-values induction* for them. Our negative result (Section 3.1) appears to be a real difficulty in adapting the techniques described in [8, 10] for efficient constructors of Mendler-style coinductive types, as the type coercion enabling an efficient recursion scheme used by their (dependent version of a) Mendler-style algebra comes as a consequence of induction, which itself implies *Lambek's lemma*.

To address this difficulty, we use monotone recursive types in which the introduction and elimination forms require certain *monotonicity witnesses*. Monotone recursive types of this sort were extensively studied by Matthes [19, 20] as a way of guaranteeing strong normalization for extensions of System F that support efficient recursion schemes for inductive datatypes. These appear to be true extensions: in [29], Spławski and Urzyczyn give evidence suggesting there can be no efficiency-preserving translation of the recursion scheme for datatypes in System F. In contrast, monotone recursive types with the desired computational behavior are derivable within CDLE (shown by Jenkins and Stump in [15]), meaning no true extensions of CDLE are required (and thus no new meta-theory is needed). We can therefore define a variant Mendler-style coalgebra equipped with a zero-cost type coercion whose codomain is the type codatatype being recursively defined, enabling an efficient corecursion scheme and a constant-time codata constructor (Figure 6).

The use of recursive types for constant-time destructors for lambda encodings of inductive types goes back (at least) to Parigot [26] (see also [18, 22]). In [13], Geuvers analyzed the Church, Scott, and Parigot

(alternatively "Church-Scott") method of lambda encodings for both inductive and coinductive data using a category- and type-theoretic account developed in [11], and he similarly uses recursive types to guarantee efficient codata constructors for the Scott and Parigot encoding. In comparison to the present work, Parigot-encoded codata supports the corecursion scheme by requiring recursive occurrences of the codatatype in the type of its constructors to be embedded in a coproduct type, which requires an additional case-analysis to access. This translates to run-time overhead linear in the number of observations made on codata generated from coiteration alone. Additionally, [13] does not treat indexed and course-of-values coiteration.

The present work leans upon the account of Mendler-style (co)inductive datatypes in type theory by Mendler [24] and in category theory by Uustalu and Vene [35, 36]. The examples of functional programming with codata via different generation schemes (Section 4) come directly from Vene's thesis [37] (in which solutions were given using ordinary final F-coalgebras). In [36], Uustalu and Vene advocated for the Mendler-style approach as being a more semantic approach to termination checking of total functional programs. We agree whole-heartedly, and consider as interesting future work the design of a high-level language with copatterns using the Mendler-style approach as the basis for productivity checking, similar to the use of sized types by Abel et al. [2]. Indeed, Mendler-style recursion schemes were reported by Barthe et al. [4] as the inspiration for type-based termination checking with sized types.

7 Conclusion

In this paper, we have derived coinductive types generically using a Mendler-style of lambda encoding in CDLE, an impredicative Curry-style pure type system. Codata so derived enjoy direct support for schemes of generation that are both *expressive* and *efficient*: the *corecursion* scheme is facilitated by a constant-time type coercion between the concrete and abstract type of the codata being generated, and the *course-of-values* coiteration scheme is facilitated by a codata constructor with run-time overhead that is constant in the number of observations made. *Indexed coiteration* is also supported, which we demonstrate with examples of proofs of some standard coinductive properties of streams. We also showed of our encoding of coinductive datatypes that while the equation given by one direction of *Lambek's lemma* (which states that final F-coalgebras are isomorphisms) holds by definitional equality, showing the opposite direction (and thus, coinduction in the sense of "bisimilarity implying equality") appears to require extending CDLE with an extensional equality type.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau & Anton Setzer (2013): *Copatterns: Programming Infinite Structures by Observations*. *SIGPLAN Not.* 48(1), p. 27–38, doi:10.1145/2480359.2429075.

[2] Andreas M. Abel & Brigitte Pientka (2013): *Wellfounded Recursion with Copatterns: A Unified Approach to Termination and Productivity*. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, Association for Computing Machinery, New York, NY, USA, p. 185–196, doi:10.1145/2500365.2500591.

