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

Christopher Jenkins

March 19, 2019

1 Introduction

This document describes *Cedilleum*, a general-purpose dependently typed programming language with inductive datatypes. Unlike most languages of this description, the underlying theory of Cedilleum is *not* the Calculus of Inductive Constructions (CIC)[PM15]. Instead, Cedilleum is designed so that it may easily be translated to *Cedille Core* – a compact core theory in which induction is derivable for lambda-encoded datatypes – while still providing high-level features like pattern-matching and recursive definitions. That said, the formal specification of Cedilleum as a self-contained language has a lot in common with CIC – see in particular Section 8 of [Inr18], which served as the basic template for much of this document's formal development.

1.1 Data-type Declarations

Before diving into the details, let us take a bird's-eye view of the language by showing some simple example data-type definitions and functions over them.

```
-- Non-recursive
data Bool: ⋆ =
  | tt: Bool
  | ff: Bool
.
-- Recursive
data Nat: ⋆ =
  | zero: Nat
  | succ: Nat → Nat
.
-- Recursive, parameterized, indexed
data Vec (A: ⋆): Nat → ⋆ =
  | vnil : Vec zero
  | vcons: ∀ n: Nat. A → Vec n → Vec (succ n)
.
```

Figure 1: Definition of natural numbers and length-indexed lists

Figure 1 shows some definitions of inductive datatypes, and modulo differences in syntax should seem straightforward to programmers used to languages like Agda, Idris, or Coq. Some key differences are:

- In constructor type signatures, recursive occurrences of the inductive data-type being defined (such as in `suc :` $\underline{\texttt{Nat}}$ `→ Nat`) must be positive, *but not strictly positive.*

- In parameterized types (like `Vec` with parameter `(A: ⋆)`) occurrences of the inductive type being defined are not written applied to its parameters.

  For example, the constructor declaration `vnil : Vec zero` results in the term `vnil` having type `∀ A: ⋆. Vec ·A zero` (with · denoting type application)

- In the constructor declaration `vcons : ∀ n: Nat. A → Vec n → Vec (succ n)`, the argument `n` is *computationally irrelevant* (or *erased*). This is because it is introduced by the irrelevant dependent function former ∀, as opposed to the relevant function former Π. More will be said of this when we discuss the type system of Cedilleum, but for now it suffices to say that implicit quantification comes from the *Implicit Calculus of Constructions*[Miq01].

1.2 Function Definitions

```
-- Non-recursive
ite : ∀ X: ⋆. Bool → X → X → X
  = Λ X. λ b. λ then. λ else. μ' b {
      | tt → then
      | ff → else
      }.
-- Recursive
add : Nat → Nat → Nat
  = λ n. λ m.
      μ rec. n @(λ x: Nat. Nat) {
      | zero   → m
      | succ p → succ (rec p)
      }.
```

Figure 2: Functions over inductive datatypes

Figure 2 shows functions defined over inductive datatypes using pattern matching and recursion. The first difference to note between the definitions is that `ite` performs "mere" pattern matching on its argument by using `μ'`, whereas `add` uses μ which provides combined pattern-matching and fix-point recursion. In `add`, μ binds `rec` as the name of the fixpoint function for recursion on `n`. From this alone the reader might expect that `μ'` is merely syntactic sugar for the more verbose μ but without recursion. Actually the difference is a bit more subtle that this, as we will see below in Section 1.3

The first major departure of Cedilleum from other languages with inductive datatypes can be seen in the type of `rec`. The type that the reader might expect it to have is `Π x: Nat. Nat` (corresponding to the motive `(λ x: Nat. Nat)`), but in Cedilleum, its type is `rec/type → Nat` (where we read `rec/type` as a single identifier) and by extension for the expression `rec p` to be well-typed, the variable `p` bound in the pattern `succ p` must have type `rec/type`. The name `rec/type` is locally bound in the body of the μ-expression (delimited by curly braces `{...}`) and is automatically generated by Cedilleum by using the name of the recursive function bound by μ (here `rec`).

Why introduced this new type for the predecessor and, in general, for the recursive occurrences of the data-type in constructor sub-data? The answer is: for *termination checking.* Typically, languages that support pattern-matching and fixpoint-style recursion for inductive data-types require all recursive calls be made on "obviously smaller" arguments. In Cedilleum, the syntactic check is replaced by type-checking, and types like `rec/type` encode the fact that some data is indeed "obviously smaller" than the previous invocation and thus legal to recurse on. These "recursive-occurence" types appear in the types of sub-data (such as `p` in the example) in the constructor patterns of the case branches introduced by μ (but not necessarily μ'), replacing all occurrences of the inductive type itself.

```
-- Recursive, parameterized, indexed
vappend : ∀ A: ⋆. ∀ m: Nat. ∀ n: Nat. Vec ·A m → Vec ·A n → Vec ·A (add m n)
  = Λ A. Λ m. Λ n. λ xs. λ ys.
      μ rec. xs @(λ i: Nat. λ zs: Vec ·A i. Vec ·A (add i n)) {
      | vnil             → ys
      | vcons -m' x xs' → [ zs = rec -m' xs' ] - vcons -(add m' n) x zs
      }.
```

Figure 3: Dependent functions over inductive datatypes

Figure 3 shows the classic dependent function `vappend` over `Vec`, the type of length-indexed lists. Like `add`, it is defined by fixpoint recursion, here over the argument `xs`. Here the fixpoint function `rec` has type `∀ i: Nat. Π zs: rec/type i. Vec ·A (add i n)`, where the recursive-occurence type has kind `Nat → ⋆`. Note again the missing parameter `A` in the type `rec/type i` – this is not a typo, but rather an indication that `A` is "baked-in" to the type `rec/type`. Aside from this the two cases of `vappend` are mostly straightforward: in the `vnil` branch the expected type is `Vec ·A (add zero n)` which converts to `Vec ·A n`, so `ys` suffices; in the `vcons` branch we bind subdata `m': Nat`, `x: A`, and `xs': rec/type m'`, with `-m'` indicating that `m'` is bound *irrelevantly*, then we make a local biding `zs` by invoking recursive function `rec` on `m'` and `xs'` (where here `-m'` indicates `m'` is an irrelevant *argument* to `rec`) before producing a result whose type is convertible with the expected `Vec ·A (add (suc m') n)`.

1.3 Course-of-Value Recursion

By supporting type-guided termination checking, Cedilleum enables a form of reasoning more powerful than what is available with a simple syntax-based termination checker: "course-of-value" recursion (and induction), allowing the programmer to recurse on arbitrarily deeply-nested sub-data in a provably terminating way. A compelling example of this can be seen when trying to implement division of natural numbers through iterated subtraction. In a language like Haskell, one may simply write:

```
-- Haskell snippet
(/) 0 d = 0
n / 0 = n
n / d = 1 + ((n - d) / d)
```

This definition of division is terminating for all input; however, syntactic termination checkers will usually have a hard time seeing that the argument `n - d` of the recursive call to `divide` is legal, as it is in general hard to determine whether some arbitrary function produces a structurally smaller result than one of its inputs. Intuitively, we understand that subtracting `d` from `n` never produces a number larger than `n`, and one advantage that Cedilleum's type-guided termination checking has is that it allows us use this intuition to define a version of division in Figure 4 very similar to the one above and which is "obviously" terminating.

Our first function, `pred'`, is crucial for defining `divide` further below. The type `Nat/Mu ·R` in its type signature is the type of *witnesses* that terms of type `R` can be pattern-matched (using μ') as if they had type `Nat`; in the definition of `pred'`, this witness is given name `muWit`. All such witnesses are introduced in only one of two ways: once *globally* for each defined data-type (like `Nat` itself – in the definition of `pred` the term `Nat/mu` has type `Nat/Mu ·Nat`); and *locally* for each recursive-occurence type introduce by μ (in the body of `divide`, term `rec/mu` has type `Nat/Mu ·rec/type`). In the definition of `pred'`, the notation `μ'<muWit>` indicates that the witness `muWit` is given explicitly to enable (mere) pattern-matching on argument `r`. After this, the definition of `pred` is easy – it is an instance of `pred'` where `R` is specialized to `Nat` itself with evidence `Nat/mu`.

Next we define `minus'`. One intuition for the type signature `∀ R: ⋆. Nat/Mu ·R ⇒ R → Nat → R` of `minus'` is that it says "this function never returns a number larger than it's `R` argument." That is to say, to return a term of type `R`, `minus'` can only return its `R` argument directly or some sub-data produced by

```
pred' : ∀ R: ⋆. Nat/Mu ·R ⇒ R → R
  = Λ R. Λ muWit. λ r. μ'<muWit> r {| zero → r | succ p → p}.

pred : Nat → Nat = pred' -Nat/mu .

minus' : ∀ R: ⋆. Nat/Mu ·R ⇒ R → Nat → R
  = Λ R. Λ muWit. λ m. λ n. μ rec. n @(λ _: Nat. R) {
  | zero   → m
  | succ n' → pred' -muWit (rec n')
  }.

minus : Nat → Nat → Nat = minus' -Nat/Rec .

lt : Nat → Nat → Bool
  = λ m. λ n. μ' (minus m n) {
  | zero   → ff
  | succ _ → tt
  }.

divide : Nat → Nat → Nat
  = λ n. λ d. μ rec. n @(λ _: Nat. Nat) {
  | zero   → zero
  | succ n' → succ (rec (minus' -rec/mu n' (pred d)))
  }.
```

Figure 4: Course-of-value recursion and division

pattern-matching on it. In `pred'` pattern-matching (via μ') is done only once; in `minus'` it is done `n` times by recursion on subtrahend `n` by invoking `pred'` each time.

