The Cedilleum Language Specification

Syntax, Typing, Reduction, and Elaboration

Christopher Jenkins

June 27, 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

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.

e	$::=$	u	variables
		$\lambda\, u.\, e$	functions
		$e\ e'$	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 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.

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

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 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 $\Box$), 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 | (the first 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)
  .
```

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.

C	::=	K	kinds
		T	types
K	::=	$\Pi\ x : C\ .\ K$	explicit product
		$C \to K$	kind arrow
		$\star$	the kind of types that classify terms
		(K)	
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
		$\{\ e\ \simeq\ e'\}$	untyped equality
		(T)	
		X	type variable
		$\bullet$	hole for incomplete types

Figure 4: Kinds and types

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 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), a primitive equality type for untyped terms, and a "hole" type whose sole purpose is to help programmers writing code when they are not sure how to fill in a type or term sub-expression ($\bullet$ generates an immediate error, but type-checking continues as usual in order to provide the user with useful feedback.)

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 to identity (after erasure, discussed below) – 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 = • .
```

Terms

Erasure

[1] Cite

s	::=	t	term
		T	type
t	::=	$\lambda\, x\ class^?.\, t$	normal abstraction
		$\Lambda\, x\ class^?.\, t$	erased abstraction
		[$defTermOrType$] - t	let
		$\rho\ t$ - 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
		$\mu\ t\ motive^?$ { $case^*$ }	pattern match and fixpoint
		μ' $t\ motive^?$ { $case^*$ }	auxiliary pattern match
		u	term variable
		(t)	
		$\bullet$	hole for incomplete term
$vararg$	::=	u	normal constructor argument
		-u	erased constructor argument
		·X	type constructor argument
$class$	::=	: C	
$motive$	::=	@ T	motive for induction
$case$	::=	| $id\ vararg^* \mapsto t$	pattern-matching cases

Figure 5: Annotated Terms

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

Figure 6: Erasure for annotated terms