Cedille 1.1.0 Datatype System Specification
Syntax, Typing, Reduction, and Elaboration

Christopher Jenkins

March 19, 2019

1 Introduction

This document describes the datatype subsystem of Cedille, to be introduced in version 1.1.0. Cedille is programming language with dependent types based on the *Calculus of Dependent Lambda Eliminations*(CDLE)[Stu17] – a compact and Curry-style pure type theory which extends the Calculus of Constructions (CC)[CH86] with additional typing constructs, and in which induction for datatypes can be *generically derived*[FBS18] rather than taken as primitive. The datatype system described in this document provides users of Cedille convenient access to this generic development by providing high-level syntax for declaring datatypes and defining functions over them; Cedille can then elaborate these features to *Cedille Core*[Stu18b], a minimal implementation of CDLE.

1.1 Background: CDLE

We first review CDLE, the type theory of Cedille; a more complete treatment can be found in [Stu18a]. CDLE is an extension of the impredicative, Curry-style (i.e. extrinsically typed) Calculus of Constructions (CC) that adds three new typing constructs: equality of untyped terms ($\{t \simeq t'\}$); the dependent intersection type ($\iota\, x : T.\, T'$) of [Kop03]; and the implicit (erased) product type ($\forall\, x : T.\, T'$) of [Miq01]. The pure term language of CDLE is just that of the untyped λ-calculus; to make type checking algorithmic, terms in Cedille are given type annotations, and definitional equality of terms is modulo erasure of these annotations. The kinding, typing, and erasure rules for the fragment of CLDE containing these type constructs are given in Figure 1. We briefly describe these below:

- $\{t_1 \simeq t_2\}$ is the type of proofs that t_1 and t_2 are equal (modulo erasure). It is introduced with β (erasing to $\lambda\, x.\, x$), proving $\{t \simeq t\}$ for any untyped term t. Combined with definitional equality, β can be used to prove $\{t_1 \simeq t_2\}$ for any $\beta\eta$-convertible t_1 and t_2 whose free variables are declared in the typing context. Equality types can be eliminated with ρ, φ, and δ.
    - $\rho\ t\ @\ x.T - t'$ (erasing to $|t'|$) rewrites a type by an equality: if t proves that $\{t_1 \simeq t_2\}$ and t' has type $[t_2/x]T$, then the ρ expression has type $[t_1/x]T$, with the guide $@\ x.T$ indicating the occurrences of t_2 rewritten in the type of t'.
    - $\varphi\ t - t_1\ \{t_2\}$ (erasing to $|t_2|$) casts t_2 to the type of t_1 when t proves t_1 and t_2 equal.
    - $\delta\ T\ - t$ (erasing to $|t|$) has type T when t proves that Church-encoded *true* equals *false*, enabling a form of proof by contradiction. While this is adequate for CDLE, Cedille makes δ more practical by implementing the Böhm-out algorithm[**?**] so δ can be used on any proof that $\{t_1 \simeq t_2\}$ for closed, normalizing, and $\beta\eta$-inconvertible terms t_1 and t_2.
- $\iota\, x : T.\, T'$ is the type of terms t which can be assigned both type T and $[t/x]T'$, and in the annotated language is introduced by $[t, t']$, where t has type T, t' has type $[t/x]T'$, and $|t| \simeq_{\beta\eta} |t'|$. Dependent intersections are eliminated with projections $t.1$ and $t.2$, selecting resp. the view that term t has type T or $[t.1/x]T'$

(a) Novel CDLE type constructs

$$\frac{FV(|t|\ |t'|) \subseteq dom(\Gamma)}{\Gamma \vdash \{t \simeq t'\} : \star} \quad \frac{\Gamma \vdash T : \star \quad \Gamma, x : T \vdash T' : \star}{\Gamma \vdash \iota\, x{:}T.\, T' : \star} \quad \frac{\Gamma \vdash T : \star \quad \Gamma, x : T \vdash T' : \star}{\Gamma \vdash \forall\, x{:}T.\, T' : \star}$$

(b) Equality

$$\frac{\Gamma \vdash \{t \simeq t\} : \star}{\Gamma \vdash \beta : \{t \simeq t\}} \qquad \frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t' : [t_2/x]T}{\Gamma \vdash \rho\ t\ @\ x.T\ \text{-}\ t' : [t_1/x]T}$$

$$\frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t_1 : T}{\Gamma \vdash \varphi\ t -\ t_1\ \{t_2\} : T} \quad \frac{\Gamma \vdash t : \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\} \quad \Gamma \vdash T : \star}{\Gamma \vdash \delta\ T - t : T}$$

$$\begin{array}{lcl} |\beta| & = & \lambda\, x.\, x \\ |\rho\ t\ @\ x.T - t'| & = & |t'| \\ |\varphi\ t - t_1\ \{t_2\}| & = & |t_2| \\ |\delta\ T - t| & = & |t| \end{array}$$

(c) Dependent Intersection

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : [t_1/x]T_2 \quad \Gamma \vdash \iota\, x{:}T_1.\, T_2 : \star \quad |t_1| = |t_2|}{\Gamma \vdash [t_1, t_2] : \iota\, x{:}T_1.\, T_2}$$

$$\frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.1 : T_1} \quad \frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.2 : [t.1/x]T_2}$$

$$\begin{array}{lcl} |[t, t']| & = & |t| \\ |t.1| & = & |t| \\ |t.2| & = & |t| \end{array}$$

(d) Implicit Products

$$\frac{\Gamma, x : T \vdash t' : T' \quad x \notin FV(|t'|) \quad \Gamma \vdash \forall\, x{:}T.\, T' : \star}{\Gamma \vdash \Lambda\, x{:}T.\, t' : \forall\, x{:}T.\, T'} \quad \frac{\Gamma \vdash t : \forall\, x{:}T'.\, T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t\ \text{-}t' : [t'/x]T}$$

$$\begin{array}{lcl} |\Lambda\, x{:}T.\, t| & = & |t| \\ |t\ \text{-}t'| & = & |t| \end{array}$$

Figure 1: Kinding, typing, and erasure for a fragment of CDLE

- $\forall\, x{:}T.\, T'$ is the implicit product type, the type of functions with an erased argument x of type T and a result of type T'. Implicit products are introduced with $\Lambda\, x{:}T.\, t$, provided x does not occur in $|t|$, and are eliminated with erased application t -t. Due to the restriction that bound variable x cannot occur in the body t of $\Lambda\, x{:}T.\, t$, erased arguments play no computational role and thus exist solely for the purposes of typing.

