The Cedilleum Language Specification

Syntax, Typing, Reduction, and Elaboration

Christopher Jenkins

October 20, 2018

1 Introduction

The document describes *Cedilleum*, a general-purpose dependently typed programming language with inductive datatypes. Unlike most languages of this description, the underlying theory of Cedilleum is *not* the Calculus of Inductive Constructions (CIC)[PM15]. Instead, Cedilleum is designed so that it may easily be translated to *Cedille Core* – a compact core theory in which induction is derivable for lambda-encoded datatypes – while still providing high-level features like pattern-matching and recursive definitions. That said, the formal specification of Cedilleum as a self-contained language has a lot in common with CIC – see in particular Section 8 of [Inr18], which served as the basic template for much of this document's formal development.

1.1 Data-type Declarations

Before diving into the details, let us take a bird's-eye view of the language by showing some simple example data-type definitions and functions over them.

```
-- Non-recursive
data Bool: ⋆ =
  | tt: Bool
  | ff: Bool
.
-- Recursive
data Nat: ⋆ =
  | zero: Nat
  | succ: Nat → Nat
.
-- Recursive, parameterized, indexed
data Vec (A: ⋆): Nat → ⋆ =
  | vnil : Vec zero
  | vcons: ∀ n: Nat. A → Vec n → Vec (succ n)
.
```

Figure 1: Definition of natural numbers and length-indexed lists

Figure 1 shows some definitions of inductive datatypes, and modulo differences in syntax should seem straightforward to programmers that have used languages like Agda, Idris, or Coq. The key things to note are:

- In constructor type signatures, recursive occurrences of the inductive data-type being defined (such as in `suc :` <u>`Nat`</u> `→ Nat`) must be positive, *but not strictly positive.*

- In parameterized types (like `Vec` with parameter `(A: ⋆)`) occurrences of the inductive type being defined are not written applied to its parameters.

  For example, the constructor declaration `vnil : Vec zero` results in the term `vnil` having type `∀ A: ⋆. Vec ·A ⋆` (with · denoting type application)

- In the constructor declaration `vcons : ∀ n: Nat. A → Vec n → Vec (succ n)`, the argument `n` is *computationally irrelevant* (also called *erased*). This is because it is introduced by the irrelevant dependent function former ∀, as opposed to the relevant function former Π. More will be said of this when we discuss the type system of Cedilleum, but for now it suffices to say that this idea comes from the *Implicit Calculus of Constructions*[Miq01]

1.2 Function Definitions

```
-- Non-recursive
ite : ∀ X: ⋆. Bool → X → X → X
  = Λ X. λ b. λ then. λ else. μ' b {
      | tt ↦ then
      | ff ↦ else
      }.
pred : Nat → Nat
  = λ n. μ' n {
      | zero    ↦ n
      | succ n' ↦ n'
      }
-- Recursive
add : Nat → Nat → Nat
  = λ n. λ m.
      μ add-rec. n @(λ x: Nat. Nat) {
      | zero   ↦ m
      | succ r ↦ succ (add-rec r)
      }.
```

Figure 2: Functions over inductive datatypes

Figure 2 shows functions defined over inductive datatypes using pattern matching and recursion. The first difference to note between the definitions is that functions `ite` and `pred` perform "mere" pattern matching on their arguments by using `μ'`, whereas `add` uses μ which provides combined pattern-matching and fix-point recursion. In `add`, μ binds `add-rec` as the name of the fixpoint function for recursion on `n`. From this alone the reader might expect that `μ'` is merely syntactic sugar for the more verbose μ. Actually the difference is a bit more subtle that this, as we will see shortly.

The first major departure of Cedilleum from other languages with inductive datatypes can be seen in the type of `add-rec`. Ordinarily, its type would be `Π x: Nat. Nat` (corresponding to the motive `(λ x: Nat. Nat)`) – but in Cedilleum, its type is `Nat/add-rec → Nat` (where we read `Nat/add-rec` as a single identifier) and by extension for the expression `add-rec r` to be well-typed, the `r` bound in the pattern `succ r` must have type `Nat/add-rec` and not the more usual `Nat`. Why? For recursive functions in Cedilleum, *termination is guaranteed by the type system* and not by a separate syntactic check for structurally-decreasing arguments. The type `Nat/add-rec` indicates the types of those terms which the function `add-rec` may legally take as arguments without risking non-termination, and within a case branch variables of this type are introduced in the place of recursive arguments to constructors – that is, `r` in the pattern `succ r`. In Section 1.3, we will see more of how to use "recursive occurence" types like `Nat/add-rec`, including how to do further pattern matching on them and perform conversion back to the original type. For now it suffices to consider them an artifice for type-guided termination.

```
-- Recursive, parameterized, indexed
vappend : ∀ A: ⋆. ∀ m: Nat. ∀ n: Nat. Vec ·A m → Vec ·A n → Vec ·A (add m n)
  = Λ A. Λ m. Λ n. λ xs. λ ys.
      μ vappend-rec. xs @(λ i: Nat. λ zs: Vec ·A i. Vec ·A (add i n)) {
      | vnil            ↦ ys
      | vcons -m' x xs' ↦ [ zs = vappend-rec -m' xs' ] - vcons -(add m' n) x zs
      }.
```

Figure 3: Dependent functions over inductive datatypes

Figure 3 shows the classic dependent function `vappend` over length-indexed lists `Vec`. Like `add`, it is defined by fixpoint recursion, here over the argument `xs`, and as with `add-rec` the fixpoint function `vappend-rec` has type `∀ i: Nat. Π zs: Vec/vappend-rec i. Vec ·A (add i n)`, where the ∀ indicates that `i` is an *irrelevant* argument. Note again the missing parameter `A` in the type `Vec/vappend-rec i` – this is not a typo, but rather an indication that `A` is "baked-in" to the type `Vec/vappend-rec`. Aside from this the two cases of `vappend` are mostly straightforward: in the `vnil` branch the expected type is `Vec ·A (add zero n)` which converts to `Vec ·A n`, so `ys` suffices; in the `vcons` branch we bind subdata `m': Nat`, `x: A`, and `xs': Vec/vappend-rec m'`, with `-m'` indicating that `m'` is bound *irrelevantly*, then we make a local biding `zs` by invoking `vappend` recursively on `m'` and `xs'` (where again the `-m'` indicates `m'` is an *irrelevant* argument to `vappend-rec`) before producing a result whose type is convertible with the expected `Vec ·A (add (suc m') n)`.

