Cedille 1.1.0 Datatype System Specification

Syntax, Typing, Reduction, and Elaboration

Christopher Jenkins

March 23, 2019

Contents

(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

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

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

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

(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

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 $[t.1/x]T'$

- $\forall\, x\!:\!T.\, T'$ 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: ★ =
| 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 (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 Nat in `suc` : Nat → `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` $\cdot$ A. Used outside of the datatype declaration, `nil` has its usual type: $\forall\, A\!:\!\star.$ `List` $\cdot\, A$.

- 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 $\texttt{Vec} \cdot A\ (\texttt{add}\ \texttt{zero}\ n)$ by the usual index refinement of pattern matching on indexed types; thanks to the reduction behavior of `add` this is convertible with $\texttt{Vec} \cdot A\ n$, the type of `ys`. Similarly, in the `vcons` branch the expected type is $\texttt{Vec} \cdot A\ (\texttt{add}\ (\texttt{suc}\ m')\ n)$, convertible with the type $\texttt{Vec} \cdot A\ (\texttt{suc}\ (\texttt{add}\ m'\ n))$ 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 $\texttt{Vec} \cdot A\ (\texttt{add}\ (\texttt{suc}\ m')\ n)$ 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 $\texttt{suc}\ (\texttt{add}\ m'\ n)$, we need a rule for μ-reduction. μ-reduction is a combination of fixpoint unrolling and case branch selection. Here, because the scrutinee is $\texttt{suc}\ m'$, 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.

- `is/Nat` : `Is/Nat` $\cdot$ `Nat` is the trivial `Is/Nat` witness.

- `to/Nat` : $\forall\, N\!:\!\star.\, \forall\, is\!:$`Is/Nat` $\cdot\, N.\, N \to$ `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.

```
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

In `predCV` the witness is of type $\texttt{Is/Nat} \cdot N$ 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 $\texttt{Is/Nat} \cdot \texttt{Nat}$ 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):

```
_ : {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. $\texttt{Is/Nat} \cdot N$ 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`

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

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` $\to$ `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 `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. <u>Here is where CoV induction is used</u>: 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`.

```
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

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 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

```
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

(`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`.

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.
```

```
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

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 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

```
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`

- 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.

```
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`

2 Syntax

We now turn to a more formal treatment of Cedille's datatype system. We begin by describing the syntax, where for completeness we present many of the same constructs described in [Stu18a]. Figure 11 shows the different grammatical categories of identifiers: the two new additions are c (constructor names) and D (datatype names).

In Figure 12 we extend the syntax of pure (erased) terms in Cedille with constructors, recursive definitions (μ) and case analysis (μ'). For convenience we also introduce an auxiliary category of sequences of expressions $\overline{s}$, used for (among other things) describing the sequence of variables bound by constructor patterns in μ and μ' expressions. (The notation using i ranging over $1..n$ (for some n) is explained below.)

Figure 13 lists the full syntax of annotated expressions in Cedille (kinds, types, and terms). The datatype system adds datatype names D, μ and μ'-expressions, and argument sequences. We explain μ- and μ'-expressions in more detail:

- $\mu\ x.\ t\ @P\ \{\texttt{|}\ c_i\ \overline{a_i}\}_{i=1..n}$

  x is the name of the recursive expression being defined by the μ-expression, in scope of the body (delimited by curly braces).

  t is the scrutinee: the expression which is being pattern-matched upon and whose subdata will be legal arguments for recursion using x. It must have a concrete datatype

a, u, x, y, z	term variables
X, Y, Z, R	type variables
κ	kind variables
c	constructors
D	datatype names

Figure 11: Identifiers

$$
\begin{array}{lcll}
p & ::= & x & \text{variables} \\
 & & \lambda\, u.\, p & \text{functions} \\
 & & c & \text{constructors} \\
 & & p\ p' & \text{applications} \\
 & & \mu\ u.\ p\ \{\mid\ c_i\ \overline{a_i} \to p_i\}_{i=1..n} & \text{recursive definitions} \\
 & & \mu'\ p\ \{\mid c_i\ \overline{a_i} \to p_i\}_{i=1..n} & \text{case analysis} \\
\overline{s} & ::= & \emptyset \mid s\ \overline{s} &
\end{array}
$$

Figure 12: Untyped terms

$@P$ is the guide. P must be a type-level λ-expression abstracting over a datatype and its indices and returning a type.

$c_i\ \overline{a_i} \to t_i$ where $i = 1..n$, describes a case tree – a collection of n constructor patterns (constructors c_i applied to variable arguments $\overline{a_i}$) associated with expressions (t_i) within which the constructor variable arguments are bound

- μ'<x> t $@P$ $\{\mid\ c_i\ \overline{a_i}\}_{i=1..n}$

  x is the witness that the scrutinee has a type legal for CoV pattern matching

  t is the scrutinee

  $@P$ is the guide, similar to above

  $c_i\ \overline{a_i} \to t_i$ where $i = 1..n$, is a case tree as above

Figure 14 lists the syntax for typing contexts. The construct $\text{IndEl}[D, \Gamma^{\text{P}}, \Gamma^{\text{I}}, R, \Delta, \Theta, \mathcal{E}]$ is explained in more detail in a later section; for now it suffices to say it is the internal representation of a declared datatype D with parameters Γ^{P} and indices Γ^{I}, constructors bound in context Δ, CoV globals in Θ, and elaborations in $\mathcal{E}$.

The syntax for datatype declarations is as expected, and given in Figure **??**

data D (Γ^{P}): K =
  | c_1 : T_1
  | ...
  | c_n : T_n .

- D, the datatype name
- Γ^{P}, the context of parameters. Each identifier-classifier pair is separated by parenthesis
- K the index-sort of the datatype
- c_i : T_i ($i = 1..n$), the constructors and their types. Elsewhere we will enforce that each T_i have a valid type for being a constructor of datatype D

Kinds K	$::=$	$\star$	the kind of types that classify terms
		$\Pi\, X\!:\!K.\, K'$	product over types
		$\Pi\, x\!:\!T.\, K$	product over terms
Types S, T, P	$::=$	X	type variables
		$\Pi\, x\!:\!S.\, T$	product over terms
		$\forall\, x\!:\!S.\, T$	implicit product over terms
		$\forall\, X\!:\!K.\, T$	implicit product over types
		$\lambda\, x\!:\!S.\, T$	term-to-type function
		$\lambda\, X\!:\!K.\, T$	type-to-type function
		$T\ t$	type-to-term application
		$T\ \cdot S$	type-to-type application
		$\iota\, x\!:\!T.\, T'$	Dependent intersection
		$\{p_1 \simeq p_2\}$	Equality of untyped terms
		D	datatypes
Classifiers A	$::=$	$T \mid K$	
Terms s, t	$::=$	x	variables
		$\lambda\, x.\, t$	term abstraction
		$\Lambda\, x.\, t$	erased term abstraction
		$\Lambda\, X.\, t$	type abstraction
		$t\ s$	term application
		t -s	erased term application
		$t \cdot T$	type application
		$[t, s]$	intro dependent intersection
		$t.1$	dep. intersection left projection
		$t.2$	dep. intersection right projection
		β	reflexivity of equality
		$\rho\ t\ @x.T\ - s$	rewrite by equality
		$\varphi\ t - t_1\ \{t_2\}$	cast by equality
		$\delta\ T - t$	anything by absurd equality
		c	data constructors
		$\mu\ x.\ t\ @P\ \{\mid\ c_i\ \overline{a_i} \to t_i\}_{i=1..n}$	recursive def. over datatype
		μ'<x> $t\ @P\ \{\mid\ c_i\ \overline{a_i} \to t_i\}_{i=1..n}$	case analysis over datatype
Argument Sequence $\overline{s}$	$::=$	$\emptyset \mid s\ \overline{s} \mid\ \cdot S\ \overline{s} \mid$ -$s\ \overline{s}$	for constructor patterns and applications

Figure 13: Syntax for Cedille kinds, types, terms

$$\text{Typing contexts } \Gamma \quad ::= \quad \emptyset \mid \Gamma, x\!:\!T \mid \Gamma, X\!:\!K \mid \Gamma, \mathrm{IndElI}[D, \Gamma^{\mathrm{P}}, \Gamma^{\mathrm{I}}, R, \Delta, \Theta, \mathcal{E}]$$