Figure 1 omits the typing and erasure rules for the more familiar term and type constructs of CC. When reasoning about definitional equality of term constructs in CC, all types in type annotations, quantifications, and applications are erased. Types are quantified over with $\forall$ within types and abstracted over with Λ in terms, similar to implicit products; the application of a term t to type T is written $t \cdot T$, and similarly application of type S to type T is written $S \cdot T$. In term-to-term applications, we omit type arguments when these are inferable from the types of term arguments.

1.2 Datatype Declarations

We begin with a bird's-eye view of the new language features by showing some simple example data-type definitions and functions over them.

```
data Bool: ★ =                data List (A: ★): ★ =
| tt: Bool                    | nil: List
| ff: Bool.                   | cons: A → List → List.

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

Figure 2: Example datatype declarations

Declarations Figure 2 shows the definitions in Cedille for some well-known types. Modulo differences in syntax, the general scheme for declaring datatypes in Cedille should be straightforward to anyone familiar with GADTs in Haskell or with dependently typed languages like Agda, Coq, or Idris. Some differences from these languages to note are that:

- In constructor types, recursive occurrences of the inductive datatype (such as <u>`Nat`</u> in `suc` : <u>`Nat`</u> → `Nat` must be positive, but *need not be* strictly positive.
- Occurrences of the inductive type being defined are not written applied to its parameters. E.g, the constructor `nil` is written with signature `List` rather than `List` · A. Used outside of the datatype declaration, `nil` has its usual type: $\forall\, A{:}\star.$ `List` · A.
- Declarations can only refer to the datatype itself and prior definitions. Inductive-recursive and inductive-inductive definitions are not part of this proposal.

1.3 Function Definitions

```
pred: Nat → Nat
= λ n. μ' n {    -- scrutinee: n
  | zero → n    -- case: n is zero
  | suc n' → n' -- case: n is successor to some n'
  }.

add: Nat → Nat → Nat
= λ m. λ n. μ addN. m {    -- scrutinee: m, recursive definition: addN
  | zero → n               -- case: m is zero
  | suc m' → suc (addN m') -- case: m is successor to some m'
  }.

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. -- explicit motive given below with @
  μ vappendYs. xs @(λ i: Nat. λ x: Vec ·A i. Vec ·A (add i m)) {
  | vnil →              -- expected: Vec ·A n
    ys
  | vcons -m' hd xs' → -- expected: Vec ·A (suc (add m' n))
    vcons -(add m' n) hd (vappendYs -m' xs')
  }.
```

Figure 3: Predecessor, addition, and vector append

Figure 3 shows a few standard examples of functional and dependently-typed programming in Cedille. Function `pred` introduces operator μ' for *course-of-values (CoV) pattern matching*, which will be explained in greater detail below. Here it is used for standard pattern matching: μ' is given scrutinee `n` of type `Nat` and a sequence of case branches for each constructor of `Nat`. Functions `add` and `vappend` introduce operator μ for *CoV induction* by combined pattern matching and recursion; the distinction between pattern matching by μ and μ' will also be made clear below. Here, μ is used for standard structurally recursive definitions, with `vappend` showing its use on indexed type `Vec` to define recursive function `vappendYs`, semantically appending *ys* to its argument. In the `vnil` branch, the expected type is `Vec` · A (`add zero` n) by the usual index refinement of pattern matching on indexed types; thanks to the reduction behavior of `add` this is convertible with `Vec` · A n, the type of `ys`. Similarly, in the `vcons` branch the expected type is `Vec` · A (`add` (`suc` m') n), convertible with the type `Vec` · A (`suc` (`add` m' n)) of the body.

1.4 Reduction Rules of μ and μ'

In the discussion of `vappend` above we omitted some details about checking convertibility of terms defined using μ and μ'. In the `vcons` case, the expected type `Vec` $\cdot$ A (`add` (`suc` m') n) reduces, by β-reduction and erasure alone, to

```
Vec ·A (μ addN. (suc m') {zero → n | suc m' → suc (addN m')})
```

For the length index of this type to be convertible with `suc` (`add` m' n), we need a rule for μ-reduction. μ-reduction is a combination of fixpoint unrolling and case branch selection. Here, because the scrutinee is `suc` m', the case branch selected is the successor case. The recursive call *addN* m' that occurs in this branch is replaced (by substitution on the μ-bound *addN*) with another copy of the μ-expression that defines `add`. Therefore, the fully normalized type in the `vcons` case of `vappened` is

```
Vec ·A (suc (μ addN. m' {zero → n | suc m' → suc (addN m')}))
```

which is convertible with the type of the expression given in that branch.

1.5 Course-of-Value Recursion

This section explains *(CoV) pattern matching* in Cedille, which is used to implement *semantic* (type-based) termination checking and which facilitates reuse for functions used in ordinary and CoV induction.

The definitions of `add` and `vappend` in Figure 3 only require *structural* recursion – recursive calls are made directly on subdata revealed by one level of pattern matching. Cedille's datatype system allows programmers to use the much more powerful form of course-of-values recursion, which allows recursive calls to be made on arbitrary subdata of a scrutinee. CoV recursion subsumes recursion subdata produced by a static number of cases analyzed, such as in the case of `fib` below:

```
fib: Nat → Nat
= λ n. μ fib. n {
  | zero → suc zero
  | suc n' → μ' n'. {| zero → suc zero | suc n' → add (fib n') (fib n'')}
 }.
```

CoV recursion allows the programmer to recurse on subdata computed *dynamically*, as well as statically. A good intuitive example is the definition of division by iterated subtraction. In a Haskell-like language, we may simply write:

```
0 / d = 0
n / 0 = n
n / d = if (n < d) then zero else 1 + ((n - d) / d)
```

This definition is guaranteed to terminate for all inputs, as the first argument to the recursive call, n - d, is smaller than the original argument n (d is guaranteed to be non-zero). In Cedille, we are able to write a version of division close to the intuitive way, requiring a few more typing annotations to enable the termination checker to see that the expression n-d is some subdata of n.

CoV Globals We first explain the types and definitions of `predCV` and `minsuCV`. In `predCV` we see the first use of predicate `Is/Nat`. Every datatype declaration in Cedille introduces, in addition to itself and its constructors, three global names derived from the datatype's name. For `Nat`, these are:

- `Is/Nat` : $\star \to \star$

  A term of type `Is/Nat` $\cdot$ N is a witness that any term of type N may be treated as if has type `Nat` for the purposes of case analysis.

```
predCV: ∀ N: ⋆. ∀ is: Is/Nat ·N. N → N
= Λ N. Λ is. λ n. μ'<is> n {| zero → n | suc n' → n'}.

minusCV: ∀ N: ⋆. ∀ is: Is/Nat ·N. N → Nat → N
= Λ N. Λ is. λ m. λ n. μ mMinus. n {
  | zero → m
  | suc n' → predCV -is (mMinus n')
  }.
minus = minusCV -is/Nat.

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

ite: ∀ X: ⋆. Bool → X → X → X
= Λ X. λ b. λ t. λ f. μ' b {| tt → t | ff → f}.

divide: Nat → Nat → Nat
= λ n. λ d. μ divD. n {
  | zero → zero
  | suc pn →
    [pn' = to/Nat -isType/divD pn] -
    [diff = minusCV -isType/divD pn (pred d)] -
      ite (lt (suc pn') d) zero (suc (divD diff))
  }.
```

Figure 4: Division using course-of-values recursion

- `is/Nat` : `Is/Nat` · `Nat` is the trivial `Is/Nat` witness.
- `to/Nat` : $\forall N{:}\star.\forall is{:}$`Is/Nat`$\cdot N. N \rightarrow$ `Nat`

  `to/Nat` is a function that coerces a term of type N to `Nat`, given a witness *is* that N "is" `Nat`. We will later see that `to/Nat` and all other such cast functions elaborate to terms definitionally equal (modulo erasure) to $\lambda\, x.\, x$. Cedille internalizes this fact: equation $\{$`to/Nat` $\simeq \lambda\, x.\, x\}$ is true definitionally in the surface language. Notice that this is possible in part because there is only one *unerased* argument to `to/Nat`. This property is important for CoV induction further on.

In `predCV` the witness *is* of type `Is/Nat` · N is given explicitly to μ' with the notation μ'<*is*>, allowing argument n (of type N) to be a legal scrutinee for `Nat` pattern matching. Reasoning by parametricity, the only ways `predCV` can produce an N output (i.e, preserve the abstract type) are by returning n itself or some subdata produced by CoV pattern matching on it – the predecessor n' also has type N. Thus, the type signature of `predCV` has the following intuitive reading: it produces a number no larger than its argument, as an expression like `suc` (`to/Nat` -*is* n) would be type-incorrect to return.

Code Reuse What is the relation is between `predCV` and the earlier `pred` of Figure 3? The fully annotated μ'-expression of the latter is:

```
μ'<is/Nat> n @(λ x: Nat. Nat) {| zero → n | suc n' → n'}
```

In `pred`, the global witness `is/Nat` of type `Is/Nat` · `Nat` need not be passed explicitly, as it is inferable[1] by the type `Nat` of the scrutinee n. Furthermore, the erasures of `pred` and `predCV` are definitionally equal, a fact provable in Cedille (where _ indicates an anonymous proof):

[1]The same holds for the inferability of the local witness (discussed below) introduced in the body of `fib`.