1.3 Histomorphic Recursion

We now return to the strange types `Nat/add-rec` and `Vec/append-rec`. The reader may well ask whether they serve any purpose other than marking what recursive calls are legal in a function – and the answer is yes! Cedilleum uses types like these to support *histomorphic* or *course-of-values* recursion, which is to say recursion schemes that can dig arbitrarily deeply into the recursive occurences of data to compute results. As a motivating example, we now show how we can define division on natural numbers in Cedilleum using histomorphic recursion.

Languages with inductive datatypes and recursive function definitions that also wish to have their type systems interpreted as sound logics must address the issue of *termination*, because the principle of general recursion `∀ A: ⋆. (A → A) → A` allows any one to inhabit every type. To that end, most such languages perform some terminiation check separate from type checking that usually involves making sure that arguments to recursive calls are getting *structurally smaller* to ensure that eventually a base case is reached. This check is necessarily conservative (i.e. it will not accept some terminating function), and the classic example of a function that is not "obviously" terminating is division on natural numbers by iterated subtraction. Intuitively, we understand that subtracting `n` from `m` never produces a number larger than `m` – but it can be tricky to explain this to the termination checker!

The definition of division is given in Figure 4. We start by defining some straightforward preliminaries: `iterate` to apply a function some number of times to a base case, `minus` for subtraction, and `lt` a (boolean) predicate testing whether its first argument is less than its second. In `divide` we kick off recursion on the numerator `m`, and in the base case where it is `zero` we return `zero`. When it is non-zero, we locally define `m'` by using `fromNat/divide-rec` to cast the sub-data `r` of type `Nat/divide-rec` back to `Nat`, where `fromNat/divide-rec` is automatically in-scope of the body of the μ definition. We then use `ite` (mnemonic for "if-then-else") to see whether our current numerator `m'` is less than the denominator, and if so return `zero`.

The real action happens when the current numerator is larger than our denominator, as we must make a recursive call to `divide-rec` after subtracting (via iterated predecessor) `n` from `m'`. We declare a version

```
iterate : ∀ R: ⋆. Nat → (R → R) → R → R
  = Λ R. λ n. λ f. λ x.
      μ iterate-rec. n @(λ _: Nat. Nat) {
      | zero    ↦ r
      | succ r' ↦ f (iterate-rec r')
      }.
minus : Nat → Nat → Nat
  = λ m. λ n.
      μ minus-rec. n @(λ _: Nat. Nat) {
      | zero   ↦ m
      | succ r ↦ pred (minus-rec r)
      }.
lt : Nat → Nat → Bool
  = λ m. λ n.
      μ' (minus m n) {
      | zero   ↦ ff
      | succ r ↦ tt
      }.
divide : Nat → Nat → Nat
  = λ m. λ n.
      μ divide-rec. m @(λ _: Nat. Nat) {
      | zero   ↦ zero
      | succ r ↦ [ m' = succ (fromNat/divide-rec r) ]
        - ite (lt m' n) zero
            ([ pred' = λ x: Nat/divide-rec.
                 μ' x {| zero ↦ x | succ x' ↦ x'} ]
             - succ (divide-rec (iterate (pred n) pred' r)))
      }.
```

Figure 4: Histomorphic recursion and division

of predecessor `pred'` to operator over a term of type `Nat/divide-rec` and define it in terms of μ`'`. This is the key to enabling histomorphic recursion – μ`'` allows for pattern matching on both the "concrete" `Nat` and "abstracted" `Nat/divide-rec` and can be iterated arbitrarily (here `pred n` times), but it *cannot* be used to kick off its own recursion and *cannot* produce terms of the recursion-abstracted type `Nat/divide-rec` that is larger than its input. Having define `pred'` this way, the rest of the definition is straightforward: recursively call `divide-rec` after performing "abstracted" subtraction via iterated predecessor on our denominator (`iterate (pred n) pred' r`), and increment the result.

2 Syntax

Identifiers Figure 5 gives the metavariables used in our grammar for identifiers. We consider all identifiers as coming from two distinct lexical "pools" – regular identifiers (consisting of identifiers *id* given for modules and definitions, term variables u, and type variables X) and kind identifiers κ. In Cedilleum source files (as in the parent language Cedille) kind variables should be literally prefixed with κ – the suffix can be any string that would by itself be a legal non-kind identifier. For example, `myDef` is a legal term / type variable and a legal name for a definition, whereas κ`myDeff` is only legal as a kind definition.

Untyped Terms The grammar of pure (untyped) terms the untyped λ-calculus augmented with a primitives for combination fixed-point and pattern-matching definitions (and an auxiliary pattern-matching con-

id			identifiers for definitions
u, c			term variables
X			type variables
κ			kind variables
x	$::=$	$id \mid u \mid X$	non-kind variables
y	$::=$	$x \mid \kappa$	all variables

Figure 5: Identifiers

p	$::=$	u	variables
		$\lambda\, u.\, p$	functions
		$p\ p$	applications
		$\mu\ u\ .\ p\ \{pcase^*\}$	fixed-point and pattern matching
		$\mu'\ p\ \{pcase^*\}$	simple pattern matching
$pcase$	$::=$	$\mid u\ u^* \mapsto p$	

Figure 6: Untyped terms

struct).

mod	$::=$	**module** id **.** $imprt^*\ cmd^*$	module declarations
$imprt$	$::=$	**import** id **.**	module imports
cmd	$::=$	$defTermOrType$	definitions
		$defDataType$	
		$defKind$	
$defTermOrType$	$::=$	$id\ checkType^? = t$ **.**	term definition
		$id : K = T$ **.**	type definition
$defKind$	$::=$	$\kappa = K$	kind definition
$defDataType$	$::=$	**data** $id\ param^* : K = constr^*$ **.**	datatype definitions
$checkType$	$::=$	$: T$	annotation for term definition
$param$	$::=$	$(x : C)$	
$constr$	$::=$	$\mid id : T$	

Figure 7: Modules and definitions

Modules and Definitions All Cedilleum source files start with production *mod*, which consists of a module declaration, a sequence of import statements which bring into scope definitions from other source files, and a sequence of *commands* defining terms, types, and kinds. As an illustration, consider the first few lines of a hypothetical `list.ced`:

```
module list .

import nat .
```

Imports are handled first by consulting a global options files known to the Cedilleum compiler (on *nix systems `~/.cedille/options`) containing a search path of directories, and next (if that fails) by searching the directory containing the file being checked.