[3] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo & E. Moran (2006): *Innovations in computational type theory using Nuprl*. *J. Applied Logic* 4(4), pp. 428–469, doi:10.1016/j.jal.2005.10.005.

[4] Gilles Barthe, Maria João Frade, Eduardo Giménez, Luís Pinto & Tarmo Uustalu (2004): *Type-based termination of recursive definitions*. *Mathematical Structures in Computer Science* 14(1), pp. 97–141, doi:10.1017/S0960129503004122.

[5] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones & Stephanie Weirich (2016): *Safe zero-cost coercions for Haskell*. *J. Funct. Program.* 26, p. e15, doi:10.1017/S0956796816000150.

[6] C. Böhm and M. Dezani-Ciancaglini and P. Peretti and S.Ronchi Della Rocca (1979): *A discrimination algorithm inside λ-β-calculus*. *Theoretical Computer Science* 8(3), pp. 271 – 291, doi:10.1016/0304-3975(79)90014-8.

[7] Thierry Coquand & Gérard Huet (1988): *The calculus of constructions*. *Information and Computation* 76(2/3), pp. 95 – 120, doi:10.1016/0890-5401(88)90005-3.

[8] Denis Firsov, Richard Blair & Aaron Stump (2018): *Efficient Mendler-Style Lambda-Encodings in Cedille*. In Jeremy Avigad & Assia Mahboubi, editors: *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*, *Lecture Notes in Computer Science* 10895, Springer International Publishing, Cham, pp. 235–252, doi:10.1007/978-3-319-94821-8_14.

[9] Denis Firsov, Larry Diehl, Christopher Jenkins & Aaron Stump (2018): *Course-of-Value Induction in Cedille*. Available at `https://arxiv.org/abs/1811.11961`. (manuscript).

[10] Denis Firsov & Aaron Stump (2018): *Generic Derivation of Induction for Impredicative Encodings in Cedille*. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, Association for Computing Machinery, New York, NY, USA, p. 215–227, doi:10.1145/3167087. Available at `https://doi-org.proxy.lib.uiowa.edu/10.1145/3167087`.

[11] Herman Geuvers (1992): *Inductive and Coinductive types with Iteration and Recursion*. In B. Nordström, K. Pettersson & G. Plotkin, editors: *Informal Proceedings of the Workshop on Types for Proofs and Programs, TYPES '92*, Dept of Computing Science, Chalmers Univ. of Tehnology and Göteborg Univ., pp. 193–217. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.9758`.

[12] Herman Geuvers (2001): *Induction Is Not Derivable in Second Order Dependent Type Theory*. In Samson Abramsky, editor: *International Conference on Typed Lambda Calculi and Applications*, Springer, Berlin, Heidelberg, pp. 166–181, doi:10.1007/3-540-45413-6_16.

[13] Herman Geuvers (2014): *The Church-Scott representation of inductive and coinductive data*. Available at `http://citeseerx.ist.psu.edu/viewdoc/citations?doi=10.1.1.705.2662`. (manuscript).

[14] Tatsuya Hagino (1987): *A categorical programming language*. Ph.D. thesis, The University of Edinburgh.

[15] Christopher Jenkins & Aaron Stump (2020): *Monotone recursive types and recursive data representations in Cedille*. Available at `https://arxiv.org/abs/2001.02828`. (manuscript).

[16] Alexei Kopylov (2003): *Dependent Intersection: A New Way of Defining Records in Type Theory*. In: *Proceedings of 18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada*, LICS '03, IEEE Computer Society, pp. 86–95, doi:10.1109/LICS.2003.1210048.

[17] Joachim Lambek (1968): *A Fixpoint Theorem for Complete Categories*. *Mathematische Zeitschrift* 103(2), pp. 151–161, doi:10.1007/bf01110627.