```
_ : {pred ≃ predCV} = β.
```

This leads to a style of programming where, when possible, functions are defined over an abstract type N for which e.g. `Is/Nat` $\cdot$ N holds, and the usual version of the functions *reuse* these as a special case. Indeed, this is how `minus` is defined – in terms of the more general `minsuCV` specialized to the trivial witness `is/Nat`. The type signature of `minsuCV` yields a similar reading that it produces a result no larger than its first argument. In the successor case, `predCV` is invoked and given the (erased) witness `is`. That `minsuCV` preserves the type of its argument after `n` uses of `predCV` is precisely what allows it to appear in expressions given as arguments to recursive functions. Function `minus` is used to define `lt`, the Boolean predicate deciding whether its first argument is less than its second; `ite` is the usual definition of a conditional expression by case analysis on `Bool`.

CoV Locals The last definition, `divide`, is as expected except for the successor case. Here, we make a let binding (the syntax for which in Cedille is $[x = t] - t'$, analogous to `let` $x = t$ `in` t') for pn', the coercion to `Nat` of the predecessor of the dividend pn (using the as-yet unexplained `Is/Nat` witness `isType/divD`), and for $diff$, the difference (using `minsuCV`) between pn and `pred` d. Note that when d is non-zero, $diff$ is equal to the different between the dividend and divisor, and otherwise it is equal to pn; in both cases, it is smaller than the original pattern `suc` pn. Finally, we test whether the dividend is less than the divisor: if so, return `zero`, if not, divide $diff$ by d and increment. The only parts of `divide` requiring further explanation, then, are the witness `isType/divD` and the type of pn, which are the keys to CoV recursion and induction in Cedille.

Within the body of the μ-expression defining recursive function `divD` over scrutinee n of type `Nat`, the following names are automatically bound:

- `Type/divD` : $\star$, the type of recursive occurrences of `Nat` in the types of variables bound in constructor patterns (such as pn).
- `isType/divD` : `Is/Nat` $\cdot$ `Type/divD`, a witness that terms of the recursive-occurrence type may used for further CoV pattern matching.
- `divD` : `Type/divD` $\rightarrow$ `Nat`, the recursive function being defined, accepting only terms of the recursive occurrence type `Type/divD`. This restriction guarantees that `divD` is only called on expressions smaller than the previous argument to recursion.

The reader is now invited to revisit the definitions of Figure 3, keeping in mind that in the μ-expressions of `add` and `vappend` constructor subdata m' and xs' in pattern guards `suc` m' and `vcons` -m' hd xs' have abstract types (the subdata of the successor case of the μ'-expression of `pred` has the usual type `Nat`), and that recursive definitions `addN` and `vappendYs` only accept arguments of such a type. With this understood, so to is the definition `divide`: predecessor pn has type `Type/divD`, witness `isType/divD` has type `Is/Nat` $\cdot$`Type/divD` and so the local variable $diff$ has type `Type/divD`, exactly as required by `divD`.

1.6 Course-of-values Induction

CoV recursion is not enough – in a dependently typed language, one also wishes sometimes to *prove* properties of recursive definitions. Cedille enables this with *CoV induction*, which we explain with an example proof below. Figure 5 shows its use in `leDiv` to prove that the result of division is no larger than its first argument.

We first encode the relation "less than or equal" as a datatype `LE` and prove two properties of it (definitions omitted, indicated by `<..>`): that it is transitive (`leTrans`) and that `minus` produces a result less than or equal to its first argument (`leMinus`). In the proof of `leDiv` itself, we define a recursive function (also named `leDiv`) over n. When it is zero, the goal becomes `LE zero zero`, provable by constructor `leZ`. When it is the successor of some number pn, the expression `divide` (`suc` pn') d in the type of the goal reduces to a conditional branch on whether the dividend is less than the divisor. We use μ' to match on the result of `lt` (`suc` pn') d to determine which branch is reached: if it is true, the goal type reduces further to

```
data LE: Nat → Nat → ⋆ =
  | leZ: Π n: Nat. LE zero n
  | leS: Π n: Nat. Π m: Nat. LE n m → LE (suc n) (suc m).

leTrans: Π l: Nat. Π m: Nat. Π n: Nat. LE l m → LE m n → LE l n = <..>
leMinus: Π m: Nat. Π n: Nat. LE (minus m n) m = <..>

leDiv: Π n: Nat. Π d: Nat. LE (divide n d) n
  = λ n. λ d. μ leDiv. n @(λ x: Nat. LE (divide x d) x) {
  | zero → leZ zero
  | suc pn →
    [pn' = to/Nat -isType/leDiv pn] -
    [diff = minus pn' (pred d)] -
    [l = divide diff d] -
      μ' (lt (suc pn') d) @(λ x: Bool. LE (ite x zero (suc l)) (suc pn')) {
      | tt → leZ (suc pn')
      | ff →
        [ih: LE l diff   = leDiv (minus' -isType/leDiv pn (pred d))] -
        [mi: LE diff pn' = leMinus pn' (pred d)] -
          leS l pn' (leTrans l diff pn' ih mi)
      }
  }.
```

Figure 5: Example of course-of-values induction

`LE zero` (`suc` pn'), which is again provable by `leZ`; otherwise, the goal is `LE` (`suc` l) (`suc` pn'), where l is defined as *diff* divided by d. Here is where CoV induction is used: to define *ih* we invoke the inductive hypothesis on `minus' -isType/leDiv` pn (`pred` d), a term that is equal (modulo erasure) to *diff* but has the required abstract type `Type/leDiv`, letting us prove `LE` l *diff* . We combine this and a proof of `LE` *diff* pn' (bound to *mi*) with the proof that `LE` is transitive, producing a proof that `LE` l pn'. The the final obligation `LE` (`suc` l) (`suc` pn') is proved by constructor `leS`.

1.7 Subtyping and Coercions

In the preceding code examples, every time we wished to use some term of the abstract recursive-occurrence type (such as `Type/divD` in `divide`) as if it had the concrete datatype (such as `Nat`), we explicitly cast the term (using e.g. `to/Nat`). We now take a moment to describe a feature we desire to implement in the near future: automatic inference of these coercions via subtyping. As an example, we provide two different implementations of the function factorial in Figure 6: `fact1` using an explicit cast and `fact2` where these would be inferred.

In the successor case of `fact1`, we know that the number we are considering is equal to the `suc`cessor of another number m. We wish to multiply *suc m* with the factorial of m. However, μ provides access to the subdata m at an abstract type; this allows m to be a legal argument for a recursive call as in `fac` m, but not as an argument to constructor `suc` which requires a `Nat`. Thus, in order to multiply the two expressions, we first cast m to `Nat` using the CoV global cast function `to/Nat` and CoV local evidence `isType/fact` (of type `Is/Nat` · `Type/fact`).

Alternatively, we should be able to infer this coercions by equipping type inference with a form of *subtyping*. In the successor case of `fact2` (which is currently not a legal Cedille definition), when we see that the expected type of m is `Nat`, and its actual type is `Type/fact`, we could search the typing context for evidence of type `Is/Nat` · `Type/fact` and, finding this in the form of `isType/fact`, accept this definition.

The story becomes more complex in the presence of non-strictly positive datatypes. Figure 7 presents a

```
mult : Nat → Nat → Nat
= λ m. λ n. μ multN. m {
| zero → zero
| suc m → add n (multN m)
}.

fact1 : Nat → Nat
= λ n. μ fact. n {
| zero → suc zero
| suc m →
  mult (suc (to/Nat -isType/fact m)) (fact m)
}.

