Elaborating course-of-values induction in Cedille

Christopher Jenkins[1][0000−0002−5434−5018], Denis Firsov[2][0000−0003−1267−7898], Larry Diehl[3], Colin McDonald[1], and Aaron Stump[1][0000−0002−9720−0003]

[1] The University of Iowa, Iowa City IA 52246, USA
{christopher-jenkins,aaron-stump,colin-mcdonald}@uiowa.edu
[2] Talinn University of Technology, Talinn, Estonia denis.firsov@taltech.ee
[3] Symbiont.io, Inc., New York NY 10012, USA larry.diehl@symbiont.io

Abstract. We present a datatype subsystem for Cedille, a surface language for the *calculus of dependent lambda eliminations* (CDLE). This subsystem supports *course-of-values induction*, an expressive proof scheme in which the inductive hypothesis may be invoked on predecessors of arbitrary depth, using type-based termination checking. We justify our surface language constructs using elaborating type inference rules, translating them to pure lambda expressions in CDLE, and show this translation is type and value preserving. This datatype subsystem and elaborator are implemented in Cedille, establishing for the first time a full translation of inductive types to a small pure typed lambda calculus.

Keywords: lambda encodings · course-of-values · datatypes · Cedille

1 Introduction

Algebraic datatypes (ADTs) are a popular feature of functional languages that combine a concise scheme for declaring datatypes and a mechanism for defining functions over them by pattern matching and recursion. This popularity extends to implementations of proof assistants based on dependent type theories, wherein proofs are given using these same mechanisms. However, wrinkles in bringing ADTs to proof assistants are the issues of *termination checking* and *positivity checking*, as non-well-founded recursion and mixed variant datatypes can lead to non-termination and logical unsoundness.

Approaches to both termination and positivity checking may be classified as either *syntactic* or *semantic*. For termination, syntactic analysis can be used to determine whether some expression is a legal argument to the recursively defined function, such as requiring it to be a pattern variable [15]. For positivity checking, the syntactic approach tracks in the types of constructor arguments the number of arrows of which recursive occurrences of a datatype appears to the left (c.f. [18, Section 4.5.2]). Such analyses must usually be implemented in the prover's core theory, running afoul of the *de Bruijn criterion* [14], i.e., that the prover produces proof objects checkable by an implementation of a *small* kernel theory. This situation is especially unfortunate for termination checking, as simple syntactic guards are brittle, incentivising implementors to make analyses more sophisticated to grow the set of acceptable definitions.

Semantic approaches address such concerns. For positivity checking, one might translate datatypes down into a closed universe of strictly positive types [7] or generate *monotonicity witnesses* [24] in the kernel theory. For termination there is *type-based* termination checking, the most popular formulation arguably being *sized types* [1, 5], in which structural decrease of a datatype in recursive calls is enforced by requiring decrease in the size index. Many expressive recursion schemes, like *course-of-values iteration* in which recursive calls can be made on predecessor values of arbitrary depth, can easily be expressed using sized types. However, adding this feature to a type theory requires rework of existing meta-theoretic results, and in languages with both sized and unsized variants of datatypes ergonomic reuse between the variants becomes an issue.

Semantic approaches to positivity and termination in CDLE The setting of the present work is Cedille, a dependently typed programming language. Cedille's kernel theory, CDLE (the *calculus of dependent lambda eliminations*), is a compact pure extrinsic type theory with no primitive notion of datatypes and which can be implemented in ∼1K Haskell LoC [32]. Firsov et al. [11] showed that it is possible to generically derive lambda encodings of datatypes and their induction principles using a Mendler encoding that features constant-time predecessors and linear-space representation. Their development uses a derived notion of positivity similar to Matthes's monotonicity witnesses [23], and the Mendler encoding naturally lends itself to the Mendler style of coding recursive functions [36], a type-based form of termination checking that uses only polymorphic typing.

The result of Firsov et al. [11] demonstrates that the foundations for semantic positivity and termination checking are present in CDLE already. However, it does not address how these foundations might be used in the design of a convenient surface language for ADTs in Cedille. Additionally, the induction scheme their encoding directly supports allows recursive calls made only on immediate predecessors. Can this be extended to richer schemes, such as course-of-values induction, and if so could the extension be supported in the surface language?

Contributions This paper[4] answers these questions affirmatively. We present a datatype subsystem for Cedille with the expected convenience of compact notation for declaring inductive types and with type-based termination checking that supports course-of-values induction. In particular, we:

- extend Firsov et al. [11] with a generic encoding of course-of-values datatypes, deriving implicitly restricted existentials and singleton types to achieve this;
- design a surface language for inductive types with type-based termination checking based on *course-of-values pattern matching*, a feature allowing recursive calls on arbitrarily deep predecessors;
- elaborate datatypes, positivity checking, recursive functions, and course-of-values pattern matching to CDLE;
- and prove that elaboration is type- and value-preserving

[4] This paper is a combination of two previously unpublished manuscripts [12, 19].

The datatype system and elaborator are implemented in Cedille[5], demonstrating that inductive definitions in constructive type theory can be soundly translated down to a small pure typed lambda calculus. This paper and its proof appendix treats only the elaboration of non-indexed, non-parameterized datatypes.

Organization In Section 2, we informally describe course-of-values pattern matching using an extended example of both implementing division and proving a certain property concerning it. We review CDLE and Firsov et al. [11] in Section 3. Section 4 gives the derivation of course-of-values datatypes and characterizes their computational behavior. Section 5 treats formally the surface language constructs for datatypes in Cedille by elaborating type inference rules states type- and value-preservation and the normalization guarantee for elaborations of datatype expressions. Finally, Section 6 discusses related work and Section 7 concludes the paper and discusses future work.

2 Course-of-values pattern matching

We explain course-of-values pattern matching, the key feature of Cedille's type-based termination checker enabling course-of-values induction, with a demonstration of its use to both implement division and to prove that the quotient is no greater than the dividend. This also establishes the expressive power of course-of-values induction: our motivation for choosing division is that its natural implementation as iterated subtraction is neither obviously well-founded by syntactic analysis, nor can it be simulated by nested pattern matching as the predecessor at each recursive step is dynamically computed.

2.1 Implementing division

Figure 1 lists an implementation of division in Cedille. The declaration of the datatype Nat is straightforward, following a similar format as for declaring GADTs in Haskell. In addition to bringing into scope the type Nat and terms zero and succ, this declaration also exports the following automatically generated names that are used for termination checking:

- $\mathsf{Is/Nat} : \star \to \star$, a predicate on types that, when true of T, allows terms of type T to be case analyzed with predecessors preserving the type T;
- $\mathsf{to/Nat} : \forall\, R : \star.\, \mathsf{Is/Nat} \cdot R \Rightarrow R \to \mathsf{Nat}$, a type coercion that converts any term of type T for which $\mathsf{Is/Nat}$ holds into a term of type Nat (the center dot denotes application to a type and the open arrow forms the type of functions whose arguments are computationally irrelevant, discussed in Section 3);
- $\mathsf{is/Nat} : \mathsf{Is/Nat} \cdot \mathsf{Nat}$, a proof that Nat itself satisfies the predicate $\mathsf{Is/Nat}$.

The first function of the figure, predCV, implements the predecessor operation for naturals and shows how $\mathsf{Is/Nat}$ is used with the term construct σ for course-of-values pattern matching. Given proof *is* that the predicate holds for the type

[5] https://github.com/cedille/cedille

```
data Nat : ⋆ = zero : Nat | succ : Nat → Nat .

predCV: ∀ N: ⋆. Is/Nat ·N ⇒ N → N = Λ N. Λ is. λ n.
  σ<is> n {zero → n | succ n' → n'}.

minusCV: ∀ N: ⋆. Is/Nat ·N ⇒ N → Nat → N = Λ N. Λ is. λ m. λ n.
  μ rec. n { zero → m | succ n' → predCV -is (rec n') }.

pred = predCV -is/Nat .
minus = minusCV -is/Nat .

data Bool: ⋆ = tt : Bool | ff : Bool .

ite: ∀ X: ⋆. Bool → X → X → X
= Λ X. λ b. λ t. λ f. σ b {tt → t | ff → f}.

lt : Nat → Nat → Bool = λ m. μ lt. (succ m) {
  | zero → λ n. tt
  | succ m → λ n. σ n { zero → ff | succ n → lt m n} } .

divide: Nat → Nat → Nat = λ n. λ d. μ divD. n {
| zero → zero
| succ p → [p' = to/Nat -isType/divD p] -
  ite (lt (succ p') d) zero
    (succ (divD (minusCV -isType/divD p (pred d)))) } .
```

Fig. 1. Division in Cedille using course-of-values pattern matching

argument N, the argument n is analyzed: in the case it is zero, n is returned, and if it is the successor to some n' then n' is returned. In both cases, the branch bodies have type N as required.

With a parametric reading, the type of predCV suggests the only possible way to return an expression of type N is by either giving back the first argument or some predecessor of it. This reading also applies to the type of minusCV. Subtraction is defined by recursion, using the construct μ, over its second argument and iteratively applies the type-preserving predCV to its first argument. Ordinary predecessor and subtraction are special cases where the proof is is/Nat (the hyphen indicates that is/Nat is an irrelevant argument to predCV and minusCV).

Analogously to datatype declarations, μ expressions export names within the scope of the given case tree. In minusCV, these are:

- Type/rec:$\star$, the type of predecessors of the scrutinee n;
- isType/rec : Is/Nat · Type/rec, evidence permitting the coercion to Nat, and course-of-values pattern matching, for terms of type Type/rec; and
- rec: Type/rec $\to N$, the handle for recursion restricted to predecessors of n

The last definitions needed before implementing division is the declaration of booleans, Bool), their conditional *if-then-else* function, ite, and the natural

```
Lte : Nat → Nat → ⋆ = λ m: Nat. λ n: Nat. { lt m (succ n) ≃ tt } .

lteTrans : Π l: Nat. Π m: Nat. Π n: Nat.
  Lte l m → Lte m n → Lte l n = <..>

lteMinus : Π m: Nat. Π n: Nat. Lte (minus m n) m = <..>

lteDivide : Π n: Nat. Π d: Nat. Lte (divide n d) n = λ n. λ d.
  μ ih. n {
  | zero → β
  | succ p →
    [p' = to/Nat -isType/ih p] -
    [s = minusCV -isType/ih p (pred d)] - [s' = to/Nat -isType/ih s] -
      σ (lt (succ p') d)
      @(λ x: Bool. Lte (ite x zero (succ (divide s' d))) (succ p')) {
      | ff → lteTrans (divide s' d) s' p' (ih s) (lteMinus p' (pred d))
      | tt → β } } .
```

Fig. 2. Proof of $n/d \leq n$

number less-than comparator, lt. Note that in Cedille, the global witness is/Bool can be inferred and does not need to be given explicitly to σ expressions.