Finally, we turn to the definition of `divide` itself. At a high level, we recurse on dividend `n`, and in the `zero` case simply return `zero`. In the successor case, we subtract the (predecessor of) the divisor from the (predecessor of) the dividend, call `rec` on the result, and then add one with `succ`. We must use `minus'` to perform subtraction so that the call to `rec` is well-typed; note how this prevents the user from writing `rec (minus (succ n') d)`, which if typeable would diverge when the divisor `d` is `zero`.

1.4 Subtyping and Coercions

The reader may wonder at this point what other ways `Nat` and recursive-occurence types like `rec/type` are related when there is a witness of type `Nat/Mu ·rec/type`. In addition to allowing the user to pattern-match on terms of type `rec/type` as they would with e.g. `Nat`, Cedilleum provides a way for users to coerce (with zero runtime cost) such terms to the concrete data-type. As a motivating example, consider trying to implement the factorial function: in the constructor case `succ n'`, we want to multiply the original number by the factorial (calculated via recursion) of its predecessor. Two implementations of factorial are shown in Figure 5, showing how this conversion is done in Cedilleum.

In the first version, `fact1`, an explicit cast `Nat/cast -rec/mu n'` is used to convert the `n'` of type `rec/type` to a `Nat`; `Nat/cast` is a function automatically generated from the definition of datatype `Nat` and has type `∀ R: ⋆. Nat/Mu ·R ⇒ R → Nat`. Furthermore, Cedilleum's built-in equality type recognizes that this term is equal to the identity function (after erasure – recall that `-rec/mu` inducates that the witness is given as an irrelevant or erased argument to `Nat/cast`); this is witnessed by the proof `NatCast-id`, which holds by conversion checking.

Frequently, the insertion of such casts can be deduced merely by comparing the "expected" and "actual"

```
mult : Nat → Nat → Nat
  = λ m. λ n. μ rec. m {
    | zero    → zero
    | succ m' → add n (rec m')
  }.
fact1 : Nat → Nat
  = λ n. μ rec. m {
    | zero    → succ zero
    | succ n' → mult (succ (Nat/cast -rec/mu n')) (rec n')
  }.

NatCast-id : {Nat/cast ≃ λ x. x} = β.

fact2 : Nat → Nat
  = λ n. μ rec. m {
    | zero    → succ zero
    | succ n' → mult (succ n') (rec n')
  }.
```

Figure 5: Factorial with explicit and implicit coercions

type of an expression, and having these coercions stated explicitly is both tedious to write and to read. To that end, the type system of Cedilleum implements a form of subtyping between the recursive-occurence types and the concrete data-type. Behind the scenes, type inference will automatically insert the appropriate coercions (possibly after η-expanding the expression). A simple example of this is shown in definition `fact2`: the variable `n'` has "actual" type `rec/type`, and its "expected" type in expression `succ n'` is `Nat`, so the expression is well typed through type subsumption since `rec/type <: Nat`, as witnessed by evidence `rec/mu` of type `Nat/Mu ·rec/type` (here bound but not used explicitly).

1.5 Reasoning via Induction

```
add-zero-r : Π m: Nat. {add m zero ≃ m}
  = λ m. μ ih. m @(λ x: Nat. {add x zero ≃ x}) {
         | zero   → β
         | succ r → χ {succ (add r zero) ≃ succ r} - ρ (ih r) - β
         } .
```

Figure 6: A proof via induction

Figure 6 shows a simple proof that `zero` is the right identity of `add` using induction on `Nat`. In the base case, pattern `zero` is substituted for `x` in the motive, and the expected result type of the branch is {`add zero zero ≃ zero`}, which is true by reflexivity (notated β) after conversion. In the step case, pattern `succ r` (equivalently `succ (Nat/cast -ih/mu r)`) is substituted in for `x` in the motive, and the expected result type of the branch is {`add (succ r) zero ≃ succ r`} (`Nat/cast -ih/mu r` reduces to `r`). Operator χ allows users to write type annotations, and here it is used to converted the expected type to {`succ (add r zero) ≃ succ r`}. Next, the ρ operator allows the user to rewrite the expected type using an equation, and here the equation used is the one given by the inductive hypothesis `ih r` of type {`add r zero ≃ r`}. After rewriting, the expected type is simply {`succ r ≃ succ r`} which holds by β.

1.6 Reduction Rules of μ and μ'

Section 1.5 omits some details about checking convertibility of terms defined using μ and μ'. For example, in Figure 6 the expected type corresponding to the branch `succ r` in the definition of `add-zero-r` is {`add (succ r) zero` ≃ `succ r`}. By β-reduction and erasure alone, this reduces to

```
{ μ rec. (succ r) {
  | zero → zero
  | succ p → succ (rec p)
  } ≃ succ r
}
```

To get the left-hand side of this equation to be convertible with `succ (add r zero)`, we need a μ-reduction rule. μ-reduction is a combination of fix-point unrolling and case-branch selection, the latter of which is usually called δ-reduction for languages with inductive data-types. Here, because the scrutinee is `succ r`, then entire μ-expression reduces to the body of the case-branch guarded by `succ p` (case-branch selection), with recursive function `rec` replaced by the entire μ-expression itself (fixpoint unrolling). Thus, the equation above reduces to

```
{ succ (μ rec. r {
  | zero → zero
  | succ p → succ (rec p)
  }) ≃ succ r
}
```

where the left-hand side is now convertible with `succ (add r zero)`. μ'-reduction only performs case-branch selection.

1.7 Non-strictly Positive Datatypes

In the preceeding sections, we have that seen "cast" functions like `Nat/cast -ih/mu` (in Section 1.5) show up in the the expected type of a case branch, and also have noted already that Cedilleum allows for positive but not strictly positive data type defintions. We now examine how these two things interact.

Figure 7 presents a definition of `PTree`, an infinitary tree which is not strictly positive in the `node` constructor, and two proofs of induction for it, one using explicit coercions and one utilizing subtyping to infer these coercions. As a type, `PTree` is a somewhat contrived example, but one intuition for what kind of terms inhabit it is "at a `node`, there must be some way of selecting some sub-tree using a predicate `PTree → Bool`".

In both versions, the branch given by patterm `leaf` corresponds to the `base` case, requiring a proof of `P leaf`. For the `step` case, the expected type is `P (node s)` (equivalently `P (node s2)`, where `s2` is locally defined in `indPTree1`). Here, `s` has type `(ih/type → Bool) → ih/type`, and the two different occurences of `s` in the arguments to `step` require it to have two different but related types, corresponding resp. to the types of `s2` and `s1` in `indPTree1`. Again, the subtyping problems `(ih/type → Bool) → ih/type <: (PTree → Bool) → PTree` and `(ih/type → Bool) → ih/type <: (PTree → Bool) → ih/type` can be solved, with coercions implicitly inserted, algorithmically by inverting the type constructors of the sub- and super-types, so definition `indPTree2` is also admissable.

1.8 Program Reuse

We conclude our informal introduction to Cedilleum with a somewhat more complex example: how to support program reuse over different data-types at zero run-time cost. Often, when working in a setting with dependent types, programmers find they must write several different versions of a datatype depending upon the invariants they wish to enfore by the type system. A classic example is non-indexed `List` and length-indexed `Vec` – if the programmer has writen several functions for the former, and discovers that they

```
data PTree : ⋆ =
  | leaf : PTree
  | node : ((PTree → Bool) → PTree) → PTree
.

indPTree1 : ∀ P: PTree → ⋆.
    P leaf → (∀ s: (PTree → Bool) → PTree. (Π p: PTree → Bool. P (s p)) → P (node s)) →
    Π t: PTree. P t
  = Λ P. λ base. λ step. λ t. μ ih. t @(λ x: PTree. P x) {
  | leaf → base
  | node s →
    [ s1 : (PTree → Bool) → ih/type = λ p. s (λ t. p (Nat/cast -ih/mu p)) ]
  - [ s2 : (PTree → Bool) → PTree = λ p. Nat/cast -ih/mu (s1 p) ]
  - step -s2 (λ p. ih (s1 p))
  }.

indPTree2 : ∀ P: PTree → ⋆.
    P leaf → (∀ s: (PTree → Bool) → PTree. (Π p: PTree → Bool. P (s p)) → P (node s)) →
    Π t: PTree. P t
  = Λ P. λ base. λ step. λ t. μ ih. t @(λ x: PTree. P x) {
  | leaf → base
  | node s → step -s (λ p. ih (s p))
  }.
```

Figure 7: A non-strictly positive infinitary tree

must rework their code because they need the latter, their choices are usually either to re-implement the existing `List` functions for `Vec`, or to write conversion functions between `List` and `Vec` to re-use the existing functions. For this second option, such conversion functions usually must tear down one structure while rebuilding the other, taking linear time. In Cedileum it is possible to define *zero-cost* coercions between `List` and `Vec` and indeed between many different types, provided that certain conditions hold.

