The Cedilleum Language Specification

Syntax, Typing, Reduction, and Elaboration

Christopher Jenkins

July 11, 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. κ`func = ⋆ → ⋆`

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 List (A : ⋆) : ⋆ =
  | nil  : List
  | cons : A → List → List
  .
data Vec (A : ⋆) : Nat → ⋆ =
  | vnil  : Vec Z
  | vcons : ∀ n : Nat . A → Vec n → Vec (S n)
  .
```

Sorts S	::=	$\Box$	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	::=	$\Pi\ x : T\ .\ T$	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" ($\lambda\ \Lambda\ \Pi\ \forall$) 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 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 → ⋆) → ⋆ = • .
```

[1] Cite

[2] Module erasure, discussed below

`castE` $\triangleleft$ $\forall$ `A : ` $\star$ `. ` $\forall$ `B : A` $\rightarrow$ $\star$ `. CastE` $\cdot$ `A` $\cdot$ `B` $\Rightarrow$ Π `a : A . B a = ` $\bullet$ `.`

Subjects s	::=	t	term
		T	type
Terms t	::=	$\lambda\, x\ class^?.\, t$	normal abstraction
		$\Lambda\, x\ class^?.\, t$	erased abstraction
		[$defTermOrType$] - t	let definitions
		$\rho\ t$ - t'	equality elimination by rewriting
		$\phi\ t$ - t' {t''}	type cast
		$\chi\ T$ - t	check a term against a type
		δ - t	ex falso quodlibet
		$\theta\ t\ t'^*$	elimination with a motive
		$t\ t'$	applications
		t -t'	application to an erased term
		t ·T	application to a type
		β {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
		-u	erased constructor argument
		·X	type constructor argument
$class$	::=	: 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 that will be discussed in more detail (and whose purpose will become more apparent) in Section 3.

2 Erasure

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)

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

[4]Where `tt` $= \lambda\, x.\, \lambda\, y.\, x$ and `ff` $= \lambda\, x.\, \lambda\, y.\, y$

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

Figure 6: Erasure for annotated terms

The 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.

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.

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

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

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

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

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

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

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

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

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

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

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

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

(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 `Ind` $[p]$ $(\Gamma_I := \Gamma_C)$ where Γ_I is a context binding *one* type variable I (representing the inductive datatype being defined), Γ_C represents the data constructors of type I, and p is the number

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

of parameters to I. For example, consider the `List` and `Vec` definitions from 1. These will be represented in the AST as

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

and

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

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 $id \in \Gamma_C$ must be *types of constructors of* I
- The definition must satisfy the *non-strict* positivity condition.

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)
- $\lfloor\Gamma\rfloor$ indicates converting Γ to a sequence of variable arguments. When given as arguments to some expression t whose type T is known, it is assumed that each variable in the sequence is given with the right "flavor" of application (type, explicit, erased).
- Γ_μ is a specially designated context tracking both global inductive datatype definitions as well local "abstracted" versions of these for uses of μ (see further below).

p-arity A kind K is a p-arity if it can be written as $\Pi\,\Gamma_P.\,K'$ for some Γ_P and K', where $\|\Gamma_p\| = p$ and $\Pi\,\Gamma_P$ represents the explicit quantification of all term and type variables in Γ_P (similarly, $\forall\,\Gamma_P$ represents the *implicit* quantification of these variables). For an inductive definition $\texttt{Ind}\ [p]\ (\Gamma_I := \Gamma_C)$, requiring that the kind of 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 if

- 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

Well-formed patterns Operators μ and μ' both take a sequence of *cases* as their last argument, with each case of the form $|\ c\ \lfloor \Gamma_A \rfloor \mapsto t$. Like with inductive data-type definitions, this is a somewhat inconvenient notation, so in the AST we will represent uses of μ and μ' more compactly. Given an inductive datatype definition $\texttt{Ind}\ [\Gamma_P]\ (\Gamma_I := \Gamma_c)$ for type I, the expression $\mu'\ t\ @P\ \{(|\ c_i\ \lfloor \Gamma_{A_i} \rfloor \mapsto t_i)_{i=1..\|\Gamma_c\|}\}$ will be written as $\mu'(t, P, (\lambda\Lambda\, \Gamma_{A_i}.\, f_i)_{i=1..\lfloor \Gamma_C \rfloor})$

The *pattern* of the case is $c\ \overline{x}$. Given some (well-formed) inductive datatype definition $\texttt{Ind}\ [\Gamma_P]\ (\Gamma_I := \Gamma_C)$ of some type I, we say that some pattern is well-formed ($WF\text{-}Pat(I, c\ \overline{x}, \Gamma_A)$) when

- c is a data constructor of I: $c \in \Gamma_c$
- Γ_A is a context binding $\overline{x}$ to the appropriate types: $\lfloor \Gamma_A \rfloor = \overline{x}$
- $\overline{x}$ matches the shape of the type T of $c\ \lfloor \Gamma_c \rfloor$ – that is, not only does it contain an argument for each quantification in T, but each argument is given erased / unerased according to the corresponding quantifiers in T.