Figure 14: Contexts

3 Erasure

The definition of the erasure function given in Figure 15 takes the annotated terms from Figures 13 to the untyped terms of Figure 12. Specifically, for the new datatype constructs:

$$
\begin{array}{lcl}
|x| & = & x \\
|\star| & = & \star \\
|\{t \simeq t'\}|| & = & \{|t| \simeq |t'|\} \\
|\beta| & = & \lambda\, x.\, x \\
|\delta\ T - t| & = & |t| \\
|\varphi\ t\ \text{-}\ t'\ \{t''\}| & = & |t''| \\
|\rho\ t'\ @\ x.T\ \text{-}\ t| & = & |t| \\
|\iota\, x{:}T.\, T'| & = & \iota\, x{:}|T|.\, |T'| \\
|[t, t']| & = & |t| \\
|t.1| & = & |t| \\
|t.2| & = & |t| \\
|\Pi\, x{:}T.\, T'| & = & \Pi\, x{:}|T|.\, |T'| \\
|\lambda\, x.\, t| & = & \lambda\, x.\, |t| \\
|t\ t'| & = & |t|\ |t'| \\
|\forall\, x{:}T.\, T'| & = & \forall\, x{:}|T|.\, |T'| \\
|\Lambda\, x{:}T.\, t| & = & |t| \\
|t\ \text{-}t'| & = & |t| \\
|\forall\, X{:}K.\, T| & = & \forall\, X{:}|K|.\, |T| \\
|\Lambda\, X{:}K.\, t| & = & |t| \\
|t\ \cdot T| & = & |t| \\
|\Pi\, x{:}T.\, K| & = & \Pi\, x{:}|T|.\, |K| \\
|\lambda\, x{:}T.\, T'| & = & \lambda\, x{:}|T|.\, |T'| \\
|T\ t| & = & |T|\ |t| \\
|\Pi\, X{:}K.\, K'| & = & \Pi\, X{:}|K|.\, |K'| \\
|\lambda\, X{:}K.\, T| & = & \lambda\, X{:}|K|.\, |T| \\
|T \cdot T'| & = & |T| \cdot |T'| \\
|c| & = & c \\
|\mu\ u.\ t\ @P\ \{\mid\ c_i\ \overline{a_i} \to t_i\}_{i=1..n}| & = & \mu\ u.\ |t|\ \{\mid\ c_i\ \overline{|a_i|} \to |t_i|\}_{i=1..n} \\
|\mu'\text{<}u\text{>}\ t\ @P\ \{\mid\ c_i\ \overline{a_i} \to t_i\}_{i=1..n}| & = & \mu'\ |t|\ \{\mid c_i\ \overline{|a_i|} \to |t_i|\}_{i=1..n} \\
 & & \\
|\varnothing| & = & \varnothing \\
|s\ \overline{s}| & = & s\ |\overline{s}| \\
|\text{-}s\ \overline{s}| & = & |\overline{s}| \\
|\cdot\ S\ \overline{s}| & = & |\overline{s}|
\end{array}
$$

Figure 15: Erasure for annotated terms

- in μ-expressions, the guide $@P$ is erased;
- in μ'-expressions, the witness u and guide $@P$ are erased;
- in the case trees of both, argument sequences are erased (type and implicit term variables bound in constructor patterns are erased);

4 Convertibility

$$\frac{1 \leq j \leq n \quad \#\overline{s} = \#\overline{a_j}}{\mu' \; (c_j \; \overline{s}) \; \{ \mid c_i \; \overline{a_i} \to t_i \}_{i=1..n} \leadsto \overline{[s/a_j]} t_j} \qquad \frac{1 \leq j \leq n \quad \#\overline{s} = \#\overline{a_j} \quad t^{\mathrm{rec}} = \lambda \, y. \, \mu \; x. \; y \; \{ \mid c_i \; \overline{a_i} \to t_i \}_{i=1..n}}{\mu \; x. \; (c_j \; \overline{s}) \; \{ \mid c_i \; \overline{a_i} \to t_i \}_{i=1..n} \leadsto \overline{[s/a_j]} [t^{\mathrm{rec}}/x] t_j}$$

Figure 16: Reduction rules for μ and μ'

$$\frac{\begin{array}{cc} \mathrm{IndEll}[D, R, \Gamma^{\mathrm{P}}, \Gamma^{\mathrm{I}}, \Delta, \Theta, \mathcal{E}] \in \Gamma & c_j : ^{\Pi}_{\forall} \overline{a_j : A_j}. \, T \in \Delta \\ \mathrm{IndEll}[D', R, \Gamma^{\mathrm{P}\prime}, \Gamma^{\mathrm{I}\prime}, \Delta', \Theta', \mathcal{E}'] \in \Gamma & c_k{}' : ^{\Pi}_{\forall} \overline{a_k : A_k{}'}. \, T' \in \Delta' \\ j = k, \#\Delta = \#\Delta' & \#|\overline{a_j}| = \#|\overline{a_k}| \end{array}}{\Gamma \vdash c_j \cong c_k{}'} \qquad \frac{\mathrm{IndEll}[D, \Gamma^{\mathrm{P}}, \Gamma^{\mathrm{I}}, R, \Delta, \Theta, \mathcal{E}] \in \Gamma \quad \texttt{to/}D \in \Theta}{\Gamma \vdash \texttt{to/}D \cong \lambda \, x. \, x}$$

Figure 17: Extension of definitional equality for data declarations

Notation In Figures 16 and 17, a metavariable c denotes a datatype constructor, $\overline{s}$ a sequence of type and (mixed-erasure) term arguments, $\overline{a}$ a sequence of type and (mixed-erasure) term variables bound by pattern guards, $\#\overline{s}$ the length of $\overline{s}$, $\{ \mid c_i \; \overline{a_i} \to t_i \}_{i=1..n}$ a collection of n branches guarded by patterns $c_i \; \overline{a_i}$ with bodies t_i, $|\overline{a}|$ the erasure of type and erased-term variables in the sequence $\overline{a}$, and $\overline{[s/a]}$ the simultaneous and capture-avoiding substitution of terms and types $\overline{s}$ for variables $\overline{a}$.

The convertibility relation $\cong$ for types is the relation described in [Stu18a] with term convertibility augmented with these rules.

μ' reduction The first rule of Figure 16 is μ'-reduction, which is simply case branch selection: if the scrutinee is some constructor c_j applied to arguments $\overline{s}$, and the case tree lists c_j applied to the same number of (variable) arguments $\overline{a_j}$, the corresponding expression t_j of that branch is selected with constructor arguments $\overline{s}$ replacing variables $\overline{a_j}$

μ reduction The second rule of Figure 16 is μ-reduction, a combination of case branch selection and fixpoint unrolling. The fixpoint unrolling is done by binding term meta-variable t^{rec} to a λ-expression that takes an argument y and makes it the scrutinee of another μ-expression, with the same case branches as before. t^{rec} replaces the μ-bound variable x in the selected case branch t_j.

Constructor convertibility The first rule of Figure 17 shows how Cedille determines whether two constructors are convertible.

- The constructors must be associated with a datatypes D and D' declared in Γ

- They must have the same entry ($j = k$) in their respective constructor lists, and these lists must be equal in length $\#\Delta = \#\Delta'$
- They must take the same number of unerased arguments

Coercion convertibility Any coercion $\texttt{to}/D$ bound by a datatype declaration is convertible with $\lambda\, x.\, x$.

5 Elaborating Type Inference Rules (without Datatypes)

This section lists the elaborating type inference rules for Cedille without datatypes (i.e., congruence elaboration rules for Cedille 1.0.0), and the auxiliary elaboration rules for elaborating telescopes formed by a sequence of expressions. To simplify the presentation we do not show elaboration to *Cedille Core*, whose terms require significantly more type annotations that would clutter the inference rules. Terms in Cedille 1.0.0 maps straightforwardly to Cedille Core terms. We describe each judgment form and (briefly) a few of the rules comprising it.