To start, we give the definitions for `List` and for function `l2v'` which one could write in virtually any dependently typed language.

```
data List (A: ⋆): ⋆ =
  | nil : List
  | cons : A → List → List
  .

len : ∀ A: ⋆. List ·A → Nat
  = Λ A. λ xs. μ rec. xs {
  | nil → zero
  | cons x xs → succ (rec xs)
  }.

append : ∀ A: ⋆. List ·A → List ·A → List ·A
  = Λ A. λ xs. λ ys. μ rec. xs {
  | nil → ys
  | cons x xs → cons x (rec xs)
 }.
```

```
l2v' : ∀ A: ⋆. Π xs: List ·A. Vec ·A (len xs)
  = Λ A. λ xs. μ ih. xs @(λ x: List ·A. Vec ·A (len x)) {
  | nil       → vnil ·A
  | cons x xs → vcons -(len (List/cast -ih/mu xs)) x (ih xs)
  }.
```

Next we see something unique to Cedilleum: the pairs of constructors `nil` and `vnil`, and `cons` and `vcons`, are provably equal. This is necessary (though not sufficient) for proving `l2v-reflection`, which states that conversion function `l2v` behaves extensionally like an identity function.

```
-- constructor equalities
l2v-eq-nil  : {nil  ≃ vnil}  = β.
l2v-eq-cons : {cons ≃ vcons} = β.

-- reflection law
l2v-reflection : ∀ A: ⋆. Π xs: List ·A. {l2v xs ≃ xs}
  = Λ A. λ xs. μ ih. xs @(λ x: List ·A. {l2v x ≃ x}) {
  | nil       → χ {nil ≃ vnil} - β
  | cons x xs → χ {vcons x (l2v xs) ≃ cons x xs}
  - ρ (ih xs) - β
  }.
```

How is it that Cedilleum decides these constructors are convertible? The precise details depend upon the shape of the λ-terms to which Cedilleum elaborates the data-type declarations, but, if `c` is a constructor of type `C` and `d` is a constructor of type `D` (not necessarily distinct for `C`), then the general rules are:

1. `C` and `D` must have the same number of constructors.

   In the definition `data Unit: ⋆ = | triv : Unit .`, `triv` would not be equal to any constructor of any data-type seen so far in this document, as `Unit` has only one constructor and `Nat`, `Bool`, `List`, `Vec`, and `PTree` all have two constructors.

2. `c` and `d` must occur in the same order in the constructor list of their respective data-type declaration

   For example, `nil` and `vnil` are both the first listed constructor for their types, but for the data-type

   ```
   data List' (A: ⋆): ⋆ =
     | cons' : A → List' → List'
     | nil'  : List'
   .
   ```

   which is isomorphic to `List`, constructor `nil'` is *not* convertible with `nil` and zero-cost reuse between `List` and `List'` is *not* possible. This same condition also prevents different constructors of the same type from being seen as convertible (e.g. `tt` and `ff` of type `Bool` are provably distinct).

3. constructors `c` and `d` must take the same number of unerased arguments. Erased arguments, and even the types of unerased arguments, do not matter.

   This is how, in particular, `cons` and `vcons` can be equated even though `vcons` takes an additional (erased) `Nat` argument. This also means that some strange constructor equalities hold:

   ```
   eq-zero-tt   : {zero ≃ tt} = β. -- {succ ≃ ff} is not provable
   eq-zero-leaf : {zero ≃ leaf} = β.
   eq-succ-node : {succ ≃ node} = β.
   ```

Despite the fact that each constructor of `Nat` is convertible with a constructor of `PTree`, it is not possible to define a conversion function `n2pt : Nat → PTree` for which `∀ x: Nat. {n2pt x ≃ x}` is provable, which (we will see next) is needed for zero-cost conversions.

To recapitulate, we have a linear-time conversion function `l2v'` that we proved behaves extensionally like an identity function. With this, and with the term construct ϕ, we can write a conversion function that after erasure is *intensionally* equal to the identity:

```
l2v : ∀ A: ⋆. Π xs: List ·A. Vec ·A (len xs)
  = Λ A. λ xs. φ (l2v-reflection xs) - (l2v' xs) {xs}.

eq-l2v-id = {l2v ≃ λ x. x} = β.
```

In `l2v`, the entire ϕ expression erases to the term in curly braces (`{xs}`), has the type `Vec ·A (len xs)` of subexpression `l2v' xs`, and requires a proof that these two terms are equal (which is satisfied by `l2v-reflection xs`).

From this point, the reuse is straight-forward: define reuse in the other direction (re-use of `Vec` as `List`) with `v2l`, lemma `v2l-len` relating the vector index to the length of a list after conversion, lemma `append-len` relating the length of the list computed by `append` to its two input lists, and finally `vappend'` which works by casting its two input `Vec`s to `List`s, `append`ing them, and casting the result back to `Vec`. The payoff comes at the last line of the code listing below: `v2l-eq-append` proves that function `vappend'` is definitionally equal to `append`, meaning our conversions between the two data-structures required no run-time cost!

```
v2l' : ∀ A: ⋆. ∀ n: Nat. Vec ·A n → List ·A
  = Λ A. Λ n. λ xs. μ rec. xs {
  | vnil → nil ·A
  | vcons -i x xs → cons x xs
  }.

v2l-reflection : ∀ A: ⋆. ∀ n: Nat. Π xs: Vec ·A n. {v2l' xs ≃ xs}
  = Λ A. Λ n. λ xs. μ ih. v @(λ i: Nat. λ x: Vec ·A i. {v2l' x ≃ x}) {
  | vnil → β
  | vcons -i x xs →
    χ {cons x (v2l' xs) ≃ vcons x xs}
  - ρ (ih -i xs) - β
  }.

v2l : ∀ A: ⋆. ∀ n: Nat. Vec ·A n → List ·A
  = Λ A. Λ n. λ xs. φ (v2l-reflection -n xs) - (v2l' -n xs) {xs}.

v2l-len : ∀ A: ⋆. ∀ n: Nat. Π xs: Vec ·A n. {n ≃ len (v2l xs)}
  = Λ A. Λ n. λ xs. μ ih. xs @(λ i: Nat. λ x: Vec ·A i. {i ≃ len (v2l x)}) {
  | vnil → β
  | vcons -n' x xs → ρ (ih -n' xs) - β
  }.

append-len : ∀ A: ⋆. Π xs: List ·A. Π ys: List ·A.
    {add (len xs) (len ys) ≃ len (append xs ys)}
  = Λ A. λ xs. λ ys. μ ih. xs @(λ x: List ·A. {add (len x) (len ys) ≃ len (append x ys)}) {
  | nil → β
  | cons x xs →
    χ {succ (add (len xs) (len ys)) ≃ succ (len (append xs ys))}
  - ρ (ih xs) - β
  }.

vappend' : ∀ A: ⋆. ∀ m: Nat. ∀ n: Nat. Vec ·A m → Vec ·A n → Vec ·A (add m n)
  = Λ A. Λ m. Λ n. λ xs. λ ys.
    [ xs' = v2l -m xs] - [ m-eq = v2l-len -m xs ]
  - [ ys' = v2l -n ys] - [ n-eq = v2l-len -n ys ]
  - ρ m-eq - ρ n-eq - ρ (append-len (v2l -m xs) (v2l -n ys))
  - l2v (append xs' ys').

v2l-eq-append : {append ≃ vappend'} = β.
```

2 Syntax

id			identifiers for definitions
u			term variables
c			constructors
X, Y, Z, R			type variables
κ			kind variables
x	$::=$	$id \mid u \mid X$	non-kind variables
y	$::=$	$x \mid \kappa$	all variables

Figure 8: Identifiers

Identifiers We now turn to a more formal treatment of Cedilleum. Figure 8 gives the metavariables used in our grammar for identifiers. For convenience 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 only legal as term and type identifier, and κ`myDeff` is only legal as a kind identifier.

f, p	$::=$	u, v	variables
		$\lambda\, u.\, p$	functions
		c	constructors
		$f\ p$	applications
		$\mu\ u\ .\ p\ \{c_i\ \overline{a_i} \to p_i\}_{i=1..n}$	recursive definitions
		$\mu'\ p\ \{c_i\ \overline{a_i} \to p_i\}_{i=1..n}$	case analysis

Figure 9: Untyped terms

Untyped Terms The grammar of pure (untyped) terms that of the untyped λ-calculus augmented with primitive μ for combined pattern-matching and fixpoint recursion and μ' for "mere" pattern-matching.