Term and type definitions are given with an identifier, a classifier (type or kind, resp.) to check the definition against, and the definition. For term definitions, giving classifier (i.e. the type) is optional. As an example, consider the definitions for the type of Church-encoded lists and two variants of the nil constructor, the first with a top-level type annotation and the second with annotations sprinkled on binders:

```
cList : ⋆ → ⋆
      = λ A : ⋆ . ∀ X : ⋆ . (A → X → X) → X → X .

cNil  : ∀ A : ⋆ . cList · A
      = Λ A . Λ X . λ c . λ n . n .
cNil' = Λ A : ⋆ . Λ X : ⋆ . λ c : A → X → X . λ n : X . n .
```

Kind definitions are given without classifiers (all kinds have super-kind □), e.g. `κfunc = ⋆ → ⋆`

Inductive datatype definitions take a set of *parameters* (term and type variables which remain constant throughout the definition) well as a set of *indices* (term and type variables which *can* vary), followed by zero or more constructors. Each constructor begins with "|" (though the grammar can be relaxed so that the first of these is optional) and then an identifier and type is given. As an example, consider the following two definitions for lists and vectors (length-indexed lists).

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

data Nat : ⋆ =
  | zero : Nat
  | suc  : Nat → Nat
  .

data List (A : ⋆) : ⋆ =
  | nil  : List
  | cons : A → List → List
  .
data Vec (A : ⋆) : Nat → ⋆ =
  | vnil  : ∀ n : Nat . {n ≃ zero} ⇒ Vec n
  | vcons : ∀ n : Nat . ∀ m : Nat . A → Vec n → {m ≃ suc n} ⇒ Vec m
  .
```

Types and Kinds In Cedilleum, the expression language is stratified into three main "classes": kinds, types, and terms. Kinds and types are listed in Figure 8 and terms are listed in Figure 9 along with some auxiliary grammatical categories. In both of these figures, the constructs forming expressions are listed from lowest to highest precedence – "abstractors" ($\lambda\ \Lambda\ \Pi\ \forall$) bind most loosely and parentheses most tightly. Associativity is as-expected, with arrows ($\rightarrow \Rightarrow$) and applications being left-associative and abstractors being right-associative.

The language of kinds and types is similar to that found in the Calculus of Implicit Constructions[1]. Kinds are formed by dependent and non-dependent products (Π and $\rightarrow$) and a base kind for types which can classify terms ($\star$). Types are also formed by the usual (dependent and non-dependent) products (Π and $\rightarrow$) and also *implicit* products ($\forall$ and $\Rightarrow$) which quantify over erased arguments (that is, arguments that disappear at run-time). Π-products are only allowed to quantify over terms as all types occurring in terms

[1] Cite

Sorts S	$::=$	$\square$	sole super-kind
		K	kinds
Classifiers C	$::=$	K	kinds
		T	types
Kinds K	$::=$	$\Pi\ x : C\ .\ K$	explicit product
		$C \to K$	kind arrow
		$\star$	the kind of types that classify terms
		(K)	
Types T	$::=$	$\Pi\ x : T\ .\ T$	explicit product
		$\forall\ x : C\ .\ T$	implicit product
		$\lambda\ x : C\ .\ T$	type-level function
		$T \Rightarrow T'$	arrow with erased domain
		$T \to T'$	normal arrow type
		$T \cdot T'$	application to another type
		$T\ t$	application to a term
		$\{\ p\ \simeq\ p'\}$	untyped equality
		(T)	
		X	type variable
		$\bullet$	hole for incomplete types

Figure 8: Kinds and types

are erased at run-time, but $\forall$-products can quantify over types *and* terms because terms can be erased. Meanwhile, non-dependent products ($\to$ and $\Rightarrow$) can only "quantify" over terms because non-dependent type quantification does not seem particularly useful. Besides these, Cedilleum features type-level functions and applications (with term and type arguments), and a primitive equality type for untyped terms. Last of all is the "hole" type ($\bullet$) for writing partial type signatures or incomplete type applications. There are term-level holes as well, and together the two are intended to help facilitate "hole-driven development": any hole automatically generates a type error and provides the user with useful contextual information.

We illustrate with another example: what follows is a module stub for **DepCast** defining dependent casts – intuitively, functions from $a : A$ to $B\ a$ that are also equal[2] to identity – where the definitions `CastE` and `castE` are incomplete.

```
module DepCast .

CastE ◁ Π A : ⋆ . (A → ⋆) → ⋆ = ● .

castE ◁ ∀ A : ⋆ . ∀ B : A → ⋆ . CastE · A · B ⇒ Π a : A . B a = ● .
```

Annotated Terms Terms can be explicit and implicit functions (resp. indicated by λ and Λ) with optional classifiers for bound variables, let-bindings, applications $t\ t'$, t -t', and $t\ \cdot T$ (resp. to another term, an erased term, or a type). In addition to this there are a number of useful operators for equaltional reasoning, type casting, providing annotations, and pattern matching. Each operator will be discussed in more detail in Section 4, but a few concrete programs in Cedilleum are given below merely to give a better idea of the syntax of the language.

```
isvnil : ∀ A : ⋆ . ∀ n : Nat . Vec · A n → Bool
       = Λ A . Λ n . λ xs .
```

[2]Module erasure, discussed below

Subjects s	$::=$	t	term
		T	type
Terms t	$::=$	$\lambda\, x\ class^?\mathbf{.}\, t$	normal abstraction
		$\Lambda\, x\ class^?\mathbf{.}\, t$	erased abstraction
		$[\ defTermOrType\ ]\ \mathbf{-}\ t$	let definitions
		$\rho\ t\ \mathbf{-}\ t'$	equality elimination by rewriting
		$\phi\ t\ \mathbf{-}\ t'\ \{t''\}$	type cast
		$\chi\ T\ \mathbf{-}\ t$	check a term against a type
		$\delta\ \mathbf{-}\ t$	ex falso quodlibet
		$\theta\ t\ t'^*$	elimination with a motive
		$t\ t'$	applications
		$t\ \mathbf{-}t'$	application to an erased term
		$t\ \cdot T$	application to a type
		$\beta\ \{t\}$	reflexivity of equality
		$\varsigma\ t$	symmetry of equality
		$\mu\ u\ .\ t\ motive^?\ \{case^*\}$	type-guarded pattern match and fixpoint
		$\mu'\ t\ motive^?\ \{case^*\}$	auxiliary pattern match
		u	term variable
		(t)	
		$\bullet$	hole for incomplete term
$case$	$::=$	$\mid c\ vararg^* \mapsto t$	pattern-matching cases
$vararg$	$::=$	u	normal constructor argument
		$\mathbf{-}u$	erased constructor argument
		$\cdot X$	type constructor argument
$class$	$::=$	$\mathbf{:}\ C$	
$motive$	$::=$	$@\ T$	motive for induction

Figure 9: Annotated Terms