For divide, we have dividend n and divisor d, and define with a μ expression the recursive function divD over n dividing its argument by d. In the successor case, the pattern variable p has type Type/divD, so to compare its successor with d we make a local definition p' coercing it to type Nat with to/Nat and the automatically available witness isType/divD of Is/Nat · Type/divD. This local definition of p' is given with the syntax $[x = t_1] - t_2$, Cedille's notation let-bindings (as in let $x = t_1$ in t_2).

When it is not the case that succ $p' < d$, we make essential use of type-preserving subtraction with minusCV to make a recursive call on the difference between p and pred d. This is course-of-values iteration: the expression minusCV -isType/divD p (pred d) dynamically computes a predecessor of p. Though it is not obviously smaller than p using syntactic analysis, it has type Type/divD, the domain of divD, making it a legal argument for recursion.

2.2 Proof concerning division

Figure 2 shows how course-of-values *induction* in Cedille is used to prove that the quotient of division is no greater than the dividend. It begins by defining the relation Lte using the comparator lt and Cedille's primitive equality type $\{t_1 \simeq t_2\}$ (see Section 3). The next definitions, lteTrans and lteMinus, are lemmas whose definitions are omitted (indicated by <..>) proving resp. that Lte is transitive and that $m - n \leq m$ for all naturals m and n.

In lteDivide, we inductively define over n a proof that $n/d \leq n$. In the case that n is zero, the goal becomes $\{\mathsf{tt} \simeq \mathsf{tt}\}$, which is proven by the introduction

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

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

$$\frac{FV(t) \cup FV(t') \subseteq dom(\Gamma)}{\Gamma \vdash \beta\langle t\rangle\{t'\} : \{t \simeq t\}} \qquad \begin{array}{rr} |\beta\langle t\rangle\{t'\}| = |t'|, & |[t_1, t_2]| = |t_1|, \\ |t.1| = |t|, & |t.2| = |t|, \\ |\Lambda\, x{:}T.\, t| = |t|, & |t\ \text{-}t'| = |t| \end{array}$$

Fig. 3. Typing and erasure for a fragment of CDLE

form β for the equality type. In the successor case, we locally define p' by coercing the predecessor, as we did in divide itself, but now also make two definitions for the difference $p - (\mathsf{pred}\ d)$. The first of these, s, has type Type/ih, and the second, s', has type Nat. The σ expression following this inspects the result of comparing $\mathsf{succ}\ p'$ with d. A type annotation for inferring the types of the bodies of case branches is given explicitly with @, as these types depend upon the result of the comparison — the type of the entire σ expression is

$$\mathsf{Lte}\ (\mathsf{ite}\ (\mathsf{lt}\ (\mathsf{succ}\ p')\ d)\ \mathsf{zero}\ (\mathsf{divide}\ s'\ d))\ (\mathsf{succ}\ p')$$

which is definitionally equal to the expected type $\mathsf{Lte}\ (\mathsf{divide}\ (\mathsf{succ}\ p')\ d)\ (\mathsf{succ}\ p')$ (strictly speaking, it is not p' but its definition that occurs in the expected type; inference for the insertion of coercions is given later in Figure 12).

In the branch for ff, we use transitivity of $\leq$ to combine a proof of $s'/d \leq s'$ with a proof of $s' \leq p'$ to obtain the desired result, and the second of these proofs is given by the lemma IteMinus. Course-of-values induction is used for the first: the difference s has type Type/ih, and ih s proves that $s'/d \leq d$, as required.

3 Background

We now review CDLE, Cedille's kernel theory. CDLE extends the impredicative extrinsically typed *calculus of constructions* (CC), overcoming historical difficulties of lambda encodings (e.g., underivability of induction [13]) by adding three new type constructs: equality of untyped terms; the dependent intersections of Kopylov [21]; and the implicit products of Miquel [27]. The pure term language of CDLE is untyped lambda calculus, but to make type checking algorithmic terms are presented with typing annotations. Definitional equality of terms t_1 and t_2 is $\beta\eta$-equivalence modulo erasure of annotations, denoted $|t_1| =_{\beta\eta} |t_2|$. The type and erasure rules for the fragment of CDLE used in this paper are given in Figure 3, with a full listing given in the proof appendix (see also Stump [31]).

Equality: $\{t_1 \simeq t_2\}$ is the type of proofs that t_1 and t_2 are equal. It is introduced with $\beta\langle t\rangle\{t'\}$, erasing to the erasure of an unrelated t' and proving $\{t \simeq t\}$ for any term t whose free variables are declared in the typing context. Combined with definitional equality, $\beta\langle t\rangle\{t'\}$ proves $\{t_1 \simeq t_2\}$ for any t_1 and t_2 whose erasures are $\beta\eta$-convertible with the erasure of t. CDLE has inference rules also for *using* equality proofs, not only introducing them, but these are omitted.

By having $\beta\langle t\rangle\{t'\}$ to erase to $|t'|$, we effectively add a top type to the language as every well-scoped untyped lambda term proves equations that hold definitionally. This idea, which we call the *Kleene trick*, can be found in Kleene's later definitions of numeric realizability [20]. In code listings, we omit $\langle t\rangle$ when t can be inferred; when $\{t'\}$ is omitted, the erasure is $\lambda\, x.\, x$.

Dependent intersection: $\iota\, x : T_1.\, T_2$ is the type of terms t which can be assigned both type T_1 and $T_2[t/x]$, and in the annotated language is introduced by $[t_1, t_2]$, where t_1 has type T_1, t_2 has type $T_2[t_1/x]$, 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_1 or $T_2[t.1/x]$. In this paper, dependent intersections occur explicitly only in the elaboration of datatype signatures (Section 5.2).

Implicit product: $\forall\, x : T.\, T'$ is the type of dependent functions taking an erased argument t of type T and returning a result of type $T'[t/x]$. 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. When x does not occur free in T' we write $T \Rightarrow T'$, similar to writing $T \to T'$ for $\Pi\, x : T.\, T'$ under the same condition.

Omitted typing constructs Figure 3 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 type arguments in instantiations of polymorphic functions (written $t \cdot S$) are 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 in terms when these can be inferred.

Meta-theoretic results for CDLE We list two of CDLE's meta-theoretic results, *logical consistency* and *call-by-name normalization of closed functions.*

Proposition 1 (Stump [31]). *There t such that $\emptyset \vdash t : \forall\, X : \star.\, X \to X$.*

Proposition 2 (Stump [31]). *Suppose $\Gamma \vdash t : T$, t is closed, and there exists a closed t' that erases to $\lambda\, x.\, x$ and whose type is $T \to \Pi\, x : T_1.\, T_2$ for some T_1, T_2. Then $|t|$ is call-by-name normalizing.*

In Theorem 2, the condition on T that there be a *retyping function* (*á la* Mitchel [28]) to a function excludes non-termination introduced by the Kleene trick; requiring t be closed concerns the derivability of *zero-cost coercions* (Cast, Figure 4), which can cause non-termination in inconsistent contexts [2].

$$\frac{\Gamma \vdash S : \star \quad \Gamma \vdash T : \star}{\Gamma \vdash \mathsf{Cast} \cdot S \cdot T : \star} \qquad \frac{\Gamma \vdash t : \mathsf{Cast} \cdot S \cdot T}{\Gamma \vdash \mathsf{elimCast}\ \text{-}t : S \to T}$$

$$\frac{\Gamma \vdash t_1 : S \to T \quad \Gamma \vdash t_2 : \Pi\, x : S.\, \{f\ x \simeq x\}}{\Gamma \vdash \mathsf{intrCast}\ \text{-}t_1\ \text{-}t_2 : \mathsf{Cast} \cdot S \cdot T}$$

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

$$\frac{\Gamma \vdash F : \star \to \star}{\Gamma \vdash \mathsf{Mono} \cdot F : \star} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Mono} \cdot F \quad \Gamma \vdash t_2 : \mathsf{Cast} \cdot S \cdot T}{\Gamma \vdash \mathsf{elimMono}\ \text{-}t_1\ \text{-}t_2 : F \cdot S \to F \cdot T}$$

$$\frac{\Gamma \vdash t : \forall\, X : \star.\, \forall\, Y : \star.\, \mathsf{Cast} \cdot X \cdot Y \Rightarrow \mathsf{Cast} \cdot (F \cdot X) \cdot (F \cdot Y)}{\Gamma \vdash \mathsf{intrMono}\ \text{-}t : \mathsf{Mono} \cdot F}$$

$$|\mathsf{elimCast}\ \text{-}t| = \lambda\, x.\, x, \quad |\mathsf{elimMono}\ \text{-}t_1\ \text{-}t_2| = \lambda\, x.\, x$$

Fig. 4. Type inclusions and positivity (resp. Cast and Mono)

3.1 Generic encoding of datatypes

Firsov et al. [11] provide a generic development for efficient encodings of datatypes and their induction principles. Genericity here means *parametricity*: the development works for an arbitrary positive type scheme $F : \star \to \star$, with positivity expressed internally as a predicate Mono; efficiency means both that data destructors compute in constant time and lambda encodings have size linear in the size of the data they encode. For brevity, we present axiomatically definitions of their framework that we use with formation, introduction, and elimination rules, and (when relevant) erasure and computation rules. We stress that all typing constructs discussed are *definitional* extensions to CDLE, and the laws of each type construct can be proven internally for the underlying definition.

Type inclusions and positivity Figure 4 gives the rules associated to type inclusions and monotonicity witnesses (in Firsov et al., resp. Id and IdMapping).