[18] Daniel Leivant (1989): *Contracting proofs to programs*. In P. Odifreddi, editor: *Logic and Computer Science*, pp. 279–327, doi:10.1184/R1/6604463.v1.

[19] Ralph Matthes (1998): *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. Ph.D. thesis, Ludwig-Maximilians-Universität München.

[20] Ralph Matthes (1998): *Monotone Fixed-Point Types and Strong Normalization*. In Georg Gottlob, Etienne Grandjean & Katrin Seyr, editors: *Computer Science Logic, 12th International Workshop, CSL '98, Annual Conference of the EACSL, Brno, Czech Republic, August 24-28, 1998, Proceedings*, *Lecture Notes in Computer Science* 1584, Springer, pp. 298–312, doi:10.1007/10703163_20.

[21] Ralph Matthes (1999): *Monotone (co)inductive types and positive fixed-point types*. *RAIRO - Theoretical Informatics and Applications* 33(4-5), p. 309–328, doi:10.1051/ita:1999120.

[22] N. P. Mendler (1987): *Recursive Types and Type Constraints in Second-Order Lambda Calculus*. In: *Proceedings of the Symposium on Logic in Computer Science*, (LICS '87), IEEE Computer Society, Los Alamitos, CA, pp. 30–36.

[23] N. P. Mendler (1991): *Predictive type universes and primitive recursion*. In: *Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, LICS '91, pp. 173–184, doi:10.1109/LICS.1991.151642.

[24] Nax Paul Mendler (1991): *Inductive types and type constraints in the second-order lambda calculus*. *Annals of Pure and Applied Logic* 51(1), pp. 159 – 172, doi:10.1016/0168-0072(91)90069-X.

[25] 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, p. 344–359, doi:10.1007/3-540-45413-6_27. Available at `http://dl.acm.org/citation.cfm?id=1754621.1754650`.

[26] Michel Parigot (1988): *Programming with Proofs: A Second Order Type Theory*. In Harald Ganzinger, editor: *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, *Lecture Notes in Computer Science* 300, Springer, Berlin, Heidelberg, pp. 145–159, doi:10.1007/3-540-19027-9_10.

[27] Michel Parigot (1989): *On the Representation of Data in Lambda-Calculus*. In Egon Börger, Hans Kleine Büning & Michael M. Richter, editors: *CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany, October 2-6, 1989, Proceedings*, *Lecture Notes in Computer Science* 440, Springer, Berlin, Heidelberg, pp. 309–321, doi:10.1007/3-540-52753-2_47.

[28] Andrew M. Pitts (1998): *Existential Types: Logical Relations and Operational Equivalence*. In Kim Guldstrand Larsen, Sven Skyum & Glynn Winskel, editors: *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, *Lecture Notes in Computer Science* 1443, Springer, pp. 309–326, doi:10.1007/BFb0055063.

[29] Zdzislaw Splawski & Pawel Urzyczyn (1999): *Type Fixpoints: Iteration vs. Recursion*. In Didier Rémy & Peter Lee, editors: *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*, ACM, pp. 102–113, doi:10.1145/317636.317789.

[30] Aaron Stump (2017): *The calculus of dependent lambda eliminations*. *J. Funct. Program.* 27, p. e14, doi:10.1017/S0956796817000053.

[31] Aaron Stump (2018): *From realizability to induction via dependent intersection*. *Ann. Pure Appl. Logic* 169(7), pp. 637–655, doi:10.1016/j.apal.2018.03.002.

[32] Aaron Stump (2018): *Syntax and Semantics of Cedille*. Available at `http://arxiv.org/abs/1806.04709`.

[33] Aaron Stump (2018): *Syntax and Typing for Cedille Core*. Available at `http://arxiv.org/abs/1811.01318`.

[34] Alfred Tarski (1955): *A lattice-theoretical fixpoint theorem and its applications*. *Pacific Journal of Mathematics* 5(2), pp. 285–309, doi:10.2140/pjm.1955.5.285.