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	::=	$\textbf{module}\ id\ \textbf{.}\ imprt^*\ cmd^*$	module declarations
$imprt$	::=	$\textbf{import}\ id\ \textbf{.}$	module imports
cmd	::=	$defTermOrType$	definitions
		$defDataType$	
		$defKind$	
$defTermOrType$	::=	$id\ checkType^?\ \textbf{=}\ t\ \textbf{.}$	term definition
		$id\ \textbf{:}\ K\ \textbf{=}\ T\ \textbf{.}$	type definition
$defKind$	::=	$\kappa = K$	kind definition
$defDataType$	::=	$\textbf{data}\ id\ param^*\ \textbf{:}\ K\ \textbf{=}\ constr^*\ \textbf{.}$	datatype definitions
$checkType$	::=	$\textbf{:}\ T$	annotation for term definition
$param$	::=	$\textbf{(}x\ \textbf{:}\ C\textbf{)}$	
$constr$	::=	$\mid id\ \textbf{:}\ T$	

Figure 10: 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 in constructor type signatures), 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  : Vec zero
  | vcons : ∀ n: Nat. A → Vec n → Vec (succ n)
  .
```

Sorts $\mathcal{S}$	::=	$\Box$	sole super-kind
		K	kinds
Classifiers C	::=	K	kinds
		T	types
Kinds K	::=	$\Pi\ x : C\ .\ K$	explicit product
		$C \to K$	kind arrow
		$\star$	the kind of types that classify terms
Types S,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
		X	type variable

Figure 11: 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 11 and terms are listed in Figure 12 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 abstractors being right-associative and applications being left-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 (●) 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.

[1]Cite

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	::=	$\lambda\, x.\, t$	normal abstraction
		$\Lambda\, x.\, t$	erased abstraction
		$[\ defTermOrType\ ]$ - t	let definitions
		$\rho\ t\ @x.T$ - t'	equality elimination by rewriting
		$\varphi\ t$ - t' $\{t''\}$	type cast
		$\chi\ T$ - t	check a term against a type
		δ - t	ex falso quodlibet
		$t\ t'$	applications
		t -t'	application to an erased term
		$t\ \cdot T$	application to a type
		β	reflexivity of equality
		$\mu\ u\ .\ t\ @P\ \{c_i\ \overline{a_i} \to t_i\}_{i=1..n}$	recursive definitions
		$\mu'\ t\ @P\ \{case^*\}$	auxiliary pattern match
		u	term variable
		(t)	
		●	
case	::=	$\mid c\ vararg^* \to t$	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 12: 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 Section 4, 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          → tt
           | vcons -n x xs → ff
           }.
```

[2]Module erasure, discussed below

```
vlength : ∀ A: ⋆. ∀ n: Nat. Vec ·A n → Nat
        = Λ A. Λ n. λ xs. μ len . xs @(Λ n . λ x . Nat) {
            | vnil          → zero
            | vcons -n x xs → suc (len -n xs)
            }.
```

3 Erasure and Reduction

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

Figure 13: Erasure for annotated terms

The definition of the erasure function given in Figure 13 takes the annotated terms from Figures 11 and 12 to the untyped terms of Figure 9. The last two equations indicate how the sequence of variables (*varargs*) bound by a constructor pattern are erased. The additional constructs introduced in the annotated term language such as β, ϕ, and ρ, are also all erased to the language of pure terms.

Reduction rules are defined for the untyped term language. In essence, to run a Cedilleum program you first erase it, then reduce it. Full conversion in Cedilleum is defined as the compatible closure of $\rightsquigarrow\ =\ \rightsquigarrow_{\beta} \bigcup \rightsquigarrow_{\mu'} \bigcup \rightsquigarrow_{\mu}$

β-reduction

$$(\lambda\, x.\, p_1)\ p_2 \rightsquigarrow_{\beta} [p_2/x]p_1$$

The rule for β-reduction is standard: those expressions consisting of a λ-abstraction as the left component of an application reduce by having their bound variable substituted away by the given argument (where $[p_2/x]$ is the simultaneous and capture-avoiding substitution of p_2 for x)

μ'-reduction

$$\mu'\ (c_i\ p_1...p_n)\ \{...\ |c_i\ u_1...u_n \mapsto f\ |...\} \rightsquigarrow_{\mu'} [p_1...p_n/u_1...u_n]f$$

μ'-reduction is a simple pattern-matching reduction rule: if the scrutinee of μ' is some variable-headed application $c_i\ p_1...p_n$ where the head c_i matches one of the branch patterns, replace the entire expression with the branch body f after substituting each of the bound variables of the branch pattern $u_1...u_n$ with the scrutinee's arguments $p_1...p_n$

μ-reduction

$$\mu\ u.(c_k\ \overline{t})\ \{c_i\ \overline{x_i} \mapsto f_i\}_{i=1..n} \rightsquigarrow_{\mu} [p_\mu/u][\overline{t/x_k}]\ f_k$$

where $p_\mu = \lambda\, v.\, \mu\ u.v\{c_i\ \overline{x_i} \mapsto f_i\}_{i=1..n}$

μ-reduction is similar to μ'-reduction, but combines with it fixpoint reduction. Again, if the scrutinee $c\ p_1...p_n$ matches one of the branch patterns $c_i\ u_{i1}...u_{ij_i}$ (for some i, where $j_i = n$), then we replace the original μ expression with the matched branch, replacing each of the pattern variables $u_1...u_n$ with the scrutinee's arguments $p_1...p_n$, but *in addition* we also replace the μ-bound variable u (which represents the entire μ expression itself) with a function p_μ that takes its argument v and re-creates the original μ expression by scrutinizing v.

4 Type System (sans Inductive Datatypes)

Figure 14: Contexts

$$\text{Typing contexts } \Gamma \quad ::= \quad \emptyset \mid x\!:\!C, \Gamma \mid x = s\!:\!C, \Gamma$$

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

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

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

Figure 15: Sort checking $\boxed{\Gamma \vdash C : S}$

The inference rules for classifying expressions in Cedilleum are stratified into two judgments. Figure 15 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 $\texttt{ff} = \lambda\, x.\, \lambda\, y.\, y$

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

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

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

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

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

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

Figure 16: Type checking $\boxed{\Gamma \vdash_\delta s : C}$ (sans inductive datatypes)

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 remaining primitive χ allows the user to provide an explicit top-level annotation for a term.

5 Inductive Datatypes

While the grammatical rule *defDataType* gives the concrete syntax for datatype definitions, it is not a very useful notation for representing and manipulating such an object in the AST. We begin this section, then, by describing a more concise syntax for datatype definitions. The notation used in this section borrows heavily from the conventions of the Coq documentations [5]. One additional abuse of notation we shall use heavily throughout the remainder of this document is for application and abstraction of a sequence of terms and types. If Γ is an ordered context binding term and type variables, then

- $t\ \Gamma$ and $T\ \Gamma$ represent the application of term t (resp. type T) to each variable in Γ in order of appearance. The erasure modality of the application – that is, for each variable x in Γ, whether it is passed as a relevant or irrelevant argument to t – will always be disambiguated by the type of term t (there is no erased application at the type level).
- $\overset{\lambda}{\Lambda}\,\Gamma.\, t$ and $\lambda\,\Gamma.\, T$ represents a sequence of abstractions at the term (resp. type) level, followed by term t (resp. type T). At the term level, the appropriate abtraction (erased or unerased) is determined by

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

the expected type of the expression and the sort of the variable (e.g. at the term level all types are abstracted over erased).

5.1 Representation of Datatype Definition in AST

Notation $\texttt{Ind}[I, \Gamma_P, \Gamma_K, R, \Sigma, \Gamma_G]$ represents a declaration of an inductive datatype named I where:

- Γ_P is the context of parameters;
- Γ_K binds the indices of type I; that is to say type $I\ \Gamma_P$ has kind $\Pi\ \Gamma_K : \star$;
- R is a (fresh) type variable of kind $\Pi\ \Gamma_K.\star$, serving as a placeholder for recursive occurrences of the inductively defined type in the type signatures of the data constructors;
- Σ is the context associating constructors with their type signatures;
- Γ_G binds additional fresh (automatically generated) identifiers in the global context which help enable CoV induction – more on this below.

