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

Christopher Jenkins

June 20, 2018

1 Syntax

id			identifiers for definitions
u			term variables
X			type variables
k			kind variables
x	$::=$	$id \mid u \mid X$	non-kind 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 k. In Cedilleum source files (as in the parent language Cedille) kind variables should be prefixed with κ.

$uterms$	$::=$	u	variables
		$\lambda\ u.\ uterm$	functions
		$uterm\ uterm$	applications

Figure 2: Untyped terms

Untyped Terms The grammar of pure (untyped) terms is that of the λ-calculus.

Modules and Definitions All Cedilleum source files start with *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 vec .

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.

mod	::=	**module** *id* **.** $imprt^*$ cmd^*	module declarations
imprt	::=	**import** *id* **.**	module imports
cmd	::=	*defTermOrType*	definitions
		defDataType	
		defKind	
defTermOrType	::=	*id* $checkType^?$ = *term* **.**	term definition
		id **:** *kind* = *type* **.**	type definition
defKind	::=	*k* = *kind*	kind definition
defDataType	::=	**data** *id* $param^*$ **:** *kind* = $constr^*$ **.**	datatype definitions
checkType	::=	**:** *type*	annotation for term definition
param	::=	(*x* **:** *typeOrKind*)	
typeOrKind	::=	*type*	
		kind	
constr	::=	| *id* **:** *type*	

Figure 3: Modules and definitions

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 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 type annotation and the second without:

```
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 |[1] 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)
  .
```

Expression Language 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

[1]The first of these is optional.

$kind$	$::=$	$\Pi\ x : typeOrKind\ .\ kind$	explicit product
		$typeOrKind \to kind$	kind arrow
		$\star$	the kind of types that classify terms
		$(kind)$	
$type$	$::=$	$\Pi\ x : type\ .\ type$	explicit product
		$\forall\ x : typeOrKind\ .\ type$	implicit product
		$\lambda\ x : typeOrKind\ .\ type$	type-level function
		$type \Rightarrow type$	arrow with erased domain
		$type \to type$	normal arrow type
		$type \cdot type$	application to another type
		$type\ term$	application to a term
		$\{\ uterm\ \simeq\ uterm\}$	untyped equality
		$(type)$	
		X	type variable
		$\bullet$	hole for incomplete types

Figure 4: Kinds and types

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 are right-associative.

$term$	::=	$\lambda\ x\ class^?\ \textbf{.}\ term$	normal abstraction
		$\Lambda\ x\ class^?\ \textbf{.}\ term$	erased abstraction
		$[\ defTermOrType\ ]\ \textbf{-}\ term$	let
		$\rho\ term\ \textbf{-}\ term$	equality elimination by rewriting
		$\phi\ term\ \textbf{-}\ term\ \boldsymbol{\{}term\boldsymbol{\}}$	type cast
		$\chi\ type^?\ \textbf{-}\ term$	check a term against a type
		$\delta\ \textbf{-}\ term$	ex falso quodlibet
		$\theta\ term\ term^*$	elimination with a motive
		$term\ term$	applications
		$term\ \textbf{-}\ term$	application to an erased term
		$term \cdot type$	application to a type
		$\beta\ \boldsymbol{\{}term\boldsymbol{\}}$	reflexivity of equality
		$\varsigma\ term$	symmetry of equality
		$\mu\ term\ motive^?\ \boldsymbol{\{}\ case^*\ \boldsymbol{\}}$	pattern match and fixpoint
		u	term variable
		$\textbf{(}term\textbf{)}$	
		$\bullet$	hole for incomplete term
$vararg$	::=	u	normal constructor argument
		$\textbf{-}\ u$	erased constructor argument
		$\cdot\ X$	type constructor argument
$class$	::=	$\textbf{:}\ typeOrKind$	
$motive$	::=	$\textbf{@}\ type$	motive for induction
$case$	::=	$\mid id\ arg^* \mapsto term$	pattern-matching cases

Figure 5: Annotated Terms