Rules marked by `<:` indicate rules part of the subtyping proposal for Cedille 1.1.1

5.1 Main judgments

(a) $\boxed{\Gamma \vdash K \hookrightarrow K'}$ Kind elaboration

$$\frac{}{\Gamma \vdash \star \hookrightarrow \star} \qquad \frac{\Gamma \vdash K_1 \hookrightarrow K_1' \quad \Gamma, X\!:\!K_1 \vdash K_2 \hookrightarrow K_2'}{\Gamma \vdash \Pi\, X\!:\!K_1.\, K_2 \hookrightarrow \Pi\, X\!:\!K_1'.\, K_2'} \qquad \frac{\Gamma \vdash T : K_2 \hookrightarrow T' \quad \Gamma, x\!:\!T \vdash K_1 \hookrightarrow K_1'}{\Gamma \vdash \Pi\, x\!:\!T.\, K_1 \hookrightarrow \Pi\, x\!:\!T'.\, K_1'}$$

(b) $\boxed{\Gamma \vdash T : K \hookrightarrow T'}$ Type Elaboration (sans datatypes)

$$\frac{FV(p_1\ p_2) \subseteq dom(\Gamma) \quad \Gamma \vdash p_1 \hookrightarrow p_1' \quad \Gamma \vdash p_2 \hookrightarrow p_2'}{\Gamma \vdash \{p_1 \simeq p_2\} : \star \hookrightarrow \{p_1' \simeq p_2'\}} \qquad \frac{\Gamma \vdash T_1 : \star \hookrightarrow T_1' \quad \Gamma, x : T_1 \vdash T_2 : \star \hookrightarrow T_2'}{\Gamma \vdash \iota\, x\!:\!T_1.\, T_2 : \star \hookrightarrow \iota\, x\!:\!T_1'.\, T_2'}$$

$$\frac{\Gamma \vdash T_1 : \star \hookrightarrow T_1' \quad \Gamma, x : T_1 \vdash T_2 : \star T_2'}{\Gamma \vdash \forall\, x\!:\!T_1.\, T_2 : \star \hookrightarrow \forall\, x\!:\!T_1'.\, T_2'} \qquad \frac{\Gamma \vdash T_1 : \star \hookrightarrow T_1' \quad \Gamma, x : T_1 \vdash T_2 : \star \hookrightarrow T_2'}{\Gamma \vdash \Pi\, x\!:\!T_1.\, T_2 : \star \hookrightarrow \Pi\, x\!:\!T_1'.\, T_2'}$$

$$\frac{\Gamma \vdash K \hookrightarrow K' \quad \Gamma, X : K \vdash T : \star \hookrightarrow T'}{\Gamma \vdash \forall\, X\!:\!K.\, T : \star \hookrightarrow \forall\, X\!:\!K'.\, T'} \qquad \frac{\Gamma \vdash S : K_1 \hookrightarrow S' \quad \Gamma, x\!:\!S \vdash T : K_2 \hookrightarrow T'}{\Gamma \vdash \lambda\, x\!:\!S.\, T : \Pi\, x\!:\!S.\, K_2 \hookrightarrow \lambda\, x\!:\!S'.\, T'}$$

$$\frac{\Gamma \vdash K_1 \hookrightarrow K_1' \quad \Gamma, X\!:\!K_1 \vdash T : K_2 \hookrightarrow T'}{\Gamma \vdash \lambda\, X\!:\!K_1.\, T : \Pi\, X\!:\!K_1.\, K_2 \hookrightarrow \lambda\, X\!:\!K_1'.\, T'} \qquad \frac{\Gamma \vdash T_1 : \Pi\, X\!:\!K_2.\, K_1 \hookrightarrow T_1' \quad \Gamma \vdash T_2 : K_2 \hookrightarrow T_2'}{\Gamma \vdash T_1 \cdot T_2 : [T_2/X]K_1 \hookrightarrow T_1' \cdot T_2'}$$

$$\frac{\Gamma \vdash T : \Pi\, x\!:\!S.\, K \hookrightarrow T' \quad \Gamma \vdash t : S \hookrightarrow t'}{\Gamma \vdash T\ t : [t/x]K \hookrightarrow T'\ t'} \qquad \frac{}{\Gamma \vdash X : \Gamma(X) \hookrightarrow X}$$

$$\frac{\Gamma \vdash T : K_1 \hookrightarrow T' \quad \Gamma \vdash K_1 \mathrel{\texttt{<:}} K_2 \nearrow S \quad \Gamma \vdash S : \Pi\, X\!:\!K_1.\, K_2 \hookrightarrow S'}{\Gamma \vdash T : K_2 \hookrightarrow S' \cdot T'}\ \texttt{<:}$$

- $\Gamma \vdash K \hookrightarrow K'$

  Read: "Under context Γ, K is a well-formed kind and elaborates to K'". Consider the second rule: to elaborate $\Pi\, X\!:\!K_1.\, K_2$ first elaborate K_1 to K_1', then elaborate K_2 to K_2' under a contextnd extended by $X\!:\!K_1$; the result is $\Pi\, X\!:\!K_1'.\, K_2'$

$$\frac{}{\Gamma \vdash x : \Gamma(x) \hookrightarrow x} \qquad \frac{\Gamma \vdash S : \star \hookrightarrow S' \quad \Gamma, x{:}S \vdash t : T \hookrightarrow t'}{\Gamma \vdash \lambda\, x.\, t : \Pi\, x{:}S.T \hookrightarrow \lambda\, x.\, t'}$$

$$\frac{\Gamma \vdash S : \star \hookrightarrow S' \quad x \notin FV(|t|) \quad \Gamma, x{:}S \vdash t : T \hookrightarrow t'}{\Gamma \vdash \Lambda\, x.\, t : \forall\, x{:}S.T \hookrightarrow \Lambda\, x.\, t'} \qquad \frac{\Gamma \vdash K \hookrightarrow K' \quad X \notin FV(|t|) \quad \Gamma, X{:}K \vdash t : T \hookrightarrow t'}{\Gamma \vdash \Lambda\, X.\, t : \forall\, X{:}K.\, T \hookrightarrow \Lambda\, X.\, t'}$$

$$\frac{\Gamma \vdash t : \Pi\, x{:}S.T \hookrightarrow t' \quad \Gamma \vdash s : S \hookrightarrow s'}{\Gamma \vdash t\ s : [s/x]T \hookrightarrow t'\ s'} \qquad \frac{\Gamma \vdash t : \forall\, X{:}K.T \hookrightarrow t' \quad \Gamma \vdash S : K \hookrightarrow S'}{\Gamma \vdash t \cdot S : [S/X]T \hookrightarrow t' \cdot S'}$$

$$\frac{\Gamma \vdash t : \forall\, x{:}S.T \hookrightarrow t' \quad \Gamma \vdash s : S \hookrightarrow s'}{\Gamma \vdash t\ \text{-}s : [s/x]T \hookrightarrow t'\ \text{-}s'} \qquad \frac{\begin{array}{cc}\Gamma \vdash t : S \hookrightarrow t' & \Gamma \vdash T : \star \hookrightarrow T' \\ S \mathrel{\texttt{<:}} T \nearrow s & \Gamma \vdash s : S \to T \hookrightarrow s'\end{array}}{\Gamma \vdash t : T \hookrightarrow s'\ t'}\ \texttt{<:}$$

$$\frac{\Gamma \vdash t_1 : T_1 \hookrightarrow t_1' \quad \Gamma \vdash t_2 : [t_1/x]T_2 \hookrightarrow t_2' \quad t_1' \cong t_2'}{\Gamma \vdash [t_1, t_2] : \iota\, x{:}T_1.\, T_2 \hookrightarrow [t_1', t_2']} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.1 : T_1 \hookrightarrow t'.1}$$

$$\frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2 \hookrightarrow t'}{\Gamma \vdash t.2 : [t.1/x]T_2 \hookrightarrow t'.2} \qquad \frac{\Gamma \vdash \{|t| \simeq |t|\} : \star \hookrightarrow \{p \simeq p\} \quad FV(p_1) \subseteq dom(\Gamma) \quad \Gamma \vdash p_1 \hookrightarrow p_1'}{\Gamma \vdash \beta\{p_1\} : \{|t| \simeq |t|\} \hookrightarrow \beta\{p_1'\}}$$

$$\frac{\begin{array}{cc}\Gamma \vdash s : \{|t_1| \simeq |t_2|\} \hookrightarrow s' & T_1 \cong T_2 \\ \Gamma \vdash t : [t_1/x]T_1 \hookrightarrow t' & \Gamma \vdash [t_2/x]T_2 : \star \hookrightarrow [t_2'/x]T_2'\end{array}}{\Gamma \vdash \rho\ s\ @\ x.T_2\ \text{-}\ t : [t_2/x]T_2 \hookrightarrow \rho\ s'\ @\ x.T_2'\ \text{-}\ t'} \qquad \frac{\Gamma \vdash s : \{|t_1| \simeq |t_2|\} \hookrightarrow s' \quad \Gamma \vdash t_1 : T \hookrightarrow t_1' \quad \Gamma \vdash t_2 \hookrightarrow t_2'}{\Gamma \vdash \varphi\ s - t_1\ \{t_2\} : T \hookrightarrow \varphi\ s' - t_1'\ \{t_2'\}}$$

$$\frac{\Gamma \vdash t : \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, x\} \hookrightarrow t' \quad \Gamma \vdash T : \star \hookrightarrow T'}{\Gamma \vdash \delta\ T - t : T \hookrightarrow \delta\ T' - t'}$$