Type inclusions For types S and T, $\mathsf{Cast} \cdot S \cdot T$ is the proposition that S is *included* into (or a *subtype* of) T, i.e., every term of type S has type T. A proof t of $\mathsf{Cast} \cdot S \cdot T$ is used with the elimination form elimCast -t, which has type $S \to T$ and is definitionally equal to $\lambda\, x.\, x$. The hyphen in the expression marks t as an *erased* argument: if S is included in T and t has type S, then t also has type T regardless of the choice of witness of the inclusion, giving Cast a form of proof-irrelevance. To introduce $\mathsf{Cast} \cdot S \cdot T$, intrCast requires a function $t_1 : S \to T$ and a proof t_2 that t_1 behaves extensionally as the identity function on terms of type S. Finally, we list operations castRefl and castTrans showing resp. that type inclusions are reflexive and transitive.

$$\frac{\Gamma \vdash F : \star \to \star}{\Gamma \vdash \mu F : \star} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Mono} \cdot F \quad \Gamma \vdash t_2 : F \cdot \mu F}{\Gamma \vdash \mathsf{in}\ \text{-}t_1\ t_2 : \mu F} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Mono} \cdot F \quad \Gamma \vdash t_2 : \mu F}{\Gamma \vdash \mathsf{out}\ \text{-}t_1\ t_2 : F \cdot \mu F}$$

$$\begin{aligned} &\mathsf{PrfAlg} : \Pi\, F : \star \to \star.\, \mathsf{Mono} \cdot F \to (\mu F \to \star) \to \star \\ &\mathsf{PrfAlg} \cdot F\ t \cdot P = \forall\, R : \star.\, \forall\, c : \mathsf{Cast} \cdot R \cdot \mu F.\, (\Pi\, x : R.\, P\ (\mathsf{elimCast}\ \text{-}c\ x)) \\ &\qquad\qquad \to \Pi\, xs : F \cdot R.\, P\ (\mathsf{in}\ \text{-}t\ (\mathsf{elimMono}\ \text{-}t\ \text{-}c\ xs)) \end{aligned}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Mono} \cdot F \quad \Gamma \vdash t_2 : \mathsf{PrfAlg} \cdot F\ t_1 \cdot P}{\Gamma \vdash \mathsf{ind}\ \text{-}t_1\ t_2 : \Pi\, x : \mu F.\, P\ x} \qquad \begin{aligned} |\mathsf{out}\ (\mathsf{in}\ t)| &=_{\beta\eta} |t| \\ |\mathsf{ind}\ t_1\ (\mathsf{in}\ t_2)| &=_{\beta\eta} |t_1\ (\mathsf{ind}\ t_1)\ t_2| \end{aligned}$$

Fig. 5. Interface for encodings of inductive types provided by [11]

Monotonicity Type $\mathsf{Mono} \cdot F$ is the proposition that type scheme F is positive, or *monotone*. To introduce $\mathsf{Mono} \cdot F$ we provide a proof that, for all types X and Y such that X is included into Y, we can show $F \cdot X$ is included into $F \cdot Y$. Finally, elimMono allows one to use a proof of $\mathsf{Mono} \cdot T$ as a function of type $F \cdot S \to F \cdot T$ that is definitionally equal to $\lambda\, x.\, x$. Mono enjoys the same form of proof-irrelevance as does Cast.

Mendler-style encoding We next review definitions for generic Mendler-style encodings of datatypes in CDLE, presented axiomatically in Figure 5.

Mendler-style inductive types It is well-known that an inductive type with signature functor F has as its semantics in category theory the initial F-algebra $(\mu F, \mathsf{in}_{\mu F})$, where an F-algebra is a pair (X, ϕ) with X (the *carrier*) an object and $\phi : F\ X \to X$ (the *action*) a morphism. As an example, for the datatype Nat the signature F is $X \mapsto 1 + X$ (with $+$ forming a binary coproduct) and the algebra actions are morphisms $\phi : 1 + X \to X$.

Mendler-style F-algebras, developed by Uustalu and Vene [35] and based on an impredicative encoding of Mendler [26], give an alternative semantics of datatypes where the action of the algebra is a natural transformation $\Phi_R : \mathcal{C}(R, X) \to \mathcal{C}(F\ R, X)$, i.e., a mapping (natural in R) of morphisms $f : R \to X$ in $\mathcal{C}$ to morphisms $\Phi_R(f) : F\ R \to X$. In polymorphic type theory, this is expressed by $\forall\, R : \star.\, (R \to X) \to (F \cdot R \to X)$, where naturality comes as a "theorem for free" (c.f. Wadler [37]) from parametricity. Writing recursive functions in this fashion is known as the Mendler style of coding recursion [35], which can guarantee termination by polymorphic typing alone.

Datatype constructor and destructor Returning to Figure 5, the formation rule for datatype μF requires only that F is a type scheme. Positivity of F is required by the generic constructor in and destructor out, in the form of a proof t_1 of $\mathsf{Mono} \cdot F$. The constructor in takes also a term t_2 of type $F \cdot \mu F$, building a successor value from an "F-collection" of μF predecessors. The destructor out undoes this, taking an expression of type μF and unravelling it to its predecessors.

Mendler-style induction The definition of PrfAlg generalizes Mendler-style F-algebras to dependent "proof algebras". The carrier P is a predicate over μF, and the action is a function polymorphic in R mapping dependent functions of type $\Pi\, x : R.\, P\ (\mathsf{elimCast}\ \text{-}c\ x)$ to functions $\Pi\, xs : F \cdot R.\, P\ (\mathsf{in}\ \text{-}t\ (\mathsf{elimMono}\ \text{-}t\ \text{-}c\ xs))$, where c proves R is included into μF and t proves F is positive. The key intuition is to view R as a subtype of μF containing predecessors for which it is legal to invoke the inductive hypothesis $\Pi\, x : R.\, (P\ x)$ (coercions omitted); one assumes such an inductive hypothesis and must show that P holds for the successor $\mathsf{in}\ xs$ of an arbitrary $xs : F \cdot R$. With PrfAlg understood, so to is the induction principle: $\mathsf{ind}\ \text{-}t_1\ t_2$ is a proof that predicate P holds for all μF when t_1 proves F positive and t_2 is a proof algebra for P.

Characterization Proposition 3 gives the computational character of in, out, and ind. The first result listed is the *cancellation law*, stating how ind computes over values constructed with in. The second and third are the two halves of *Lambek's lemma* [22], stating that out and in are mutual inverses. Crucial to the claim that this encoding is efficient (and summarized in Figure 5) is the first and second hold by definitional equality alone (the third is an extensionality principle).

Proposition 3 (Firsov et al. [11]). *For $F : \star \to \star$ and $m : \mathsf{Mono} \cdot F$:*

- *For all term t_1 and t_2, $|\mathsf{ind}\ t_1\ (\mathsf{in}\ t_2)| =_{\beta\eta} |t_1\ (\mathsf{ind}\ t_1)\ t_2|$*
- *For all terms t, $|\mathsf{out}\ (\mathsf{in}\ t)| =_{\beta\eta} |t|$.*
- *If t is a term of type μF then $\{\mathsf{in}\ \text{-}m\ (\mathsf{out}\ \text{-}m\ t) \simeq t\}$ is provable.*

4 Course-of-values induction in CDLE

We now present the first listed contribution of the paper, the derivation of course-of-values induction, and of implicitly restricted existentials and singleton types used to achieve this, in CDLE. This result serves as the basis of type-based termination checking for the datatype subsystem for Cedille, as the surface language is justified via elaboration to developments in this section.

Course-of-values iteration enables functions to be defined in terms of the entire "course" of previously computed values. It is translated into the Mendler style of coding recursion with the type $\forall\, R : \star.\, (R \to F \cdot R) \to (R \to X) \to F \cdot R \to X$, the idea being to equip the function with a type-preserving destructor $R \to F \cdot R$ to dig arbitrarily deeply into predecessors. We may think to derive course-of-values induction from ordinary induction by augmenting the datatype signature F with an abstract destructor, forming the signature $R \mapsto (R \to F\ R) \times F\ R$, but this attempt faces two difficulties.

- *This signature is not positive*, and while Mendler-style inductive types can be safely formed from mixed-variant signatures, the desired destructor is definable only for positive signatures in logically consistent type theories.

$$\frac{\Gamma \vdash H : \star \to \star \quad \Gamma \vdash F : \star \to \star}{\Gamma \vdash \mathsf{RExt} \cdot H \cdot F : \star} \qquad \frac{\Gamma \vdash S : \star \quad \Gamma \vdash s : H \cdot S \quad \Gamma \vdash t : F \cdot S}{\Gamma \vdash \mathsf{pack} \cdot S \ \text{-}s \ t : \mathsf{RExt} \cdot H \cdot F}$$

$$\frac{\Gamma \vdash P : \mathsf{RExt} \cdot H \cdot F \to \star \quad \Gamma \vdash t : \forall\, X : \star . \forall\, x : H \cdot X . \Pi\, y : F \cdot X . P\ (\mathsf{pack} \cdot X \ \text{-}x\ y)}{\Gamma \vdash \mathsf{unpack} \cdot P\ t : \Pi\, z : \mathsf{RExt} \cdot H \cdot F . P\ z}$$

$$|\mathsf{pack} \cdot S \ \text{-}s\ t| = (\lambda\, x . \lambda\, f . f\ x)\ |t|,\ |\mathsf{unpack} \cdot P\ t| = (\lambda\, f . \lambda\, x . x\ f)\ |t|$$

Fig. 6. Derived implicily restricted existential types

– *This signature defines a* superset *of course-of-values datatypes.* The first component of the product, $R \to F\ R$, is not restricted to be just the datatype destructor, leaving open the possibility that each predecessor is tupled together with different "destructors". This interferes with describing the computational character of course-of-values induction.

This first difficulty was solved by Uustalu and Vene [35] with restricted existential types (categorically, restricted coends) used to form the covariant signature $R \mapsto \exists\, X . (X \to R) \times (X \to F\ X) \times F\ X$. To solve the second difficulty, we present a novel derivation of singleton types in CDLE, called View, enabling us to specify the abstract destructor is precisely the desired one.

4.1 Implicitly restricted existentials

Figure 6 axiomatically summarizes the derivation of RExt, a family of *implicitly* restricted existential types. This derivation follows a recipe similar to that given by Stump [30] and can be found in the proof appendix. The type $\mathsf{RExt} \cdot H \cdot F$ existentially quantifies over types S such that both $F \cdot S$ *and* the restriction $H \cdot S$ holds. Differing from the usual presentation of restricted existentials, the evidence for the restriction $H \cdot S$ is erased and cannot be used in computation.

To form this type, type schemes H and F must have kind $\star \to \star$. The introduction form pack takes a type argument S, a term s of type $H \cdot S$ as an erased argument, and a term t of type $F \cdot S$. The (dependent) elimination form unpack takes a property P over $\mathsf{RExt} \cdot H \cdot F$ and a proof that P holds of $\mathsf{pack} \cdot X \ \text{-}x\ y$ for all types X, erased $x : H \cdot S$, and $y : F \cdot S$. The whole expression is a proof that P holds for all $\mathsf{RExt} \cdot H \cdot F$. Finally, the erasure rules for implicitly restricted existentials give the erasures that the Cedille tool reports for the CDLE definitions of pack and unpack in the code repository.

4.2 Singleton types

We now describe the derived family of singleton types in CDLE that we use in the choice of restriction made to define course-of-values datatypes. Recall from Section 3 that the Kleene trick enables the definition of a type for all untyped

$$\mathsf{Top} = \{\lambda\, x.\, x \simeq \lambda\, x.\, x\} \qquad \frac{\Gamma \vdash S : \star \quad \Gamma \vdash t : \mathsf{Top}}{\Gamma \vdash \mathsf{View} \cdot S\ t}$$

$$\frac{\Gamma \vdash t : \mathsf{Top} \quad \Gamma \vdash t_1 : S \quad \Gamma \vdash t_2 : \{t_1 \simeq t\}}{\Gamma \vdash \mathsf{intrView}\ t\ \text{-}t_1\ \text{-}t_2 : \mathsf{View} \cdot S\ t} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Top} \quad \Gamma \vdash t_2 : \mathsf{View} \cdot S\ t_1}{\Gamma \vdash \mathsf{elimView}\ t_1\ \text{-}t_2 : S}$$

$$|\mathsf{intrView}\ t\ \text{-}t_1\ \text{-}t_2| = (\lambda\, x.\, x)\ |t|, \qquad |\mathsf{elimView}\ t_1\ \text{-}t_2| = (\lambda\, x.\, x)\ |t_1|$$

Fig. 7. Derived singleton types

lambda calculus terms. We give such a type as Top, with which we can form predicates over terms *before* they are known to have any other type.

