# Understanding eliminators

Davide Peressoni

Eliminators are undoubtedly the most difficult aspect of type theory. In this article we will try to make more clear how they work, but most importantly why we need them. This articles will present some analogies between eliminators and constructs of programming languages (such Python, Rust, Scala, …) which can help who knows some basic of computer programming understanding eliminators. Agda will be used to implement all eliminators, since it is a language (and proof assistant) built on top of Martin-Löf's type theory. Don't worry if you don't know how to develop: the only prerequisite is to know how to define a math function by cases, which all of us studied at school.

## Type theory

Before starting our journey, let's do a brief recap of what is type theory. Type theory is a math field which aims to:

- Solve some issues in lambda-calculus (a computation model) by adding types to variables
- Provide a small set of axioms to prove foundations of mathematics: the less we postulate, the more we can prove, the more we can trust

For this last point, type theory is used as a basis for many proof assistants. A proof assistant (such Agda, Coq, Idris, Isabelle, …) is both a programming language (usually functional) and an environment to help producing and validating mathematical proofs and algorithms correctness ones.

## Eliminators purpose

Before talking about what is an eliminator and how it works, let's think about why we need eliminators. Let's take an overview of which are the instruments type theory give us.

- **Formation rule**
  With the formation rule we introduce a type: we state that a type exists, under a certain context. The context is useful to fix preconditions, hypothesis on the variables used in the definition of the type. Even if context is directly useful only for dependent types, we always put a generic context,

also in formation rules of non-dependent types: this way we can always restrict the context and chain rules.

$$\text{F-}\mathsf{Unit})\ \frac{\Gamma \text{ context}}{\mathsf{Unit} \text{ type } [\Gamma]}$$

The above says that given a generic context $\Gamma$, $\mathsf{Unit}$ is a type. For who has not seen this notation before, you can read the "fraction" as an implication: above the are the premises and below the consequences. On the left we have a superfluous label.

$$\text{F-}\mathsf{List})\ \frac{A \text{ type } [\Gamma]}{\mathsf{List}(A) \text{ type } [\Gamma]}$$

Here we see a dependent type definition: for each type $A$ under context $\Gamma$, $\mathsf{List}(A)$ is a type, under the same context. As we can see here we have a precondition.

- **Introduction rules**
  An empty type is not very useful (except the *empty type* itself), so we need to introduce members of types. These constructors are defined in introduction rules, which simply state that, under a context, a certain object is member of a certain type.

  $$\text{I-}\mathsf{Unit})\ \frac{\Gamma \text{ context}}{\mathrm{tt} \in \mathsf{Unit} \ [\Gamma]}$$

  The unique constructor of the unit type is tt. Its introduction rule tells us that given a generic context $\Gamma$, tt is an element of the unit type.

With those two tools we can create variables of a certain type, but we cannot use them to do computations. We are missing the basic operations. If we were using classical natural numbers we could operate on them with their basic operations (i.e. addition, multiplication, …) and easily write functions, e.g. $\lambda(a, b).a + b$.[1] But we don't have such operators defined for our types in type theory.

The objective of eliminators is allowing us to define functions to do computations with the variables of our types. More precisely eliminators are functions which take a variable of some type and give us a result, based on the value of the variable we gave to it. We can so compare our eliminator as a `switch` (in C, C++, C#, Java, …), `match` (in Python, Rust, Scala, …) or `case` (in Haskell, Agda, …).[2] Indeed defining a function using eliminators will be like defining a function by cases.

[1] For those not adept to lambda calculus $\lambda(a, b).a + b$ (or more formaly $\lambda(a).\lambda(b).a + b$) is a definition of a function which takes two parameters ($a$ and $b$) and returns their sum. It is somewhat equivalent to $(a, b) \mapsto a + b$, or $f(a, b) = a + b$.

[2] As we will see `match` from Rust and Scala, and `case` from Haskell and Agda are indeed an accurate implementation of eliminators.

## How eliminators work

As we said before, eliminators allow us to define functions with variables of our type, defining them by cases. In general terms an eliminator has $n+1$ parameters, where $n$ is the number of constructors of the type. The first parameter is the variable we want to match to, the others are the values we would like to return if we match a certain constructor.

$$El(x, y_1, y_2, \dots, y_n) = \begin{cases} y_1 & \text{if } x = \text{constuctor 1} \\ y_2 & \text{if } x = \text{constuctor 2} \\ \dots & \\ y_n & \text{if } x = \text{constuctor } n \end{cases}$$

Let's now see some examples to understand better

### Singleton

As said before, the singleton, or unit type, has only one constructor (tt). So singleton eliminator will have only two arguments: the variable to match, and wath we will return if we match that constructor.

$$El_{\mathsf{Unit}}(x, y) = \begin{cases} y & \text{if } x = \text{tt} \end{cases}$$

We can write that in Python as

```
class Unit(Enum):
    tt = 1

    def eliminator(self, y):
        match self:
            case Unit.tt:
                return y
```

We have defined the unit type as an enumeration with only one value (`tt`). Then we have added the `eliminator` method, which matches on `self` for all possible constructor (in this case only `tt`) and returns the appropriate value (`y`).