-- not yet supported
fact2 : Nat → Nat
= λ n. μ fact. n {
| zero → suc zero
| suc m → mult (suc m) (fact m)
}.
```

Figure 6: Factorial with explicit and implicit coercions

definition of `PTree`, an infinitary tree which a non-strict positive recursive occurrence 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 pattern `leaf` corresponds to the given assumption l proving P `leaf`. In the `node` case of `indPTree1`, the expected type is P (`node` s). The pattern-bound variable s has type (`Type/ih` → `Bool`) → `Type/ih`, and the two different occurrences of s in the arguments to the assumed proof n require casting s to two different types, corresponding to the two explicit type coercions of s locally bound to $s1$ and $s2$ (note that these two expressions are $\beta\eta$-convertible with s).

In the `node` case of `indPTree2`, the two occurrences of s in the arguments to n correspond to two subtyping problems:

- (`Type/ih` → `Bool`) → `Type/ih <:` (`PTree` → `Bool`) → `PTree`
- (`Type/ih` → `Bool`) → `Type/ih <:` (`PTree` → `Bool`) → `Type/ih`

Such subtyping problems can solved algorithmically and the necessary coercions to the desired type inserted automatically.

1.8 Program Reuse

We conclude our informal introduction to Cedille's datatype system with a somewhat more complex example: how to support program reuse over different data-types at zero run-time cost. For datatypes encoded as λ-terms in Cedille, it is possible that some constructor between the two types are definitionally equal. For example, for λ-encoded `List` and `Vec` constructors `nil` (`cons`) and `vnil` (`vcons`) are indeed equal modulo erasure. When Cedille elaborates the *declared datatypes* `List` and `Vec`, this correspondence also holds. Cedille's datatype system internalizes this fact, meaning the declared constructors `nil` (`cons`) and `vnil` (`vcons`) are *themselves definitionally equal.* This is shown in the following example with manual zero-cost reuse of `map` for `List` in `vmap` for `Vec`.

```
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 l. λ n. λ t. μ ih. t @(λ x: PTree. P x) {
  | leaf → l
  | node s →
    [s1 : (PTree → Bool) → Type/ih = λ p. s (λ t. p (to/PTree -isType/ih t))]
  - [s2 : (PTree → Bool) → PTree   = λ p. to/PTree -isType/ih (s1 p)]
  - n -s2 (λ p. ih (s1 p))
  }.

-- not yet implemented
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 l. λ n. λ t. μ ih. t @(λ x: PTree. P x) {
| leaf → l
| node s → n -s (λ p. ih (s p))
}.
```

Figure 7: Subtyping for a non-strictly positive type

Manual zero-cost reuse of `map` for `vmap` Figure 8 gives the definitions of the linear-time conversion functions `v2l` and `l2v`, as well as the types for list operations `len` and `map` (`List` is given in Figure 2, `<..>` and _ indicate resp. an omitted def. and anonymous proof). First, and as promised, Cedille considers the corresponding constructors of `List` and `Vec` definitionally equal:

```
_ : {nil  ≃ vnil}  = β.
_ : {cons ≃ vcons} = β.
```

This means that the linear-time functions `v2l` and `l2v` merely return a term equal to their argument at a different type. Indeed, this is provable in Cedille by easy inductive proofs `vl2Id` and `l2vId` (Figure 9), rewriting the expected branch type by ρ (Figure 1b) in the `cons` and `vcons` cases using the inductive hypothesis and making implicit use of constructor equality. Thanks to φ (casting a term to the type of another it is proven equal to, Figure 1b), these proofs give rise to coercions `v2l!` and `l2v!` between `List` and `Vec` that erase to identity functions – meaning there is no performance penalty for using them! By notational convention, identifiers suffixed with the bang (!) character indicate zero-cost coercions between types.

With `v2l!` and `l2v!` and the two lemmas `mapPresLen` and `v2lPresLen` resp. stating that `map` and `v2l!` preserve the length of their inputs, we can now define `vmap` (Figure 10) over `Vec` by reusing `map` for `List` with no run-time cost, demonstrating that Cedille's datatype system does not prevent use of this desirable property derived in its core theory CDLE.

Definitional Equality of Constructors Under what conditions should users expect Cedille to equate constructors of different datatypes? Certainly they should *not* be required to know the details of elaboration to use features like zero-cost reuse that depend on this. Fortunately, there is a simple, high-level explanation for when different constructors are considered equal that makes reference only to the shape of the datatype

```
len: ∀ A: ⋆. List ·A → Nat = <..>
map: ∀ A B: ⋆. (A → B) → List ·A → List ·B = <..>

v2l: ∀ A: ⋆. ∀ n: Nat. Vec ·A n → List ·A
  = Λ A. Λ n. λ xs. μ v2l. xs {
  | vnil → nil ·A
  | vcons -n' hd tl → cons hd (v2l -n' tl)
  }.

