The Cedilleum Language Specification
Syntax, Typing, Reduction, and Elaboration

Christopher Jenkins

July 18, 2018

1 Syntax

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

Figure 1: Identifiers

Identifiers Figure 1 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.

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

Figure 2: Untyped terms

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 construct).

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`:

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 3: Modules and definitions

```
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 ≃ Z} ⇒ Vec · A n
  | vcons : ∀ n : Nat . ∀ m : Nat . A → Vec n → { m ≃ S n} ⇒ Vec m
  .
```

Sorts S	$::=$	$\square$	sole super-kind
		K	kinds
Classifiers C	$::=$	K	types
		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, P	$::=$	$\Pi\ x : T\ .\ T$	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 4: Kinds and types

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 4 and terms are listed in Figure 5 along with some auxiliary grammatical categories. In both of these figures, the constructs forming expressions are listed from lowest to highest precedence – "abstractors" (λ Λ Π $\forall$) bind most loosely and parentheses most tightly. Associativity is as-expected, with arrows ($\to$ $\Rightarrow$) 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 $\to$) 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 $\to$) 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 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

[1] Cite

and applications (with term and type arguments), and a primitive equality type for untyped terms. Last of all is the "hole" type (•) 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 = • .
```

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
		$\mu\ u, X, u_I\ .\ t\ motive^?\ \{case^*\}$	type-guarded pattern match and fixpoint
		$\mu'\ t\ motive^?\ \{case^*\}$	auxiliary pattern match
		u	term variable
		(t)	
		$\bullet$	hole for incomplete term
$case$	$::=$	$\mid\ id\ vararg^*\ \mapsto t$	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 5: Annotated Terms

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

[2] Module erasure, discussed below

Section 3, 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 .
           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)
              }
```

2 Erasure

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

Figure 6: Erasure for annotated terms

The definition of the erasure function given in Figure 6 takes the annotated terms from Figures 4 and 5 to the untyped terms of Figure 2. 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.

3 Type System (sans Inductive Datatypes)

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

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

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

Figure 7: Sort checking $\boxed{\Gamma \vdash C : S}$

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

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

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

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

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

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

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

Figure 8: Type checking $\boxed{\Gamma \vdash_{\delta} s : C}$ (sans inductive datatypes)

The inference rules for classifying expressions in Cedilleum are stratified into two judgments. Figure 7 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.

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

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

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 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)

4 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 $\texttt{Ind}*[p](\Gamma_I := \Sigma)$ where Γ_I is a context binding *one* type variable I (representing the inductive datatype being defined), Σ contains the data constructors of type I, and p is the number of parameters to I. For example, consider the `List` and `Vec` definitions from 1. These will be represented in the AST as

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

and

$$\texttt{Ind}_{\mathrm{C}}[1](Vec : \star \to Nat \to \star := \begin{array}{lcl} vnil & : & \forall A : \star . Vec \cdot A\ Z \\ vcons & : & \forall A : \star . \forall n : Nat . A \to Vec \cdot A\ n \to Vec \cdot A\ (S\ n) \end{array})$$

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

- 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. For example, consider the pattern-matches given in the code listings for `isvnil` and `vlength` above. These would be translated into the AST as

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

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

and

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

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 below):

- 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 case t_i must be in a bijection with the n constructors Γ_{C} of I

4.1 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 $\lambda\,\Gamma.\, t = \lambda\, x_1 : T_1.\, \lambda\, x_2 : T_2.\, t$
- $\|\Gamma\|$ 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.