Obviously the variable $y$ could be of whatever type, this is highlighted by the usage of generics in Rust:[3]

[3] Rust already have a singleton type `()`, whose only value is `()`. We declared an enum to see better how the eliminator behaves.

```
enum Unit {
    Tt
}

impl Unit {
    fn eliminator<T>(self, y: T) -> T {
        match self {
            Unit::Tt => y
        }
    }
}
```

Paying attention at the definition of the eliminator we can see the type $T$ of the variable $y$ could be a dependent type on the variable we are matching on.

$$\text{E-Unit)}\ \frac{T(z)\ \text{type}\ [\Gamma,\ z \in \mathsf{Unit}] \qquad x \in \mathsf{Unit}\ [\Gamma] \qquad y \in T(\text{tt})\ [\Gamma]}{El_{\mathsf{Unit}}(x, y) \in T(x)\ [\Gamma]}$$

This dependency could potentially be useful, in types with multiple constructors, to have multiple return types based on the input.

Python and Rust do not support dependent types, so here we will see our eliminator implemented in Agda:

```
data Unit : Set where
  tt : Unit

ElUnit : {T : Unit → Set} → (x : Unit) → T tt → T x
ElUnit tt y = y
```

Here we clearly state that our generic type `T` is a dependent type on a value of type `Unit` (recall in the context we have fixed $z \in \mathsf{Unit}$). `y` is of type $T(\text{tt})$, and the return value of type $T(x)$. In this case this is not important since the unique value of type `Unit` is `tt`, but we will later see how this is still useful when chaining eliminators together.

We can rewrite the Agda code to compare it better with the eliminator and conversion rules.[4]

```
ElUnit : {T : (z : Unit) → Set} →  -- T(z) type [z   Unit]
         (x : Unit) →              -- x   Unit
         (y : T tt) →              -- y   T(tt)
         T x                       -- ElUnit(x,y)   T(x)
ElUnit tt y = y                    -- ElUnit(tt,y) = y
```

$$\text{C-Unit)}\ \frac{T(z)\ \text{type}\ [\Gamma,\ z \in \mathsf{Unit}] \qquad y \in T(\text{tt})\ [\Gamma]}{El_{\mathsf{Unit}}(\text{tt}, y) = y \in T(\text{tt})\ [\Gamma]}$$

[4]Eliminators are defined by two rules: the eliminator rule declare the return type of the eliminator, the conversion rules instead define the actual case split.

This example is great to start, but it is not really interesting because splitting on a single constructor makes no practical sense. So let's add some fun trying with two constructors.

### Booleans

The bool type represent truth values (`true` and `false`). It's not a canonical type, since it could be constructed from canonical types[5], but we will use it for its simplicity.

$$\text{F-Bool)}\ \frac{\Gamma \text{ context}}{\mathsf{Bool} \text{ type } [\Gamma]}$$

$$\text{I}_1\text{-Bool)}\ \frac{\Gamma \text{ context}}{\text{true} \in \mathsf{Bool}\ [\Gamma]}$$

$$\text{I}_2\text{-Bool)}\ \frac{\Gamma \text{ context}}{\text{false} \in \mathsf{Bool}\ [\Gamma]}$$

The eliminator, intuitively, would be:

$$El_{\mathsf{Bool}}(x, y_1, y_2) = \begin{cases} y_1 & \text{if } x = \text{true} \\ \\ y_2 & \text{if } x = \text{false} \end{cases}$$

This seem more interesting than the one of singleton. Computer scientists could also have recognized how this eliminator in no more less than the `if` construct.

```
function ElBool(x, y1, y2)
  if (x)       // x is true
    return y1
  else         // x is false
    return y2
```

Before implementing the eliminator, let's give a formal definition of it:

$$\text{E-Bool)}\ \frac{\begin{array}{cc} T(z) \text{ type } [\Gamma,\, z \in \mathsf{Bool}] & x \in \mathsf{Bool}\ [\Gamma] \\ y_1 \in T(\text{true})\ [\Gamma] & y_2 \in T(\text{false})\ [\Gamma] \end{array}}{El_{\mathsf{Bool}}(x, y_1, y_2) \in T(x)\ [\Gamma]}$$

$$\text{C}_1\text{-Bool)}\ \frac{T(z) \text{ type } [\Gamma,\, z \in \mathsf{Bool}] \qquad y_1 \in T(\text{true})\ [\Gamma] \qquad y_2 \in T(\text{false})\ [\Gamma]}{El_{\mathsf{Bool}}(\text{true}, y_1, y_2) = y_1 \in T(\text{true})\ [\Gamma]}$$