For all types S, $\mathsf{View} \cdot S$ is such a predicate, a derived type construct presented axiomatically in Figure 7. $\mathsf{View} \cdot S\ t$ is the type of proofs that the Top-typed term t can be annotated (via $\mathsf{elimView}$) to have type S. Alternatively, it can be read as the singleton type on S of term equal to t. The introduction form $\mathsf{intrView}$ takes a Top-typed term t, an erased argument t_1 of type S, and a proof t_2 that t equals t_1. The elimination form $\mathsf{elimView}$ takes a Top-typed term t_1 and a proof t_2 of $\mathsf{View} \cdot S\ t_1$ as an erased argument. As with the Cast and Mono (Figure 4), View enjoys a form of proof-irrelevance suitable to be part of an implicitly restricted existential. The erasure of the elimination form reinforces this, with the expression $\mathsf{elimView}\ t_1\ \text{-}t_2$ erasing to $(\lambda\, x.\, x)\ |t_1|$.

4.3 Course-of-values datatypes

We bring together the preceding results to define course-of-values datatypes. To begin, notice that there is subtle circularity in proposing to define the generic signature CV_F for such datatypes in terms of their destructor $\mathsf{outCV} : \mu\mathsf{CV}_F \to F \cdot \mu\mathsf{CV}_F$. This circularity is broken using the following observations.

- CDLE's equality is *modulo erasure of typing annotations*. We need only define CV_F in terms of some term equal to the subsequent definition of outCV. We find it expedient to give a Top-typed version of the destructor, outCVU.
- The destructor out (Figure 5) takes an erased monotonicity witness, meaning it *performs no computation specific to the datatype signature*. No prior operation on CV_F is needed to express computation of out for $\mu\mathsf{CV}_F$.

The full CDLE derivation using these observations is given in Figure 8, with some syntax modified to aid in readability (e.g., sugar for derived product types $S \times T$ with projections π_1 and π_2, and removal of some type annotations). The code listing is parameterized in a type scheme $F : \star \to \star$ and monotonicity proof $m : \mathsf{Mono} \cdot F$. We summarize the key aspects of the derivation.

Definition of CV The Top-typed definition outCVU gives the underlying computational content of the destructor for course-of-values datatypes. Reading it with

```
outCVU : Top = β{| λ x. unpack (λ xs. xs) (out x) |} .

RCV : ⋆ → ⋆ → ⋆
= λ X: ⋆. λ R: ⋆. (Cast ·R ·X) × View ·(R → F ·R) outCVU .

CV : ⋆ → ⋆ = λ X: ⋆. REext ·(RCV ·X) ·F .

mCV : Mono ·CV = intrMono -(Λ X. Λ Y. Λ c.
  intrCast -(unpack Λ R. Λ r. λ xs.
        pack -((castTrans -(π1 r) -c), π2 r) xs)
    -(unpack Λ R. Λ r. λ xs. β)) .

outCV : μ CV → F ·(μ CV) = λ x.
  unpack (Λ R. Λ r. λ xs. elimMono -m -(π1 r) xs) (out -mCV x) .

inCV : F ·(μ CV) → μ CV = λ xs. in -mCV
    (pack -(castRefl, intrView outCVU -outCV -β) xs) .

PrfAlgCV : (μ CV → ⋆) → ⋆ = λ P: μ CV → ⋆.
  ∀ R: ⋆. ∀ c: Cast ·R ·(μ CV). View ·(R → F ·R) outCVU →
  (Π x: R. P (elimCast -c x)) → Π xs: F ·R. P (inCV (elimMono -m -c xs)) .

indCV : ∀ P: μ CV → ⋆. PrfAlgCV ·P → Π x: μ CV. P x = Λ P. λ alg.
  ind -mCV Λ R. Λ c. λ ih. unpack Λ S. Λ r. λ xs.
      alg ·S -(castTrans -(π1 r) -c)
        (intrView outCVU -(elimView outCVU -(π2 r)) -β)
        (λ x. ih (elimCast -(proj1 r) x)) xs .
```

Fig. 8. Generic encoding of course-of-values datatypes in CDLE

the intended typing, we have a function mapping datatypes $\mu\mathsf{CV}$ to $F \cdot \mu\mathsf{CV}$ by destructing them with out (giving a result of type $\mathsf{CV} \cdot (\mu\mathsf{CV})$ and unpacking the restricted existential to return the predecessor values. With outCVU, we define the type family of restrictions RCV, whose first argument X is the parameter in which we will take the fixpoint and whose second argument R serves as the existentially quantified type variable. The body of RCV is the product of proofs that R is included into X and that outCVU has type $R \to F \cdot R$.

Finally, we form the signature CV and give proof mCV that it is positive. We do not detail this proof except to note the critical point where the existential is re-packed: there, transitivity of type inclusions is used (castTrans, Figure 4) to combine the hypothetical inclusion $c : \mathsf{Cast} \cdot X \cdot Y$ y with the inclusion $\pi_1\ r : \mathsf{Cast} \cdot R \cdot X$ that is part of the restriction r.

Datatype destructor and constructor As promised, we can define an informatively typed destructor outCV that is definitionally equal to outCVU. That this equality holds follows from the erasure of the elimination form elimMono (Figure 4).

Finally, we can define the constructor inCV for course-of-values datatypes. In the argument to `in` we are required to give a restricted existential $\mathsf{CV} \cdot \mu\mathsf{CV}$. We chose for the existentially quantified type $\mu\mathsf{CV}$, provide a proof that $\mu\mathsf{CV}$ is included in itself (castRefl), and a proof that outCVU can be viewed at type $\mu\mathsf{CV} \to F \cdot \mu\mathsf{CV}$, given with intrView — outCV equals outCVU and has this type.

Course-of-values induction To understand the definition PrfAlgCV of Mendler-style course-of-values proof algebras, recall the definition in type theory of Mendler-style course-of-values algebras: $\forall\, R : \star.\, (R \to F \cdot R) \to (R \to T) \to F \cdot R \to T$. PrfAlgCV translates this into a dependently typed setting by:

- adding (erased) assumption c that R is included into $\mu\mathsf{CV}$;
- restricting the destructor to be precisely outCVU;
- taking the handle for making recursive calls $R \to T$ to an induction hypothesis $\Pi\, x : R.\, P$ (elimCast -c x);
- making the goal showing that P holds for every $\mu\mathsf{CV}$ built from arbitrary $xs : F \cdot R$, lifting of the assumed c to an inclusion of $\mathsf{CV} \cdot R$ into $\mathsf{CV} \cdot \mu\mathsf{CV}$.

The generic course-of-values induction principle, indCV, is defined as a composition of ind (Figure 5) and the eliminator for restricted existentials. After applying both and introducing assumptions, the goal is to show P holds for in (pack xs) (convertible with inCV xs; erased arguments are here omitted), accomplished by giving the proof algebra *alg*:

- a proof of inclusion of the existentially quantified S into $\mu\mathsf{CV}$, using transitivity on the inclusion of S into R and of R into $\mu\mathsf{CV}$;
- a proof that outCVU has type $S \to F \cdot S$;
- and an induction hypothesis formed by restricting the domain of ih to S.

Computing with course-of-values datatypes To characterize our derivation of course-of-values datatypes, it is essential to abstract away from lambda encodings and express their computational properties in terms of inCV, outCV, and indCV themselves. The expected computation laws arise from the setting of category theory as the *cancellation law* and one half of *Lambek's lemma* (the second half being an extensionality principle or η-law). Our encoding enjoys the both cancellation law and Lambek's lemma (in full).

Theorem 1 (Characterization). *For all $F : \star \to \star$ and $m : \mathsf{Mono} \cdot F$:*

- *for all terms t_1 and t_2, $|\mathsf{indCV}\ t_1\ (\mathsf{inCV}\ t_2)| =_{\beta\eta} |t_1\ \mathsf{outCV}\ (\mathsf{indCV}\ t_1)\ t_2|$;*
- *for all terms t, $|\mathsf{outCV}\ (\mathsf{inCV}\ t)| =_{\beta\eta} |t|$*
- *for all terms t of type $\mu(\mathsf{CV} \cdot F)$, $\{\mathsf{inCV}\ \text{-}m\ (\mathsf{outCV}\ \text{-}mt) \simeq t\}$ is provable.*

Proof. These are proven with respect to CDLE's equality type in the code repository associated with this paper. For the first two, the proof holds by the β-axiom alone (Figure 3), meaning the terms are definitionally equal. The last comes from an inductive proof using indCV.

```
Case : (μ CV → ⋆) → Π R: ⋆. RCV ·(μ CV) ·R → ⋆
= λ P: μ CV → ⋆. λ R: ⋆. λ r: RCV ·(μ CV) ·R.
  Π xs: F ·R. P (inCV (elimMono -m -(π1 r) xs)) .

sigma : ∀ P: μ CV → ⋆. ∀ R: ⋆. ∀ r: RCV ·(μ CV) ·R.
  Π x: R. Case ·P ·R r → P (elimCast -(π1 r) x) = <..>

ByInd : (μ CV → ⋆) → ⋆ = λ P: μ CV → ⋆. ∀ R: ⋆.
  ∀ r: RCV ·(μ CV) ·R. (Π x: R. P (elimCast -(π1 r) x)) → Case ·P ·R r .

mu : ∀ P: μ CV_F → ⋆. Π x: μ CV. ByInd ·P → P x = <..>
```

Fig. 9. Variants sigma and mu

4.4 Towards surface language constructs

It is preferable to not expose the datatype signature F to the user. In particular, this means that outCV should be replaced by a general construct for *proof by cases*, and should not appear in the reduction of recursive definitions in the surface language, as it does in Proposition **??**. We end this section by describing variants of outCV and indCV, resp. sigma and mu in Figure 9, that correspond directly to resp. the surface constructs σ and μ (Section 5.3). The variations are minor, so we describe only their types and properties.

The type family Case expresses a generalized shape of proofs by cases of a predicate P over a type R for which the restriction $\mathsf{RCV} \cdot (\mu\mathsf{CV})$ holds. Its body is the type of functions taking an F-collection of R values and returning a proof that P holds for the value from them using inCV, after inserting the appropriate coercion. The type of sigma then promises that for all such P and R, given a proof by cases of P we have that P holds for all values of type R (the definition is omitted, indicated by <..>). The type family ByInd gives a variation of Mendler-style course-of-values proof algebras in terms of the restriction RCV, given as an erased argument. The definition mu is a variant of indCV that takes a ByInd argument.

Properties We can characterize sigma and mu (and the extensionality principle of sigma) without referring to the encodings, as we did with outCV and indCV.

Theorem 2 (Characterization (mu, sigma)).