For example, the datatype declaration for `Vec` in the concrete syntax:

```
data Vec (A: ⋆): Nat → ⋆ =
  | vnil  : Vec zero
  | vcons : ∀ n: Nat. A → Vec n → Vec (succ n)
  .
```

corresponds to the following object in the abstract syntax:

$$\texttt{Ind}[\texttt{Vec}, \texttt{A}{:}\star, \Pi\ \texttt{n}{:}\texttt{Nat}.\star, \texttt{R}, \Sigma, \Gamma_G]$$

where

$$\Sigma \quad = \quad \begin{array}{lcl} \texttt{vnil} & : & \forall \texttt{A}{:}\star.\texttt{Vec}\ \cdot\texttt{A zero} \\ \texttt{vcons} & : & \forall \texttt{A}{:}\star.\forall \texttt{n}{:}\texttt{Nat}.\texttt{A} \rightarrow \texttt{R n} \rightarrow \texttt{Vec}\ \cdot\texttt{A (S n)} \end{array}$$

$$\Gamma_G \quad = \quad \begin{array}{lcl} \texttt{Is/Vec} & : & \Pi\ \texttt{A}{:}\ \star.\ \ (\texttt{Nat} \rightarrow \star) \rightarrow \star \\ \texttt{is/Vec} & : & \forall\ \texttt{A}{:}\star.\ \ \texttt{Is/Vec}\ \cdot\texttt{A}\ \cdot(\texttt{Vec}\ \cdot\texttt{A}) \\ \texttt{to/Vec} & : & \forall\ \texttt{A}{:}\star.\ \ \forall\ \texttt{R}{:}\ \texttt{Nat} \rightarrow \star.\texttt{Is/Vec}\ \cdot\texttt{A}\ \cdot\texttt{R} \Rightarrow \forall\ \texttt{n}{:}\texttt{Nat}.\texttt{R n} \rightarrow \texttt{Vec}\ \cdot\texttt{A n} \\ & = & \lambda\ \texttt{x.x} \end{array}$$

In the above definition for Γ_G, understand that

- `Is/Vec` is an automatically-generated type of "witnesses" that some type can be pattern-matched upon just like `Vec` can; this is exists to support CoV induction
- `is/Vec` is the (trivial) witness that `Vec` behaves like `Vec` as far as pattern-matching is concerned
- `to/Vec` is a coercion from some type `R` to `Vec ·A`, provided there is an `Is/Vec ·A ·R` witness.

  This coercions is "zero-cost" in the sense that it is defined to be equal to λ`x.x`

The purposes of these global definitions will become more clear when we give a formal treatment of μ (combined fixpoint and pattern-matchin) and μ' ("mere" pattern matching) below.

5.2 Well-formedness of Datatype Definition

For an inductive datatype definition $\texttt{Ind}[I, \Gamma_P, \Gamma_K, R, \Sigma, \Gamma_G]$ to be well-formed, it must satisfy the following conditions:

- I must have (well-formed) kind $\Pi\,\Gamma_P.\,\Pi\,\Gamma_K.\,\star$
  Ensuring this is trivial from the concrete syntax
- The type T of each constructor $c{:}T \in \Sigma$ must be a *type of constructor of* I (c.f. Section 5.5)
- The type T of each constructor $c{:}T \in \Sigma$ must satisfy the (non-strict) positivity condition for R (c.f. Section 5.5)
- Γ_G must bind precisely the following (these are added to the global context):
  - $\texttt{Is/}I\texttt{:}\ \ \Pi\ \Gamma_P\texttt{.}\ \ \texttt{K} \to \star$
    The name bound here is literally the string concatenation of "`Is/`" with the user-given name for the data-type I
  - $\texttt{is/}I\texttt{:}\ \ \forall\ \Gamma_P\texttt{.}\ \ \texttt{Is/}I\ \Gamma_P\ \cdot(I\ \Gamma_P)$
  - $\texttt{to/}I\texttt{:}\ \ \forall\ \Gamma_P\texttt{.}\ \ \forall\ \texttt{R: K. Is/}I\ \Gamma_P\ \texttt{R} \Rightarrow \forall\ \Gamma_K\texttt{.}\ \ \texttt{R}\ \Gamma_K \to I\ \Gamma_P\ \Gamma_K\ \texttt{=}\ \lambda\ \texttt{x.x}$

  Collision with user-given definitions is avoided by prohibiting such user-supplied names from having the character "`/`" present.

We will write judgment $\texttt{Ind}[I, \Gamma_P, \Gamma_K, R, \Sigma, \Gamma_G]\ wf$ to indicate that a datatype declaration is well-formed.

5.3 Fixpoint-style recursion and Pattern Matching

Similarl to datatype declarations, the notation used in the concrete syntax of Cedilleum for μ (for combined fixpoint recursion and pattern matching) and μ' (for mere pattern matching) is inconveneient. In the AST we will represent a μ' expression as

$$\mu'[t_s, w, P, \bar{t}]$$

where

- t_s is the scrutinee for case analysis;
- w is the witness that t_s is valid for case-analysis
- P is the motive for (dependent) pattern matching;
- $\bar{t}$ are the case branches;

For a simple example, the μ'-expression in the body of predecessor in Figure 4, `pred'`, would be represented as

$$\mu'[\texttt{r}, \texttt{muWit}, \lambda\texttt{x:Nat.R}, \texttt{r}, \lambda\texttt{p.p}]$$

μ-expressions are represented in the AST as

$$\mu[x_\mu, t_s, P, \Gamma_L, \bar{t}]$$

where

- x_μ is the name given for the function being defined in fixpoint style
- t_s is the scrutinee for case-analysis and whose recursive subdata will recursed upon
- P is the motive for (dependent) pattern-matching

- $\bar{t}$ are the case branches
- Γ_L are (automatically generated) definitions in-scope of the case branches

As an example, in the definition of subtraction in Figure 4, `minus'`, the μ-expressions would be represented as

$$\mu[\texttt{rec}, \texttt{n}, \Gamma_L, \lambda\texttt{n:Nat.R}, \begin{array}{c} \texttt{m} \\ \lambda\ \texttt{n'.pred' -muWit (rec n')} \end{array}]$$

where

$$\Gamma_P \;=\; \begin{array}{lcl} \texttt{Type/rec} & : & \star \\ \texttt{isType/rec} & : & \texttt{Is/Nat}\ \cdot\texttt{Type/rec} \\ \texttt{rec} & : & \Pi\texttt{x:Type/rec.}\ \ \texttt{rec/Type} \end{array}$$

which is to say that μ introduces a fresh type `Type/rec`, a witness `isType/rec` that terms of this type can be further case analysed, and binds recursive function (inductive hypothesis) `rec` which can operate only on terms of the appropriate (recursive) type.

5.4 Well-formedness of μ- and μ'-expressions

5.5 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 $\lambda\,\Gamma.\,t = \lambda\,x_1 : T_1.\,\lambda\,x_2 : T_2.\,t$. Additionally, we will also write $\overset{\Pi}{\forall}\,\Gamma$ to indicate an arbitrary mixture of Π and $\forall$ quantified variables. Note that *if* $\overset{\Pi}{\forall}\,\Gamma$ *occurs multiple times within a definition or inference rule*, the intended interpretation is that *all occurrences have the same mixture of* Π *and* $\forall$ *quantifiers.*
- $\|\Gamma\|$ 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. Implicit in this notation is the removal of typing annotations from the variables Γ when these variables are 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 flavor of application 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 Section 5.11) For example, given just the (global) data type declaration of Vec, we would have $\Delta(Vec) = \texttt{Ind}_{\mathrm{C}}[1](\Gamma_{Vec} : \Sigma =)$, 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 $\texttt{Ind}_M[p](\Gamma_I : \Sigma =)$, 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

5.6 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 17 states when an inductive datatype definition may be introduced, provided that the following side conditions hold:

Figure 17: Introduction of inductive datatype

$$\frac{\emptyset \vdash \Gamma_I(I) : \Box \quad \|\Gamma_P\| = p \quad (\Gamma_I, \Gamma_P \vdash T_i : \star)_{i=1..n}}{\mathtt{Ind}_M[p](\Gamma_I : \Sigma =)wf}$$

- Names I and $c_1..c_n$ are distinct from any other inductive datatype type or constructor names, and distinct amongst themselves
- Each of $T_1..T_n$ is a type of constructor of I which satisfies the positivity condition for I. Furthmore, each occurence of I in T_i is one which is applied to the parameters Γ_P.
- Identifiers I, $c_1, ..., c_n$ are fresh w.r.t the global context, and do not overlap with each other nor any identifiers in Γ_P.

When an inductive data-type has been defined using the *defDataType* production, it is understood that this always a concrete inductive type, and it (implicitly) adds to a global typing context the variable bindings in Γ_I and Σ. Similarly, when checking that the kind $\Gamma_I(I)$ and type T_i are well-sorted and well-kinded, we assume an (implicit) global context of previous definitions.

5.7 Valid Elimination Kind

Figure 18: Valid elimination kinds

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

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 of kind K' can be eliminated through pattern-matching with a motive of kind K". This judgment is defined by the simple rules in Figure 18. For example, a valid elimination kind for the indexed type family $Vec \cdot X$ (which has kind $\Pi\, n{:}Nat.\, \star$) is $\Pi\, n{:}Nat.\, \Pi\, x{:}Vec \cdot X\ n.\, \star$

5.8 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 indicate the type corresponding to the (possibly partially applied) constructor c of I and its type T. 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 c 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.