We furthermore say that a set of patters is well-formed ($WF\text{-}Pats(I, (c\ \overline{x}_i, \Gamma_{A_i})_{i=1..\|\Gamma_C\|})$) when

- Each pattern is well-formed: $WF\text{-}Pat(I, c_i\ \overline{x}_i, \Gamma_{A_i})$
- The constructors in the patterns are distinct: for all i, j, $c_i = c_j$ means $i = j$

4.2 Well-formed inductive definitions

Let Γ_P, Γ_I, and Γ_C be contexts such that Γ_I associates a single type-variable I to kind $\Pi\, \Gamma_P.\, K_I$ and Γ_c associates term variables $c_1...c_n$ with corresponding types $\forall\, \Gamma_P.\, T_1, ...\forall\, \Gamma_P.\, T_n$. 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

$$\frac{\Gamma_P \vdash K_I : \Box \quad (\Gamma_I, \Gamma_P \vdash T_{c_i} : \star)_{i=1..n}}{\Gamma \vdash \texttt{Ind}\ [p]\ (\Gamma_I := \Gamma_C)\ wf}$$

- Names I and $c_1..c_n$ are distinct from any other inductive datatype type or constructor names, and distinct amongst themselves
- $\|\Gamma_P\| = p$
- 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 Γ_C

Figure 10: Use of an inductive datatype `Ind` $[p]$ $(\Gamma_I := \Gamma_C)$

$$\frac{}{\Gamma \vdash I : \Gamma_I(I)} \quad \frac{}{\Gamma \vdash_\delta c : \Gamma_C(c)}$$

$$\Gamma \vdash_{\Uparrow} t : I\ \lfloor \Gamma_P \rfloor\ \overline{s} \qquad \Gamma_\mu(I) = \texttt{Ind}\ [\Gamma_P]\ (\Gamma_I := \Gamma_C), \|\Gamma_C\| = n$$

$$\Gamma, \Gamma_D, x{:}I\ \lfloor \Gamma_P \rfloor\ \lfloor \Gamma_D \rfloor \vdash T : \star \qquad \mathit{WF\text{-}Pats}(I, (c_i\ \lfloor \Gamma_{A_i} \rfloor, \Gamma_{A_i})_{i=1..n})$$

$$\frac{(\Gamma, \Gamma_{A_i} \vdash t_i : [\overline{s_i}/\lfloor \Gamma_D \rfloor, (c_i\ \lfloor \Gamma_P \rfloor\ \lfloor \Gamma_{A_i} \rfloor)/x]\ T \quad \mathit{erased}(\Gamma_{A_i}) \cap FV(|t_i|) = \varnothing)_{i=1..n}}{\Gamma, \Gamma_\mu \vdash_\delta \mu'\ t\ @\lambda \Gamma_D.\, \lambda\, x{:}I\ \lfloor \Gamma_P \rfloor\ \lfloor \Gamma_D \rfloor.\, T\ \{(|\ c_i\ \lfloor \Gamma_{A_i} \rfloor \mapsto t_i)\} : [\overline{s}/\lfloor \Gamma_D \rfloor, t/x]\ T}$$

4.3 Typing Rules

Assuming that an inductive definition `Ind` $[p]$ $(\Gamma_I := \Gamma_C)$ is well-formed and has been defined, the typing rules of Figure 10 govern its usage.

The rules for typing uses of μ and μ' are fairly involved. We will start with the latter, since its the simpler of the two and overlaps a good deal with the former. Before diving into the details of these rules, we need to understand a few notational conventions. First, Γ_{mu} is a special context of type variables tracking which ones are inductive by associating them with their inductive declaration. Second, the notation $\lfloor \Gamma \rfloor$ indicates a sequence of variable arguments given by context Γ. For example, if $\Gamma_P = A : \star$ and $\Gamma_D = n : Nat$, then the type written $Vec\ \lfloor \Gamma_P \rfloor\ \lfloor \Gamma_D \rfloor$ is equivalent to $Vec \cdot A\ n$ [6] By convention, Γ_D is meant to be read as the "context of type in**d**ices". Finally, $\|\Gamma\|$ represents the number of associated variables in Γ.

The μ' operator performs simple pattern-matching and has three components. The first component is the scrutinee t, and the first and second premises of this typing rule ensure that t really is an inductive data type by checking that it is indeed well-typed, that its type is some variable-headed application, and that this variable head has a corresponding inductive definition. The next component is the *motive*, which is preceded by the @ symbol. Essentially, type T is the property the programmer wishes to prove, and μ' allows them to do so by *case analysis*. We check that the motive is well-kinded in the third premise. The last component of μ' is the cases covering the constructors of I. Each constructor case c_i comes with its own sub-data held in Γ_{A_i}; the fourth premise checks that each c_i really is a constructor and the left-hand side patterns are type-correct (applied to the right number of arguments and of the right flavor of application.) Left implicit in these premises is the condition that each c_i be mutually distinct. The fifth premise checks that each right-hand side of the case is well-typed, given the new arguments introduced by Γ_A, and that arguments of a case analysis introduced in an erased position are used correctly. Each t_i is expected to have a type derived from T, where the indices Γ_D have been replaced by the particular constructor's $\overline{s_i}$ and the abstracted subject of case analysis x by c_i applied to its arguments.

[6] With the kind of Vec guiding which flavor of application is appropriate.