- *For all terms* t_1 *and* t_2, $|\mathsf{mu}\ (\mathsf{inCV}\ t_1)\ t_2| =_{\beta\eta} |t_2\ (\lambda\, x.\, \mathsf{mu}\ x\ t_2)\ t_1|$, *where* $x \notin FV(t_2)$
- *For all terms* t_1 *and* t_2, $|\mathsf{sigma}\ (\mathsf{inCV}\ t_1)\ t_2| =_{\beta\eta} |t_2\ t_1|$
- *For all terms* t *of type* $\mu\mathsf{CV}$, $\{\mathsf{sigma}\ t\ \mathsf{inCV} \simeq t\}$ *is provable.*

Proof. Given in the code repository associated with this paper.

5 Inductive definitions

We now turn to the syntax for Cedille's datatype system and its translational semantics into the encodings of the previous section. In this paper and its proof appendix, we treat formally only non-indexed, non-parameterized datatypes (the Cedille tool supports both of these); in this paper, we simplify the presentation elaboration and further restrict ourselves to datatypes of one constructor taking one argument. Multiple constructors can be simulated with a single constructor taking a coproduct argument, and multiple arguments simulated with products (and erased term and type arguments simulated with implicit restricted existentials). We emphasize that Cedille's surface language requires no such simulation.

Definition 1 (Datatype declaration). *A* datatype declaration *is a triple, written* $Ind[D, R, c{:}T \to D]$*, where:*

- D *(the datatype name) is a unique identifier supplied by the user;*
- R *is a freshly generated type variable;*
- *and* $c{:}T \to D$ *is the constructor declaration, with* c *(the constructor name) is a unique identifier, and where all occurrences of* D *in the user-supplied constructor argument type* $T[D/R]$ *have been replaced by* R*.*

Example 1. Natural numbers are declared in Cedille as

```
data Nat : ⋆ = zero : Nat | succ : Nat → Nat .
```

and would be simulated by

```
data Nat : ⋆ = mkNat : Unit + Nat → Nat
```

which would be represented by $\mathrm{Ind}[\mathsf{Nat}, R, \mathsf{mkNat}{:}\mathsf{Unit} + R \to \mathsf{Nat}]$ where '+' is sugar for the encoding of binary product types and Unit the singleton type.

We require that the type of the constructor argument is well-kinded and that R occurs only positively within it. We describe the algorithm for checking these requirements by a collection of judgments whose inference rules are mutually inductively defined. The main judgments are:

- $\Gamma \vdash \mathrm{Ind}[D, R, c : T \to D] \dashv \Gamma, \mathrm{IndEl}[D, R, c : T \to D, m, L, \Theta, \mathcal{E}]$, which checks that the datatype declaration $\mathrm{Ind}[D, R, c : T \to D]$ is well-formed and adds to the current typing context (notated by $\dashv$) the declaration, a positivity witness m and predicate transformer L used during elaboration, additional global constants Θ, and a mapping $\mathcal{E}$ of declarations in the surface language elaborations to their CDLE elaborations;
- $\Gamma \vdash F \overset{+}{\hookrightarrow} m$, checking F is positive and elaborating witness $m : \mathsf{Mono} \cdot F$;
- and $\Gamma \vdash t : T \hookrightarrow t'$, meaning that term t has type T in the surface language and elaborates to the CDLE term t'.

Our notational convention is that judgments with the hooked arrow $\hookrightarrow$ are for the surface language (which includes CDLE), whereas those without concern only pure CDLE constructs. Due to space constraints, we cannot list the full set of inference rules for elaboration here (see the proof appendix). Figure 10 lists remaining judgments needed for the statements of our main meta-theoretic results, type and value preservation for elaborated terms. In particular, the definitions of term, type, kind, and context elaboration consist mostly of congruence rules, with the elaborations of global definitions exported by datatype declarations given by their mappings in $\mathcal{E}$ of the elaborated declaration.

5.1 Positivity checking

The single inference rule for the judgment $\Gamma \vdash F \overset{+}{\hookrightarrow} m$ for positivity checking is given in Figure 11. It appeals to the auxiliary judgments for checking subtyping and subkinding based on a given coercion, listed resp. in Figures 12 and 13. The positivity judgment is defined only for type schemes of the form $\lambda\, X : \star.\, T$, and attempts to confirm monotonicity of this scheme following the introduction forms of intrMono and intrCast (Figure 4) by assuming arbitrary types R_1 and R_2 such that R_1 is included into R_2, witnessed by z, and using the coercion elimCast -z to produce a coercion s from $T[R_1/R]$ to $T[R_2/R]$.

Subtyping and subkinding relation In the judgment $\Gamma\ ;\ s \vdash S \leqslant T \hookrightarrow s'$ for subtyping, input s is a coercion — a base subtyping assumption – and output s' is a coercion from S to T derived from it. The invariant this judgment maintains is that if $|s| =_{\beta\eta} \lambda\, x.\, x$, then so too does $|s'|$. We do not walk through the entire set of inference rules for the subtyping judgment, but discuss a few interesting cases. The first rule is simply a statement of reflexivity for type inclusions (c.f. [31] for a description of the convertibility relation between types). The last rule of the figure is for type quantification. If K_2 is a subkind of K_1 with kind coercion S, and if after extending the current context with a type variable $Y : K_2$ we have that replacing all occurrences of X in T_2 with$S \cdot Y$ (of kind K_1) is a subtype of $T[Y/X]$ with type coercion s, then $\forall\, X : K_1.\, T_1$ is a subtype of $\forall\, X : K_2.\, T_2$. The coercion produced in the conclusion takes a term f of the subtype and type argument $X : K_2$, instantiates f with $S \cdot X$ and coerces the result to type T_2.

The subkinding judgment $\Gamma\ ;\ s \vdash K_1 \leqslant K_2 \hookrightarrow S$ is simpler, appealing in the case of term quantification back to the subtyping judgment. We have that, with coercion s, type $\forall\, X : K_1.\, T_1$ is a subtype of $\forall\, X : K_2.\, T_2$ when the kind K_2 is a subkind of K_1, witnessed by the kind coercion S, and for arbitrary $Y : K_2$, $[S \cdot Y/X]T_1$ is a subtype of $[Y/X]T_2$.

Soundness Our positivity checker is *sound*, meaning that term it produces when checking type scheme F really does prove that F is monotone.

Theorem 3 (Soundness of positivity checker). *For all Γ and F, if $\Gamma \vdash F : \star \to \star \hookrightarrow F'$ for some F' and $\Gamma \vdash F \overset{+}{\hookrightarrow} m$, then $\Gamma \vdash m : \mathsf{Mono} \cdot F \hookrightarrow m'$ for some m'.*

$\boxed{\Gamma \vdash t : T \hookrightarrow t'}$ Terms $\quad$ $\boxed{\Gamma \vdash T : K \hookrightarrow T'}$ Types $\quad$ $\boxed{\Gamma \vdash K \hookrightarrow K'}$ Kinds $\quad$ $\boxed{\vdash \Gamma \hookrightarrow \Gamma'}$ Contexts

$\boxed{\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \dashv \Gamma, \mathrm{IndEl}[D, R, c{:}T \to D, m, L, \Theta, \mathcal{E}]}$ Datatype declarations

$\boxed{\Gamma \vdash F \overset{+}{\hookrightarrow} m}$ Positivity checking $\quad$ $\boxed{\Gamma \vdash S \leqslant T \hookrightarrow s}$ Subtyping $\quad$ $\boxed{\Gamma \vdash K_1 \leqslant K_2 \hookrightarrow S}$ Subkinding

Fig. 10. Summary of elaboration judgments

$$\frac{\Gamma, R_1 : \star, R_2 : \star, z : \mathsf{Cast} \cdot R_1 \cdot R_2 \ ; \ \mathsf{elimCast}\ \text{-}z \vdash T[R_1/R] \leqslant T[R_2/R] \hookrightarrow s}{\Gamma \vdash \lambda\, R{:}\star.\, T \overset{+}{\hookrightarrow} \mathsf{intrMono}\ \text{-}(\Lambda\, R_1.\, \Lambda\, R_2.\, \lambda\, z.\, \mathsf{intrCast}\ \text{-}s\ \text{-}(\lambda\, y.\, \beta\{\lambda\, x.\, x\}))}$$

Fig. 11. Positivity checking

$$\frac{\Gamma \vdash T_1 \cong T_2}{\Gamma; s \vdash T_1 \leqslant T_2 \hookrightarrow\ \lambda\, x.\, x} \qquad \frac{\Gamma \vdash s : S \to T \hookrightarrow \_}{\Gamma; s \vdash S \leqslant T \hookrightarrow s}$$

$$\frac{\Gamma; s' \vdash S_2 \leqslant S_1 \hookrightarrow s \quad \Gamma, y{:}S_2; s' \vdash T_1[(s\ y)/x] \leqslant T_2[y/x] \hookrightarrow t}{\Gamma; s' \vdash \Pi\, x{:}S_1.\, T_1 \leqslant \Pi\, x{:}S_2.\, T_2 \hookrightarrow \lambda\, f.\, \lambda\, x.\, t[x/y]\ (f\ (s\ x))}$$

$$\frac{\Gamma; s' \vdash S_2 \leqslant S_1 \hookrightarrow s \quad \Gamma, y{:}S_2; s' \vdash T_1[(s\ y)/x] \leqslant T_2[y/x] \hookrightarrow t}{\Gamma; s' \vdash \forall\, x{:}S_1.\, T_1 \leqslant \forall\, x{:}S_2.\, T_2 \hookrightarrow \lambda\, f.\, \Lambda\, x.\, t[x/y]\ (f\ \text{-}(s\ x))}$$

$$\frac{\Gamma; s' \vdash S_1 \leqslant S_2 \hookrightarrow s \quad \Gamma, y{:}S_1; s' \vdash T_1[y/x] \leqslant T_2[(s\ y)/x] \hookrightarrow t}{\Gamma; s' \vdash \iota\, x{:}S_1.\, T_1 \leqslant \iota\, x{:}S_2.\, T_2 \hookrightarrow \lambda\, u.\, [s\ u.1, t[u.1/y]\ u.2]}$$

$$\frac{\Gamma; s' \vdash K_2 \leqslant K_1 \hookrightarrow S \quad \Gamma, Y : K_2; s' \vdash T_1[(S \cdot Y)/X] \leqslant T_2[Y/X] \hookrightarrow s}{\Gamma; s' \vdash \forall\, X{:}K_1.\, T_1 \leqslant \forall\, X{:}K_2.\, T_2 \hookrightarrow \lambda\, f.\, \Lambda\, X.\, s[X/Y]\ (f \cdot (S \cdot X))}$$

Fig. 12. Subtyping with type inclusion

$$\frac{}{\Gamma; s \vdash \star\ \leqslant\ \star \hookrightarrow \lambda\, X.\, X} \qquad \frac{\Gamma; s' \vdash S_2 \leqslant S_1 \hookrightarrow s \quad \Gamma, y : S_2; s' \vdash K_1[(s\ y)/x] \leqslant K_2[y/x] \hookrightarrow S}{\Gamma; s' \vdash \Pi\, x{:}S_1.\, K_1 \leqslant \Pi\, x{:}S_2.\, K_2 \hookrightarrow \lambda\, P.\, \lambda\, x.\, S[x/y] \cdot (P\ (s\ x))}$$

$$\frac{\Gamma; s \vdash K_1' \leqslant\ K_1 \hookrightarrow S_1 \quad \Gamma, Y : K_1'; s \vdash K_2[(S_1 \cdot Y)/X]\ \leqslant\ K_2'[Y/X] \hookrightarrow S_2}{\Gamma; s \vdash \Pi\, X{:}K_1.\, K_2 \leqslant\ \Pi\, X{:}K_1'.\, K_2' \hookrightarrow \lambda\, P.\, \lambda\, X.\, S_2[X/Y] \cdot (P \cdot (S_1 \cdot X))}$$

Fig. 13. Subkinding with type inclusion

Proof. Given in the proof appendix.