5.9 Well-formed Patterns

Figure 19: Well-formedness of a pattern

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

Figure 19 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, and 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.

5.10 Generation of Abstracted Inductive Definitions

Cedilleum supports *historphic* recursion (that is, having access to all previous recursive values) 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 $\mathit{Hist}(\Delta, I, \overline{T}, I')$ producing an abstracted (well-formed) inductive definition $\mathtt{Ind}_{\mathrm{A}}[0](\Gamma_{I'} : \Sigma' =)$, where

- $\Gamma_{I'}(I') = \forall\, \Gamma_D .\, \star$ if $\Gamma_I(I) = \forall\, \Gamma_P .\, \forall\, \Gamma_D .\, \star$ (and $\|\Gamma_P\| = \|\overline{T}\| = p$)

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

- $\Sigma' = c_1' {:} \forall\, \Gamma_D .\, \overset{\Pi}{\forall}\, \Gamma_{A_1'} .\, I'\ \Gamma_D, ..., c_n' {:} \forall\, \Gamma_D .\, \overset{\Pi}{\forall}\, \Gamma_{A_n'} .\, I\ \overline{T}\ \Gamma_D,$

  when each of the concrete constructors c_i in Σ are associated with type $\forall\, \Gamma_P .\, \forall\, \Gamma_D .\, \overset{\Pi}{\forall}\, \Gamma_{A_i} .\, I\ \Gamma_P\ \Gamma_D$ and each $\Gamma_{A_i'} = [\lambda\, \Gamma_P .\, I'/I, \overline{T}/\Gamma_P]\Gamma_{A_i}$.

That is, trasforming the concrete constructors of the inductive datatype I to "abstracted" constructors involves replacing each recursive occurrence of $I\ \Gamma_P$ with the fresh type variable I, and instantiating each of the parameters Γ_P with $\overline{T}$.

Users of Cedilleum will see "punning" of the concrete constructors c_i and abstracted constructors c_i'. In particular, when using fix-point pattern matching branch labels will be written with the constructors for the concrete inductive data-type, and the expected type of a branch given by the motive will pretty-print using the concrete constructors. In the inference rules, however, we will take more care to distinguish the abstract constructors (see Subsection 5.11).

5.11 Typing Rules

Figure 20: Use of an inductive datatype $\texttt{Ind}_M[p](\Gamma_I : \Sigma =)$

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

$$\frac{\Gamma \vdash_{\Uparrow} t : I\ \overline{T}\ \overline{s} \quad \Delta(I) = \texttt{Ind}_{\mathrm{C}}[p](I : K = \Sigma) \quad \Gamma_I(I) = \Pi\, \Gamma_P.\, \Pi\, \Gamma_{\mathrm{D}}.\, \star, \|\Gamma_P\| = p \quad \mathit{Hist}(\Delta, I, \overline{T}, I') = \texttt{Ind}_{\mathrm{A}}[0](I' : K = \Sigma')}{\Gamma, \Delta \vdash_{\delta} \mu(x_{\mathrm{rec}}, I', x_{\mathrm{to}}, t, P, t_{i=1..n}) : P\ \overline{s}\ t}$$

The first rule of Figure 20 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 $\mathit{Hist}(\Delta, I, \overline{T}, I')$ 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 $\Lambda\Gamma_D$ 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 $P'\ \Gamma_D\ x$ utilizes x_{to} in P' to be well-typed, as the x in this expression has type $I'\ \Gamma_D$, but P expects an $I\ \overline{T}\ \Gamma_D$. 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'.

6 Elaboration of Inductive Datatypes

As mentioned in Section 1, Cedilleum is not based on CIC. Rather, its core theory is the *Calculus of Dependent Lambda Eliminations* (CDLE), whose complete typing rules can are those of Section 4 plus rules for dependent intersections (see [Stu18]). That is to say, the preceding treatment for inductive datatypes (Section 5) is a high-level and convenient interface for *derivable* inductive λ-encodings. This section explains the elaboration process. Since the generic derivation of inductive data-types with course-of-value induction has been covered in-depth in [TODO], we omit these details and instead describe the *interface* such developments provide which data-type elaboration targets.

At a high level, inductive data-types in Cedilleum are first translated to *identity mappings*, which are (in the non-indexed case) a class of type schemes `F: ⋆ → ⋆` that are more general than functors. The parameter of the identity scheme replaces all recursive occurrences of the data-type in the signatures of the constructor and a quantified type variable replaces all "return type" occurrences. For example, the type scheme for data-type `Nat` is `λ R: ⋆. ∀ X: ⋆. X → (R → X) → X`, with `R` the parameter and `X` the quantified variable. For the rest of this section we assume the reader has at least a basic understanding of impredicative encodings of datatypes (see [PPM89] and [Wad90]) and taking the least fix-point of functors (see [MFP91]).

The following developments are parameterized by an indexed type scheme F of kind `(Π Γ_D. ⋆) → (Π Γ_D. ⋆)` corresponding to the kind `Π Γ_D. ⋆` of inductive data-type I declared as $\texttt{Ind}_I[p](\Gamma_I : \Sigma =)$

6.1 Identity Mappings

Our first task is to describe identity mappings, the class of type schemes `F: (Π Γ_D. ⋆) → Π Γ_D. ⋆` we concerned with. Identity mappings are similar to functors in that they come equipped with a function that resembles `fmap: ∀ Γ_D. ∀ A B: Π Γ_D. ⋆. Π f: (A ·Γ_D → B ·Γ_D). F ·(A ·Γ_D) → F ·(B ·Γ_D)` except that it need only be defined for an argument `f` that is equal to the identity function. We define the type `Id` of such functions and declare (indicated by `<..>`) its elimination principle `elimId_D`:

```
Id_D : Π A B: (Π Γ_D. ⋆). ι id: ∀ Γ_D. A Γ_D → B Γ_D. {id ≃ λ x. x}.
elimId_D : ∀ A B: (Γ_D. ⋆). Id_D ·A ·B ⇒ A → B = <..>
```

Recall that since Cedilleum has a Curry-style type system and implicit products there are many non-trivial functions that erase to identity. While the definition of `elimId_D` is omitted, it is important to note that it enjoys the property of erasing to the identity function:

```
elimId_D-prop : {elimId_D ≃ λ x. x} = β.
```

We may now define `IdMapping` as a scheme `F` that comes with a way to lift identity functions:

```
IdMapping_D : Π F: (Γ_D → ⋆) → (Γ_D → ⋆). ⋆
  = λ F. ∀ A B: (Γ_D → ⋆). Π/∀ Γ_D. Id_D ·A ·B → Id_D ·(F ·A) ·(F ·B).
```

Finally, it is convenient to define `fimap` which given an `IdMapping` and an `Id` function performs the lifting:

```
fimap_D : ∀ F: (Π Γ_D. ⋆) → (Π Γ_D. ⋆). ∀ im: IdMapping_D ·F. Cast_D ·A ·B ⇒ F ·A → F ·B
  = Λ F im c. λ f. elimId_D -(im c) f.
```

From `elimId_D-prop` it should be clear that `fimap_D` also erases to `λ x. x`.

6.2 Type-views of Terms

A crucial component of course-of-value is the ability to view some term as having two different types. The idea behind a `View` is similar to that behind the type `Id` from the previous section, except now we explicitly name the doubly-typed term:

```
View : Π A: ⋆. A → ⋆ → ⋆ = λ A a B. ι b: B. {a ≃ b}
elimView : ∀ A B: ⋆. Π a: A. View ·A a ·B ⇒ B = <..>
elimView-prop : {elimView ≃ λ x. x} = β.
```

6.3 λ-encoding Interface

This subsection describes the interface to which data-type declarations are elaborated; it is parameterized by an identity mapping.

```
module (F_D: (Π Γ_D. ⋆) → (Π Γ_D. ⋆)){im: IdMapping ·F_D}.
```

where parameters $\texttt{F}_\texttt{D}$ and `im` are automatically derived from the declaration of a positive data-type.

With these two parameters alone, the generic developments of [TODO] provide the following interface for inductive λ-encodings of data-types:

```
Fix_D : Π Γ_D. ⋆ = <..>
in_D  : ∀ Γ_D. F_D ·Fix_D Γ_D → Fix_D Γ_D = <..>
out_D : ∀ Γ_D. Fix_D Γ_D → F_D ·Fix_D Γ_D = <..>

PrfAlg_D : Π P: (Π Γ_D. Π d: Fix_D Γ_D. ⋆). ⋆
  = λ P. ∀ R: (Π Γ_D. ⋆).
      ∀ c: Id_D ·R ·Fix_D.
      Π v: View ·(∀ Γ_D. Fix_D Γ_D → F_D ·Fix_D Γ_D) out ·(∀ Γ_D. R Γ_D → F_D ·R Γ_D).
      Π ih: (∀ Γ_D. Π r: R Γ_D. P Γ_D (elimId_D -c -Γ_D r)).
      Π Γ_D. Π fr. F ·R Γ_D.
      P Γ_D (in_D -Γ_D (fimap_D -im -c fr)).
induction_D : ∀ P: (Π Γ_D. Π d: Fix_D Γ_D. ⋆). PrfAlg_D ·P → ∀ Γ_D. Π d: Fix_D Γ_D. P Γ_D d
  = <..>
```

The first three definitions give $\texttt{Fix}_\texttt{D}$ as the (least) fixed-point of $\texttt{F}_\texttt{D}$, with $\texttt{in}_\texttt{D}$ and $\texttt{out}_\texttt{D}$ representing resp. a generic set of constructors and destructors. $\texttt{induction}_\texttt{D}$ of course is the proof-principle stating that if one can provide a `PrfAlg` for property `P` (that is, `P` holds for all $\texttt{Fix}_\texttt{D}$ generated by (generic) constructor $\texttt{in}_\texttt{D}$) then this suffices to show that `P` holds for *all* $\texttt{Fix}_\texttt{D}$.

