

The Cedilleum Language Specification

Syntax, Typing, Reduction, and Elaboration

Christopher Jenkins

July 4, 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 `κmyDef` is only legal as a kind definition.

$p ::= u$	variables
$\lambda u. p$	functions
$p p'$	applications
$\mu\langle u_{rec} u_{out} u_z \rangle e_s \{pcase^*\}$	fixed-point and pattern matching
$\mu' p_s \{pcase^*\}$	auxiliary 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`:

<i>mod</i>	::= module <i>id</i> . <i>imprt</i> * <i>cmd</i> *	module declarations
<i>imprt</i>	::= import <i>id</i> .	module imports
<i>cmd</i>	::= <i>defTermOrType</i> <i>defDataType</i> <i>defKind</i>	definitions
<i>defTermOrType</i>	::= <i>id</i> <i>checkType</i> ? = <i>t</i> .	term definition
	<i>id</i> : <i>K</i> = <i>T</i> .	type definition
<i>defKind</i>	::= κ = <i>K</i>	kind definition
<i>defDataType</i>	::= data <i>id param</i> * : <i>K</i> = <i>constr</i> * .	datatype definitions
<i>checkType</i>	::= : <i>T</i>	annotation for term definition
<i>param</i>	::= (<i>x</i> : <i>C</i>)	
<i>constr</i>	::= <i>id</i> : <i>T</i>	

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 \square), e.g. $\kappa\text{func} = * \rightarrow *$

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	$::= \square$	sole super-kind
	K	kinds
Classifiers C	$::= K$	types
	T	types
Kinds K	$::= \Pi x : C . K$	explicit product
	$C \rightarrow K$	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
	$T \rightarrow T'$	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 ($\rightarrow \Rightarrow$) 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 Constructions¹. 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 are erased at run-time, but \forall -products can quantify over types *and* terms because terms can be erased. Meanwhile, non-dependent products (\rightarrow 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 equal² to identity – where the definitions **CastE** and **castE** are incomplete.

```
module DepCast .
```

```
CastE <|  $\Pi A : \star . (A \rightarrow \star) \rightarrow \star = \bullet .$ 
```

¹Cite

²Module erasure, discussed below

$\text{castE} \triangleleft \forall A : \star . \forall B : A \rightarrow \star . \text{CastE} \cdot A \cdot B \Rightarrow \Pi a : A . B a = \bullet .$

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

⁴Where we assume t does not occur anywhere in T

⁴Where $\text{tt} = \lambda x. \lambda y. x$ and $\text{ff} = \lambda x. \lambda y. y$

$ x $	$=$	x
$ \star $	$=$	\star
$ \square $	$=$	\square
$ \beta \{t\} $	$=$	$ t $
$ \delta t $	$=$	$ t $
$ \chi T^? - t $	$=$	$ t $
$ \theta t t'^* $	$=$	$ t t'^* $
$ \varsigma t $	$=$	$ t $
$ t t' $	$=$	$ t t' $
$ t - t' $	$=$	$ t $
$ t \cdot T $	$=$	$ t $
$ \rho t - 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 - t' \{t''\} $	$=$	$ t'' $
$ [x = t : T] - t' $	$=$	$(\lambda x. t') t $
$ [X = T : K] - t $	$=$	$ t $
$ \{t \simeq t'\} $	$=$	$\{ t \simeq t' \}$
$ \mu\langle u, X, u', u'' \rangle t \text{ motive}^? \{case^*\} $	$=$	$\mu\langle u, u', u'' \rangle t \{ case^* \}$
$ \mu' t \text{ motive}^? \{case^*\} $	$=$	$\mu' t \{ case^* \}$
$ id \text{ vararg}^* \mapsto t $	$=$	$id vararg^* \mapsto t $
$ -u $	$=$	
$ \cdot T $	$=$	

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.