l2v: ∀ A: ⋆. Π xs: List ·A. Vec ·A (len xs)
  = Λ A. λ xs. μ l2v. xs @(λ x: List ·A. Vec ·A (len x)) {
  | nil → vnil ·A
  | cons hd tl → vcons -(len (to/List -isType/l2v tl)) hd (l2v tl)
  }.
```

Figure 8: `len`, `map`, and linear-time conversion between `List` and `Vec`

declaration. We give this here informally, with the formal statement and soundness property given in the technical portion of this document.

If c, c' are resp. constructors of datatype D and D', then c and c' are equal iff:

- D and D' have the same number of constructors;
- the index of c in the list of constructors for D is the same as the index of c' in the list of constructors for D'; and
- c and c' take the same number of unerased arguments

That these three conditions hold for the corresponding constructors of `List` and `Vec` is readily verified: both datatypes have two constructors; `nil` (`cons`) and `vnil` (`vcons`) are each the first (second) entries in their datatype's constructor list; and `nil` and `vnil` take no arguments while `cons` and `vcons` take two unerased argument (the `Nat` argument to `vcons` is erased). It is clear also these conditions prohibit two different constructors of the same datatype from ever being equated, as their index in the constructor list would necessarily be different.

This scheme for equating data constructors perhaps leads to some counter-intuitive results. First, changing the order of the constructors of `List` prevents zero-cost reuse between it and `Vec`. Second, between two datatypes with the same number of constructors, some constructors may be equal and others not. For example, `List` and `Nat` have two constructors, and the first of both takes no arguments. Thus, equality between `zero` and `nil` holds definitionally, but is not possible for `suc` and `cons`. The very same phenomenon occurs for e.g. Church-encoded numbers and lists.

2 Syntax

Identifiers We now turn to a more formal treatment of Cedilleum. Figure 11 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.

```
v2lId: ∀ A: ⋆. ∀ n: Nat. Π vs: Vec ·A n. {v2l vs ≃ vs}
  = Λ A. Λ n. λ vs. μ v2lId. vs @(λ i: Nat. λ x: Vec ·A i. {v2l x ≃ x}) {
  | vnil → β
  | vcons -i hd tl → ρ (v2lId -i tl) @ x. {cons hd x ≃ vcons hd tl} - β
  }.

l2vId: ∀ A: ⋆. Π ls: List ·A. {l2v ls ≃ ls}
  = Λ A. λ ls. μ l2vId. ls @(λ x: List ·A. {l2v x ≃ x}) {
  | nil → β
  | cons hd tl → ρ (l2vId tl) @ x. {vcons hd x ≃ cons hd tl} - β
  }.

v2l!: ∀ A : ⋆. ∀ n: Nat. Π vs: Vec ·A n. List ·A
  = Λ A. Λ n. λ vs. φ (v2lId -n vs) - (v2l -n vs) {vs}.
_ : {v2l! ≃ λ vs. vs} = β.

l2v!: ∀ A: ⋆. Π ls: List ·A. Vec ·A (len ls)
  = Λ A. λ ls. φ (l2vId ls) - (l2v ls) {ls}.
_ : {l2v! ≃ λ ls. ls} = β.
```

Figure 9: Zero-cost conversions between `Vec` and `List`

```
mapPresLen: ∀ A: ⋆. ∀ B: ⋆. Π f: A → B. Π xs: List ·A. {len xs ≃ len (map f xs)} = <..>
v2lPresLen: ∀ A: ⋆. ∀ n: Nat. Π xs: Vec ·A n. {n ≃ len (v2l! -n xs)} = <..>

vmap: ∀ A B: ⋆. ∀ n: Nat. (A → B) → Vec ·A n → Vec ·B n
  = Λ A B n. λ f xs. ρ (v2lPresLen -n xs) - ρ (mapPresLen f (v2l! -n xs))
  - l2v! (map f (v2l! -n xs)).
_ : {vmap ≃ map} = β.
```

Figure 10: Zero-cost reuse of `map` for `Vec`

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`:

```
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 : ⋆ → ⋆
```

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 11: Identifiers

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 12: Untyped terms

```
      = 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)
  .
```

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 14 and terms are listed in Figure 15 along with some

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**)**	
$constr$	::=	**|** id **:** T	

Figure 13: Modules and definitions

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 ($\rightarrow$ $\Rightarrow$) 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[2]. Kinds are formed by dependent and non-dependent products (Π and $\rightarrow$) and a base kind for types which can classify terms ($\star$). Types are also formed by the usual (dependent and non-dependent) products (Π and $\rightarrow$) and also *implicit* products ($\forall$ and $\Rightarrow$) which quantify over erased arguments (that is, arguments that disappear at run-time). Π-products are only allowed to quantify over terms as all types occurring in terms are erased at run-time, but $\forall$-products can quantify over types *and* terms because terms can be erased. Meanwhile, non-dependent products ($\rightarrow$ and $\Rightarrow$) can only "quantify" over terms because non-dependent type quantification does not seem particularly useful. Besides these, Cedilleum features type-level functions and applications (with term and type arguments), and a primitive equality type for untyped terms. Last of all is the "hole" type ($\bullet$) for writing partial type signatures or incomplete type applications. There are term-level holes as well, and together the two are intended to help facilitate "hole-driven development": any hole automatically generates a type error and provides the user with useful contextual information.

We illustrate with another example: what follows is a module stub for **DepCast** defining dependent casts – intuitively, functions from $a : A$ to $B\ a$ that are also equal[3] 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 = ● .
```

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.