5.2 Datatype declarations

Figure 14 gives the judgment for elaborating inductive definitions with the judgment $\Gamma \vdash \mathrm{Ind}[D, R, c : T \to D] \dashv \mathrm{IndEl}[D, R, c : T \to D, m, L, \Theta, \mathcal{E}]$. For readability, it has been broken into several single-use judgments, which we refer to by the labels given to the single rule associated to each. Observe that, within the CDLE derivations of Figures 8 and 9, the signature F and monotonicity witness m were implicitly parameters. In our elaboration judgments, the elaborated signature and witness are resp. given explicitly those type and term definitions.

Datatype signature Rules F, FI, and FIX elaborate the datatype signature and the datatype itself. In F, the signature for a datatype D with single constructor $c : T \to D$ is elaborated to its usual impredicative encoding $\lambda\, R.\, \forall\, X : \star.\, (T' \to X) \to X$ (where T elaborates to T') — call this type scheme F. Rule FI further refines F, elaborating its CDLE version that supports an induction principle (non-recursive proof by cases) using dependent intersections (Figure 3). Read the body of this scheme as: terms z which have type $F \cdot R$ *and are proofs* that, for all properties X over terms of type $F \cdot R$, if X holds for terms constructed with t (the elaboration c), then X holds for z. Finally, rule FIX elaborates the datatype itself using the generic datatype former μ (Figure 5) on the inductive signature further equiped with support for course-of-values induction (CV, Figure 8).

Datatype constructor Rules cF, cFI, and cFIX elaborate the constructor for the datatype. Following the same pattern as for the datatype itself, the first rule cF elaborates the usual lambda encoding of the sole constructor of F, which is a term t of type $\forall\, R : \star.\, T' \to F \cdot R$. This is further refined in cFI: t is combined with a proof that t satisfies the induction principle given by the elaborated inductive signature of FI. Notice that these two terms are indeed definitionally equal, as required to introduce dependent intersections. Finally, cFIX elaborates the datatype constructor by combining the constructor for the inductive signature with the generic constructor inCV. *It is here that positivity is checked* for the elaborated inductive signature, as this is required by inCV.

Predicate lifting The penultimate rule is Lift, bridging the gap between the proof principle supported by the type scheme F_2 generated by FI, which concerns predicates over the type scheme F_1 generated by F, and a true induction principle. More precisely, it defines a family, over types R for which the CV-restriction holds for the datatype $\mu(\mathrm{CV} \cdot F_2)$, of transformers of predicates $P : \mu(\mathsf{CV} \cdot F_2) \to \star$ to predicates of kind $F_1 \cdot R \to \star$ by requiring that the given $x : F_1 \cdot R$ may be viewed at type $F_2 \cdot R$ and giving as codomain that P holds for the value constructed from x using inCV after inserting type coercions.

$$\frac{\Gamma, R{:}\star, X{:}\star \vdash T : \star \hookrightarrow T'}{\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{F}}{\hookrightarrow} \lambda\, R.\, \forall\, X{:}\star.\, (T' \to X) \to X}\ \mathrm{F}$$

$$\frac{}{\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{cF}}{\hookrightarrow} \Lambda\, R.\, \lambda\, y.\, \Lambda\, X.\, \lambda\, x.\, x\ y}\ \mathrm{cF}$$

$$\frac{\begin{array}{c}\Gamma, R{:}\star, X{:}\star \vdash T : \star \hookrightarrow T' \\ \Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{F}}{\hookrightarrow} F \quad \Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{cF}}{\hookrightarrow} t\end{array}}{\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{FI}}{\hookrightarrow} \lambda\, R.\, \iota\, z{:}F \cdot R.\, \forall\, X{:}F \cdot R \to \star.\, (\Pi\, y{:}T'.\, X\ (t\ y)) \to X\ z}\ \mathrm{FI}$$

$$\frac{\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{cF}}{\hookrightarrow} t}{\Gamma \vdash \mathrm{Ind}[D, R, \Delta] \overset{\mathrm{cFI}}{\hookrightarrow} \Lambda\, R.\, \lambda\, y.\, [t \cdot R\ y, \Lambda\, X.\, \lambda\, x.\, x\ y]}\ \mathrm{cFI}$$

$$\frac{\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{FI}}{\hookrightarrow} F}{\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{FIX}}{\hookrightarrow} \mu(\mathsf{CV} \cdot F)}\ \mathrm{FIX}$$

$$\frac{\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{FIX}}{\hookrightarrow} \mu(\mathsf{CV} \cdot F) \quad \Gamma \vdash F \overset{+}{\hookrightarrow} m \quad \Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{cFI}}{\hookrightarrow} t}{\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{cFIX}}{\hookrightarrow} \lambda\, y.\, \mathsf{inCV}\ \text{-}m\ (t\ y)}\ \mathrm{cFIX}$$

$$\frac{\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{F}}{\hookrightarrow} F_1 \quad \Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{FIX}}{\hookrightarrow} \mu(\mathsf{CV} \cdot F_2) \quad \Gamma \vdash F_2 \overset{+}{\hookrightarrow} m}{\begin{array}{c}\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{Lift}}{\hookrightarrow} \lambda\, P{:}\mu(\mathsf{CV} \cdot F_2) \to \star.\, \lambda\, R{:}\star. \\ \lambda\, r{:}\mathsf{RCV} \cdot F_2 \cdot \mu(\mathsf{CV} \cdot F_2) \cdot R.\, \lambda\, x{:}F_1 \cdot R.\, \forall\, y{:}\mathsf{View} \cdot (F_2 \cdot R)\ \beta\{x\}. \\ P\ (\mathsf{inCV}\ \text{-}m\ (\mathsf{elimMono}\ \text{-}m\ \text{-}(\pi_1\ r)\ (\mathsf{elimView}\ \beta\{x\}\ \text{-}y)))\end{array}}\ \textsc{Lift}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D]\ \mathit{wf} \quad \Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{FIX}}{\hookrightarrow} \mu(\mathsf{CV} \cdot F) \\ \Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{cFIX}}{\hookrightarrow} t \quad \Gamma \vdash F \overset{+}{\hookrightarrow} m \quad \Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \overset{\mathrm{Lift}}{\hookrightarrow} L\end{array}}{\Gamma \vdash \mathrm{Ind}[D, R, c{:}T \to D] \dashv \Gamma, \mathrm{IndEl}[D, R, c{:}T \to D, m, L, \Theta, \mathcal{E}]}$$

where $\Theta = (\mathsf{ls}/D : \star \to \star,\ \mathsf{is}/D : \mathtt{Is}/D \cdot D,\ \mathsf{to}/D = \lambda\, x.\, x : \forall\, R{:}\star.\, \forall\, is{:}\mathsf{ls}/D \cdot R.\, R \to D)$

$$\mathcal{E} = \left\{ \begin{array}{c} D \mapsto \mu(\mathsf{CV} \cdot F), \quad c \mapsto t, \quad \mathsf{ls}/D \mapsto \mathsf{RCV} \cdot F \cdot \mu(\mathsf{CV} \cdot F), \\ \mathsf{is}/D \mapsto (\mathsf{castRefl} \cdot \mu(\mathsf{CV} \cdot F), \mathsf{intrView}\ \mathsf{outCVU}\ \text{-}(\mathsf{outCV}\ \text{-}m)\ \text{-}\beta) \\ \mathsf{to}/D \mapsto \Lambda\, R.\, \Lambda\, is.\, \mathsf{elimCast}\ \text{-}(\pi_1\ is) \end{array} \right\}$$

Fig. 14. Elaboration of datatype declarations

Datatype globals for termination checking The final (unlabeled) rule of Figure 14 elaborates the datatype D and its constructor c, add them as identifiers to the surface language typing context and storing their CDLE encodings in the map $\mathcal{E}$. In addition, it exports several other identifiers, given in Θ (with the elaborations of these also given in $\mathcal{E}$) to support course-of-values induction. These identifiers are automatically generated based on the identifier D of the datatype, e.g., Is/Nat. In the surface language, the type family Is/D is a predicate specifying that some type S may be treated as if it were really the datatype D for the purposes of case analysis; internally, it is the restriction $\mathsf{RCV} \cdot \mu(\mathsf{CV} \cdot F)$ (Figure 8). The type inclusion that this restriction provides is accessed in the surface language by to/D, which is convertible with $\lambda\, x.\, x$, and is/D is a proof that D itself satisfies the predicate Is/D. The monotonicity witness m and predicate transformer L are not exported to the surface language, but are required for elaboration.

Soundness Datatype elaboration is *sound*, meaning that the elaborated datatype is well kinded (at $\star$), the elaborated term constructs are well typed for the types of their elaborations, and the elaboration of to/D is definitionally equal to $\lambda\, x.\, x$.