Figure 19: $\boxed{\Gamma \vdash t : T \hookrightarrow t'}$ Term elaboration (without datatypes)

- $\Gamma \vdash T : K \hookrightarrow T'$

  Read: "Under context Γ, type T has kind K and elaborates to T'". The rule for elaborating the equality type is worth explaining further as it makes use of a new judgment for pure-term elaboration $\Gamma \vdash p_1 \hookrightarrow p_1'$. To type and elaborate $\{p_1 \simeq p_2\}$, check that the free variables of p_1 and p_2 are declared by the typing context Γ; then, elaborate terms p_1 and p_2 to resp. p_1' and p_2'; the resulting elaborated type is $\{p_1' \simeq p_2'\}$

- $\Gamma \vdash t : T \hookrightarrow t'$

  Read: "Under context Γ, term t has type T and elaborates to t'". We will note a few of the rules for this judgment, with the list below indicating the rule by its subject of typing:

  - $\rho\ s\ @x.T_2\ -t$. We use a modified rule for ρ ("rewrite a type by an equation") based on the form used by Cedille Core[Stu18b]. This version of the rule ensures that the resulting expression has a well-kinded type, which is not the case for all other proposed versions.

    If s has type $\{|t_1| \simeq |t_2|\}$ (i.e. provides the equation we will rewrite by) and elaborates to s', and t (the term whose type we are rewriting) has type $[t_1/x]T_1$ and elaborates to t', and T_1 is convertible with the user-supplied type guide T_2, and finally the type $[t_2/x]T_2$ has kind $\star$ and elaborates to $[t_2/x]T_2'$, then the entire expression has type $[t_2/x]T_2$ and elaborates to $\rho\ s'\ @x.T_2' - t'$
  - $\Lambda\, x.\, t$. The domain S of this implicit function must have kind $\star$ and elaborate to a type S'. The Λ-bound variable x must not occur free in the erasure of the body t. On the assumption of $x{:}S$, t must have type T and elaborate to t'.

    (TODO) In the appendix we will show that elaborate is sound with respect to FV (e.g. the free variables of t are the same as the free variables of t', and similiarly for $|t|$ and $|t'|$).

We also explain the subtyping subsumption rule (for Cedille 1.0.0, replaced the relation `<:` with $\cong$ and the coercions s with $\lambda\, x.\, x$ – well, s was going to be that already!).

If t has type S and elaborates to S', and S is a subtype of T with the evidence of that being s, and further, subtyping evidence s has type $S \to T$ and elaborates to s', then t also has type T and elaborates to $s'\ t'$. As we will show further on, coercion s (and thus its elaboration s') will always be definitionally equal to $\lambda\, x.\, x$.

5.2 Auxiliary Judgments

(a) $\boxed{\vdash \Gamma \hookrightarrow \Gamma'}$ Elaboration of contexts

$$\frac{}{\vdash \emptyset \hookrightarrow \emptyset} \qquad \frac{\vdash \Gamma \hookrightarrow \Gamma' \quad \Gamma \vdash T : \star \hookrightarrow T'}{\vdash \Gamma, x{:}T \hookrightarrow \Gamma', x{:}T'}$$

$$\frac{\vdash \Gamma \hookrightarrow \Gamma' \quad \Gamma \vdash K \hookrightarrow K'}{\vdash \Gamma, X{:}K \hookrightarrow \Gamma', X{:}K'} \qquad \frac{\vdash \Gamma \hookrightarrow \Gamma'}{\vdash \Gamma, \mathrm{IndElI}[D, \Gamma^{\mathrm{P}}, \Gamma^{\mathrm{I}}, R, \Delta, \Theta, \mathcal{E}] \hookrightarrow \Gamma'}$$

(b) $\boxed{\Gamma\,;\,(\overline{a{:}A}) \vdash \overline{s} : (\overline{a{:}B}) \hookrightarrow \overline{s'}}$ Elaboration of type-coerced constructor argument telescope

$$\frac{}{\Gamma\,;\,\varnothing \vdash \varnothing : \varnothing \hookrightarrow \varnothing} \qquad \frac{\Gamma, x{:}S \vdash s : T \hookrightarrow s' \quad \Gamma, x{:}S\,;\,(\overline{a{:}A}) \vdash \overline{s} : ([s/x]\overline{a{:}B}) \hookrightarrow \overline{s'}}{\Gamma\,;\,(x{:}S, \overline{a{:}A}) \vdash s, \overline{s} : (x{:}T, \overline{a{:}B}) \hookrightarrow s', \overline{s'}}$$

$$\frac{\Gamma, X{:}K_1 \vdash S : K_2 \hookrightarrow S' \quad \Gamma, X{:}K_1\,;\,(\overline{a{:}A}) \vdash \overline{s} : ([S/X]\overline{a{:}B}) \hookrightarrow \overline{s'}}{\Gamma\,;\,(X{:}K_1, \overline{a{:}A}) \vdash S, \overline{s} : (X{:}K_2, \overline{a{:}B}) \hookrightarrow S', \overline{s'}}$$

- $\vdash \Gamma \hookrightarrow \Gamma'$

  Read: "Context Γ elaborates to context Γ'". These rules are straightforward except for the last: declarations of term variables of type T translate to term variables of type T' (where T elaborates to T' under the context prefix), and similarly for type variables.

  The rule for datatype declarations indicates that these disappear completely.

- $\Gamma; (\overline{a\!:\!A}) \vdash \overline{s} : (\overline{a\!:\!B}) \hookrightarrow \overline{s'}$

  Read: "Under context Γ and a dependent telescope $(\overline{a}A)$ of term and type variables, the sequence of (term and type) expressions $\overline{s}$ can be classified by the telescope $(\overline{a}B)$, elaborating expressions s'"

  It is helpful to give an example: let the telescope to the right of the $\vdash$ be $(x_1\!:\!S_1, y_1\!:\!T_1\ x_1)$ (for some declared types $S_1 : \star$ and $T_1 : S_1 \to \star$), and the one to the left of the colon : be $(x_2\!:\!S_2, y_2\!:\!T_2\ x_2)$ (for some declared types $S_2 : \star$ and $T_2 : S_2 \to \star$). The sequence of expressions we will classify with these telescopes are $\mathit{cast}\ x_1, f\ (\mathit{cast}\ x_1)$, where $\mathit{cast} : S_1 \to S_2$ and $f : \Pi\, z\!:\!S_2.\, T_2\ z$ are also declared terms.

  Using the second rule, we first check that $\mathit{cast}\ x_1$ has type S_2 under a context extended by $x_1\!:\!S_1$; let us say resulting elaborated term is $\mathit{cast}'\ x_1$. We now check the rest of the telescope under a context similarly extended, substituting $[\mathit{cast}\ x_1/x_2]$ in $T_2\ x_2$.

  Now it remains to check $f\ (\mathit{cast}\ x_1)$ against type $T_2\ (\mathit{cast}\ x_1)$ (we do not need the additional assumption $y_1\!:\!T_1\ x_1$ that the context was extended by, but we still need to make use of $x_1\!:\!S_1$). Let us say this elaborates to $f'\ (\mathit{cast}'\ x_1)$. So the whole sequence elaborates $\mathit{cast}'\ x_1,\ f'\ (\mathit{cast}'\ x_1)$

  Though not enforced by these three rules, the judgment for telescope coercions will only ever be used for elaborating (zero-cost) coercions of the variables of a telescope, meaning that in all uses of the judgment $|\overline{s}| = |\overline{a}|$.