[5] In facts the type $\mathsf{Unit} + \mathsf{Unit}$ (disjoint sum of two units) is used to represent boolean data.

$$\text{C}_2\text{-Bool})\ \frac{T(z)\ \text{type}\ [\Gamma,\ z \in \mathsf{Bool}] \qquad y_1 \in T(\text{true})\ [\Gamma] \qquad y_2 \in T(\text{false})\ [\Gamma]}{El_{\mathsf{Bool}}(\text{false}, y_1, y_2) = y_2 \in T(\text{false})\ [\Gamma]}$$

As we can see, this time the resulting type can be really different, based on the input: if $x = \text{true}$, the result would be of type $T(\text{true})$, instead with $x = \text{false}$ the result type would be $T(\text{false})$.

```
Ttrue = TypeVar('Ttrue')
Tfalse = TypeVar('Tfalse')

class Bool(Enum):
    true = 1
    false = 2

    #              Bool      T(true)    T(false)   T(x)
    def eliminator(self, y1: Ttrue, y2: Tfalse) -> Union[Ttrue, Tfalse]:
        match self:
            case Bool.true:
                return y1
            case Bool.false:
                return y2
```

Python does not have dependent types: we declared two generics (`Ttrue` and `Tfalse`) and we declared a return type of `Union[Ttrue, Tfalse]`, i.e. one of the two types. This simulates well our dependent type $T(x)$.

```
data Bool : Set where
  true  : Bool
  false : Bool

ElBool : {T  : (z : Bool) → Set} →  -- T(z) type [z  Bool]
         (x  : Bool) →              -- x  Bool
         (y1 : T true) →            -- y1  T(true)
         (y2 : T false) →           -- y2  T(false)
         T x                        -- ElBool(x,y1,y2)  T(x)
ElBool true  y1 y2 = y1             -- ElBool(true,y1,y2) = y1
ElBool false y1 y2 = y2             -- ElBool(false,y1,y2) = y2
```

With Agda dependent types, we can better understand how the bool eliminator works. In fact we can predict the output type from the value of the parameter $x$.

### Naturals

It would be impossible to have a constructor for each natural number: they are infinite. Instead we define natural numbers recursively.

$$\text{F-Nat})\ \frac{\Gamma\ \text{context}}{\mathsf{Nat}\ \text{type}\ [\Gamma]}$$

$$\mathrm{I}_1\text{-}\mathsf{Nat})\ \frac{\Gamma \text{ context}}{\text{zero} \in \mathsf{Nat}\ [\Gamma]}$$

$$\mathrm{I}_2\text{-}\mathsf{Nat})\ \frac{m \in \mathsf{Nat}\ [\Gamma]}{\text{suc}(m) \in \mathsf{Nat}\ [\Gamma]}$$

We have a base constructor (zero), which represents the number 0, and then a constructor (suc) which transforms a natural into its successor. This constructors take a natural number and returns a new natural.

Following the same reasoning of above the eliminator would be:

$$El_{\mathsf{Nat}}(n, y_1, y_2) = \begin{cases} y_1 & \text{if } n = \text{zero} \\ \\ y_2 & \text{if } n = \text{suc}(m) \end{cases}$$

But such an eliminator wouldn't be useful. For example how are we supposed to define the sum between two numbers using it?

$$\text{sum}(a, b) := El_{\mathsf{Nat}}(a, ?_1, ?_2) = \begin{cases} ?_1 & \text{if } a = \text{zero} \\ \\ ?_2 & \text{if } a = \text{suc}(a') \end{cases}$$

The first hole could be filled with $b$ (since $a$ is zero and $0 + b = b$), but what can we put in the second hole? Here $a = \text{suc}(a') = 1 + a'$ and so we have to return $a + b = (1 + a') + b = 1 + (a' + b) = \text{suc}(a' + b) = \text{suc}(\text{sum}(a', b))$, but unfortunately we cannot obtain $a'$:

$$\text{sum}(a, b) := El_{\mathsf{Nat}}(a, b, \text{suc}(\text{sum}(a', b)))$$

We can easily fix it by substituting the parameter $y_2$: instead to be a variable we can make it a function. This function takes a natural ($m$) and returns what would have been $y_2$:

$$El_{\mathsf{Nat}}(n, y, f) = \begin{cases} y & \text{if } n = \text{zero} \\ \\ f(m) & \text{if } n = \text{suc}(m) \end{cases}$$

$$\text{sum}(a, b) := El_{\mathsf{Nat}}(a, b, \lambda(a').\text{suc}(\text{sum}(a', b)))$$