Theorem 4 (Soundness of elaboration of declarations). *For all contexts Γ, unique identifiers D and c, type variables R fresh wrt Γ, and types T, if:*

- $\Gamma \vdash \mathit{Ind}[D, R, c{:}T \to D] \dashv \Gamma, \mathit{IndEl}[D, R, c{:}T \to D, m, L, \Theta, \mathcal{E}]$;
- $\mathit{and} \vdash \Gamma \hookrightarrow \Gamma'$ *for some* Γ';
- *and* $\Gamma, X{:}\star, R{:}\star \vdash T : \star \hookrightarrow T'$ *for some* T' *implies* $\Gamma', X{:}\star, R{:}\star \vdash T' : \star$,

we have that:

- $\Gamma' \vdash \mathcal{E}(D) : \star$ *and* $\Gamma' \vdash c : T'[\mathcal{E}(D)/R] \to \mathcal{E}(D)$
- $\Gamma' \vdash \mathcal{E}(\mathsf{Is}/D) : \star \to \star$ *and* $\Gamma' \vdash \mathcal{E}(\mathsf{is}/D) : \mathcal{E}(\mathsf{Is}/D) \cdot \mathcal{E}(D)$
- $\Gamma' \vdash \mathcal{E}(\mathsf{to}/D) : \forall\, R{:}\star.\, \forall\, \mathit{is}{:}\mathcal{E}(\mathsf{Is}/D) \cdot R.\, R \to \mathcal{E}(D)$ *and* $|\mathcal{E}(\mathsf{to}/D)| =_{\beta\eta} \lambda\, x.\, x$
- $\Gamma \vdash m : \mathsf{Mono} \cdot F_2$, *where* $\Gamma \vdash \mathit{Ind}[D, R, c{:}T \to D] \overset{FI}{\hookrightarrow} F_2$
- $\Gamma \vdash L : (\mu(\mathsf{CV} \cdot F_2) \to \star) \to \Pi\, R{:}\star.\, \mathsf{RCV} \cdot F_2 \cdot \mu(\mathsf{CV} \cdot F_2) \to F_1 \cdot R \to \star$, *where* F_2 *is as above and* $\Gamma \vdash \mathit{Ind}[D, R, c{:}T \to D] \overset{F}{\hookrightarrow} F_1$

5.3 Functions

We now explain the term constructs σ and μ for resp. case analysis and combined case analysis and recursion, given in Figure 15. *These are the key constructs for supporting course-of-values induction in the surface language.*

Case branches For readability, the two typing rules use an auxiliary judgment $\Gamma \vdash \{c\ y \to t_1\} : (P, \mathit{is}) \hookrightarrow (t_1', \mathit{is}')$ for checking case branches, consisting of a single rule Case. In Case, first is is checked to be a witness of type $\mathsf{Is}/D \cdot S$ for some type S and datatype D. Type S takes the place of all occurrences of R in the type of the constructor pattern variable y (these are the recursive occurrences of D in the user given datatype declarations), so a coercion s from $T[S/X]$ to $T[D/X]$ is generated using the subtyping judgment (Figure 12). Finally, the branch body t_1 is checked to be a proof of $P\ (c\ (s\ y))$, and the rule elaborates the elaboration of t_1 (with y lambda bound) and of the witness is.

$$\boxed{\Gamma \vdash \{c\ y \to t_1\} : (P, is) \hookrightarrow (t_1', is')}$$

$$\frac{\begin{array}{c}\mathrm{IndEl}[D, R, c{:}T \to D, m, L, \Theta, \mathcal{E}] \in \Gamma \quad \Gamma \vdash is : \mathsf{Is}/D \cdot S \hookrightarrow is' \\ \Gamma\ ;\ \mathsf{to}/D\ \text{-}is \vdash T[S/R] \leqslant T[D/R] \hookrightarrow s \quad \Gamma, y{:}T[S/R] \vdash t_1 : P\ (c\ (s\ y)) \hookrightarrow t_1'\end{array}}{\Gamma \vdash \{c\ y \to t_1\} : (P, is) \hookrightarrow \lambda\, y.\, t_1'}\ \textsc{Case}$$

$$\boxed{\Gamma \vdash t : T \hookrightarrow t'}$$

$$\frac{\begin{array}{c}\mathrm{IndEl}[D, R, c{:}T \to D, m, L, \Theta, \mathcal{E}] \in \Gamma \quad \Gamma \vdash is : \mathsf{Is}/D \cdot S \hookrightarrow is' \quad \Gamma \vdash t_1 : S \hookrightarrow t_1' \\ \Gamma \vdash S : \star \hookrightarrow S' \quad \Gamma \vdash P : D \to \star \hookrightarrow P' \quad \Gamma \vdash \{c\ y \to t_2\} : (P, is) \hookrightarrow \lambda\, y.\, t_2' \quad z \notin FV(t_2')\end{array}}{\begin{array}{c}\Gamma \vdash \sigma\langle is\rangle\ t_1\ @P\ \{\ c\ y \to t_2\} : P\ (\mathsf{to}/D\ \text{-}is\ t_1) \hookrightarrow \mathsf{sigma}\ \text{-}m\ \text{-}is'\ t_1' \cdot P'\ \lambda\, x. \\ x.2 \cdot (L \cdot P' \cdot S'\ is')\ (\lambda\, y.\, \Lambda\, z.\, t_2')\ \text{-}(\mathsf{intrView}\ \beta\{x.1\}\ \text{-}x\ \text{-}\beta)\end{array}}$$

$$\frac{\begin{array}{c}\mathrm{IndEl}[D, R, c{:}T \to D, m, L, \Theta, \mathcal{E}] \in \Gamma \quad \Gamma \vdash P : D \to \star \hookrightarrow P' \quad \Gamma \vdash t_1 : D \hookrightarrow t_1' \\ \Gamma' =_{\mathbf{df}} \Gamma, \mathsf{Type}/f : \star, \mathsf{isType}/f : \mathsf{Is}/D \cdot \mathsf{Type}/f, f : \Pi\, x : \mathsf{Type}/f.\, P\ (\mathsf{to}/D\ \text{-}\mathsf{isType}/f\ x) \\ \Gamma' \vdash \{c\ y \to t_2\} : (P, \mathsf{isType}/f) \hookrightarrow \lambda\, y.\, t_2' \quad z \notin FV(t_2') = \emptyset\end{array}}{\begin{array}{c}\Gamma \vdash \mu\ f.\ t_1\ @P\ \{\ c\ y \to t_2\} : P\ t_1 \hookrightarrow \mathsf{mu}\ \text{-}m\ t' \cdot P'\ \Lambda\, \mathsf{Type}/f.\, \Lambda\, \mathsf{isType}/f.\, \lambda\, f.\, \lambda\, x. \\ x.2 \cdot (L \cdot P' \cdot \mathsf{Type}/f\ (\mathsf{isType}/f)\ \lambda\, y.\, \Lambda\, z.\, t_2')\ \text{-}(\mathsf{intrView}\ \beta\{x.1\}\ \text{-}x\ \text{-}\beta)\end{array}}$$

Fig. 15. Elaboration of datatype destructor and recursor

Case analysis The term construct $\sigma\langle is\rangle\ t_1\ @P\ \{\ c\ y \to t_2\}$ provides a mechanism for case distinction that *preserves the type of its scrutinee* t_1 *for its predecessors* (by using the judgment for case branches), making it suitable for case analysis on terms of both type D as well as abstract types introduced during recursion. Since t_1 might not be given at type D, the σ expression returns a proof that P holds of t_1 coerced to type D using to/D. Similar to checking case branches, its typing rule begins by checking that the term *is* has type $\mathsf{Is}/D \cdot S$ for some D and S. It then checks that t_1 has type S and that the case branch $\{c\ y \to t_2\}$ is well-typed for predicate P. The term elaborated in the rule's conclusion uses the CDLE expression sigma (Figure 9). In the body of the function argument given to sigma, the proof principle associated to argument x for the elaborated inductive signature is invoked with the lifting predicate transformer L on the elaboration of P'. The first term argument to $x.2$ is the elaborated case branch extended with an erased assumption z introduced by lifting, and the second (erased) argument is a proof that $x.1$ can be viewed as having the type of x.

Recursion The term construct $\mu\ f.\ t_1\ @P\ \{\ c\ y \to t_2\}$ defines an anonymous recursive function (referred to by f in the case body t_2) over some data t_1 and returning a proof of $P\ t_1$. Its typing rule checks that the type of t_1 is some datatype D, then checks that the case branch is well typed for P under a context extended by: an automatically generated type variable Type/f (based on the user-supplied identifier f), a proof isType/f that this type satisfies Is/D,

and a handle f for invoking the inductive hypothesis on terms of type Type/f. *It is the introduction of the witness* isType/f *that enables further inspection of predecessors at the abstract type* Type/f*, and thus course-of-values induction.* Similar to σ, the elaboration of the μ expression uses the CDLE expression mu and the predicate transformer L for signature induction.

$$
\begin{array}{r}
\mid \sigma\langle is\rangle\ t_1\ @P\ \{\ c\ y \to t_2\}\ \mid = \sigma\ \mid t_1\mid\ \{\ c\ y \to \mid t_2\mid\}\\
\mid \mu\ f.\ t_1\ @P\ \{\ c\ y \to t_2\}\ \mid = \mu\ f.\ \mid t_1\mid\ \{\ c\ y \to \mid t_2\mid\}
\end{array}
$$

$$
\begin{array}{rcl}
\sigma\ (c\ \mid t_1\mid)\ \{\ c\ y \to \mid t_2\mid\} & \rightsquigarrow & \mid t_2[t_1/y]\mid \\
\mu\ f.\ (c\ \mid t_1\mid)\ \{\ c\ y \to \mid t_2\mid\} & \rightsquigarrow & \mid t_2[t_1/y][t/f]\mid \\
 & \text{where} & t =_{\mathbf{df}} \lambda\, x.\, \mu\ f.\ x\ \{\ c\ y \to \mid t_2\mid\}
\end{array}
$$

Fig. 16. Erasure and reduction rules μ and σ

Operational semantics As primitives of the surface language, μ and σ expressions require erasure and reduction rules. These are given in Figure 16, and are based on the erasures and convertibility behavior of the underlying CDLE expressions mu and sigma (see Proposition 2). When applied to data formed from constructor c and a case branch for that constructor, both μ and σ reduce to the branch body with the (erasure of the) constructor argument t_1 replacing all occurrences of pattern variable y. In μ, the bound variable f is replaced by a lambda expression whose body is the same μ expression but with the scrutinee $c\ t_1$ replaced by the bound variable x. We denote the convertibility relation for the surface language as $=_{\beta\eta\mu}$, the reflexive transitive congruence closure of the usual $\beta\eta$-reduction rules of lambda calculus augmented with the rules of Figure 16.

5.4 Properties of elaboration

Our elaboration rules are *type preserving*, meaning that elaborated surface language expressions can be classified in CDLE by the elaborations of their surface language classifiers, and *value preserving*, meaning that the elaborations of surface language terms and of the terms they reduce to are definitionally equal.

Theorem 5 (Value preservation). *If* $\Gamma \vdash t_1 : T_1 \hookrightarrow t_1'$*,* $\Gamma \vdash t_2 : T_2 \hookrightarrow t_2'$*, and* $\mid t_1\mid =_{\beta\eta\mu} \mid t_2\mid$ *then* $\mid t_1'\mid =_{\beta\eta} \mid t_2'\mid$

Theorem 6 (Type preservation). *If* $\vdash \Gamma \hookrightarrow \Gamma'$ *then:*

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

The datatype system also enjoys a limited termination guarantee for closed data terms (a similar limitation holds for CDLE, see Section 3).

Theorem 7 (Normalization guarantee).
If $\Gamma \vdash \mathit{Ind}[D, R, c : T \to R] \dashv \Gamma, \mathit{IndEl}[D, R, c : T \to D, m, L, \Theta, \mathcal{E}]$ *and* $\Gamma, \mathit{IndEl}[D, R, c : T \to D, m, L, \Theta, \mathcal{E}] \vdash t : D \hookrightarrow t'$ *with* $|t|$ *a closed term, then* $|t'|$ *is call-by-name normalizing.*

6 Related work

TCBs in ITPs Many interactive theorem provers (ITPs) have large trusted computing bases (TCBs). For example, Coq's kernel is ∼30K OCaml LoC, and some provers like Agda [29] (∼100K Haskell LoC) have no kernel. But, there is much interest in verifying provers themselves [9, 17] and thus practical interest in keeping their kernels small [4].