$$\begin{array}{c}
\overline{\Gamma \vdash \star : \square} \qquad \frac{\Gamma \vdash C : S \quad \Gamma, y : C \vdash C' : S'}{\Gamma \vdash \Pi y : C. C' : S'} \qquad \frac{\Gamma \vdash C : S \quad \Gamma, y : C \vdash C' : \star}{\Gamma \vdash \forall y : C. C' : \star} \\
\\
\frac{FV(p \ p') \subseteq \text{dom}(\Gamma)}{\Gamma \vdash \{p \simeq p'\} : \star} \qquad \overline{\Gamma \vdash \kappa : \Gamma(\kappa)} \qquad \overline{\Gamma \vdash X : \Gamma(X)} \\
\\
\frac{\Gamma \vdash \Pi x : C. K : \square \quad \Gamma, x : C \vdash T : K}{\Gamma \vdash \lambda x : C. T : \Pi x : C. K} \quad \frac{\Gamma \vdash T : \Pi x : K. K' \quad \Gamma \vdash T' : K}{\Gamma \vdash T \cdot T' : [T'/x]K'} \quad \frac{\Gamma \vdash T : \Pi x : T'. K \quad \Gamma \vdash_{\Downarrow} t : T'}{\Gamma \vdash T t : [t/x]K}
\end{array}$$

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

$$\begin{array}{c}
\overline{\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} \quad \frac{x \notin FV(|t|) \quad \Gamma, x : C \vdash_{\delta} t : T}{\Gamma \vdash_{\Downarrow} \Lambda x. t : \forall x : C. T} \quad \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'} \quad \frac{\Gamma \vdash_{\Uparrow} t : \forall x : T'. T \quad \Gamma \vdash_{\Downarrow} t' : T'}{\Gamma \vdash_{\delta} t \cdot t' : [t'/x]T} \quad \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] \cdot t' : T'} \quad \frac{\Gamma \vdash_{\Uparrow} t : T \quad \Gamma, id = t : T \vdash_{\delta} t' : T'}{\Gamma \vdash_{\delta} [id = t] \cdot t' : T'} \quad \frac{\Gamma \vdash_{\Uparrow} t : \{t_1 \simeq t_2\} \quad \Gamma \vdash_{\Uparrow} t' : [t_1/x] T}{\Gamma \vdash_{\delta} \rho t \cdot t' : [t_2/x] T} \quad 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] \cdot t' : T'} \quad \frac{\Gamma \vdash \{t' \simeq t'\} : \star}{\Gamma \vdash_{\Downarrow} \beta \{t\} : \{t' \simeq t'\}} \quad \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 \cdot t_1 \{t_2\} : T} \quad \frac{\Gamma \vdash_{\Downarrow} t : T}{\Gamma \vdash_{\Uparrow} \chi T \cdot t : T} \quad \frac{\Gamma \vdash_{\Downarrow} t : \{\mathbf{tt} \simeq \mathbf{ff}\}}{\Gamma \vdash_{\Downarrow} \delta \cdot t : T} \quad 4 \\
\\
\frac{\Gamma \vdash_{\Uparrow} t : ?? \quad \Gamma \vdash_{\downarrow} t'^* : ??}{\Gamma \vdash_{\Downarrow} \theta t t'^* : T}
\end{array}$$

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 borrows heavily from the convention of the Coq documentation⁵.

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

1. Γ_I consists of *one* type variable with an associated kind, representing the inductive type being defined. We will frequently refer to the sole type variable in this context as I .

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

In particular, note that in the current version of Cedilleum mutual inductive datatypes will not be definable.

2. Γ_C consists of term variables representing the constructors.
3. p represents the number of parameters to the inductive data-type definition.

For example, consider the **List** and **Vec** definitions from 1. These will be represented in the AST as

$$\text{Ind } [1] \text{ (} \textit{List} : \star \rightarrow \star := \begin{array}{ll} \textit{nil} & : \forall A : \star. \textit{List} \cdot A \\ \textit{cons} & : \forall A : \star. A \rightarrow \textit{List} \cdot A \rightarrow \textit{List} \cdot A \end{array} \text{)}$$

and

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