[2]Cite

[3]Module erasure, discussed below

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 14: Kinds and types

```
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
          }.
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

The definition of the erasure function given in Figure 16 takes the annotated terms from Figures 14 and 15 to the untyped terms of Figure 12. 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

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

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)	
		$\bullet$	
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 15: Annotated Terms

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)

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

[5]Where $\texttt{tt} = \lambda\, x.\, \lambda\, y.\, x$ and $\texttt{ff} = \lambda\, x.\, \lambda\, y.\, y$

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

Figure 16: Erasure for annotated terms

Figure 17: Contexts

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

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

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

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

Figure 18: Sort checking $\boxed{\Gamma \vdash C : S}$

The inference rules for classifying expressions in Cedilleum are stratified into two judgments. Figure 18 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

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

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

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

$$\frac{\Gamma \vdash T : K \quad \Gamma \vdash_\Downarrow t : T \quad \Gamma, id = t{:}T \vdash_\delta t' : T'}{\Gamma \vdash_\delta [id : T = t]\ \text{-}\ t' : T'} \qquad \frac{\Gamma \vdash_\Uparrow t : T \quad \Gamma, id = t{:}T \vdash_\delta t' : T'}{\Gamma \vdash_\delta [id = t]\ \text{-}\ t' : T'} \qquad \frac{\Gamma \vdash_\Uparrow t : \{t_1 \simeq t_2\} \quad \Gamma \vdash_\Uparrow t' : [t_1/x]\ T}{\Gamma \vdash_\delta \rho\ t\ \text{-}\ t' : [t_2/x]\ T}\ {}^{4}$$

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

$$\frac{\Gamma \vdash_\Downarrow t : \{|t_1| \simeq |t_2|\} \quad \Gamma \vdash_\delta t_1 : T}{\Gamma \vdash_\delta \phi\ t\ \text{-}\ t_1\ \{t_2\} : T} \qquad \frac{\Gamma \vdash_\Downarrow t : T}{\Gamma \vdash_\Uparrow \chi\ T\ \text{-}\ t : T} \qquad \frac{\Gamma \vdash_\Downarrow t : \{\texttt{tt}\ \simeq\ \texttt{ff}\}}{\Gamma \vdash_\Downarrow \delta\ \text{-}\ t : T}\ {}^{5}$$

Figure 19: Type checking $\boxed{\Gamma \vdash_\delta s : C}$ (sans inductive datatypes)

these two figures. Most of these rules are similar to what one would expect from the Calculus of Implicit Constructions, so we focus on the typing rules unique to Cedilleum.

The typing rule for ρ shows that ρ is a primitive for rewriting by an (untyped) equality. If t is an expression that synthesizes a proof that two terms t_1 and t_2 are equal, and t' is an expression synthesizing type $[t_1/x]\ T$ (where, as per the footnote, t_1 does not occur in T), then we may essentially rewrite its type to $[t_2/x]\ T$. The rule for β is reflexivity for equality – it witnesses that a term is equal to itself, provided that the type of the equality is well-formed. The rule for ς is symmetry for equality. Finally, ϕ acts as a "casting" primitive: the rule for its use says that if some term t witnesses that two terms t_1 and t_2 are equal, and t_1 has been judged to have type T, then intuitively t_2 can also be judged to have type T. (This intuition is justified by the erasure rule for ϕ – the expression erases to $|t_2|$). The last rule involving equality is for δ, which witnesses the logical principle *ex falso quodlibet* – if a certain impossible equation is proved (namely that the two Church-encoded booleans `tt` and `ff` are equal), then *any* type desired is inhabited. The 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 [6]. 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).

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

- $\overset{\lambda}{\Lambda}\,\Gamma.\,t$ 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 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

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

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

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 **??**, `predCV`, 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 ??, `minsuCV`, the μ-expressions would be represented as

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

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 $\Delta(Vec) = \texttt{Ind}_{\text{C}}[1](\Gamma_{Vec} : \Sigma =)$, 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 $\Gamma_\text{P}, \Gamma_I$, and Σ be contexts such that Γ_I associates a single type-variable I to kind $\Pi\, \Gamma_\text{p}.\, K$ and Σ associates term variables $c_1...c_n$ with corresponding types $\forall\, \Gamma_\text{P}.\, T_1, ...\forall\, \Gamma_\text{P}.\, T_n$. Then the rule given in Figure 20 states when an inductive datatype definition may be introduced, provided that the following side conditions hold:

Figure 20: Introduction of inductive datatype

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

- 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 21: 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 21. 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 $\{\{c : T\}\}^P_I$ 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.

$$\begin{array}{rcl}
\{\{c : I\ \overline{T}\ \overline{s}\}\}^P_I & = & P\ \overline{s}\ c \\
\{\{c : \forall\, x{:}T'.\, T\}\}^P_I & = & \forall\, x{:}T'.\, \{\{c\ \text{-}x : T\}\}^P_I \\
\{\{c : \forall\, x{:}K.\, T\}\}^P_I & = & \forall\, x{:}K.\, \{\{c\ \cdot x : T\}\}^P_I \\
\{\{c : \Pi\, x{:}T'.\, T\}\}^P_I & = & \Pi\, x{:}T'.\, \{\{c\ x : T\}\}^P_I
\end{array}$$

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 22: 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 22 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 *histomorphic* 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 23: Use of an inductive datatype $\mathtt{Ind}_M[p](\Gamma_I : \Sigma =)$

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

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

The first rule of Figure 23 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 [Stu18a]). 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 $\texttt{F:}\ \star \to \star$ 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 $\lambda\ \texttt{R:}\ \star.\ \forall\ \texttt{X:}\ \star.\ \texttt{X} \to (\texttt{R} \to \texttt{X}) \to \texttt{X}$, 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 $(\Pi\ \Gamma_{\texttt{D}}.\ \star) \to (\Pi\ \Gamma_{\texttt{D}}.\ \star)$ corresponding to the kind $\Pi\ \Gamma_{\texttt{D}}.\ \star$ 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 $\texttt{F:}\ (\Pi\ \Gamma_{\texttt{D}}.\ \star) \to \Pi\ \Gamma_{\texttt{D}}.\ \star$ we concerned with. Identity mappings are similar to functors in that they come equipped with a function that resembles $\texttt{fmap:}\ \forall\ \Gamma_{\texttt{D}}.\ \forall\ \texttt{A B:}\ \Pi\ \Gamma_{\texttt{D}}.\ \star.\ \Pi\ \texttt{f:}\ (\texttt{A}\cdot\Gamma_{\texttt{D}} \to \texttt{B}\cdot\Gamma_{\texttt{D}}).\ \texttt{F}\cdot(\texttt{A}\cdot\Gamma_{\texttt{D}}) \to \texttt{F}\cdot(\texttt{B}\cdot\Gamma_{\texttt{D}})$ 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 $\texttt{elimId}_{\texttt{D}}$:

```
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 $\texttt{elimId}_{\texttt{D}}$ 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 → ⋆). $\overset{\Pi}{\forall}$ 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 $\texttt{elimId}_{\texttt{D}}\texttt{-prop}$ it should be clear that $\texttt{fimap}_{\texttt{D}}$ also erases to $\lambda\ \texttt{x. x}$.

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

We now explain the definition of `PrfAlg`$_D$ in more detail:

- `R` is the type of recursive occurrences of the data-type `Fix`$_D$.

  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* `Fix`$_D$ witnessed by an identity function.

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

- `v` is evidence that the (generic) destructor `out`$_D$ 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 `out`$_D$ 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 Γ`$_D$ `(in`$_D$ `-Γ`$_D$ `(fimap`$_D$ `-im -c fr))` accounts for the return type of each case branch.

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

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 $\mathtt{Id_D} \cdot\mathtt{(F} \cdot\Gamma_\mathtt{p}\ \mathtt{R1)} \cdot\mathtt{(F} \cdot\Gamma\ \mathtt{R2)}$

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

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

where $\bullet_1\mathtt{:\ F} \cdot\Gamma_\mathtt{p}\ \mathtt{R2} \cdot\Gamma_\mathtt{D}$ and $\bullet_2\mathtt{:}\ \{\lambda\ \mathtt{fr1.}\ \bullet_1 \simeq \lambda\ \mathtt{x.\ x}\}$. As the only (non-hole) refinements we will make to $\bullet_1$ are converting terms to η-long form and applying $\mathtt{elimId_D\ -id}$ 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

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

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

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

$\mathtt{fr1}\ (\bullet_\mathtt{j}\ \mathtt{c_j})_{\mathtt{j}=1-n}$

where now $\bullet_\mathtt{j}\mathtt{:}\ (\overset{\Pi}{\forall}\ \Gamma^\mathtt{R2}{}_\mathtt{j}.\ \mathtt{X}\ \overline{\mathtt{t}}_\mathtt{j}) \to \overset{\Pi}{\forall}\ \Gamma^\mathtt{R1}{}_\mathtt{j}.\ \mathtt{X}\ \overline{\mathtt{t}}_\mathtt{j}$. 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 $\bullet\ \mathtt{c}$ η-long, as in $\overset{\lambda}{\Lambda}\ \Gamma^\mathtt{R1}.\ \bullet\ \mathtt{c}\ \Gamma^\mathtt{R1}$, then refine $\bullet\ \mathtt{c}\ \Gamma^\mathtt{R1}$ to an expression with m holes $\bullet_\mathtt{k}$ for each $\mathtt{y}_k \in \Gamma^\mathtt{R1}$ (where $m = \|\Gamma^\mathtt{R1}\|$), yielding

$\mathtt{c}\ (\bullet_\mathtt{k}\ \mathtt{y_k})_{\mathtt{k}=1-m}$

where $\bullet_\mathtt{k}\mathtt{:}\ \Gamma^\mathtt{R1}(\mathtt{y_k}) \to \Gamma^\mathtt{R2}{}_\mathtt{k}(\mathtt{y_k})$ (and the type of $\mathtt{y_k}$ and $\bullet_\mathtt{k}\ \mathtt{y_k}$ can depend resp. on any $\mathtt{y}^\mathtt{R1}{}_\mathtt{j}$ and $\bullet_\mathtt{j}\ \mathtt{y_j}$ 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 $\bullet\ \mathtt{y}$ where $\mathtt{y:\ S}$ is some sub-data to an (abstract) constructor with recursive occurence type R1 passing the positivity checker. (The expression $\bullet\ \mathtt{y}$ has type [R2/R1]S). There are two cases to consider:

1 R1 does not occur in the type of y

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

2 R1 occurs positively in the type of y

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

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

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

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

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

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

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

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

where $\bullet_{\mathtt{j}}\mathtt{:}\ \Gamma^{\mathtt{R1}}{}_{\mathtt{y}}(\mathtt{y}_{\mathtt{j}})\ \rightarrow\ \Gamma^{\mathtt{R2}}{}_{\mathtt{y}}(\mathtt{y}_{\mathtt{j}})$. 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

[CH86] Thierry Coquand and Gérard Huet. *The calculus of constructions.* PhD thesis, INRIA, 1986.

[FBS18] Denis Firsov, Richard Blair, and Aaron Stump. Efficient mendler-style lambda-encodings in cedille. In *International Conference on Interactive Theorem Proving*, pages 235–252. Springer, 2018.

[Kop03] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, LICS '03, pages 86–, Washington, DC, USA, 2003. IEEE Computer Society.

[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.

[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.

[Stu17] Aaron Stump. The calculus of dependent lambda eliminations. *Journal of Functional Programming*, 27, 2017.

[Stu18a] Aaron Stump. Syntax and semantics of cedille, 2018.

[Stu18b] Aaron Stump. Syntax and typing for cedille core. *arXiv preprint arXiv:1811.01318*, 2018.

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