```
      μ' xs @(Λ n . λ xs . Bool)
        { | vnil -n -eq ↦ tt
          | vcons -n -m x xs -eq ↦ ff
        }
vlength : ∀ A : ⋆ . ∀ n : Nat . Vec · A n → Nat
        = Λ A . Λ n . λ xs .
           μ len . xs @(Λ n . λ x . Nat)
            { | vnil -n -eq ↦ zero
              | vcons -n -m x xs -eq ↦ suc (len -n xs)
            }
```

3 Erasure

The definition of the erasure function given in Figure 10 takes the annotated terms from Figures 8 and 9 to the untyped terms of Figure 6. The last two equations indicate that any type or erased arguments in the the zero or more *vararg*'s of pattern-match case are indeed erased. The additional constructs introduced in the annotated term language such as β, ϕ, and ρ, are all erased to the language of pure terms.

$$
\begin{array}{lcl}
|x| & = & x \\
|\star| & = & \star \\
|\square| & = & \square \\
|\beta\ \{t\}| & = & |t| \\
|\delta\ t| & = & |t| \\
|\chi\ T^{?}\text{-}\ t| & = & |t| \\
|\theta\ t\ t'^{*}| & = & |t|\ |t'^{*}| \\
|\varsigma\ t| & = & |t| \\
|t\ t'| & = & |t|\ |t'| \\
|t\ \text{-}t'| & = & |t| \\
|t\ \cdot T| & = & |t| \\
|\rho\ t\ \text{-}\ t'| & = & |t'| \\
|\forall\, x\!:\!C.\, C'| & = & \forall\, x\!:\!|C|.\, |C'| \\
|\Pi\, x\!:\!C.\, C'| & = & \Pi\, x\!:\!|C|.\, |C'| \\
|\lambda\, u\!:\!T.\, t| & = & \lambda\, u.\, |t| \\
|\lambda\, u.\, t| & = & \lambda\, u.\, |t| \\
|\lambda\, X\!:\!K.\, C| & = & \lambda\, X\!:\!|K|.\, |C| \\
|\Lambda\, x\!:\!C.\, t| & = & |t| \\
|\phi\ t\ \text{-}\ t'\ \{t''\}| & = & |t''| \\
|[x = t : T]|\ \text{-}\ t'| & = & (\lambda\, x.\, |t'|)\ |t| \\
|[X = T : K]\ \text{-}\ t| & = & |t| \\
|\{t \simeq t'\}|| & = & \{|t| \simeq |t'|\} \\
|\mu\ u,\ .\ t\ \mathit{motive}^{?}\ \mathbf{\{}\mathit{case}^{*}\mathbf{\}}| & = & \mu\ u\ .\ |t|\ \mathbf{\{}|\mathit{case}^{*}|\mathbf{\}} \\
|\mu'\ t\ \mathit{motive}^{?}\ \mathbf{\{}\mathit{case}^{*}\mathbf{\}}| & = & \mu'\ |t|\ \mathbf{\{}|\mathit{case}^{*}|\mathbf{\}} \\
 & & \\
|\mathit{id}\ \mathit{vararg}^{*} \mapsto t| & = & \mathit{id}\ |\mathit{vararg}^{*}|\ \mapsto |t| \\
 & & \\
|\text{-}u| & = & \\
|\cdot T| & = &
\end{array}
$$

Figure 10: Erasure for annotated terms

4 Type System (sans Inductive Datatypes)

Figure 11: Contexts

$$\text{Typing contexts } \Gamma \quad ::= \quad \emptyset \mid x\!:\!C, \Gamma \mid x = s\!:\!C, \Gamma$$

The inference rules for classifying expressions in Cedilleum are stratified into two judgments. Figure 12 gives the uni-directional rules for ensuring types are well-kinded and kinds are well-formed. Future versions of Cedilleum will allow for bidirectional checking for both typing *and* sorting, allowing for a unification of these two figures. Most of these rules are similar to what one would expect from the Calculus of Implicit Constructions, so we focus on the typing rules unique to Cedilleum.

The typing rule for ρ shows that ρ is a primitive for rewriting by an (untyped) equality. If t is an expression that synthesizes a proof that two terms t_1 and t_2 are equal, and t' is an expression synthesizing type $[t_1/x]\ T$ (where, as per the footnote, t_1 does not occur in T), then we may essentially rewrite its type to $[t_2/x]\ T$. The rule for β is reflexivity for equality – it witnesses that a term is equal to itself, provided that

[4] Where we assume t does not occur anywhere in T

[4] Where $\texttt{tt} = \lambda\, x.\, \lambda\, y.\, x$ and $\texttt{ff} = \lambda\, x.\, \lambda\, y.\, y$

$$\frac{}{\Gamma \vdash \star : \Box} \qquad \frac{\Gamma \vdash C : S \quad \Gamma, y : C \vdash C' : S'}{\Gamma \vdash \Pi\, y{:}C.\, C' : S'} \qquad \frac{\Gamma \vdash C : S \quad \Gamma, y : C \vdash C' : \star}{\Gamma \vdash \forall\, y{:}C.\, C' : \star}$$

$$\frac{FV(p\ p') \subseteq dom(\Gamma)}{\Gamma \vdash \{p \simeq p'\} : \star} \qquad \frac{}{\Gamma \vdash \kappa : \Gamma(\kappa)} \qquad \frac{}{\Gamma \vdash X : \Gamma(X)}$$

$$\frac{\Gamma \vdash \Pi\, x{:}C.\, K : \Box \quad \Gamma, x{:}C \vdash T : K}{\Gamma \vdash \lambda\, x{:}C.\, T : \Pi\, x{:}C.\, K} \qquad \frac{\Gamma \vdash T : \Pi\, x{:}K.\, K' \quad \Gamma \vdash T' : K}{\Gamma \vdash T\ \cdot T' : [T'/x]K'} \qquad \frac{\Gamma \vdash T : \Pi\, x{:}T'.\, K \quad \Gamma \vdash_{\Downarrow} t : T'}{\Gamma \vdash T\ t : [t/x]K}$$

Figure 12: Sort checking $\boxed{\Gamma \vdash C : S}$

$$\frac{}{\Gamma \vdash_{\delta} u : \Gamma(u)} \qquad \frac{\Gamma \vdash T : K \quad \Gamma, x{:}T \vdash_{\delta} t : T'}{\Gamma \vdash_{\delta} \lambda\, x{:}T.\, t : \Pi\, x{:}T.\, T'} \qquad \frac{\Gamma, x{:}T \vdash_{\Downarrow} t : T'}{\Gamma \vdash_{\Downarrow} \lambda\, x.\, t : \Pi\, x{:}T.\, T'}$$

$$\frac{\Gamma \vdash C : S \quad x \notin FV(|t|) \quad \Gamma, x{:}C \vdash_{\delta} t : T}{\Gamma \vdash_{\delta} \Lambda\, x{:}C.\, t : \forall\, x{:}C.\, T} \qquad \frac{x \notin FV(|t|) \quad \Gamma, x{:}C \vdash_{\delta} t : T}{\Gamma \vdash_{\Downarrow} \Lambda\, x.\, t : \forall\, x{:}C.\, T} \qquad \frac{\Gamma \vdash_{\Uparrow} t : \Pi\, x{:}T'.\, T \quad \Gamma \vdash_{\Downarrow} t' : T'}{\Gamma \vdash_{\delta} t\ t' : [t'/x]T}$$

$$\frac{\Gamma \vdash_{\Uparrow} t : \forall\, X{:}K.\, T' \quad \Gamma \vdash T : K}{\Gamma \vdash_{\delta} t\ \cdot T : [T/X]T'} \qquad \frac{\Gamma \vdash_{\Uparrow} t : \forall\, x{:}T'.\, T \quad \Gamma \vdash_{\Downarrow} t' : T'}{\Gamma \vdash_{\delta} t\ \text{-}t' : [t'/x]T} \qquad \frac{\Gamma \vdash_{\Uparrow} t : T' \quad |T'| =_{\beta} |T|}{\Gamma \vdash_{\Downarrow} t : T}$$