[35] Tarmo Uustalu & Varmo Vene (1999): *Mendler-style Inductive Types, Categorically*. *Nordic J. of Computing* 6(3), pp. 343–361. Available at `http://dl.acm.org/citation.cfm?id=774455.774462`.

[36] Tarmo Uustalu & Varmo Vene (2000): *Coding Recursion a la Mendler (Extended Abstract)*. In: *Proc. of the 2nd Workshop on Generic Programming, WGP 2000, Technical Report UU-CS-2000-19*, Dept. of Computer Science, Utrecht University, pp. 69–85.

[37] Varmo Vene (2000): *Categorical programming with inductive and coinductive types*. Ph.D. thesis, University of Tartu.

[38] Philip Wadler (1990): *Recursive Types for Free!* Available at `https://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`. Unpublished.

[39] Benjamin Werner (1994): *Une théorie des constructions inductives*. Ph.D. thesis, Université Paris-Diderot.

A CDLE: additional term constructors

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : [t_1/x]T_2 \quad |t_1| = |t_2|}{\Gamma \vdash [t_1, t_2] : \iota\, x{:}T_1.\, T_2} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.1 : T_1} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.2 : [t.1/x]T_2}$$

$$\frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t_1 : T \quad FV(t_2) \subseteq dom(\Gamma)}{\Gamma \vdash \varphi\ t\ -\ t_1\ \{t_2\} : T}$$

$$\begin{aligned} |[t_1, t_2]| &= |t_1| \\ |t.1| &= |t| \\ |t.2| &= |t| \\ |\varphi\ t\ -\ t_1\ \{t_2\}| &= |t_2| \end{aligned}$$

Figure 12: Dependent intersections and φ

The code listings in Appendix B detailing the derivation of type coercions and monotone recursive rely on additional term constructs for dependent intersections and an additional eliminator φ for the equality type.

Dependent intersection $\iota\, x{:}T_1.\, T_2$ is the type of terms t which can be assigned both type T and $[t/x]T'$. In the annotated language, the introduction form is $[t_1, t_2]$, where t_1 has type T_1 and t_2 has type $T_2[t_1/x]$. This is conceptually similar to the introduction rule of a dependent pair, except that $|t_1|$ is additionally required to be definitionally equal to t_2. This allows the erasure of $[t_1, t_2]$ to be simply $|t_1|$.

Dependent intersections are eliminated with projections $t.1$ and $t.2$: if t has type $\iota\, x{:}T_1.\, T_2$, then $t.1$ has type T and erases to $|t|$ and $t.2$ has type $T_2[t.1/x]$ and also erases to $|t|$. This can be seen choosing to "view" t as either having type T_1 or $T_2[t.1/x]$.

Type coercions by equalities The expression $\varphi\ t\ -\ t_1\ \{t_2\}$ (erasing to $|t_2|$) has type T if t_1 has type T and t proves that t_1 is equal to t_2. This is similar to the direct computation rule of NuPRL (see Section 2.2 of Allen et al. [3]).

B Derived type constructors

B.1 Cast

For an indexing type `I: ⋆` (given as a module parameter) and type families `A: I → ⋆` and `B: I → ⋆`, type `Cast ·A ·B` is defined as the dependent intersection type of functions `f: ∀ i: I. A i → B i` and proofs that `f` is equal to the identity function.