We didn't talk about the recursive call to the sum function: sum is defined with a call to sum itself. This makes sum a recursive function. It is guaranteed to terminate since each time a recursive call is done one parameter (in this case the first) becomes smaller (in this case we remove a suc wrapping, or equivalently

we decrease it by 1), finally reaching the base constructor (which implies being in the base case of the function).

In type theory we often use recursion to define types, and thus to define functions to operate on them. Hence the canonical eliminators already give us the result of this recursive call, simplifying the definition of other functions:

$$El_{\mathsf{Nat}}(n, y, e) = \begin{cases} y & \text{if } n = \text{zero} \\ e(m, El_{\mathsf{Nat}}(m, y, e)) & \text{if } n = \text{suc}(m) \end{cases}$$

As we can see, this time the function we pass takes not only the unwrapped value ($m$), but also the result of the recursive call of the eliminator. We can so simplify the definition of the sum function as:

$$\text{sum}(a, b) := El_{\mathsf{Nat}}(a, b, \underbrace{\lambda(a', s).\text{suc}(s)}_{e})$$

Recall that $s = El_{\mathsf{Nat}}(a', b, e) = \text{sum}(a', b)$.

In Scala we can write this eliminator as:

```
enum Nat:
  case Zero
  case Suc (m: Nat)

  def eliminator [T] (y: T, e: (Nat, T) => T): T = this match
    case Zero    => y
    case Suc (m) => e (m, m.eliminator (y, e))

  def + (that: Nat): Nat =
    this.eliminator (
      // 0 + that = that
      that,
      // (1 + m) + that = 1 + (m + that)
      (m, m_plus_that) => Nat.Suc (m_plus_that),
    )
```

We are missing the most important thing: the type. Nonetheless we are talking about type theory.

$$\text{E-Nat)}\ \frac{\begin{array}{c} T(z)\ \text{type}\ [\Gamma,\ z \in \mathsf{Nat}] \qquad n \in \mathsf{Nat}\ [\Gamma] \\ y \in T(\text{zero})\ [\Gamma] \qquad e(m, x) \in T(\text{suc}(m))\ [\Gamma,\ m \in \mathsf{Nat},\ x \in T(m)] \end{array}}{El_{\mathsf{Nat}}(n, y, e) \in T(n)\ [\Gamma]}$$

$$\text{C}_1\text{-Nat})\ \frac{T(z)\ \text{type}\ [\Gamma,\ z \in \mathsf{Nat}] \qquad y \in T(\text{zero})\ [\Gamma] \qquad e(m,x) \in T(\text{suc}(m))\ [\Gamma,\ m \in \mathsf{Nat},\ x \in T(m)]}{El_{\mathsf{Nat}}(\text{zero}, y, e) = y \in T(\text{zero})\ [\Gamma]}$$

$$\text{C}_2\text{-Nat})\ \frac{T(z)\ \text{type}\ [\Gamma,\ z \in \mathsf{Nat}] \qquad y \in T(\text{zero})\ [\Gamma] \qquad m \in \mathsf{Nat}\ [\Gamma] \qquad e(m,x) \in T(\text{suc}(m))\ [\Gamma,\ m \in \mathsf{Nat},\ x \in T(m)]}{El_{\mathsf{Nat}}(\text{suc}(m), y, e) = e(m, El_{\mathsf{Nat}}(m, y, e)) \in T(\text{suc}(m))\ [\Gamma]}$$

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat

ElNat : {T : (z : Nat) → Set} →                    -- T(z) type [z  Nat]
        (n : Nat) →                                -- n  Nat
        (y : T zero) →                             -- y  T(zero)
        (e : (m : Nat) → (x : T m) → T (suc m)) →  -- e(m,x)  T(suc(m)) [m  Nat, x  T(m)]
        T n                                        -- ElNat(n,y,e)  T(n)
ElNat zero    y _ = y                    -- ElNat(zero,y,e) = y
ElNat (suc m) y e = e m (ElNat m y e)    -- ElNat(suc(m),y,e) = e(m, ElNat(m, y, e))

--     a     b    a+b
--     ↓     ↓     ↓
sum : Nat → Nat → Nat
sum a b = ElNat a b (  a' a'+b → suc a'+b)
--   value to     ↑    ↑    ↑       return value if a = suc a'
--  be splitted   |    |   sum-El a' b
--                |   a = succ a'
--     return value if a is zero