$$\frac{\Gamma \vdash T : K \quad \Gamma \vdash_{\Downarrow} t : T \quad \Gamma, id = t{:}T \vdash_{\delta} t' : T'}{\Gamma \vdash_{\delta} [id : T = t]\ \text{-}\ t' : T'} \qquad \frac{\Gamma \vdash_{\Uparrow} t : T \quad \Gamma, id = t{:}T \vdash_{\delta} t' : T'}{\Gamma \vdash_{\delta} [id = t]\ \text{-}\ t' : T'} \qquad \frac{\Gamma \vdash_{\Uparrow} t : \{t_1 \simeq t_2\} \quad \Gamma \vdash_{\Uparrow} t' : [t_1/x]\ T}{\Gamma \vdash_{\delta} \rho\ t\ \text{-}\ t' : [t_2/x]\ T}\ {}^{3}$$

$$\frac{\Gamma \vdash K : \Box \quad \Gamma \vdash T : K \quad \Gamma, id = T{:}K \vdash_{\delta} t' : T'}{\Gamma \vdash_{\delta} [id : K = T]\ \text{-}\ t' : T'} \qquad \frac{\Gamma \vdash \{t' \simeq t'\} : \star}{\Gamma \vdash_{\Downarrow} \beta\{t\} : \{t' \simeq t'\}} \qquad \frac{\Gamma \vdash_{\delta} t : \{t_1 \simeq t_2\}}{\Gamma \vdash_{\delta} \varsigma\ t : \{t_2 \simeq t_1\}}$$

$$\frac{\Gamma \vdash_{\Downarrow} t : \{|t_1| \simeq |t_2|\} \quad \Gamma \vdash_{\delta} t_1 : T}{\Gamma \vdash_{\delta} \phi\ t\ \text{-}\ t_1\ \{t_2\} : T} \qquad \frac{\Gamma \vdash_{\Downarrow} t : T}{\Gamma \vdash_{\Uparrow} \chi\ T\ \text{-}\ t : T} \qquad \frac{\Gamma \vdash_{\Downarrow} t : \{\mathtt{tt} \simeq \mathtt{ff}\}}{\Gamma \vdash_{\Downarrow} \delta\ \text{-}\ t : T}\ {}^{4}$$

$$\frac{\Gamma \vdash_{\Uparrow} t : \mathtt{??} \quad \Gamma \vdash_{?} t'^{*} : \mathtt{??}}{\Gamma \vdash_{\Downarrow} \theta\ t\ t'^{*} : T}$$

Figure 13: Type checking $\boxed{\Gamma \vdash_{\delta} s : C}$ (sans inductive datatypes)

the type of the equality is well-formed. The rule for ς is symmetry for equality. Finally, ϕ acts as a "casting" primitive: the rule for its use says that if some term t witnesses that two terms t_1 and t_2 are equal, and t_1 has been judged to have type T, then intuitively t_2 can also be judged to have type T. (This intuition is justified by the erasure rule for ϕ – the expression erases to $|t_2|$). The last rule involving equality is for δ, which witnesses the logical principle *ex falso quodlibet* – if a certain impossible equation is proved (namely that the two Church-encoded booleans `tt` and `ff` are equal), then *any* type desired is inhabited.

The two remaining primitives are not essential to the theory but are useful additions for programmers. The rule for χ allows the user to provide an explicit top-level annotation for a term, and θ embodies "elimination with a motive", using the expected type of an application to infer some type arguments. (TODO)

5 Inductive Datatypes

Before we can provide the typing rules for introduction and usage of inductive datatypes, some auxiliary definitions must be given. The syntax for these, and the structure of this entire section, borrows heavily from the conventions of the Coq documentation[5]. The author believes it is worthwhile to restate this development in terms of the Cedilleum type system, rather than merely pointing readers to the Coq documentation and asking them to infer the differences between the two systems.

To begin with, the production *defDataType* gives the concrete syntax for datatype definitions, but it is not a very useful notation for representing one in the abstract syntax tree. In our typing rules we will instead use the notation $\mathtt{Ind}_M[p](\Gamma_I := \Sigma)$, where

- M is a meta-variable ranging over constant labels "C" and "A" (used to distinguish **c**oncrete and **a**bstracted inductive definitions – more on this below)
- p is the number of **p**arameters of the inductive definition
- Γ_I is a typing context binding *one* type variable I, the inductive type being defined
- Σ is a typing context containing the n data constructors $c_1, ..., c_n$ of I.

For example, consider the `List` and `Vec` definitions from Section 2. These will be represented in the AST as

$$\mathtt{Ind}_{\mathrm{C}}[1](\mathit{List} : \star \to \star := \begin{array}{lll} \mathit{nil} & : & \forall A : \star . \mathit{List} \cdot A \\ \mathit{cons} & : & \forall A : \star . A \to \mathit{List} \cdot A \to \mathit{List} \cdot A \end{array})$$

and

$$\mathtt{Ind}_{\mathrm{C}}[1](Vec : \star \to Nat \to \star := \begin{array}{lll} \mathit{vnil} & : & \forall A : \star . \forall n : Nat . \{n \simeq \mathit{zero}\} \Rightarrow Vec \cdot A\ n \\ \mathit{vcons} & : & \forall A : \star . \forall n : Nat . \forall m : Nat . A \to Vec \cdot A\ n \to \{m \simeq \mathit{succ}\ n\} \Rightarrow Vec \cdot A\ m \end{array})$$

All inductive types the user will define will be concrete inductive defintions, and have global scope. Abstracted definitions are automiatically generated during fix-point pattern matching, and have local scope.

For an inductive datatype definition to be well-formed, it must satisfy the following conditions (each of which is explained in more detail in Subsections 5.1 and 5.2):

- The kind of I must be (at least) a *p-arity of kind* $\star$.
- The types of each $id \in \Sigma$ must be *types of constructors of* I
- The definition must satisfy the *non-strict* positivity condition.

