**Week 2**

In this set of lectures, we cover the concept of **propositions** as **types**, and **proofs** as **programs**. This is the goal of:

1. Verifying programs
2. Writing better, cleverer, more useful types
3. State properties that programs should satisfy (documentation).

This is known as MLTT ([Martin-Löf type theory](http://archive-pml.github.io/martin-lof/pdfs/Bibliopolis-Book-1984.pdf)).

# Logic and Propositions

* In propositional logic, we have logic symbols: not, and, or, implies, forall, exists, equals, etc.
* We can then construct propositions:
  + This proposition states that every natural number is either odd or even.
* In most programming languages, when you implement some data structure for example, you must verify that the data structure is correct.
  + Usually, this correctness is in your head or through unit tests, but in Agda we attempt to show correctness more thoroughly through programs.

## And (Conjunction)

* Suppose we have two logical statements A and B, and we want to prove both A and B at the same time.
* In propositional logic, we would use AND:
* In Agda, we will use the times symbol:
* To show this, we must provide **justifications** for our logical statements, with reasoning:
  + We have to justify that A is true, **and**
  + We have to justify that B is true.
* If we say that a : A means that a is a justification of A, and b : B means that b is a justification of B, then our proof is:
  + We place our justifications into a pair, and the type becomes the cartesian product.
* In Agda:
  + data \_x\_ (A B : Type) : Type where
  + \_,\_ : A -> B -> A x B

## Or (Disjunction)

* Suppose we have two logical statements A and B, and we want to prove either A or B is true.
* In propositional logic, we would use OR:
* In Agda, we will use binary sum: + (either in haskell)
* Given a : A and b : B, we tag our proof a in the full set of A as in l, and our proof b in the full set of B as in r.
* This results in two constructors in Agda:
  + data \_+\_ (A B : Type) : Type where
    - in l : A -> A + B
    - in r : B -> A + B
* This makes sense, as we only need one proof A or B to satisfy the proof A + B.

## Implication

* Suppose we have two logical statements A and B, and we want to prove A implies B.
* In propositional logic, we would use impl:
* In Agda, we will use the function symbol:
* Example: x is odd x + 1 is even
  + To prove implications, we assume the left side, then prove the right.
  + We assume x is odd is true, then use the assumption to prove x + 1 is even.
* **A proof of an implication is a function**
  + that transforms a justification (proof) of A, into a proof of B.

## False

* We cannot prove false or contradiction - e.g: we cannot prove 0 = 1.
* We instead use the empty type 𝟘 for this purpose, which has **no elements**, and therefore cannot be created.
* In Agda:
  + data 𝟘 : Type where
  + {no constructors as the type is empty}

## Negation

* Suppose we have a logical statement A, and we want to prove not A.
* In propositional logic, we would use the not symbol:
* The strategy in Mathematics is to **assume A**, then conclude or derive something which is absurd (or false) - e.g: 0 = 1.
* In Agda, we therefore represent this as A -> 𝟘 (A implies empty type/false).
* This is a function that transforms an element of A to the empty type, which is false.
* Example, suppose A is false, therefore it is represented by 𝟘.
  + This means that not A is true, as it would be:
  + f : 𝟘 -> 𝟘
  + To prove this function, we would have to provide a proof for all constructors of the empty type (there are none).
  + Therefore, we show this in Agda as f ()
    - () is the absurd pattern - no inputs.

## 

## Forall

* Suppose we have a logical statement A, and we want to prove it holds for all elements of a set.
* In propositional logic, we would use the universal quantifier:
* Example: (numbers are odd or even)
  + We will abbreviate this as
* In Agda, we represent forall as functions - assume an element of the input (x : A in this case), and we construct a justification of the output (Bx).
  + (x : A) -> B x
  + This is known as a **dependent function**, and B is a **dependent type** because its type output depends on the input x (remember this is the **type definition**, not the implementation).
* This makes sense, because the function says that we can take any element of the input set (forall), and we will get an element of the output set.
* Example of a dependent type:
  + B : Bool -> Type (B takes a bool, and returns a type)
  + B false = 𝟘
  + B true = 𝟙

## Exists

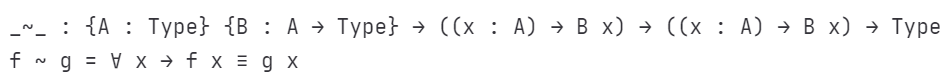
* Suppose we have a logical statement A, and we want to prove it holds for at least one element of a set.
* In propositional logic, we would use the existential quantifier:
* Example: (there exists an even natural number)
* Generally, to prove ,
  + We have to give an example of x, and
  + A justification of B x
* This means that we need a pair (x, y) where x : A and y : B x
  + This is a **dependent pair**.
* In Agda, we represent exists using **sigma** .
  + data Σ {A : Type } (B : A → Type) : Type where
  + \_,\_ : (x : A) (y : B x) → Σ {A} B
* This is a **generalisation** of the cartesian product, where the second element depends on the first.
  + Another way of writing this notation is