We now explain the definition of $\texttt{PrfAlg}_\texttt{D}$ in more detail:

- `R` is the type of recursive occurrences of the data-type $\texttt{Fix}_\texttt{D}$.

  It corresponds directly to types like `rec/Nat` when using μ in Cedilleum

- `c` is a "revealer", that is to say a proof that `R` really *is* $\texttt{Fix}_\texttt{D}$ witnessed by an identity function.

  It corresponds directly to functions like `rec/cast` when using μ

- `v` is evidence that the (generic) destructor $\texttt{out}_\texttt{D}$ can be used on the recursive occurrence type `R` for further pattern-matching.

  It corresponds directly to μ' (when used outside of μ it corresponds to the "trivial" view that $\texttt{out}_\texttt{D}$ has the type it is already declared to have).

- `ih` is the inductive hypothesis, stating that property `P` holds for all recursive occurrences `R` of an inductive case

  It corresponds directly to the μ-bound variable for fix-point recursion.

- `fr` represents the collection of constructors that each μ branch must account for.

  For example, for the data-type `Nat` we have identity mapping `fr: ∀ X: ⋆. X → (R → X) → X` and Cedilleum cases branches `{| zero → zcase | succ r → scase r }` translate to `fr zcase (λ r. scase r)`

- Finally, result type `P` $\Gamma_\texttt{D}$ ($\texttt{in}_\texttt{D}$ -$\Gamma_\texttt{D}$ ($\texttt{fimap}_\texttt{D}$ `-im -c fr`)) accounts for the return type of each case branch.

  Since `P` is phrased over $\texttt{Fix}_\texttt{D}$, and we have by assumption `fr`: $\texttt{F}_\texttt{D}$ ·`R` $\Gamma_\texttt{D}$, we must first use our identity mapping `im` to traverse `fr` and cast each recursive occurrence `R` $\Gamma_\texttt{D}$ to $\texttt{Fix}_\texttt{D}$ $\Gamma_\texttt{D}$, producing an expression of type `F` ·$\texttt{Fix}_\texttt{D}$ $\Gamma_\texttt{D}$ which we are then able to transform into $\texttt{Fix}_\texttt{D}$ $\Gamma_\texttt{D}$ using (generic) constructor $\texttt{in}_\texttt{D}$.

While the definitions of $\texttt{in}_\texttt{D}$, $\texttt{out}_\texttt{D}$, and $\texttt{induction}_\texttt{D}$ are omitted, it is important that they have the following computational behavior (guaranteed by [TODO]):

```
lambek1_D : ∀ Γ_D. Π gr: F_D Fix_D Γ_D. {out_D (in_D gr) ≃ gr} = β.
lambek2_D : ∀ Γ_D. Π d: Fix_D Γ_D. {in (out d) ≃ d}
  = induction_D ·(λ Γ_D. λ x: Fix_D Γ_D. {in (out x) ≃ x})
      (Λ R. Λ c. λ o. Λ eq. λ ih. λ gr. β).

inductionCancel_D : ∀ P: (Π Γ_D. Fix_D Γ_D → ⋆).
    Π alg: PrfAlg ·P → ∀ Γ_D. Π fr: F ·Fix_D Γ_D.
    { induction_D alg (in gr) ≃ alg out_D (induction_D alg) fr}
  = λ _. λ _. β.
```

That is, $\texttt{in}_\texttt{D}$ and $\texttt{out}_\texttt{D}$ are inverses of each other and $\texttt{induction}_\texttt{D}$ behaves like a fold (where the algebra takes the additional $\texttt{out}_\texttt{D}$ argument).

6.4 Sum-of-Products Induction

As stated above, every inductive data-type declaration $\texttt{Ind}_I[p](\Gamma_I : \Sigma =)$ is first translated to a type-scheme IF where all recursive occurrences of type I in the constructor signatures Σ have been replaced by the scheme's argument R. In this subsection describe that process more precisely and explain "sum-of-products" induction for IF

First, as the kind of I is $\Pi\ \Gamma_\texttt{p}.\ \Pi\ \Gamma_\texttt{D}.\ \star$, where $\Gamma_\texttt{p}$ are the parameters and $\Gamma_\texttt{D}$ the indices, it follows that the kind of IF is $\Pi\ \Gamma_\texttt{p}.\ \Pi\ \texttt{R}\colon\ (\Pi\ \Gamma_\texttt{D}.\ \star).\ (\Pi\ \Gamma_\texttt{D}.\ \star)$. Next, each constructor c_j has type $\Sigma(c_j)$ which we know has the form $\overset{\Pi}{\forall}\ \Gamma_j.\ I\ \Gamma_p\ \overline{t_j}$ (that is, some number of arguments Γ_j with a return type constructing the inductive data-type I). All recursive occurrences of I in Γ_j are substituted away with $\lambda\ \Gamma_\texttt{p}.\ \texttt{R}$ to produce Γ_j^R. With that, we may defined IF as

$$\lambda\ \Gamma_\texttt{p}\ \texttt{R}\ \Gamma_\texttt{D}.\ \forall X : \Pi\ \Gamma_\texttt{D}.\star.(\Pi\ c_j : (\overset{\Pi}{\forall}\Gamma_j^R.\ X\ \overline{t_j}))_{j=1..n}.\ X\ \Gamma_\texttt{D}$$

Example The data-type declaration of Vec translates to:

```
VecF : Π A: ⋆. (Nat → ⋆) → Nat → ⋆
  = λ A R n. ∀ X: Nat → ⋆. X zero → (∀ n: Nat. A → R n → X (succ n)) → X n.
```

An induction principle for each of these non-recursive sum-of-products types IF can be defined in an automated way following the recipe given by [TODO]; in general these have the following shape:

```
indIF : ∀ Γ_p. ∀ R: (Π Γ_D. ⋆). ∀ Γ_D. Π fr: IF Γ_p ·R Γ_D. ∀ P: (Π Γ_D. IF Γ_p ·R Γ_D → ⋆)
    (Π p_j: Π/∀ Γ^R_j. P (c_j Γ^R_j))_{j=1..n}. P Γ_D fr = <..>
```

A Deriving $\texttt{IdMapping}_\texttt{D}$ for a Data-type Type Scheme

A type scheme F derived from a data-type declaration has by assumption a definition following the pattern:

```
F : Π Γ_p. (Π Γ_D. ⋆) → Π Γ_D. ⋆
  = λ Γ_p R Γ_D. ∀ X: (Π Γ_D. ⋆). (Π c_j: (Π/∀ Γ^R_j. X t̄_j))_{j=1−n}. X Γ_D
```

where R occurs only positively. From this we must give a witness that F is an identity mapping over R

```
idmap : ∀ Γ_p. IdMapping_D ·(F Γ_p)
  = Λ Γ_p. Λ R1. Λ R2. Λ id. ●
```

where the expected type of $\bullet$ is $\texttt{Id}_\texttt{D}\ \cdot(\texttt{F}\ \cdot\Gamma_\texttt{p}\ \texttt{R1})\ \cdot(\texttt{F}\ \cdot\Gamma\ \texttt{R2})$

We refine $\bullet$ by the introduction rule for intersections (which $\texttt{Id}_\texttt{D}$ is) and introduce the assumption $\texttt{fr1: F}\ \cdot\Gamma_\texttt{p}\ \texttt{R1}\ \cdot\Gamma_\texttt{D}$

$[\ \Lambda\ \Gamma_\texttt{D}.\ \lambda\ \texttt{fr1}.\ \bullet_1\ ,\ \bullet_2]$

where $\bullet_1\texttt{: F}\ \cdot\Gamma_\texttt{p}\ \texttt{R2}\ \cdot\Gamma_\texttt{D}$ and $\bullet_2\texttt{: }\{\lambda\ \texttt{fr1}.\ \bullet_1\ \simeq\ \lambda\ \texttt{x. x}\}$. As the only (non-hole) refinements we will make to $\bullet_1$ are converting terms to η-long form and applying $\texttt{elimId}_\texttt{D}$ `-id` to subterms (which reduces to the identity function), we are justified in replacing $\bullet_2$ with β. We now refine the remaining $\bullet_1$ to