6 Elaborating Subtyping Rules

(a) $\boxed{\Gamma \vdash S \mathrel{\texttt{<:}} T \nearrow s'}$ Elaboration of subtyping

$$\frac{S \cong T}{\Gamma \vdash S \mathrel{\texttt{<:}} T \nearrow \lambda\, x.\, x} \qquad \frac{|\overline{s_1^{\mathrm{I}}}| \cong |\overline{s_2^{\mathrm{I}}}| \quad \mathrm{IndElI}[D, \Gamma^{\mathrm{P}}, \Gamma^{\mathrm{I}}, R, \Delta, \Theta, \mathcal{E}] \in \Gamma \\ (\texttt{Is/}D, \texttt{to/}D) \in \Theta \quad \Gamma \vdash x : \texttt{Is/}D\ \overline{s^{\mathrm{P}}}\ S}{\Gamma \vdash S\ \overline{s_1^{\mathrm{I}}} \mathrel{\texttt{<:}} D\ \overline{s^{\mathrm{P}}}\ \overline{s_2^{\mathrm{I}}} \nearrow \texttt{to/}D\ \overline{s^{\mathrm{P}}}\ \text{-}\overline{s_1^{\mathrm{I}}}\ \text{-}\ x}$$

$$\frac{\Gamma \vdash S_2 \mathrel{\texttt{<:}} S_1 \nearrow s \quad \Gamma, y\!:\!S_2 \vdash [(s\ y)/x]T_1 \mathrel{\texttt{<:}} [y/x]T_2 \nearrow t}{\Gamma \vdash \Pi\, x\!:\!S_1.\, T_1 \mathrel{\texttt{<:}} \Pi\, x\!:\!S_2.\, T_2 \nearrow \lambda\, f.\, \lambda\, x.\, [x/y]t\ (f\ (s\ x))}$$

$$\frac{\Gamma \vdash S_2 \mathrel{\texttt{<:}} S_1 \nearrow s \quad \Gamma, y\!:\!S_2 \vdash [(s\ y)/x]T_1 \mathrel{\texttt{<:}} [y/x]T_2 \nearrow t}{\Gamma \vdash \forall\, x\!:\!S_1.\, T_1 \mathrel{\texttt{<:}} \forall\, x\!:\!S_2.\, T_2 \nearrow \lambda\, f.\, \Lambda\, x.\, [x/y]t\ (f\ \text{-}(s\ x))}$$

$$\frac{\Gamma \vdash S_1 \mathrel{\texttt{<:}} S_2 \nearrow s \quad \Gamma, y : S_1 \vdash [y/x]T_1 \mathrel{\texttt{<:}} [(s\ y)/x]T_2 \nearrow t}{\Gamma \vdash \iota\, x\!:\!S_1.\, T_1 \mathrel{\texttt{<:}} \iota\, x\!:\!S_2.\, T_2 \nearrow \lambda\, u.\, [s\ u.1, [u.1/y]t\ u.2]}$$

$$\frac{\Gamma \vdash K_2 \mathrel{\texttt{<:}} K_1 \nearrow S \quad \Gamma, Y : K_2 \vdash [(S \cdot Y)/X]T_1 \mathrel{\texttt{<:}} [Y/X]T_2 \nearrow s}{\Gamma \vdash \forall\, X\!:\!K_1.\, T_1 \mathrel{\texttt{<:}} \forall\, X\!:\!K_2.\, T_2 \nearrow \lambda\, f.\, \Lambda\, X.\, [X/Y]s\ (f \cdot (S \cdot X))}$$

(b) $\boxed{\Gamma \vdash K_1 \mathrel{\texttt{<:}} K_2 \nearrow S}$ Elaboration of subkinding

$$\frac{}{\Gamma \vdash \star \mathrel{\texttt{<:}} \star \nearrow \lambda\, X\!:\!\star.\, X} \qquad \frac{\Gamma \vdash S_2 \mathrel{\texttt{<:}} S_1 \nearrow s \quad \Gamma, y : S_2 \vdash [(s\ y)/x]K_1 \mathrel{\texttt{<:}} [y/x]K_2 \nearrow S}{\Gamma \vdash \Pi\, x\!:\!S_1.\, K_1 \mathrel{\texttt{<:}} \Pi\, x\!:\!S_2.\, K_2 \nearrow \lambda\, P\!:\!(\Pi\, x\!:\!S_1.\, K_1).\, \lambda\, x\!:\!S_2.\, [x/y]S \cdot (P\ (s\ x))}$$

$$\frac{\Gamma \vdash K_1' \mathrel{\texttt{<:}} K_1 \nearrow S_1 \quad \Gamma, Y : K_1' \vdash [(S_1 \cdot Y)/X]K_2 \mathrel{\texttt{<:}} [Y/X]K_2' \nearrow S_2}{\Gamma \vdash \Pi\, X\!:\!K_1.\, K_2 \mathrel{\texttt{<:}} \Pi\, X\!:\!K_1'.\, K_2' \nearrow \lambda\, P\!:\!(\Pi\, X\!:\!K_1.\, K_2).\, \lambda\, X\!:\!K_1'.\, [X/Y]S_2 \cdot (P \cdot (S_1 \cdot X))}$$

6.1 Subtyping

Figure 21a shows Cedille's evidence-producing subtyping rules. The core rule is in the first row, second column; the rest are congruence and convertibility rules.

Read the judgment $\Gamma \vdash S \mathrel{\texttt{<:}} T \nearrow s'$ as: "under a context Γ, type S is a subtype of T and the coercion that witnesses this fact is s'". For the core rule of this judgment, we are in a case where we are trying to show some type S applied to term and type arguments $\overline{s_1^{\mathrm{I}}}$ is a subtype of datatype D (instantiated with parameters $\overline{s^{\mathrm{P}}}$ and indicies $\overline{s_2^{\mathrm{I}}}$). We check that the two sets of indices are equal, that D really is a declared datatype, and finally that we have some assumption already in the context which tells us that type S "is" a $D\ \overline{s^{\mathrm{P}}}$. The global names $\texttt{to/}D$ and $\texttt{Is/}D$ are explained in Section 8 (and in Section 1); here it suffices to say that $\texttt{to/}D$ is a zero-cost cast (modulo the same indices) from any type S to $D\ \overline{s^{\mathrm{P}}}$ when it is provided evidence that $\texttt{Is/}D\ \overline{s^{\mathrm{P}}} \cdot S$.

One helpful way to view these rules is as a type-guided construction of zero-cost casts between two types following structural subtyping rules, where our base subtyping assumptions are given by variables declared in the typing context of an appropriate type.

6.2 Subkinding

Subtyping for types that quantify over other types requires a *subkinding* judgment. Read the judgment $\Gamma \vdash K_1 \mathrel{\texttt{<:}} K_2 \nearrow S$ as: "under context Γ, K_1 is a subkind of K_2 and the coercion that witnesses this fact is S". Unlike for terms, kind annotations for the abstracted variable of λ-expressions at the type level remain after erasure; therefore we cannot say the kind coercion produced by the subkinding rules is always definitionally equal to $\lambda\, X.\, X$, as this is not even a well-formed erased type! The only invariant these rules preserve is that if $\Gamma \vdash K_1 \mathrel{<:} K_2 \nearrow S$ then $\Gamma \vdash S : K_1 \to K_2$ (i.e. the kind of S is $\Pi\, X : K_1.\, K_2$ where X does not occur in K_2). This poses no problem for the subtyping rules, as all such type coercions occur within type arguments to terms, which are erased.

We consider the second rule of the first row: the goal is to produce a coercion between the kinds $\Pi\, x : S_1.\, K_1$ and $\Pi\, x : S_2.\, K_2$. First we have the usual contravariance for functionals: it must be the case that S_2 is a subtype of S_1, with s type coercion (of type $S_2 \to S_1$ and erasing to $\lambda\, x.\, x$) witnessing this fact. Next, under context Γ extended by assumption $y : S_2$[2] we must have that K_1 with free variable x substituted with $s\ y$ is a subkind of K_2 with free variable x substituted for y. The resulting coercion S has kind $[(s\ y)/x]K_1 \to [y/x]K_2$. Finally, coercion produced in the conclusion of the judgment is formed by abstracting over type variable P of the subkind $\Pi\, x : S_1.\, K_1$ and term variable x of the domain S_2 of the superkind $\Pi\, x : S_2.\, K_2$ and then a sequence of term and type applications. Read innermost to outermost: coerce the term variable x to type S_1, and give the result to P, then coerce the kind of this expression with S (renaming y to x within S).