Dagand and McBride [7] share both this goal and general method. They describe the elaboration inductive definitions, pattern matching, and recursion to a simpler core, Martin-Löf type theory extended with a universe of positive inductive types. In comparison, CDLE has no inductive primitives and elaboration produces monotonicity witnesses *á la* Matthes [24] for positivity checking.

Goguen et al. [16] show how *dependent pattern matching* [6] can be elaborated to the dependent eliminators of datatypes. While course-of-values pattern matching as discussed in this paper is in many respects less sophisticated than dependent pattern matching, an interesting point of comparison is the treatment of course-of-values induction. They implement it by providing as the inductive hypothesis $\mathsf{Below}_D\ P\ x$, a large tuple containing proofs that P holds for all subdata of x. Functions analysing a static number of cases may easily make use of this, but accessing a proof for dynamically computed subdata (e.g. the result of minusCV in divide) requires an inductive proof of a lemma such as $\mathsf{Below}_{\mathsf{Nat}}\ P\ (\mathsf{succ}\ n) \to P\ (\mathsf{minus}\ n\ m)$ (for any m), not required in our work.

Semantic termination checking Abel [1] showed how to extend 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 datatype declaration is defined with the usual notation and automatically supports course-of-values induction. Furthermore, the Mendler style of coding recursion, on which Cedille's termination checking is based, can be expressed using polymorphic typing alone, making it suitable for a paradigm of *strong functional programming* [34] (c.f. [33] for a demonstration of this Cedille). On the other hand, sized types allow for even more powerful forms of recursive definitions. In particular, the usual implementation of merge sort is definable using sized types but is not easily expressible as course-of-values recursion, as it involves recursion on terms that are not predecessors of the original list.

The Nax language, described by Ahn [3], also takes an approach to termination based on the Mendler style. In Nax, recursive functions are defined in terms of Mendler style recursion schemes, including course-of-values recursion, whereas in Cedille the μ-operator supports course-of-values *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 declarations to positive datatypes.

7 Conclusion and future work

We have presented a datatype subsystem for Cedille with type-based termination checking for recursive functions that supports course-of-values induction, an expressive proof scheme. This subsystem required no extension to Cedille's core theory CDLE, which we demonstrated by first deriving course-of-values induction for lambda encoded datatypes in CDLE using the generic framework of Firsov et al. [11], then giving a type- and value-preserving translation of the surface language constructs to these encodings.

One immediate usability concern is the proliferation of explicit type coercions in the case branches of μ- and σ-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 12; 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. [10]. For datatypes with multiple constructors, 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 lambda expressions are. This would allow users to derive reuse *manually* for datatypes and functions. More ambitiously, a higher level syntax (such as *ornaments* [25, 8]) would allow programmers to define, e.g., length-indexed lists in terms of ordinary lists by describing the function or relation on terms of the latter to the indices of the former. Such definitions could then be elaborated using the generic zero-cost reuse combinators of Diehl et al [10].

References

1. Abel, A.: Miniagda: Integrating sized and dependent types. Electronic Proceedings in Theoretical Computer Science **43**, 14–28 (Dec 2010). https://doi.org/10.4204/eptcs.43.2, http://dx.doi.org/10.4204/EPTCS.43.2
2. Abel, A., Coquand, T.: Failure of normalization in impredicative type theory with proof-irrelevant propositional equality (2019), https://arxiv.org/abs/1911.08174
3. Ahn, K.Y.: The nax language: Unifying functional programming and logical reasoning in a language based on mendler-style recursion schemes and term-indexed types (2014)

4. Appel, A.W.: Foundational proof-carrying code. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings. pp. 247–256. IEEE Computer Society (2001). https://doi.org/10.1109/LICS.2001.932501, https://doi.org/10.1109/LICS.2001.932501
5. Barthe, G., Frade, M.J., Giménez, E., Pinto, L., Uustalu, T.: Type-based termination of recursive definitions. Mathematical structures in computer science **14**(1), 97–141 (2004)
6. Coquand, T.: Pattern matching with dependent types. In: Informal proceedings of Logical Frameworks. vol. 92, pp. 66–79 (1992)
7. Dagand, P.E., McBride, C.: Elaborating inductive definitions (2012)
8. Dagand, P.É., McBride, C.: Transporting functions across ornaments. Journal of functional programming **24**(2-3), 316–383 (2014)
9. Davis, J., Myreen, M.O.: The reflective milawa theorem prover is sound (down to the machine code that runs it). J. Autom. Reason. **55**(2), 117–183 (2015). https://doi.org/10.1007/s10817-015-9324-6, https://doi.org/10.1007/s10817-015-9324-6
10. Diehl, L., Firsov, D., Stump, A.: Generic zero-cost reuse for dependent types. Proc. ACM Program. Lang. **2**(ICFP), 104:1–104:30 (jul 2018). https://doi.org/10.1145/3236799, http://doi.acm.org/10.1145/3236799
11. Firsov, D., Blair, R., Stump, A.: Efficient mendler-style lambda-encodings in cedille. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving. pp. 235–252. Springer International Publishing, Cham (2018)
12. Firsov, D., Diehl, L., Jenkins, C., Stump, A.: Course-of-value induction in cedille (2018)
13. Geuvers, H.: Induction is not derivable in second order dependent type theory. In: Abramsky, S. (ed.) Typed Lambda Calculi and Applications. pp. 166–181. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
14. Geuvers, H.: Proof assistants: History, ideas and future. Sadhana **34**(Part 1), 3–25 (2009). https://doi.org/10.1007/s12046-009-0001-5
15. Giménez, E.: Codifying guarded definitions with recursive schemes. In: Dybjer, P., Nordström, B., Smith, J. (eds.) Types for Proofs and Programs. pp. 39–59. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
16. Goguen, H., McBride, C., McKinna, J.: Eliminating dependent pattern matching. In: Futatsugi, K., Jouannaud, J., Meseguer, J. (eds.) Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 4060, pp. 521–540. Springer (2006). https://doi.org/10.1007/11780274_27, https://doi.org/10.1007/11780274_27
17. Harrison, J.: Towards self-verification of HOL light. In: Furbach, U., Shankar, N. (eds.) Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4130, pp. 177–191. Springer (2006). https://doi.org/10.1007/11814771_17, https://doi.org/10.1007/11814771_17
18. INRIA: The Coq proof assistant, version 8.7.0 (Oct 2017). https://doi.org/10.5281/zenodo.1028037, https://doi.org/10.5281/zenodo.1028037
19. Jenkins, C., McDonald, C., Stump, A.: Elaborating inductive definitions and course-of-values induction in cedille. https://arxiv.org/abs/1903.08233 (2019), unpublished manuscript
20. Kleene, S.: Classical Extensions of Intuitionistic Mathematics. In: Bar-Hillel, Y. (ed.) LMPS 2. pp. 31–44. North-Holland Publishing Company (1965)

21. Kopylov, A.: Dependent intersection: A new way of defining records in type theory. In: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science. pp. 86–. LICS '03, IEEE Computer Society, Washington, DC, USA (2003), http://dl.acm.org/citation.cfm?id=788023.789073
22. Lambek, J.: A fixpoint theorem for complete categories. Mathematische Zeitschrift **103**(2), 151–161 (1968). https://doi.org/10.1007/bf01110627, https://doi.org/10.1007/bf01110627
23. Matthes, R.: Monotone fixed-point types and strong normalization. In: Gottlob, G., Grandjean, E., Seyr, K. (eds.) 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, vol. 1584, pp. 298–312. Springer (1998). https://doi.org/10.1007/10703163_20
24. Matthes, R.: Tarski's fixed-point theorem and lambda calculi with monotone inductive types. Synthese **133**(1-2), 107–129 (2002). https://doi.org/10.1023/A:1020831825964, https://doi.org/10.1023/A:1020831825964
25. McBride, C.: Ornamental algebras, algebraic ornaments. Journal of functional programming (2010)
26. Mendler, N.P.: Recursive types and type constraints in second-order lambda calculus. In: Proceedings of the Symposium on Logic in Computer Science. pp. 30–36. (LICS '87), IEEE Computer Society, Los Alamitos, CA (June 1987)
27. Miquel, A.: The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In: Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications. pp. 344–359. TLCA'01, Springer-Verlag, Berlin, Heidelberg (2001), http://dl.acm.org.proxy.lib.uiowa.edu/citation.cfm?id=1754621.1754650
28. Mitchell, J.C.: Polymorphic type inference and containment. Inf. Comput. **76**(2/3), 211–249 (1988). https://doi.org/10.1016/0890-5401(88)90009-0
29. Norell, U.: Dependently typed programming in agda. In: Koopman, P.W.M., Plasmeijer, R., Swierstra, S.D. (eds.) Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures. Lecture Notes in Computer Science, vol. 5832, pp. 230–266. Springer (2008). https://doi.org/10.1007/978-3-642-04652-0_5, https://doi.org/10.1007/978-3-642-04652-0_5
30. Stump, A.: From realizability to induction via dependent intersection. Annals of Pure and Applied Logic **169**(7), 637–655 (2018)
31. Stump, A.: Syntax and semantics of Cedille. https://github.com/cedille/cedille/blob/master/docs/semantics/paper.pdf (2018)
32. Stump, A.: Syntax and typing for cedille core (2018)
33. Stump, A., Jenkins, C., Spahn, S., McDonald, C.: Strong functional pearl: Harper's regular-expression matcher in Cedille. Proc. ACM Program. Lang. **4**(ICFP), 122:1–122:25 (2020). https://doi.org/10.1145/3409004, https://doi.org/10.1145/3409004
34. Turner, D.A.: Elementary strong functional programming. In: Functional Programming Languages in Education, First International Symposium, FPLE'95, Nijmegen, The Netherlands, December 4-6, 1995, Proceedings. pp. 1–13 (1995). https://doi.org/10.1007/3-540-60675-0_35, https://doi.org/10.1007/3-540-60675-0_35
35. Uustalu, T., Vene, V.: Mendler-style inductive types, categorically. Nord. J. Comput. **6**(3), 343 (1999)

36. Uustalu, T., Vene, V.: Coding recursion a la Mendler (extended abstract). In: Proc. of the 2nd Workshop on Generic Programming, WGP 2000, Technical Report UU-CS-2000-19. pp. 69–85. Dept. of Computer Science, Utrecht University (2000)
37. Wadler, P.: Theorems for free! In: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture. p. 347–359. FPCA '89, ACM, New York, NY, USA (1989). https://doi.org/10.1145/99370.99404, https://doi.org/10.1145/99370.99404