**Common Types as MLTT Types**

This lecture will cover mapping common types (Bool, Maybe, Fin, Lists, Vectors, etc) to MLTT types (using 0, 1, product, etc).

In reality, we don’t use these definitions as they reduce the **readability** of our programs.

However, by defining them as **isomorphisms**, we can prove that the functionality of our programs has not changed.

# Boolean

* The boolean type (introduction [here](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/Bool.lagda.md)) can be imagined to be an OR type, with both left and right being the unit type.
* In this definition, left is false, and right is true:



* The isomorphism definition can be constructed accordingly.
* This is shown in the notes [here](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/Bool-functions.lagda.md).

# Maybe

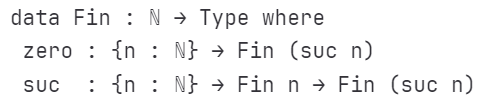
* Maybe is defined as **nothing** and **just X**.
* We can imagine the maybe type as a dependent OR type, with left being the unit type, and right being the dependent type:



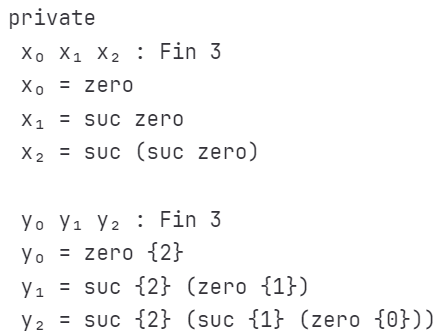
* This is shown in the notes [here](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/Maybe.lagda.md?plain=0).

# Fin

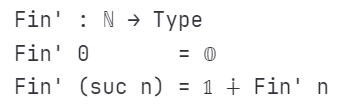
* As a reminder, the [Finite type](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/Fin.lagda.md) is defined as follows - Fin n has n elements:



* It can be used as follows:



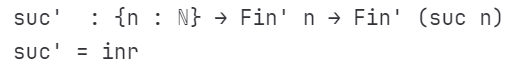
* When the dependent value n increases by 1, we have **1 new element** of that type (zero {n}).
* For the other elements of the type n - 1, we increment the values using suc.
* For Fin 3, we have zero {2}, suc {2} (zero {1}), and suc {2} (suc {1} (zero {0})).
* Notably, Fin 0 is empty, because it has no elements.
* The [video](https://bham.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=36d8b376-060b-4dda-8a4a-af9e01065fa6) contains a good explanation of how the Fin type works.
* For the [isomorphic definition](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/Fin-functions.lagda.md), we first define a type definition.
  + 0 is the empty type, as before.
  + suc n is defined as the unit type or the recursive step (therefore Fin 1 will have 1 element, Fin 2 will have 2, etc).
  + Fin’ 3 = 



* We then also define constructors for actual values of this type:
  + First, we define a zero’ function to produce zero elements of Fin’ n:



* Then, we define a suc’ function to produce suc elements of Fin’ n:
* What Agda wants here is this definition: 
* This is purely an inr call on whatever input we are given:



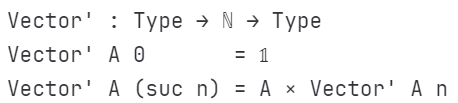
* From here, we can define an isomorphism between Fin n and Fin’ n.
  + This can be found in the notes above.
* f and g are relatively straightforward.
* gf and fg require slightly more complex recursive definition, and also use [**notation for equality reasoning**](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/identity-type.lagda.md#notation-for-equality-reasoning)
  + This notation basically allows you to repeatedly show equalities.
  + For any proof on the right, you are showing that the definition to the left is equal to the definition a row down and to the left.

# List

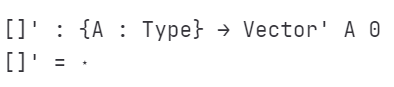
* Reminder: [lists](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/List.lagda.md) are defined as the empty list, followed by **any number of elements** prepended to that empty list.
* The lecture notes also define elimination principles for lists, both dependent and non-dependent:
  + The non-dependent elimination principle is like a **fold function** for lists.

# Vectors

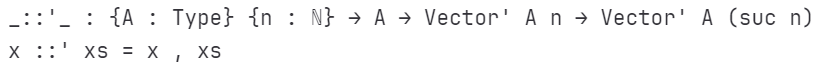
* Reminder: [vectors](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/Vector.lagda.md) are like lists, but we know how long the list is.
* We start with the empty list with length zero, and follow that with any number of elements prepended to that empty list.
* Every time we prepend an element, we increase the length of the list by 1.
* The lecture notes also define elimination principles for vectors, both dependent and non-dependent:
  + The non-dependent elimination principle is like a **fold function** for vectors.
* In Agda, it is impossible to define a **head** or **tail** function on lists.
  + This is because we don’t know how long the list is.
  + However, when using a Vector, in our type definition we can have that we require a Vector X (suc n), meaning that the list has at least one element in it.
* We can also define a type safe **indexing function** because of this principle.
* This is discussed in the following [notes](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/Vector-functions.lagda.md) file.
* Also discussed in the above notes file, is representing a **vector** as a MLTT type.
* First, we define a **type definition**:
  + The empty list is the Unit type.
  + Our successor definition is a **product** (AND) type. This allows us to chain elements together.



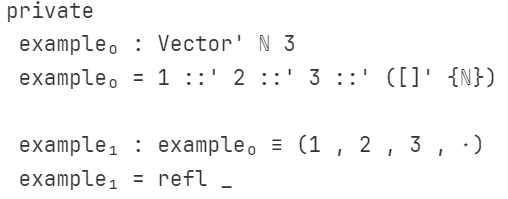
* Then, we define constructors for elements of the type:
  + For the empty Vector, we just need to produce an element of the Unit type, which is trivial:



* + For the cons (::) function, we take in a new element, and the rest of the list, and we must produce a cartesian product, which is relatively simple:



* This can be used in the following way:



* Finally, we can define an isomorphism between Agda Vectors, and the MLTT Vectors, and the notes walk through this.

## List Vector Isomorphism

* The lecture notes go on to define **isomorphisms** between lists and vectors.
  + List A is isomorphic to an existential type, where the witness is the length of this list.
  + Vectors of length n are isomorphism to Lists where the length of the list is equal to n.