$\Lambda\ \texttt{X}.\ \lambda\ \overline{\texttt{c}}.\ \bullet\ \texttt{fr1}\ \overline{\texttt{c}}$

where each abstract constructor $\texttt{c}_\texttt{j}$ in $\overline{\texttt{c}}$ has type $\overset{\Pi}{\forall}\ \Gamma^{\texttt{R2}}{}_\texttt{j}.\ \texttt{X}\ \overline{\texttt{t}}_\texttt{j}$. Note again the superscript `R2` – we are now trying to construct a term of type $\texttt{F}\ \cdot\Gamma_\texttt{p}\ \texttt{R2}\ \cdot\Gamma_\texttt{D}$ so we assume the "abstract" constructors whose recursive occurence types are `R2`. Correspondingly, this means that $\bullet\texttt{: F}\ \cdot\Gamma_\texttt{p}\ \texttt{R1}\ \cdot\Gamma_\texttt{D}\ \to\ (\Pi\ \texttt{c}_\texttt{j}\texttt{: }(\overset{\Pi}{\forall}\ \Gamma^{\texttt{R2}}{}_\texttt{j}.\ \texttt{X}\ \overline{\texttt{t}}_\texttt{j}))_{\texttt{j}=1-n}\ \to\ \texttt{X}\ \Gamma_\texttt{D}$.

Since `fr1` produces a value of type $\texttt{X}\ \Gamma_\texttt{D}$ when fed appropriate arguments, we refine $\bullet$ by n holes $\bullet_\texttt{j}$ applied to constructor $\texttt{c}_\texttt{j}$. The expression $\bullet\ \texttt{fr1}\ \overline{\texttt{c}}$ becomes

$\texttt{fr1}\ (\bullet_\texttt{j}\ \texttt{c}_\texttt{j})_{\texttt{j}=1-n}$

where now $\bullet_\texttt{j}\texttt{: }(\overset{\Pi}{\forall}\ \Gamma^{\texttt{R2}}{}_\texttt{j}.\ \texttt{X}\ \overline{\texttt{t}}_\texttt{j})\ \to\ \overset{\Pi}{\forall}\ \Gamma^{\texttt{R1}}{}_\texttt{j}.\ \texttt{X}\ \overline{\texttt{t}}_\texttt{j}$. We henceforth dispense with the subscript j numbering the constructor and treat each abstract constructor uniformly.

A.1 Conversion of the Abstract constructors

We first make the expression $\bullet\ \texttt{c}$ η-long, as in $\overset{\lambda}{\Lambda}\ \Gamma^{\texttt{R1}}.\ \bullet\ \texttt{c}\ \Gamma^{\texttt{R1}}$, then refine $\bullet\ \texttt{c}\ \Gamma^{\texttt{R1}}$ to an expression with m holes $\bullet_\texttt{k}$ for each $\texttt{y}_k \in \Gamma^{\texttt{R1}}$ (where $m\ =\ \|\Gamma^{\texttt{R1}}\|$), yielding

$\texttt{c}\ (\bullet_\texttt{k}\ \texttt{y}_\texttt{k})_{\texttt{k}=1-m}$

where $\bullet_\texttt{k}\texttt{: }\Gamma^{\texttt{R1}}(\texttt{y}_\texttt{k})\ \to\ \Gamma^{\texttt{R2}}{}_\texttt{k}(\texttt{y}_\texttt{k})$ (and the type of $\texttt{y}_\texttt{k}$ and $\bullet_\texttt{k}\ \texttt{y}_\texttt{k}$ can depend resp. on any $\texttt{y}^{\texttt{R1}}{}_\texttt{j}$ and $\bullet_\texttt{j}\ \texttt{y}_\texttt{j}$ where $j < k$). We now dispense with the subscript k for arguments and handle each constructor sub-data uniformly.

A.2 Conversion of Constructor Sub-data With Positive Recursive Occurences

We now consider $\bullet\ \texttt{y}$ where `y: S` is some sub-data to an (abstract) constructor with recursive occurence type `R1` passing the positivity checker. (The expression $\bullet\ \texttt{y}$ has type `[R2/R1]S`). There are two cases to consider:

1 `R1` does not occur in the type of `y`

Refine $\bullet$ to $\texttt{unit: }\forall\ \texttt{X: }\star.\ \texttt{X}\ \to\ \texttt{X}\ =\ \Lambda\ \texttt{X}.\ \lambda\ \texttt{x. x}$ and finish.

2 `R1` occurs positively in the type of `y`

This means S has the shape $\overset{\Pi}{\forall}\ \Gamma^{\texttt{R1}}{}_x.\ \texttt{T}$ (where `T` is not formed by an arrow) with `R1` occurring *only negatively* in the type of the $\texttt{x}_\texttt{j} \in \Gamma^{R1}_x$ (where $j = 1..\|\Gamma^{R1}_x\|$). Make $\bullet\ \texttt{y}$ η-long and refine the expression to $\|\Gamma^{R1}_x\|$ holes $\bullet_\texttt{j}$ such that the expression is now

$\overset{\lambda}{\Lambda}\ \Gamma^{\texttt{R2}}{}_x.\ \bullet\ \texttt{y}\ (\bullet_\texttt{j}\ \texttt{x}_\texttt{j})_{\texttt{j}=1-n}$

Where here $\texttt{x}_\texttt{j}$ is bound by $\Gamma^{\texttt{R2}}$ and thus has negative occurences of `R2`. Note that we still require $\bullet$ since it might be the case that $\texttt{T = R1}\ \Gamma_\texttt{D}$ (handled below); it has type $\texttt{S}\ \to\ \overset{\Pi}{\forall}\ \Gamma^{\texttt{R1}}{}_x.\ \texttt{[R1/R2]T}$. Each $\bullet_\texttt{j}$ has type $\Gamma^{\texttt{R2}}{}_x(\texttt{x}_\texttt{j})\ \to\ \Gamma^{\texttt{R1}}{}_x(\texttt{x}_\texttt{j})$.

Perform the steps outlined in Section A.3 to fill in each $\bullet_\texttt{j}$ producing from $\bullet_\texttt{j}\ \texttt{x}_\texttt{j}$ the sequence of arguments $\overline{\texttt{t}}_\texttt{j}$ of type $\Gamma^{\texttt{R1}}{}_x$ that erase to $\texttt{x}_{\texttt{j}=1-n}$ Finally, refine $\bullet$ to either `unit` or $\lambda\ \texttt{y}.\ \lambda\ \texttt{x}_\texttt{j}.\ \texttt{elimId -c (y}\ \texttt{x}_\texttt{j})$ depending on whether $\texttt{T = R1}\ \Gamma_\texttt{D}$

A.3 Conversion of Constructor Sub-data With Negative Recursive Occurences

We consider $\bullet$ `x` where `x`: $\overset{\Pi}{\forall}$ $\Gamma^{\mathtt{R2}}{}_{\mathtt{y}}$. `S`, `S` is not an arrow and does not contain `R2`, and `R2` occurs positively in the types of the variables bound by $\Gamma^{\mathtt{R2}}{}_{\mathtt{y}}$. The expression $\bullet$ `x` has type $\overset{\Pi}{\forall}$ $\Gamma^{\mathtt{R1}}{}_{\mathtt{y}}$. `S`.

Make $\bullet$ `x` η-long and introduce holes $\bullet_{\mathtt{j}}$ to apply to the sub-data as in

$$\overset{\lambda}{\Lambda}\ \Gamma^{\mathtt{R1}}{}_{\mathtt{y}}.\ \mathtt{x}\ (\bullet_{\mathtt{j}}\ \mathtt{y}_{\mathtt{j}})_{\mathtt{j}=1-n}$$

where $\bullet_{\mathtt{j}}$: $\Gamma^{\mathtt{R1}}{}_{\mathtt{y}}(\mathtt{y}_{\mathtt{j}}) \to \Gamma^{\mathtt{R2}}{}_{\mathtt{y}}(\mathtt{y}_{\mathtt{j}})$. Perform the steps outlined by Section A.2 to fill in each $\bullet_{\mathtt{j}}$ producing from $\bullet_{\mathtt{j}}$ $\mathtt{y}_{\mathtt{j}}$ the sequence of arguments $\overline{\mathtt{t}}$ that erase to $\mathtt{y}_{\mathtt{j}=1-n}$.

gygygy

References

[Inr18] Inria. The Coq Documentation. `https://coq.inria.fr/refman/index.html`, 2018.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[Miq01] Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications*, TLCA'01, pages 344–359, Berlin, Heidelberg, 2001. Springer-Verlag.

[PM15] Christine Paulin-Mohring. Introduction to the calculus of inductive constructions, 2015.

[PPM89] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 209–228. Springer, 1989.

[Stu18] Aaron Stump. Syntax and semantics of cedille, 2018.

[Wad90] Philip Wadler. Recursive types for free!, 1990.