  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 what flavor of application is 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 below.) For example, given just the (global) data type declaration of Vec, we would have $\Delta = \mathtt{Ind}_{\mathrm{C}}[1](\Gamma_{Vec} := \Sigma :=)$, 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 $\mathtt{Ind}_*[p](\Gamma_I := \Sigma)$, 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

4.2 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 9 states when an inductive datatype definition may be introduced, provided that the following side conditions hold:

Figure 9: Introduction of inductive datatype

$$\frac{\Gamma_P \vdash \Gamma_I(I) : \Box \quad \Sigma = c_1 \!:\! \forall\, \Gamma_P.\, T_1, ..., c_n \!:\! \forall\, \Gamma_P.\, T_n \quad \|\Gamma_P\| = p \quad (\Gamma_I, \Gamma_P \vdash T_i : \star)_{i=1..n}}{\Gamma \vdash \mathtt{Ind}{*}[p](\Gamma_I := \Sigma)\ wf}$$

- Names I and $c_1..c_n$ are distinct from any other inductive datatype type or constructor names, and distinct amongst themselves
- $\|\Gamma_{\mathrm{P}}\| = p$
- Each of $T_1..T_n$ is a type of constructor of I which satisfies the positivity condition for I
- No other previously defined inductive datatypes I' nor constructors $c'_1..c'_{n'}$ occur anywhere in Γ_{P},Γ_I, or Σ

When an inductive (concrete) data-type has been defined using the *defDataType* production, it is understood (and left implicit) that this adds to a global typing context the variable bindings in Γ_I and Σ.

4.3 Valid Elimination Kind

Figure 10: Valid elimination kinds

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

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 can be eliminated through pattern-matching with a motive of kind K". This judgment is defined by the simple rules in Figure 10. For example, a valid elimination kind for the indexed type family $Vec\ \cdot X$ (which has kind $\Pi\, n\!:\!Nat.\, \star$) is $\Pi\, n\!:\!Nat.\, \Pi\, x\!:\!Vec\ \cdot X\ n.\, \star$

4.4 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 indcate the type coressponding to the (possibly partially applied) constructor c and its type T of an inductive data-type I. 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 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.

4.5 Well-formed Patterns

Figure 11: Well-formedness of a pattern

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

Figure 11 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, 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.

4.6 Generation of Histomorphic Datatype Definitions

Cedilleum supports *histomorphic* recursion 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 $\mathit{Hist}(\Delta, I, I')$ producing an abstracted (well-formed) inductive definition $\mathtt{Ind}_{\mathrm{H}}[0](\Gamma_{I'} := \Sigma')$, where

- $\Gamma_{I'}(I') = \forall\,\Gamma_D.\, \star$ if $\Gamma_I(I) = \forall\,\Gamma_P.\, \forall\,\Gamma_D {:} \star$. (and $\|\Gamma_P\| = p$)

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

- $\Sigma' = c_1 : T'_1, ..., c_n : T'_n$ when $\Sigma = c_1 : \forall\,\Gamma_P.\, T_1, ..., c_n : \forall\,\Gamma_P.\, T_n$ and $T'_i = [\lambda\,\Gamma_P.\, I'/I]\ [\overline{T}/\Gamma_P]\ T_i$ for all $i = 1..n$

  That is, the new signature of constructiors Σ' is taken by replacing all occurences of $I\ \Gamma_P$ with I' and instantiating each parameter variable in Γ_P with the corresponding parameter of $\overline{T}$

For convenience, Σ' binds the same constructor names as did Σ. There is however no risk of ambiguity in typing an expression from doing this: as we will see below in Section 4.7, these variables are *not* added to the typing context, meaning that when writing programs there are no constructors for the abstracted inductive type I'. Instead, Σ' serves merely to ensure that pattern-matching branches are well-typed for the abstrated inductive data type.

4.7 Typing Rules

Figure 12: Use of an inductive datatype $\mathtt{Ind}_*[p](\Gamma_I := \Sigma)$

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

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

The first rule of Figure 12 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 $\mathit{Hist}(\Delta, I, I')$ 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 over Γ_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 $P\ \Gamma_D\ (x_{\mathrm{to}}\ \Gamma_D\ x)$ utilizes x_{to} to be well-typed. 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'.

Note that there is a subtle issue in checking that the branches t_i have the right type for the fix-point pattern-matching case. Within the definition of $Wf\text{-}Pat$, the type computed by $c_i\}\}^{P'}_{I}{}'$