[2] We α-convert the abstracted variable to improve readability

7 Datatype Elaboration Interface

Figure 22: Definitions used by positivity checker

```
module Positivity (ΓP)
Id: (ΓI → ⋆) → (ΓI → ⋆) → ⋆
= λ A: ΓI → ⋆. λ B: ΓI → ⋆.
  Σ f: (∀ ΓI. A ΓI → B ΓI). {f ≃ id}.

intrId: ∀ A: ΓI → ⋆. ∀ B: ΓI → ⋆.
  Π f: (∀ ΓI. A ΓI → B ΓI). (∀ ΓI. Π x: A ΓI. {f x ≃ x}) → Id ·A ·B
= <..>

elimId: ∀ A: ΓI → ⋆. ∀ B: ΓI → ⋆. ∀ _: Id ·A ·B. ∀ ΓI. A ΓI → B ΓI
= <..>.
_ : {elimId ≃ λ x. x} = β.

IdMapping: ((ΓI → ⋆) → (ΓI → ⋆)) → ⋆
= λ F: (ΓI → ⋆) → (ΓI → ⋆).
  ∀ A: ΓI → ⋆. ∀ B: ΓI → ⋆. (Id ·A ·B) → (Id ·(F ·A) ·(F ·B)).

imap: ∀ F: (ΓI → ⋆) → (ΓI → ⋆).
  ∀ _: IdMapping ·F. ∀ A: ΓI → ⋆. ∀ B: ΓI → ⋆. ∀ _: Id ·A ·B. ∀ ΓI. F ·A ΓI → F ·B ΓI
= <..>
_ : {imap ≃ λ x. x} = β.
```

7.1 Generic Framework

The elaborator interface is implemented using the definitions in Figures 22 and 23, which list resp. a set of utilities and the primary results provided by the generic framework. Our implementation regenerates these definitions for each datatype to support large indices (type-indexed types), not handled by the generic framework. All such definitions are definable in Cedille *without datatypes* and thus map straightforwardly to CDLE. For the utilities, we have:

- `Id`, `intrId`, `elimId`: `Id` is a generalized type of identity functions in CDLE using the fact that a term may have multiple types. A term c of type $\texttt{Id} \cdot A \cdot B$ is a dependent pair (Σ, also definable) whose first projection $\pi_1\ c$ has type $A \to B$ and second $\pi_2\ c$ is a proof that $\{\pi_1\ c \simeq \lambda\, x.\, x\}$. Its eliminator `elimId` is convertible with $\lambda\, x.\, x$.
- `IdMapping`, `imap`: `IdMapping` is a generalization of a functor where the mapping need only be defined for identity functions. A term of type $\texttt{IdMapping} \cdot F$ can be viewed as a proof that F is positive. Our elaborator produces such a proof when checking datatype positivity. `imap` maps such an identity function over F and is convertible with $\lambda\, x.\, x$.

For the generic framework, taking module parameters $(F : \star \to \star)$ and $\{im : \texttt{IdMapping} \cdot F\}$ (with curly braces indicating im is an erased parameter):

- `Fix`, `D`, `in`, `out`: resp. the type-level fixpoint function, the least fixed-point of F, and its rolling and unrolling functions.

Figure 23: Generic constructs for λ-encoded inductive types

```
import Positivity.

module GenericInd (Γ^P) (F: (Γ^I → ⋆) → (Γ^I → ⋆)) {im: IdMapping Γ^P ·F}

Fix: Π F: (Γ^I → ⋆) → (Γ^I → ⋆). IdMapping Γ^P ·F → Γ^I → ⋆ = <..>

D: Γ^I → ⋆ = Fix ·F im.
in: ∀ Γ^I. F ·D Γ^I → D Γ^I = <..>
out: ∀ Γ^I. D Γ^I → F ·D Γ^I = <..>

PrfAlg: (Π Γ^I. D Γ^I → ⋆) → ⋆
= λ P: (Π Γ^I. D Γ^I → ⋆). ∀ R: Γ^I → ⋆. ∀ c: Id ·R ·D.
  Π o: ∀ Γ^I. R Γ^I → F ·R Γ^I. ∀ oeq: {o ≃ out}.
  (∀ Γ^I. Π r: R Γ^I. P Γ^I (elimId -c Γ^I r)) →
  ∀ Γ^I. Π fr: F ·R Γ^I. P Γ^I (in Γ^I (imap -im -c Γ^I fr)).

induction: ∀ P: Π Γ^I. D Γ^I → ⋆. PrfAlg ·P → ∀ Γ^I. Π d: D Γ^I. P Γ^I d = <..>

inductionComp: ∀ P: Π Γ^I. D Γ^I → ⋆. Π alg: PrfAlg ·P. ∀ Γ^I. Π fd: F ·D Γ^I.
  {induction alg (in fd) ≃ alg out (induction alg) fd}
= Λ P. λ alg. λ fd. β.

lambek1: ∀ Γ^I. Π fd: F ·D Γ^I. {fd ≃ out (in fd)} = β.
lambek2: ∀ Γ^I. Π d: D Γ^I. {d ≃ in (out d)}
= λ d. induction ·P (Λ R. Λ c. λ o. Λ eq. λ ih. Λ Γ^I. λ fr. β) d
```

- `PrfAlg`: an inductive version of the Mendler-style CoV algebra. Its additional (erased) arguments are c, an identity function from R to `D`, and oeq, a proof that the abstract destructor o is equal to `out`. Argument c is required to be even able state the result of `PrfAlg`: that P holds of the `in` of fr coerced (using `imap`) to type $F \cdot$ `D`.
- `induction`, `inductionComp`: the generic induction principle for `D` and its computation law
- `lambek1` and `lambek2`, the proofs of *Lambek's lemma* that `in` and `out` are mutual inverses.

7.2 Elaborator Interface

There is a discrepancy between the facilities of the generic framework and the design of the surface language. In the former, a function like `minus'` (Figure 4) must be given o, the abstract `out` of type $N \to \texttt{Nat}^{\texttt{FI}} \cdot N$ (where N is the type variable bound in the `PrfAlg` given to `induction`, and $\texttt{Nat}^{\texttt{FI}}$ is the inductive signature functor of `Nat`), directly. Doing the same in the surface language has the undesirable consequence of exposing $\texttt{Nat}^{\texttt{FI}}$ to the user. Worse still, `minus'` o is not definitionally equal to `minus` = `minus' out`, and so proofs like `leDiv` (Figure 5) would require explicit use of $oeq : \{o \simeq \texttt{out}\}$, as they do when using the framework directly.

```
module DataInterface (F: ⋆ → ⋆) {im: IdMapping · F}.
import GenericInd ·F -im.

View: Π A: ⋆. Π a: A. ⋆ → ⋆ = λ A: ⋆. λ a: A. λ B: ⋆. Σ b: B. {a ≃ b}.
intrView: ∀ A: ⋆. ∀ B: ⋆. ∀ a: A. Π b: B. {b ≃ a} → View ·A a ·B
  = Λ A. Λ B. Λ x. λ y. λ eq. (y , eq).
elimView: ∀ A: ⋆. ∀ B: ⋆. Π a: A. ∀ _: View ·A a ·B. B
  = Λ A. Λ B. λ a. Λ v. φ (π_2 v) - (π_1 v) {a}.
_ : {elimView ≃ λ x. x} = β.

IsD: ⋆ → ⋆ = λ R: ⋆. (Id ·R ·D) × (View ·(D → F ·D) out ·(R → F ·R)).
isD: IsD ·D = (intrId (λ x. x) (λ _. β) , intrView -out out β).
toD: ∀ R: ⋆. ∀ _: IsD ·R. R → D = Λ R. Λ is. λ r. elimId -(π₁ is) r.
toFD: ∀ R: ⋆. ∀ _: IsD ·R. F ·R → F ·D = Λ R. Λ is. λ fd. imap -pos -(π₁ is) fd.

ByCases: (D → ⋆) → Π R: ⋆. IsD ·R → ⋆
  = λ P: D → ⋆. λ R: ⋆. λ is: IsD ·R. Π fr: F ·R. P (in (toFD -is fr)).
mu': ∀ R: ⋆. ∀ is: IsD ·R. Π r: R. ∀ P: D → ⋆. ByCases ·P ·R is → P (toD -is r)
  = Λ R. Λ is. λ r. Λ P. λ case. ρ (lambek2 (toFD -is r)) - case (elimView out -(π₁ is) r)

ByInd: (D → ⋆) → ⋆
  = λ P: D → ⋆. ∀ R: ⋆. ∀ is: IsD ·R. (Π r: R. P (toD -is r)) → ByCases ·P ·R is.
mu: Π d: D. ∀ P: D → ⋆. ByInd · P → P d
  = λ d. Λ P. λ ind. induction ·P (
    Λ R. Λ c. λ o. λ oeq. λ ih. λ fr. ind -(c , intrView -out o oeq) ih fr) d
```