-- T n = Nat, in fact we can make it explicit as:
-- sum a b = ElNat a { _ → Nat} b (  a' a'+b → succ a'+b)
```

As we said in the introduction some languages such Rust, Scala, Haskell, Agda, … natively support eliminators:

```
  def + (that: Nat): Nat = this match
    case Zero    => that
    case Suc (m) => Nat.Suc (m + that)
```

```
--     a     b    a+b
--     ↓     ↓     ↓
sum : Nat → Nat → Nat
sum zero     b = b                -- return value if a is zero
sum (suc a') b = suc (sum a' b)   -- return value if a = suc a'
```

### Lists

Lists are similar to natural numbers: we have the empty list (*nil*) and the *cons*, i.e. the addition of an element to a list. The added difficulty is that we have to track the type of the elements of the list.

$$\text{F-List)}\ \frac{A\ \text{type}\ [\Gamma]}{\mathsf{List}(A)\ \text{type}\ [\Gamma]}$$

$$\text{I}_1\text{-List)}\ \frac{\mathsf{List}(A)\ \text{type}\ [\Gamma]}{\text{nil} \in \mathsf{List}(A)\ [\Gamma]}$$

$$\text{I}_2\text{-List)}\ \frac{s \in \mathsf{List}(A)\ [\Gamma] \qquad a \in A\ [\Gamma]}{\text{cons}(s,a) \in \mathsf{List}(A)\ [\Gamma]}$$

Also the eliminator is similar to that of naturals number, but the passed function, other than the recursive call, takes two unwrapped parameters. These are the prefix list ($s$), called *head*, and the tail element ($a$):

$$El_{\mathsf{List}}(x,y,e) = \begin{cases} y & \text{if } x = \text{nil} \\ \\ e(s,a,El_{\mathsf{List}}(s,y,e)) & \text{if } x = \text{cons}(s,a) \end{cases}$$

Don't forget the types!

$$\text{E-List)}\ \frac{T(z)\ \text{type}\ [\Gamma,\ z \in \mathsf{List}(A)] \qquad x \in \mathsf{List}(A)\ [\Gamma] \qquad y \in T(\text{nil})\ [\Gamma] \qquad e(s,w,t) \in T(\text{cons}(s,w))\ [\Gamma,\ s \in \mathsf{List}(A),\ w \in A,\ t \in T(s)]}{El_{\mathsf{List}}(x,y,e) \in T(x)\ [\Gamma]}$$

$$\text{C}_1\text{-List)}\ \frac{T(z)\ \text{type}\ [\Gamma,\ z \in \mathsf{List}(A)] \qquad y \in T(\text{nil})\ [\Gamma] \qquad e(s,w,t) \in T(\text{cons}(s,w))\ [\Gamma,\ s \in \mathsf{List}(A),\ w \in A,\ t \in T(s)]}{El_{\mathsf{List}}(\text{nil},y,e) = y \in T(\text{nil})\ [\Gamma]}$$

$$\text{C}_2\text{-List)}\ \frac{T(z)\ \text{type}\ [\Gamma,\ z \in \mathsf{List}(A)] \qquad y \in T(\text{nil})\ [\Gamma] \qquad s \in \mathsf{List}(A)\ [\Gamma] \qquad a \in A\ [\Gamma] \qquad e(s,w,t) \in T(\text{cons}(s,w))\ [\Gamma,\ s \in \mathsf{List}(A),\ w \in A,\ t \in T(s)]}{El_{\mathsf{List}}(\text{cons}(s,a),y,e) = e(s,a,El_{\mathsf{List}}(s,y,e)) \in T(\text{cons}(s,a))\ [\Gamma]}$$

```
data List (A : Set) : Set where
  nil  : List A
  cons : List A → A → List A

ElList : {A : Set}
         {T : List A → Set} →                    -- T(z) type [z  List(A)]
         (x : List A) →                          -- x  List(A)
         (y : T nil) →                           -- y  T(nil)
                                                 -- e(s,w,t)  T(cons(s,w)) [s  List(A), w  A,
```