Similarly, the notation in the grammar of Cedilleum μ' and μ for pattern matching is inconvenient, and we will represent them in the AST as resp. $\mu'(t, P, t_{i=1..n})$ and $\mu(x_{\mathrm{rec}}, I', x_{\mathrm{to}}, t, P, t_{i=1..n})$. Translation from the form given in the grammar to this form is discussed in detail below, but is as expected. In particular, we enforce that patterns are exhaustive and non-overlapping, and that I' and x_{to} (which are not present in the grammar but an automatically generated identifier) are fresh w.r.t the global and local context. For example, consider the pattern-matches given in the code listings for `isvnil` and `vlength` above. These would be translated into the AST as

$$\mu'(xs, \Lambda\, n.\, \lambda\, x.\, Bool, \begin{array}{l} \Lambda\, n.\, \Lambda\, eq.\, tt \\ \Lambda\, n.\, \Lambda\, m.\, \lambda\, x.\, \lambda\, xs.\, \Lambda\, eq.\, ff \end{array})$$

and

$$\mu(len, Vec/len, toVec/len, xs, \Lambda\, n.\, \lambda\, x.\, Nat, \begin{array}{l} \Lambda\, n.\, \Lambda\, eq.\, zero \\ \Lambda\, n.\, \Lambda\, m.\, \lambda\, x.\, \lambda\, xs.\, \Lambda\, eq.\, suc\ (len\ \text{-}n\ xs) \end{array})$$

[5]https://coq.inria.fr/refman/language/cic.html#inductive-definitions

In general, the generated name for I' and x_{to} that users will write in Cedilleum programs will be of the form "I/x_{rec}" and "toI/x_{rec}".

For a pattern construct (μ or μ') in the AST to be well-formed, it must satisfy the following conditions (each of which is, again, explained in more detail in Subsections 5.3, 5.5, and 5.6):

- The motive P must be well-kinded
- P must be a legal motive to be used in eliminating the inductive type I of the scrutinee t
- Each branch t_i must have the type expected given the constructor $c_i \in \Sigma$ and the motive P.

5.1 Auxiliary Definitions

Contexts To ease the notational burden, we will introduce some conventions for writing contexts within terms and types.

- We write $\lambda\,\Gamma$, $\Lambda\,\Gamma$, $\forall\,\Gamma$, and $\Pi\,\Gamma$ to indicate some form of abstraction over each variable in Γ. For example, if $\Gamma = x_1\!:\!T_1, x_2\!:\!T_2$ then $\lambda\,\Gamma.\,t = \lambda\,x_1\!:\!T_1.\,\lambda\,x_2\!:\!T_2.\,t$. Additionally, we will also write $\overset{\Pi}{\forall}\,\Gamma$ to indicate an arbitrary mixture of Π and $\forall$ quantified variables. Note that *if* $\overset{\Pi}{\forall}\,\Gamma$ *occurs multiple times within a definition or inference rule*, the intended interpretation is that *all occurrences have the same mixture of* Π *and* $\forall$ *quantifiers.*
- $\|\Gamma\|$ denotes the length of Γ (the number of variables it binds)
- We write $s\ \Gamma$ to indicate the sequence of variable arguments in Γ given as arguments to s. Implicit in this notation is the removal of typing annotations from the variables Γ when these variables are given as arguments to s.

  Since in Cedilleum there are three flavors of applications (to a type, to an erased term, and to an unerased term), we will only us this notion when the type or kind of s is known, which is sufficient to disambiguate the flavor of application intended for each particular binder in Γ. For example, if s has type $\forall\,X\!:\!\star.\,\forall\,x\!:\!X.\,\Pi\,x'\!:\!X.\,X$ and $\Gamma = X\!:\!\star, x\!:\!X, x'\!:\!X$ then $s\ \Gamma = s\ \cdot X\ \text{-}x\ x'$
- Δ and Δ' are notations we will use for a specially designated contexts associating type variables with both global "concrete" and local "abstracted" inductive data-type declarations. The purpose of this latter sort of declaration is to enable type-guided termination of definitions using fixpoints (see Section 5.7) For example, given just the (global) data type declaration of Vec, we would have $\Delta(Vec) = \texttt{Ind}_{\text{C}}[1](\Gamma_{Vec} := \Sigma)$, where $\Gamma_{Vec} = Vec\!:\!\star \to Nat \to \star$ and Σ binds data constructors *vnil* and *vcons* to the appropriate types.

p-arity A kind K is a p-arity if it can be written as $\Pi\,\Gamma.\,K'$ for some Γ and K', where $\|\Gamma\| = p$. For an inductive definition $\texttt{Ind}_M[p](\Gamma_I := \Sigma)$, requiring that the kind $\Gamma_I(I)$ is a p-arity of $\star$ ensures that I *really does have* p parameters.

Types of Constructors T is a *type of a constructor of* I iff

- it is $I\ s_1 ... s_n$
- it can be written as $\forall\,s\!:\!C.\,T$ or $\Pi\,s\!:\!C.\,T$, where (in either case) T is a type of a constructor of I

Positivity condition The positivity condition is defined in two parts: the positivity condition of a type T of a constructor of I, and the positive occurence of I in T. We say that a type T of a constructor of I satisfies the positivity condition when

- T is $I\ s_1 ... s_n$ and I does not occur anywhere in $s_1 ... s_n$
- T is $\forall\, s\!:\!C.\, T'$ or $\Pi\, s\!:\!C.\, T'$, T' satisfies the positivity condition for I, and I occurs *only* positively in C

We say that I occurs only positively in T when

- I does not occur in T
- T is of the form $I\ s_1 ... s_n$ and I does not occur in $s_1 ... s_n$
- T is of the form $\forall\, s\!:\!C.\, T'$ or $\Pi\, s\!:\!C.\, T'$, I occurs only positively in T', and I *does not* occur positively in C

5.2 Well-formed inductive definitions