`intrCast` takes an argument `f: ∀ i: I. A i → B i` and a proof that `f` behaves extensionally like an identity function (`∀ i: I. Π a: A i. {f a ≃ a}`). In the body of its definition, we introduce a dependent intersection. For the first argument, we assume `a: A i` and use φ to cast `a` to the type `B i` of the expression `f -i a` with the equality `eq -i a` proving these two expressions are equal. By the

```
module utils/cast (I: ⋆).

Cast : (I → ⋆) → (I → ⋆) → ⋆
= λ A: I → ⋆. λ B: I → ⋆. ι f: ∀ i: I. A i → B i. {f ≃ λ x. x}.

intrCast
: ∀ A: I → ⋆. ∀ B: I → ⋆.
  ∀ f: ∀ i: I. A i → B i. (∀ i: I. Π a: A i. {f -i a ≃ a}) ⇒ Cast ·A ·B
= Λ A. Λ B. Λ f. Λ eq. [ Λ i. λ a. φ (eq -i a) - (f -i a) {a} , β].

elimCast : ∀ A: I → ⋆. ∀ B: I → ⋆. Cast ·A ·B ⇒ ∀ i: I. A i → B i
= Λ A. Λ B. Λ c. φ c.2 - c.1 {λ x. x}.

-- underscore for anonymous definitions
_ : { intrCast ≃ λ x. x } = β .
_ : { elimCast ≃ λ x. x } = β .

castRefl : ∀ A: I → ⋆. Cast ·A ·A
= Λ A. intrCast -(Λ _. λ x. x) -(Λ _. λ _. β).

castTrans
: ∀ A: I → ⋆. ∀ B: I → ⋆. ∀ C: I → ⋆. Cast ·A ·B ⇒ Cast ·B ·C ⇒ Cast ·A ·C
= Λ A. Λ B. Λ C. Λ c1. Λ c2.
  intrCast -(Λ i. λ a. elimCast -c2 -i (elimCast -c1 -i a)) -(Λ _. λ _. β).

_ : { castRefl ≃ λ x. x } = β .
_ : { castTrans ≃ λ x. x } = β .

Mono : ((I → ⋆) → I → ⋆) → ⋆
= λ F: (I → ⋆) → I → ⋆. ∀ A: I → ⋆. ∀ B: I → ⋆. Cast ·A ·B ⇒ Cast ·(F ·A) ·(F ·B).

intrMono
: ∀ F: (I → ⋆) → I → ⋆.
  (∀ A: I → ⋆. ∀ B: I → ⋆. Cast ·A ·B ⇒ Cast ·(F ·A) ·(F ·B)) ⇒ Mono ·F
= Λ F. Λ m. Λ A. Λ B. Λ c. intrCast -(elimCast -(m -c)) -(Λ i. λ a. β) .

elimMono : ∀ F: (I → ⋆) → I → ⋆. ∀ A: I → ⋆. ∀ B: I → ⋆.
  Mono ·F ⇒ Cast ·A ·B ⇒ ∀ i: I. F ·A i → F ·B i
= Λ F. Λ A. Λ B. Λ cm. Λ c. Λ i. λ f. elimCast -(cm -c) -i f.

_ : { intrMono ≃ λ x. x } = β .
_ : { elimMono ≃ λ x. x } = β .
```

Figure 13: Type coercions and monotonicity witnesses (`utils/cast.ced`)

erasure of φ, the resulting expression erases to λ `a. a`, so the proof that this is equal to the identity function (needed for the second component of the dependent intersection) is trivial, given by β. As β also erases to λ `x. x`, the two components of the intersection are definitionally equal, as required.

`elimCast` takes an erased cast argument of type `Cast ·A ·B` and produces a function of type $\forall$ `i: I. A i → B i`. Its definition simply uses φ to cast λ `x. x` to the type of `c.1` by the proof `c.2` that `{ c.1 ≃ λ x. x }`. This means that `elimCast` itself erases to λ `x. x`.

`castRefl` **and** `castTrans` are straight-forward: there is always a cast between two definitionally equal types, and the composition of identity functions produces an identity function.

B.2 Mono

For an indexed type scheme `F: (I → ⋆) → I → ⋆`, we define `Mono ·F` to be the property that any type coercion between indexed type families `A: I → ⋆` and `B: I → ⋆` can be lifted to a coercion between the type families `F ·A` and `F ·B`.