## Equality

* Suppose we have two logical statements A and B, and we want to prove that A = B.
* In propositional logic, we would use equals: x = y
* In Agda, we will use the **identity type (Id):** ≡
* We imagine that the equality type is empty if x != y, and has one element if x = y.
* Given X : Type and x y : X (two elements part of X), we can form a new type Id X x y.
* We assume as an axiom that anything is equal to itself:
  + refl x : Id X x x (for reflexivity)
  + Note: Id X x x is a type (not a boolean). It is the type of justifications that x = x.
* Refl can only be used to create equality when you are sure you have two of the same element.
* The Agda specific definition is as follows:
  + data \_≡\_ {X : Type} : X -> X -> Type where:
  + refl (x : X) -> x ≡ x
* This is a type family, because it is a collection of types where, depending on the left and right elements, you get a specific type which may be empty, or have an element (refl). This means that:
  + 3 ≡ 17 is empty (you cannot find any element of the type to create).
  + 3≡ 3 has one element.

### Pointwise Equality

* This is a modified version of equality which defines whether two **functions** are equal - intuitively, they are equal if they have the same value for all inputs:



* It is not provable or disprovable in Agda that f ~ g implies f = g.
  + This is known as **function extensionality**, and while we cannot prove it in some cases we may assume it

# Summary

<https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/curry-howard.lagda.md>

| **Logic** | **English** | **Type theory** | **Agda** | **Handouts** | **Alternative terminology** |
| --- | --- | --- | --- | --- | --- |
| ⊥ | false | ℕ₀ | 𝟘 | empty type |  |
| ⊤ | true (\*) | ℕ₁ | 𝟙 | unit type |  |
| A ∧ B | A and B | A × B | A × B | binary product | cartesian product |
| A ∨ B | A or B | A + B | A ∔ B | binary sum | coproduct,  binary disjoint union |
| A → B | A implies B | A → B | A → B | function type | non-dependent function type |
| ¬ A | not A | A → ℕ₀ | A → 𝟘 | negation |  |
| ∀ x : A, B x | for all x:A, B x | Π x : A , B x | (x : A) → B x | product | dependent function type |
| ∃ x : A, B x | there is x:A such that B x | Σ x ꞉ A , B x | Σ x ꞉ A , B x | sum | disjoint union,  dependent pair type |
| x = y | x equals y | Id x y | x ≡ y | identity type | equality type,  propositional equality |

Finally, we have elimination rules to get rid of the types. E.G:

* For , we pattern match on inl x and inr y to get our output.
* Others for and, functions, exists, etc

Examples:

* To prove the odd/even case
* We can say that f(10) = inl (5, \*)
  + f(13) = inr (6, \*)
* Inl/inr tells us whether the number is odd even.
* The input to the exists essentially performs integer division by 2.
* From there the equality (both odd and even) is trivial, because Agda can verify that the y\* 2 (+ 1 if odd) is equal to x trivially.

You can view a type definition for functions both computationally and mathematically:

* The computational definition is as usual, takes an element of the type X and returns Y.
* The mathematical definition is "Given any element of X, we are able to derive an element of Y"

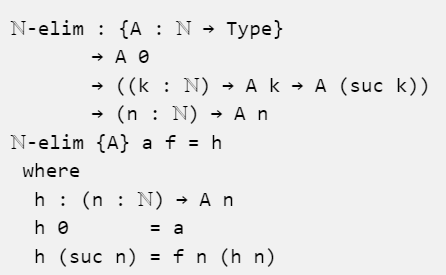
A good example of this is the or-elim proof:

* Logically to for all A+B->C, we must have a function for both A and B to C, then an element of A+B, and we can get an element of C out.
* Computationally, we match on cases - we see which element of A+B we have (inl or inr) and use the appropriate function to get an element of C

# 

# Elimination Principles

* In the introductions for each type above, we have shown the **introduction** principles, how to get an element of the type.
* However, for each of these, there is also an **elimination principles**.
* These tell us how to define dependent functions **out** of the given type.
* For example, we can define an elimination rule for natural numbers which essentially provides **primitive recursion**.
  + Given a base case a : A 0, and a step function f : (k : Nat) -> A K -> A (suc k), we get a function h : (n : Nat) -> A n
    - A is a predicate on natural numbers.



* We can also define **non-dependent** elimination functions which are not of the predicate form (and therefore do not require a dependent type input).
  + This is much similar to standard primitive recursion:
  + We take a base case a : A, a step function f : Nat -> A -> A, and we get a function N -> A.