```
        (e : (s : List A) → (w : A) → (t : T s) → T (cons s w)) →
        T x                                        -- ElList(x,y,e)   T(x)
ElList nil         y _ = y                         -- ElList(nil,c,e) = y
ElList (cons s a) y e = e s a (ElList s y e)       -- ElList(cons(s,a),y,e) = e(s,a,ElList(s,y,e
```

### Disjoint sum

The disjoint sum type (known also as *Either*) allow us to represent values of two different types, taking note of the type of the current value. For example a variable of type $\mathsf{Unit} + \mathsf{Nat}$ could store a unit or a natural, and it knows which one is stored.[6]

$$\text{F-+)}\ \frac{B \text{ type } [\Gamma] \qquad C \text{ type } [\Gamma]}{B + C \text{ type } [\Gamma]}$$

$$\text{I}_1\text{-+)}\ \frac{B \text{ type } [\Gamma] \qquad C \text{ type } [\Gamma] \qquad b \in B\ [\Gamma]}{\text{inl}(b) \in B + C \text{ type } [\Gamma]}$$

$$\text{I}_2\text{-+)}\ \frac{B \text{ type } [\Gamma] \qquad C \text{ type } [\Gamma] \qquad c \in C\ [\Gamma]}{\text{inr}(c) \in B + C \text{ type } [\Gamma]}$$

As we can see we have two constructors: one for each type. So the eliminator will take two functions: one unwrapping the left type, the other unwrapping the right type.

$$El_+(x, e_1, e_2) = \begin{cases} e_1(x_1) & \text{if } x = \text{inl}(x_1) \\ \\ e_2(x_2) & \text{if } x = \text{inr}(x_2) \end{cases}$$

As you may have noticed here we don't have the recursive call. This because the disjoint sum type is not recursively defined, and so the functions which operate on its values will not be recursive.

$$\text{E-+)}\ \frac{\begin{array}{cc} T(z) \text{ type } [\Gamma,\, z \in B + C] & x \in B + C\ [\Gamma] \\ e_1(x_1) \in T(\text{inl}(x_1))\ [\Gamma,\, x_1 \in B] & e_2(x_2) \in T(\text{inr}(x_2))\ [\Gamma,\, x_2 \in C] \end{array}}{El_+(x, e_1, e_2) \in T(x)\ [\Gamma]}$$

$$\text{C}_1\text{-+)}\ \frac{\begin{array}{cc} T(z) \text{ type } [\Gamma,\, z \in B + C] & b \in B\ [\Gamma] \\ e_1(x_1) \in T(\text{inl}(x_1))\ [\Gamma,\, x_1 \in B] & e_2(x_2) \in T(\text{inr}(x_2))\ [\Gamma,\, x_2 \in C] \end{array}}{El_+(\text{inl}(b), e_1, e_2) = e_1(b) \in T(\text{inl}(b))\ [\Gamma]}$$

[6] In some programming languages such Haskell, Rust, … the disjoint type is implemented with a tagged union. In other languages (such C and derived) you can simulate this with and (untagged) union, but here you have to manually take note of the type of the current value.

$$\text{C}_2\text{-}+)\ \frac{\begin{array}{cc} T(z)\ \text{type}\ [\Gamma,\ z \in B + C] & c \in C\ [\Gamma] \\ e_1(x_1) \in T(\text{inl}(x_1))\ [\Gamma,\ x_1 \in B] & e_2(x_2) \in T(\text{inr}(x_2))\ [\Gamma,\ x_2 \in C] \end{array}}{El_+(\text{inr}(c), e_1, e_2) = e_2(c) \in T(\text{inr}(c))\ [\Gamma]}$$

```
data _+_ (B C : Set) : Set where
  inl : B → B + C
  inr : C → B + C

El+ : {B C : Set}
      {T : B + C → Set} →              -- T(z) type [z   B+C]
      (x : B + C) →                    -- x   B+C
      (e  : (x  : B) → T (inl x )) → -- e (x )   T(inl(x )) [x   B]
      (e  : (x  : C) → T (inr x )) → -- e (x )   T(inr(x )) [x   C]
      T x                              -- El+(x,e ,e )   T(x)
El+ (inl b) e  _   = e  b             -- El+(inl(b),e ,e ) = e (b)
El+ (inr c) _  e  = e  c              -- El+(inr(c),e ,e ) = e (c)
```

A naïve example of how to apply this eliminator to define a function is the following. It mirrors a disjoint sum: it maps the left element to the right and viceversa. Obviously the input and output types are different, even if are both disjoint sums of the same two types, but in different order.

$$\text{swap}(x) = \begin{cases} \text{inr}(x_1) & \text{if } x = \text{inl}(x_1) \\ \text{inl}(x_2) & \text{if } x = \text{inr}(x_2) \end{cases} = El_+(x, \lambda(x_1).\text{inr}(x_1), \lambda(x_2).\text{inl}(x_2))$$