Let $\Gamma_{\mathrm{P}}, \Gamma_I$, and Σ be contexts such that Γ_I associates a single type-variable I to kind $\Pi\, \Gamma_{\mathrm{p}}.\, K$ and Σ associates term variables $c_1 ... c_n$ with corresponding types $\forall\, \Gamma_{\mathrm{P}}.\, T_1, ... \forall\, \Gamma_{\mathrm{P}}.\, T_n$. Then the rule given in Figure 14 states when an inductive datatype definition may be introduced, provided that the following side conditions hold:

Figure 14: Introduction of inductive datatype

$$\frac{\emptyset \vdash \Gamma_I(I) : \Box \quad \|\Gamma_P\| = p \quad (\Gamma_I, \Gamma_P \vdash T_i : \star)_{i=1..n}}{\mathtt{Ind}_M[p](\Gamma_I := \Sigma)\ wf}$$

- Names I and $c_1..c_n$ are distinct from any other inductive datatype type or constructor names, and distinct amongst themselves
- Each of $T_1..T_n$ is a type of constructor of I which satisfies the positivity condition for I. Furthmore, each occurence of I in T_i is one which is applied to the parameters Γ_P.
- Identifiers I, $c_1, ..., c_n$ are fresh w.r.t the global context, and do not overlap with each other nor any identifiers in Γ_P.

When an inductive data-type has been defined using the *defDataType* production, it is understood that this always a concrete inductive type, and it (implicitly) adds to a global typing context the variable bindings in Γ_I and Σ. Similarly, when checking that the kind $\Gamma_I(I)$ and type T_i are well-sorted and well-kinded, we assume an (implicit) global context of previous definitions.

5.3 Valid Elimination Kind

Figure 15: Valid elimination kinds

$$\frac{}{[\![T : \star \mid T \to \star]\!]} \qquad \frac{[\![T\ s : K \mid K']\!]}{[\![T : \Pi\, s\!:\!C.\, K \mid \Pi\, s\!:\!C.\, K']\!]}$$

When type-checking a pattern match (either μ or μ'), we need to know that the given motive P has a kind K for which elimination of a term with some inductive data-type I is permissible. We write this judgment as $[\![T : K' | K]\!]$, which should be read "the type T of kind K' can be eliminated through pattern-matching with a motive of kind K". This judgment is defined by the simple rules in Figure 15. For example, a valid elimination kind for the indexed type family $Vec \cdot X$ (which has kind $\Pi\, n\!:\!Nat.\, \star$) is $\Pi\, n\!:\!Nat.\, \Pi\, x\!:\!Vec \cdot X\ n.\, \star$

5.4 Valid Branch Type

Another piece of kit we need is a way to ensure that, in a pattern-matching expression, a particular branch has the correct type given a particular constructor of an inductive data-type and a motive. We write $\{\{c : T\}\}_I^P$ to indicate the type corresponding to the (possibly partially applied) constructor c of I and its type T. We abbreviate this notation to $\{\{c\}\}^P$ when the inductive type variable I, and the type T of c, is known from the (meta-language) context.

$$\begin{array}{rcl} \{\{c : I\ \overline{T}\ \overline{s}\}\}_I^P & = & P\ \overline{s}\ c \\ \{\{c : \forall\, x{:}T'.\, T\}\}_I^P & = & \forall\, x{:}T'.\, \{\{c\ \text{-}x : T\}\}_I^P \\ \{\{c : \forall\, x{:}K.\, T\}\}_I^P & = & \forall\, x{:}K.\, \{\{c\ \cdot x : T\}\}_I^P \\ \{\{c : \Pi\, x{:}T'.\, T\}\}_I^P & = & \Pi\, x{:}T'.\, \{\{c\ x : T\}\}_I^P \end{array}$$

where we leave implicit the book-keeping required to separate the parameters $\overline{T}$ from the indicies $\overline{s}$.

The biggest difference bewteen this definition and the similar one found in the Coq documentation is that types can have implicit and explicit quantifiers, so we must make sure that the types of branches have implicit / explicit quantifiers (and the subjects c have applications for types, implicit terms, and explicit terms), corresponding to those of the arguments to the data constructor for the pattern for the branch.

5.5 Well-formed Patterns

Figure 16: Well-formedness of a pattern

$$\frac{\Gamma \vdash P : K \quad \Sigma = c_1 {:} \forall\, \Gamma_P.\, T_1, ..., c_n {:} \forall\, \Gamma_P.\, T_n \quad \|\overline{T}\| = \|\Gamma_p\| = p \quad [\![I\ \overline{T}\ : \Gamma(I) \mid K]\!] \quad (\Gamma, \Delta \vdash_{\Downarrow} t_i : \{\{c_i\ \overline{T}\}\}^P)_{i=1..n}}{\mathit{WF\text{-}Pat}(\Gamma, \Delta, \mathtt{Ind}_M[p](\Gamma_I := \Sigma), \overline{T}, \mu'(t, P, t_{i=1..n}))}$$

Figure 16 gives the rule for checking that a pattern $\mu'(t, P, t_{i=1..n})$ is well-formed. We check that the motive P is well-kinded at kind K, that the given parameters $\overline{T}$ match the expected number p from the inductive data-type declaration, that an inductive data-type I instantiated with the given parameters $\overline{T}$ can be eliminated to a type of kind K, and that the given branches t_i account for each of the constructors c_i of Σ and have the required branch type $\{\{c_i\ \overline{T}\}\}^P$ under the given local context Γ and context of inductive data-type declarations Δ.

5.6 Generation of Abstracted Inductive Definitions

Cedilleum supports *histomorphic* recursion (that is, having access to all previous recursive values) where termination is ensured through typing. In order to make this possible, we need a mechanism for tracking the global definitions of *concrete* inductive data types as well the locally-introduced *abstract* inductive data type representing the recursive occurences suitable for a fixpoint function to be called on.