`intrMono` is defined the way it is to make the axiomatic presentation given in Figure 2b intelligible. Its type signature is equal to the type:

`∀ F: (I → ⋆) → I → ⋆. Mono ·F ⇒ Mono ·F`

So, `intrMono` takes an erased proof `cm` that there is an `F`-lifting for any type coercion, and an erased type coercion `c: Cast ·A ·B`, and constructs a new cast from family `F ·A` to family `F ·B`. As all arguments are erased and the definition uses `intrCast`, `intrMono` is definitionally equal to λ `x. x`.

`elimMono` takes an erased proof `cm: Mono ·F` and an erased `c: Cast ·A ·B` and produces a function of type $\forall$ `i: I. F ·A i → F ·B i` by eliminating the cast `cm -c` of type `Cast ·(F ·A) ·(F ·B)` on the assumed `f: F ·A i`. As the definition uses `elimCast`, `elimMono` is definitionally equal to λ `f. f`.

B.3 `Rec`

The definition of `Rec`, the recursive type former, is described in [15] in detail and follows the proof of Tarski's least fixed-point theorem for monotone functions over a complete lattice [34]. Interpreting impredicative quantification as set intersection, read the definition of `Rec` as the intersection over all type families `X: I → ⋆` of all types `X i` such that there exists a type coercion from families `F ·X` to `X`.

`recFold` takes an erased cast `c` from `F ·X` to `X` and produces a cast from `Rec` to `X` by simply giving `c` to the assumed `x: Rec` as an erased argument.

`recIn` assumes that `F` is monotonic and produces a cast from families `F ·Rec` to `Rec`. The function `f` implementing this cast assumes `xs: F ·Rec i` and to produce an expression of type `Rec` assumes `c: Cast ·(F ·X) ·X`, and first coerces the type of `xs` to `F ·X i` (by using `recFold` and the monotonicity witness to lift this to a type coercion from `F ·Rec i` to `F ·X i`), then uses `c` to coerce this to type `X i`. `x`

```
module utils/rec (I: ⋆) (F: (I → ⋆) → I → ⋆).

import top.
import cast ·I.

Rec : I → ⋆
= λ i: I. ∀ X: I → ⋆. Cast ·(F ·X) ·X ⇒ X i.

recFold : ∀ X: I → ⋆. Cast ·(F ·X) ·X ⇒ Cast ·Rec ·X
= Λ X. Λ c. intrCast -(Λ i. λ x. x -c) -(Λ i. λ x. β).

recIn : Mono ·F ⇒ Cast ·(F ·Rec) ·Rec
= Λ im.
  [f : ∀ i: I. F ·Rec i → Rec i
    = Λ i. λ xs. Λ X. Λ c.
      elimCast -c -i (elimMono -im -(recFold -c) -i xs)]
- intrCast -f -(Λ i. λ xs. β) .

recOut : Mono ·F ⇒ Cast ·Rec ·(F ·Rec)
= Λ im.
  [f : ∀ i: I. Rec i → F ·Rec i
    = Λ i. λ x. x ·(F ·Rec) -(im -(recIn -im))]
- intrCast -f -(Λ i. λ x. β) .

roll : Mono ·F ⇒ ∀ i: I. F ·Rec i → Rec i
= Λ im. elimCast -(recIn -im) .

unroll : Mono ·F ⇒ ∀ i: I. Rec i → F ·Rec i
= Λ im. elimCast -(recOut -im).

_ : {roll   ≃ λ x. x} = β.
_ : {unroll ≃ λ x. x} = β.
```

Figure 14: Monotone recursive types (`utils/rec.ced`)

`recOut` assumes that `F` is monotonic and produces a cast from families `Rec` to `F ·Rec`. The function `f` implementing this cast assumes `x: Rec i` and gives this a type coercion from families `F ·(F ·Rec)` to `F ·Rec` by combining `recIn` with the assumption of monotonicity.

`roll` **and** `unroll` are simply defined by eliminating the casts `recIn` and `recOut`, resp., so it is immediate that these both erased to `λ x. x`.