```
swap : {B C : Set} → B + C → C + B
swap x = El+ x inr inl
```

### Empty type

The empty type is a type with no inhabitants. It represents the logic value of *false*, and as from false we can derive anything, also from values of the empty type (which don't exist) we can derive anything.

$$\text{F-Empty})\ \frac{\Gamma\ \text{context}}{\mathsf{Empty}\ \text{type}\ [\Gamma]}$$

$$\text{E-Empty})\ \frac{T(z)\ \text{type}\ [\Gamma,\ z \in \mathsf{Empty}] \qquad x \in \mathsf{Empty}\ [\Gamma]}{El_{\mathsf{Empty}}(x) \in T(x)\ [\Gamma]}$$

```
data Empty : Set where

ElEmpty : {T : (z : Empty) → Set} →  -- T(z) type [z   Empty]
          (x : Empty) →              -- x   Empty
          T x                        -- ElEmpty(x)   T(x)
ElEmpty ()
```

Since there are no values of the empty type, there is nothing to match, and so we don't have any conversion rule.

$$El_{\mathsf{Empty}}(x) = \{$$

Side note: only few rustaceans know the Rust never type `!`, which is in fact an empty type. It is usually placed as a type for functions which don't terminate.

```
fn eliminator<T>(x: !) -> T {
    match x {
    }
}
```

### Propositional equality

The type of propositional equality states if two values are the same. If the values are not equal it is like the empty type. If instead the values are equal it is like a singleton with a unique element: its only constructor. This constructor represent a proof of the equality.

$$\text{F-}\mathsf{Id})\ \frac{A \text{ type } [\Gamma] \qquad a \in A\ [\Gamma] \qquad b \in A\ [\Gamma]}{\mathsf{Id}(A, a, b) \text{ type } [\Gamma]}$$

$$\text{I-}\mathsf{Id})\ \frac{a \in A\ [\Gamma]}{\mathrm{id}(a) \in \mathsf{Id}(A, a, a)\ [\Gamma]}$$

As we can see $\mathrm{id}(a)$ is the only constructor of the propositional equality type between $a$ and $a$ itself, for other equality types there is no constructor. So the eliminator would be as easy as the singleton and empty type ones.

$$El_{\mathsf{Id}}(x, y) = \Big\{ y \quad \text{if } x = \mathrm{id}(a)$$

Since many times it is useful to access the value of $a$, we redefine the eliminator making it accept a function.

$$El_{\mathsf{Id}}(x, e) = \Big\{ e(a) \quad \text{if } x = \mathrm{id}(a)$$

$$\text{E-}\mathsf{Id})\ \frac{\begin{array}{c} T(z_1, z_2, z_3) \text{ type } [\Gamma,\ z_1, z_2 \in A,\ z_3 \in \mathsf{Id}(A, z_1, z_2)] \qquad a, b \in A\ [\Gamma] \\ x \in \mathsf{Id}(A, a, b)\ [\Gamma] \qquad e(w) \in T(w, w, \mathrm{id}(w))\ [\Gamma,\ w \in A] \end{array}}{El_{\mathsf{Id}}(x, e) \in T(a, b, x)\ [\Gamma]}$$

$$\text{C-}\mathsf{Id})\ \frac{\begin{array}{c} T(z_1, z_2, z_3) \text{ type } [\Gamma,\ z_1, z_2 \in A,\ z_3 \in \mathsf{Id}(A, z_1, z_2)] \\ a \in A\ [\Gamma] \qquad e(w) \in T(w, w, \mathrm{id}(w))\ [\Gamma,\ w \in A] \end{array}}{El_{\mathsf{Id}}(\mathrm{id}(a), e) = e(a) \in T(a, a, \mathrm{id}(a))\ [\Gamma]}$$