If I is an inductive type such that $\Delta(I) = \mathtt{Ind}_{\mathrm{C}}[p](\Gamma_I := \Sigma)$ and I' is a fresh type variable, then we define function $\mathit{Hist}(\Delta, I, \overline{T}, I')$ producing an abstracted (well-formed) inductive definition $\mathtt{Ind}_{\mathrm{A}}[0](\Gamma_{I'} := \Sigma')$, where

- $\Gamma_{I'}(I') = \forall\, \Gamma_D.\, \star$ if $\Gamma_I(I) = \forall\, \Gamma_P.\, \forall\, \Gamma_D.\, \star$ (and $\|\Gamma_P\| = \|\overline{T}\| = p$)

  That is, the kind of I' is the same as the kind of $I\ \overline{T}$

- $\Sigma' = c_1' {:} \forall\, \Gamma_D.\, \overset{\Pi}{\forall}\, \Gamma_{A_1'}.\, I'\ \Gamma_D, ..., c_n' {:} \forall\, \Gamma_D.\, \overset{\Pi}{\forall}\, \Gamma_{A_n'}.\, I\ \overline{T}\ \Gamma_D$,

  when each of the concrete constructors c_i in Σ are associated with type $\forall\, \Gamma_P.\, \forall\, \Gamma_D.\, \overset{\Pi}{\forall}\, \Gamma_{A_i}.\, I\ \Gamma_P\ \Gamma_D$ and each $\Gamma_{A_i'} = [\lambda\, \Gamma_P.\, I'/I, \overline{T}/\Gamma_P]\Gamma_{A_i}$.

That is, trasforming the concrete constructors of the inductive datatype I to "abstracted" constructors involves replacing each recursive occurrence of $I\ \Gamma_P$ with the fresh type variable I, and instantiating each of the parameters Γ_P with $\overline{T}$.

Users of Cedilleum will see "punning" of the concrete constructors c_i and abstracted constructors c'_i. In particular, when using fix-point pattern matching branch labels will be written with the constructors for the concrete inductive data-type, and the expected type of a branch given by the motive will pretty-print using the concrete constructors. In the inference rules, however, we will take more care to distinguish the abstract constructors (see Subsection 5.7).

5.7 Typing Rules

Figure 17: Use of an inductive datatype $\mathtt{Ind}_M[p](\Gamma_I := \Sigma)$

$$\frac{\Gamma \vdash_{\Uparrow} t : I\ \overline{T}\ \overline{s} \quad \mathit{WF\text{-}Pat}(\Gamma, \Delta, \Delta(I), \overline{T}, \mu'(t, P, t_{i=1..n}))}{\Gamma, \Delta \vdash_{\delta} \mu'(t, P, t_{i=1..n}) : P\ \overline{s}\ t}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\Uparrow} t : I\ \overline{T}\ \overline{s} \quad \Delta(I) = \mathtt{Ind}_{\mathrm{C}}[p](\Gamma_I := \Sigma) \quad \Gamma_I(I) = \Pi\, \Gamma_P .\, \Pi\, \Gamma_{\mathrm{D}} .\, \star, \|\Gamma_P\| = p \quad \mathit{Hist}(\Delta, I, \overline{T}, I') = \mathtt{Ind}_{\mathrm{A}}[0](\Gamma_{I'} := \Sigma') \\ \Gamma' = \Gamma, \Gamma_{I'}, x_{\mathrm{to}} = \Lambda\, \Gamma_D .\, \lambda\, x .\, x : \forall\, \Gamma_{\mathrm{D}} .\, I'\ \Gamma_{\mathrm{D}} \to I\ \overline{T}\ \Gamma_{\mathrm{D}}, x_{\mathrm{rec}} : \forall\, \Gamma_{\mathrm{D}} .\, \Pi\, x : I'\ \Gamma_{\mathrm{D}} .\, P\ \Gamma_{\mathrm{D}}\ (x_{\mathrm{to}}\ \Gamma_D\ x) \quad \Delta' = \Delta, \mathit{Hist}(\Delta, I, \overline{T}, I') \\ \mathit{WF\text{-}Pat}(\Gamma', \Delta', \Delta'(I'), \varnothing, \mu'(t, P, t_{i=1..n}))\end{array}}{\Gamma, \Delta \vdash_{\delta} \mu(x_{\mathrm{rec}}, I', x_{\mathrm{to}}, t, P, t_{i=1..n}) : P\ \overline{s}\ t}$$

The first rule of Figure 17 is for typing simple pattern matching with μ'. We need to know that the scrutinee t is well-typed at some inductive type $I\ \overline{T}\ \overline{s}$, where $\overline{T}$ represents the parameters and $\overline{s}$ the indicies. Then we defer to the judgment $\mathit{WF\text{-}Pat}$ to ensure that this pattern-matching expression is a valid elimination of t to type P.

The second rule is for typing pattern-matching with fix-points, and is significantly more involved. As above we check the scrutinee t has some inductive type $I\ \overline{T}\ \overline{s}$. We confirm that I is a *concrete* inductive data-type by looking up its definition in Δ, and then generate the abstracted definition $\mathit{Hist}(\Delta, I, \overline{T}, I')$ for some fresh I'. We then add to the local typing context $\Gamma_{I'}$ (the new inductive type I' with its associated kind) and two new variables x_{to} and x_{rec}.

- x_{to} is the *revealer*. It casts a term of an abstracted inductive data-type $I'\ \Gamma_D$ to the concrete type $I\ \overline{T}\ \Gamma_D$. Crucially, it is an *identity* cast (the implicit quantification $\Lambda\Gamma_D$ disappears after erasure). The intuition why this should be the case is that the abstracted type I' only serves to mark the recursive occurrences of I during pattern-matching to guarantee termination.

- x_{rec} is the *recursor* (or the inductive hypothesis). Its result type $P'\ \Gamma_D\ x$ utilizes x_{to} in P' to be well-typed, as the x in this expression has type $I'\ \Gamma_D$, but P expects an $I\ \overline{T}\ \Gamma_D$. Because x_{to} erases to the identity, uses of the x_{rec} will produce expressions whose types will not interfere with producing the needed result for a given branch (see the extended example – TODO).

With these definitions, we finish the rule by checking that the pattern is well-formed using the augmented local context Γ' and context of inductive data-type definitions Δ'.

References

[Inr18] Inria. The Coq Documentation. `https://coq.inria.fr/refman/index.html`, 2018.

[Miq01] Alexandre Miquel. 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, pages 344–359, Berlin, Heidelberg, 2001. Springer-Verlag.

[PM15] Christine Paulin-Mohring. Introduction to the calculus of inductive constructions, 2015.