Figure 24: Interface for datatype elaborator

Our solution to this discrepancy is `View` (Figure 24), a novel type that (similar to `Id`) takes advantage of our Curry-style theory. `View` $\cdot A\ a \cdot B$ is the type of proofs that a specified term a (of type A) can be "viewed" as having type B (e.g. `out` at an abstract type). It is introduced (`intrView`) by providing some b of type B and a proof $\{b \simeq a\}$. Most significant of `View` is that its eliminator (`elimView`) takes the named a and an erased `View` witness and returns the a at type B, thanks to the typing and erasure of φ (Figure 1b).

The upshot is that functions like `minus'` do not need to use `out` indirectly by taking as an argument some abstract version of it. Instead, they only require *permission to use* (in the form of a `View` witness) `out` at the abstract type; this why `minus` and `minus'` (and even the *fully separate definitions* of `pred` and `pred'`) are definitionally equal.

Figure 24 lists the remaining definitions in the interface used by our datatype elaborator:

- `IsD`, `isD`, `toD`: the generic versions of the global definitions of similar name that are defined for every declared datatype. $\texttt{IsD} \cdot R$ is pair type ($\times$): the first component (of type $\texttt{Id} \cdot R \cdot \texttt{D}$) is a proof that all terms of type R have type `D`; the second component (of type $\texttt{View} \cdot (\texttt{D} \to \texttt{F} \cdot \texttt{D})\ \texttt{out} \cdot (R \to \texttt{F} \cdot R))$ is a witness that `out` can be used at type $R \to \texttt{F} \cdot R$. `isD` is (still) the trivial witness of `IsD`. `toD` casts a term of type R to `D` given a proof $\texttt{IsD} \cdot R$; since `elimId` converts with $\lambda\, x.\, x$, so too does `toD`, *justifying the convertibility of functions* `to/D` *with* $\lambda\, x.\, x$ *in the surface language.* `toFD` is not exported to the surface language (as the datatype's signature functor is not) and uses `imap` to cast $\texttt{F} \cdot R$ to $\texttt{F} \cdot \texttt{D}$.

- `ByCases`, `mu'`: type $\texttt{ByCases} \cdot P \cdot R\ is$ is the generic type of proofs of P by case analysis. Thus, type of `mu'` says that for any term r of type R where $\texttt{IsD} \cdot R$ holds, to show P holds of r (after casting r to `D`), it suffices to give a proof by case analysis on R; its definition uses `out` at the abstract type $R \to F \cdot R$ (via `elimView`) on argument r, gives this to `case` proving (with coercions omitted) P (`in` (`out` r)), and rewrites this type with ρ by Lambek's lemma.

- $\texttt{ByInd} \cdot P$ is the type of generic proofs that P holds by induction. It is defined in terms of a proof `ByCases` additionally equipped with the inductive hypothesis and evidence of $\texttt{IsD} \cdot R$ for the quantified type R. Thus, the type of `mu` says that P holds for any d given a proof by induction on `D`; its definition uses `induction`, repackaging the assumptions available to the `PrfAlg` argument for use by argument `ind`.

There is a direct mapping between the definition of `ByInd` (and thus, in the type of `mu`) and the local definitions introduced within the body of μ-expressions in the surface language. For a μ-expression recursively defining ih, the type `Type/`ih corresponds to type variable R; `isType/`ih to argument is; and the name ih itself corresponds to the inductive hypothesis.

8 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 [3]. 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 the expected type of the expression and the sort of the variable (e.g. at the term level all types are abstracted over erased).

8.1 Representation of Datatype Definition in AST

Notation $\mathrm{Ind}[I, \Gamma_P, \Gamma_K]R\Sigma\Gamma_G$ 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:

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

where

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

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

In the above definition for Γ_G, understand that

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

- `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.

8.2 Well-formedness of Datatype Definition

For an inductive datatype definition $\mathrm{Ind}[I, \Gamma_P, \Gamma_K]R\Sigma\Gamma_G$ 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 8.5)
- The type T of each constructor $c\!:\!T \in \Sigma$ must satisfy the (non-strict) positivity condition for R (c.f. Section 8.5)
- Γ_G must bind precisely the following (these are added to the global context):
  - `Is/`I`:` $\Pi\ \Gamma_P$`.` `K` $\rightarrow$ $\star$

    The name bound here is literally the string concatenation of "`Is/`" with the user-given name for the data-type I
  - `is/`I`:` $\forall\ \Gamma_P$`.` `Is/`I Γ_P $\cdot(I\ \Gamma_P)$
  - `to/`I`:` $\forall\ \Gamma_P$`.` $\forall$ `R: K. Is/`I Γ_P `R` $\Rightarrow$ $\forall\ \Gamma_K$`.` `R` $\Gamma_K \rightarrow I\ \Gamma_P\ \Gamma_K$ `=` λ `x.x`

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

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

8.3 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.

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

8.5 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 8.11) For example, given just the (global) data type declaration of Vec, we would have $\Delta(Vec) = \mathtt{Ind}_{\mathrm{C}}[1](\Gamma_{Vec} : \Sigma =)$, 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 $\mathtt{Ind}_M[p](\Gamma_I : \Sigma =)$, 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

8.6 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 25 states when an inductive datatype definition may be introduced, provided that the following side conditions hold:

Figure 25: 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. Furthermore, 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.

8.7 Valid Elimination Kind

Figure 26: 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 26. For example, a valid elimination kind for the indexed type family $Vec \cdot X$ (which has kind $\Pi\, n\!:\!Nat.\, \star$) is $\Pi\, n\!:\!Nat.\, \Pi\, x\!:\!Vec \cdot X\ n.\, \star$

8.8 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.

8.9 Well-formed Patterns

Figure 27: 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 27 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.

8.10 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 $\Delta(I) = \texttt{Ind}_{\mathrm{C}}[p](\Gamma_I : \Sigma =)$ and I' is a fresh type variable, then we define function $Hist(\Delta, I, \overline{T}, I')$ producing an abstracted (well-formed) inductive definition $\texttt{Ind}_{\mathrm{A}}[0](\Gamma_{I'} : \Sigma' =)$, 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 8.11).

8.11 Typing Rules

Figure 28: Use of an inductive datatype $\texttt{Ind}_M[p](\Gamma_I : \Sigma =)$

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

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

The first rule of Figure 28 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 $WF\text{-}Pat$ 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 $Hist(\Delta, I, \overline{T}, I')$ 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'.