```
data Id (A : Set) : A → A → Set where
  id : (a : A) → Id A a a

ElId : {A : Set}
       {T : (z  z  : A) → Id A z  z  → Set} -- T(z ,z ,z ) type [z ,z   A,  z   Id(A,z ,z )]
       {a b : A} →                          -- a,b   A
       (x : Id A a b) →                     -- x   Id(A,a,b)
       (e : (w : A) → T w w (id w)) →       -- e(w)   T(w,w,id(w)) [w   A]
       T a b x                              -- ElId(x,e)   T(a,b,x)
ElId (id a) e = e a                         -- ElId(id(a),e) = e(a)
```

### Strong indexed sum

The last type for which we will analyze the eliminator is the indexed sum. This type contains two values of two different types. The *pair* (binary *tuple*) is a type, present in many programming languages, that serves this exact purpose. But the indexed sum is more powerful than a pair, indeed a pair is a special case of indexed sum.

In an indexed sum the type of the second element is a dependent type on the value of the first element. This is a bit tricky to understand, so let's see an example. Let's say the type of the first argument is a bool: we can impose that if the value of the first element is true then the second element must be a natural, and if instead it is false the second type should be unit type. If the second type is constant, we have the usual pair.

$$\text{F-}\Sigma)\ \frac{C(w)\ \text{type}\ [\Gamma,\ w \in B]}{\Sigma_{w\in B}C(w)\ \text{type}\ [\Gamma]}$$

The indexed type used in the example is $\Sigma_{w\in\mathsf{Bool}}C(w)$, where

$$C(w) = \begin{cases} \mathsf{Nat} & \text{if } w = \text{true} \\ & = El_{\mathsf{Bool}}(w, \mathsf{Nat}, \mathsf{Unit}) \\ \mathsf{Unit} & \text{if } w = \text{false} \end{cases}$$

$$\text{I-}\Sigma)\ \frac{C(w)\ \text{type}\ [\Gamma,\ w \in B] \qquad b \in B\ [\Gamma] \qquad c \in C(b)\ [\Gamma]}{\langle b, c\rangle \in \Sigma_{w\in B}C(w)\ [\Gamma]}$$

The eliminators has nothing to do other than deconstructing the "pair", passing both members as parameters to a given function

$$El_\Sigma(x, e) = \begin{cases} e(b, c) & \text{if } x = \langle b, c\rangle \end{cases}$$

$$\text{E-}\Sigma)\ \frac{\begin{array}{cc} C(w)\ \text{type}\ [\Gamma,\ w \in B] & T(z)\ \text{type}\ [\Gamma,\ z \in \Sigma_{w\in B}C(w)] \\ x \in \Sigma_{w\in B}C(w) & e(b, c) \in T(\langle b, c\rangle)\ [\Gamma,\ b \in B,\ c \in C(b)] \end{array}}{El_\Sigma(x, e) \in T(x)\ [\Gamma]}$$

$$\text{C-}\Sigma)\ \frac{\begin{array}{cc} C(w)\ \text{type}\ [\Gamma,\ w \in B] & T(z)\ \text{type}\ [\Gamma,\ z \in \Sigma_{w\in B}C(w)] \\ b \in B \quad c \in C(b) & e(b,c) \in T(\langle b,c\rangle)\ [\Gamma,\ b \in B,\ c \in C(b)] \end{array}}{El_\Sigma(\langle b,c\rangle, e) = e(b,c) \in T(x)\ [\Gamma]}$$

```
data Σ (B : Set) (C : B → Set) : Set where
  _,_ : (b : B) → C b → Σ B C

ElΣ : {B : Set}
      {C : B → Set}                                  -- C(w) type [w  B]
      {T : Σ B C → Set} →                            -- T(z) type [z  Σ(B,C(w))]
      (x : Σ B C) →                                  -- x  Σ(B,C(w))
      (e : (b : B) → (c : C b) → T  b , c  ) →       -- e(b,c)  T( b,c ) [b  B,  c  C(b)]
      T x                                            -- ElΣ(x,e)  T(x)
ElΣ  b , c  e = e b c                                -- ElΣ( b,c ,e) = e(b,c)
```

The indexed type is very useful to fix preconditions. For example we can define the positive naturals (greater than zero) as the indexed sum of naturals and a certificate of being greater than zero. We will use as certificate an identity type between $n$ (the first term of the indexed sum) and $\text{suc}(m)$ for some $m \in \mathsf{Nat}$: obviously there is no such proof for $n = \text{zero}$.

$$\mathsf{Nat}^+ = \Sigma_{n\in\mathsf{Nat}}\lambda(n).\Sigma_{m\in\mathsf{Nat}}\lambda(m).\mathsf{Id}(\mathsf{Nat}, n, \text{suc}(m))$$

which can be read as

$$\mathsf{Nat}^+ = (n,p)\ :\ n \in \mathsf{Nat}, \underbrace{\exists\ m \in \mathsf{Nat}\ :\ n = \text{suc}(m)}_{\text{proved by } p}$$

Now we can define a total predecessor function from $\mathsf{Nat}^+$ to $\mathsf{Nat}$.

$$\begin{aligned} \text{pred}(x) &= \Big\{\big\{m \quad \text{if } p = \langle m,q\rangle \text{ , if } x = \langle n,p\rangle && (1)\\ &= El_\Sigma(x, \lambda(n,p).El_\Sigma(p, \lambda(m,q).m)) && (2) \end{aligned}$$

Recall that $p$ is $\langle m,q\rangle$ and $q$ a proof of $n = \text{suc}(m)$.

```
Nat  : Set
Nat  = Σ Nat ( n → Σ Nat ( m → Id Nat n (suc m)))

pred : Nat  → Nat
pred x = ElΣ x ( n p → ElΣ p ( m n succ[m] → m))
```