9 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 ?? plus rules for dependent intersections (see [Stu18a]). That is to say, the preceding treatment for inductive datatypes (Section 8) 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 $\texttt{Ind}_I[p](\Gamma_I : \Sigma =)$

9.1 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 $\texttt{elimId}_{\texttt{D}}$:

$$\texttt{Id}_{\texttt{D}}\ :\ \Pi\ \texttt{A B:}\ (\Pi\ \Gamma_{\texttt{D}}.\ \star).\ \iota\ \texttt{id:}\ \forall\ \Gamma_{\texttt{D}}.\ \texttt{A}\ \Gamma_{\texttt{D}} \to \texttt{B}\ \Gamma_{\texttt{D}}.\ \{\texttt{id} \simeq \lambda\ \texttt{x. x}\}.$$
$$\texttt{elimId}_{\texttt{D}}\ :\ \forall\ \texttt{A B:}\ (\Gamma_{\texttt{D}}.\ \star).\ \texttt{Id}_{\texttt{D}}\cdot\texttt{A}\cdot\texttt{B} \Rightarrow \texttt{A} \to \texttt{B} = \texttt{<..>}$$

Recall that since Cedilleum has a Curry-style type system and implicit products there are many nontrivial 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:

$$\texttt{elimId}_{\texttt{D}}\texttt{-prop}\ :\ \{\texttt{elimId}_{\texttt{D}} \simeq \lambda\ \texttt{x. x}\} = \beta.$$

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

$$\texttt{IdMapping}_{\texttt{D}}\ :\ \Pi\ \texttt{F:}\ (\Gamma_{\texttt{D}} \to \star) \to (\Gamma_{\texttt{D}} \to \star).\ \star$$
$$=\ \lambda\ \texttt{F.}\ \forall\ \texttt{A B:}\ (\Gamma_{\texttt{D}} \to \star).\ \overset{\Pi}{\forall}\ \Gamma_{\texttt{D}}.\ \texttt{Id}_{\texttt{D}}\cdot\texttt{A}\cdot\texttt{B} \to \texttt{Id}_{\texttt{D}}\cdot(\texttt{F}\cdot\texttt{A})\cdot(\texttt{F}\cdot\texttt{B}).$$

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 λ `x. x`.

9.2 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.
```

9.3 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).

9.4 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 Π $\Gamma_\texttt{p}$. Π $\Gamma_\texttt{D}$. ⋆, where $\Gamma_\texttt{p}$ are the parameters and $\Gamma_\texttt{D}$ the indices, it follows that the kind of IF is Π $\Gamma_\texttt{p}$. Π R: (Π $\Gamma_\texttt{D}$. ⋆). (Π $\Gamma_\texttt{D}$. ⋆). 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 λ $\Gamma_\texttt{p}$. 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:

$$\texttt{VecF} : \Pi\ \texttt{A}{:}\ \star.\ (\texttt{Nat} \to \star) \to \texttt{Nat} \to \star$$
$$= \lambda\ \texttt{A R n}.\ \forall\ \texttt{X}{:}\ \texttt{Nat} \to \star.\ \texttt{X zero} \to (\forall\ \texttt{n}{:}\ \texttt{Nat}.\ \texttt{A} \to \texttt{R n} \to \texttt{X (succ n)}) \to \texttt{X n}.$$

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:

$$\texttt{indIF} : \forall\ \Gamma_{\texttt{p}}.\ \forall\ \texttt{R}{:}\ (\Pi\ \Gamma_{\texttt{D}}.\ \star).\ \forall\ \Gamma_{\texttt{D}}.\ \Pi\ \texttt{fr}{:}\ \texttt{IF}\ \Gamma_{\texttt{p}}\ \cdot\texttt{R}\ \Gamma_{\texttt{D}}.\ \forall\ \texttt{P}{:}\ (\Pi\ \Gamma_{\texttt{D}}.\ \texttt{IF}\ \Gamma_{\texttt{p}}\ \cdot\texttt{R}\ \Gamma_{\texttt{D}} \to \star)$$
$$(\Pi\ \texttt{p}_{\texttt{j}}{:}\ \overset{\Pi}{\forall}\ \Gamma^{\texttt{R}}{}_{\texttt{j}}.\ \texttt{P}\ (\texttt{c}_{\texttt{j}}\ \Gamma^{\texttt{R}}{}_{\texttt{j}}))_{j=1..n}.\ \texttt{P}\ \Gamma_{\texttt{D}}\ \texttt{fr} = \texttt{<..>}$$

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:

$$\texttt{F} : \Pi\ \Gamma_{\texttt{p}}.\ (\Pi\ \Gamma_{\texttt{D}}.\ \star) \to \Pi\ \Gamma_{\texttt{D}}.\ \star$$
$$= \lambda\ \Gamma_{\texttt{p}}\ \texttt{R}\ \Gamma_{\texttt{D}}.\ \forall\ \texttt{X}{:}\ (\Pi\ \Gamma_{\texttt{D}}.\ \star).\ (\Pi\ \texttt{c}_{\texttt{j}}{:}\ (\overset{\Pi}{\forall}\ \Gamma^{\texttt{R}}{}_{\texttt{j}}.\ \texttt{X}\ \overline{\texttt{t}}_{\texttt{j}}))_{j=1-n}.\ \texttt{X}\ \Gamma_{\texttt{D}}$$

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

$$\texttt{idmap} : \forall\ \Gamma_{\texttt{p}}.\ \texttt{IdMapping}_{\texttt{D}}\ \cdot(\texttt{F}\ \Gamma_{\texttt{p}})$$
$$= \Lambda\ \Gamma_{\texttt{p}}.\ \Lambda\ \texttt{R1}.\ \Lambda\ \texttt{R2}.\ \Lambda\ \texttt{id}.\ \bullet$$

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 $\texttt{fr1}{:}\ \texttt{F}\ \cdot\Gamma_{\texttt{p}}\ \texttt{R1}\ \cdot\Gamma_{\texttt{D}}$

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

where $\bullet_1{:}\ \texttt{F}\ \cdot\Gamma_{\texttt{p}}\ \texttt{R2}\ \cdot\Gamma_{\texttt{D}}$ and $\bullet_2{:}\ \{\lambda\ \texttt{fr1}.\ \bullet_1 \simeq \lambda\ \texttt{x}.\ \texttt{x}\}$. As the only (non-hole) refinements we will make to $\bullet_1$ are converting terms to η-long form and applying $\texttt{elimId}_{\texttt{D}}\ \texttt{-id}$ 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 $\bullet{:}\ \texttt{F}\ \cdot\Gamma_{\texttt{p}}\ \texttt{R1}\ \cdot\Gamma_{\texttt{D}} \to (\Pi\ \texttt{c}_{\texttt{j}}{:}\ (\overset{\Pi}{\forall}\ \Gamma^{\texttt{R2}}{}_{\texttt{j}}.\ \texttt{X}\ \overline{\texttt{t}}_{\texttt{j}}))_{j=1-n} \to \texttt{X}\ \Gamma_{\texttt{D}}$.

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

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

where now $\bullet_{\texttt{j}}{:}\ (\overset{\Pi}{\forall}\ \Gamma^{\texttt{R2}}{}_{\texttt{j}}.\ \texttt{X}\ \overline{\texttt{t}}_{\texttt{j}}) \to \overset{\Pi}{\forall}\ \Gamma^{\texttt{R1}}{}_{\texttt{j}}.\ \texttt{X}\ \overline{\texttt{t}}_{\texttt{j}}$. 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 $\bullet_{\texttt{k}}{:}\ \Gamma^{\texttt{R1}}(\texttt{y}_{\texttt{k}}) \to \Gamma^{\texttt{R2}}{}_{\texttt{k}}(\texttt{y}_{\texttt{k}})$ (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 $\bullet$ `y` where `y: S` is some sub-data to an (abstract) constructor with recursive occurence type `R1` passing the positivity checker. (The expression $\bullet$ `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 `unit:` $\forall$ `X:` $\star$`. X` $\to$ `X =` Λ `X.` λ `x. x` and finish.

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

This means S has the shape $\overset{\Pi}{\forall}$ $\Gamma^{\mathtt{R1}}{}_{x}$`. 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 $\bullet$ `y` η-long and refine the expression to $\|\Gamma_x^{R1}\|$ holes $\bullet_{\mathtt{j}}$ such that the expression is now

$\overset{\lambda}{\Lambda}$ $\Gamma^{\mathtt{R2}}{}_{x}$. $\bullet$ `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 `T = R1` $\Gamma_{\mathtt{D}}$ (handled below); it has type `S` $\to$ $\overset{\Pi}{\forall}$ $\Gamma^{\mathtt{R1}}{}_{x}$`. [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 λ `y.` λ $\mathtt{x_j}$`. elimId -c (y` $\mathtt{x_j}$`)` depending on whether `T = R1` $\Gamma_{\mathtt{D}}$

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

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

where $\bullet_{\mathtt{j}}$: $\Gamma^{\mathtt{R1}}{}_{\mathtt{y}}(\mathtt{y_j}) \to \Gamma^{\mathtt{R2}}{}_{\mathtt{y}}(\mathtt{y_j})$. Perform the steps outlined by Section A.2 to fill in each $\bullet_{\mathtt{j}}$ producing from $\bullet_{\mathtt{j}}$ $\mathtt{y_j}$ 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.