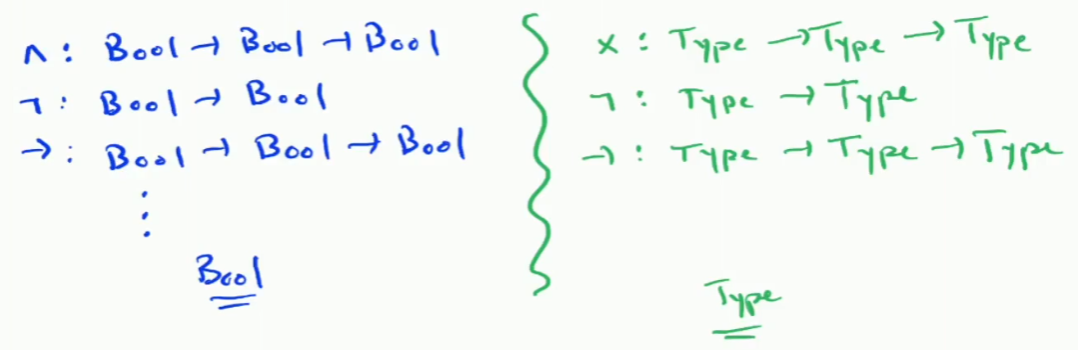
**Decidability**

# Propositions as Types vs Booleans

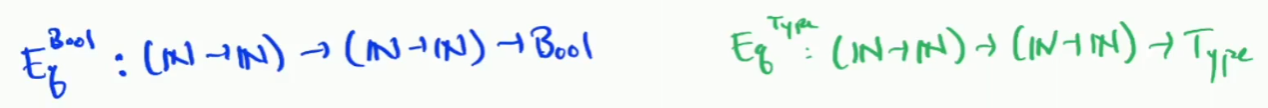
* In Haskell, we are used to defining propositions using Booleans.
  + E.G: forall on a list[A] returns true iff a function A -> Bool returns true for every element of the list.
* In Agda, we have now discussed the idea of instead using types to represent these propositions.
  + By this we mean, if we can create a statement with that type, then the proposition must be true.
  + The function with that type is essentially a **specification** for that proposition.
* Basic AND, NOT, Implies rules can be derived for Booleans, as well as for the Type of all Types (Type):



* However, Type is like an upgraded version of Bool.

## Example

* One case where using Type becomes more expressive than Boolean is with Equality.



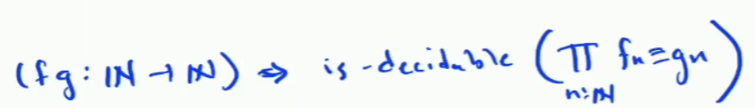
* The example shows comparing two functions from Nat to Nat to see if they are equal.
* In the boolean version, we would have to check **all** inputs to ensure that the outputs are equal, but there are **infinitely many inputs**, so this is not possible.
  + This is a form of the **halting problem**, and you could reduce this problem to the halting problem to prove it is impossible.
  + Therefore, we cannot check that f and g are equal using an algorithm.
* The Agda Type version, we would encode the proof using a forall quantifier:



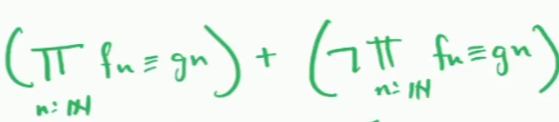
* We simply ask for proof from the type that, for any natural number, f n = g n.
  + Terminology, we say that we find a type who’s inhabitants are **witnesses** that f and g are equal - we essentially delay answering the question.
* This is trivial to do with the same function - the answer is

# Decidability

* We then define decidability:
  + A Type is decidable if I have a proof of A or not A.
* What we are essentially saying here is that, do have the decidability type, either:
  + We know how to calculate that a statement is true, OR
  + We know how to prove that there is no such statement.
* Decidability is trivial for base types (Nat, Bool, 1, 𝟘), as we have to give a proof of , and we usually have an element of A.
* For Nat, we simply have to give a proof that we know a natural number, e.g:
  + inl 0, inl 42, etc (infinitely many)
* Bool is similar with true and false.
* 1 (unit type) is decidable with \*.
* 𝟘 is different, we don’t know an element of 𝟘 (because it has no elements), but we do know an element of ¬𝟘:
  + ¬𝟘 is really the type of functions from 𝟘 → 𝟘, which is just the identity type, so inr id.
* This introduces the idea that we can have **multiple proofs** for a type. E.G: there are infinitely many proofs for Nat as there are infinitely many natural numbers.
* However, we run into a problem from here. If we go back to the previous example - prove that two functions f and g are equal, we’ve really just kicked the can down the road:



* The definition now looks like this - for all f and g, the statement “for all Natural numbers, f n = g n” is decidable. The is-decidable portion translates to:

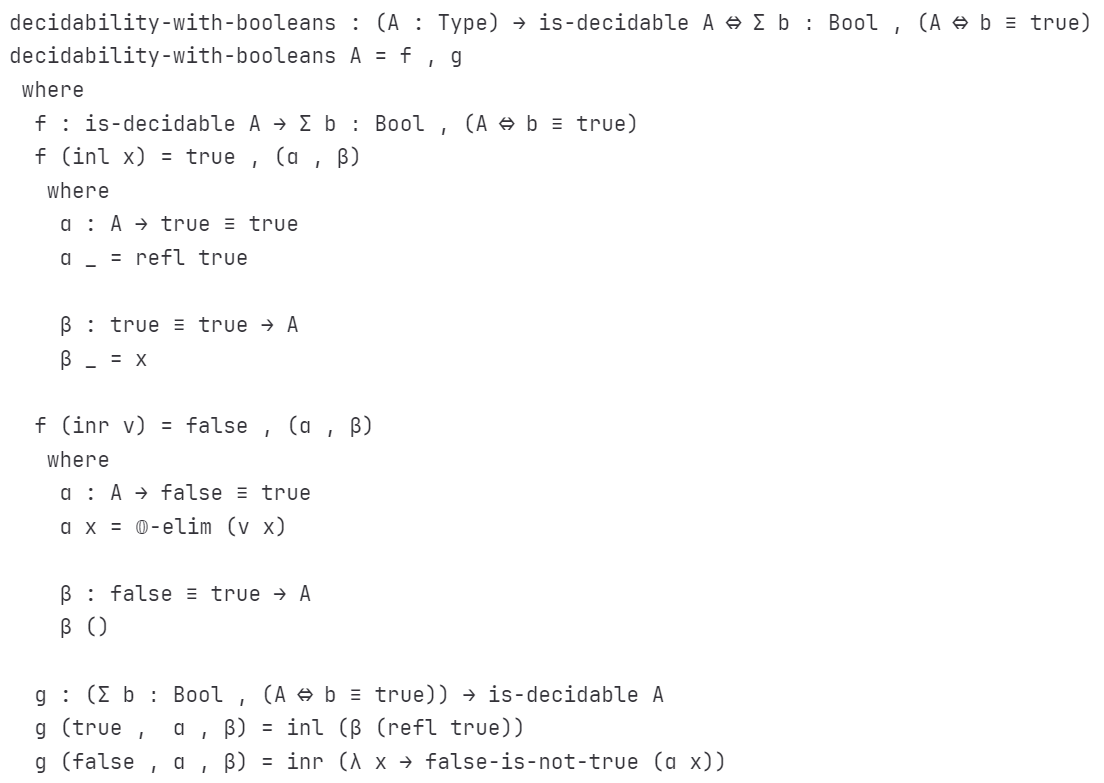


* Either we can prove f n = g n for all natural numbers, or we can prove it’s impossible.
  + This is **still** the halting problem all over again.

## 

## Boolean Decidability

* We can prove that decidability of any type is analogous to Booleans.



* The proof is quite complex, but the f function was explained in the lecture - it’s essentially a pattern match on the decidability (either A or not A).
* The A portion - top - is fairly self explanatory.
* The not A portion - bottom - uses or elim firstly (in alpha). We require a function from A -> false = true, but false = true is part of the empty type (as it’s not equal.
  + This means that we can simply use 0-elim, which allows us to prove anything we want as long as we have a theoretical element of the empty type. We apply x (an element of A) to v (a function from A -> 0), to get 0.
* Finally, the beta function works in reverse, we take a non-equality (which is the empty type) and return A.
  + This is impossible to get as an argument, so Agda is happy with () as the proof of this function.
  + You can write this as a lambda - \ { () } - but it’s not particularly readable.
  + Better to define a separate function showing the impossibility and use that.

## 

## Predicate Decidability

* We define a Predicate as some function of X -> Type (Agda) or X -> Bool (Boolean).
  + is-decidable-predicate : {X : Type} → (X → Type) → Type
  + is-decidable-predicate {X} A = (x : X) → is-decidable (A x)

Predicates are simply dependent types.

* The above Agda function defines what it means for a predicate (e.g: is-even) to be decidable, but does not define how to do this - it is a Type function.
  + Useful note, is-decidable is itself a **predicate on Types.**
* Then, we can prove some predicate A (X -> Type) is decidable iff there is a boolean valued function a (X -> Bool) which A x holds iff a x = true (literally if a x computes to true):
  + predicate-decidability-with-booleans : {X : Type} (A : X → Type)
  + → is-decidable-predicate A
  + ⇔ Σ α ꞉ (X → Bool) , ((x : X) → A x ⇔ α x ≡ true)
* Above is the Agda definition of that statement.
* We do this translation because we prefer to use type-valued-predicates everywhere (as we can always **define** type-valued-predicates) but we can only define boolean-valued-predicates in specific cases.
  + We cannot define boolean-valued-predicates for the case above.

## Iff Decidability

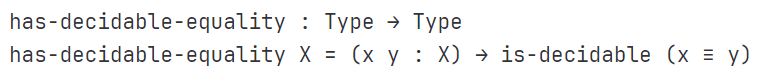
* Iff (A ⇔ B) can be used in decidability proofs.
  + In Agda, iff is represented as a pair of functions (A -> B, B ->A), you can pattern match on (f, g) to get the functions out.
* We can show that, given A ⇔ B and is-decidable A, we can get is-decidable B:
  + Pattern match on the iff (f, g), and is-decidable A which gets two cases out.
  + The first is that we have an inl a (proof of A existence), and we can provide inl (f a).
  + The second is that we have an inr nota (proof of not A), we can provide inr (\b -> nota (g b))
    - This works because nota is actually a function from A to the empty type.

## Relation Decidability

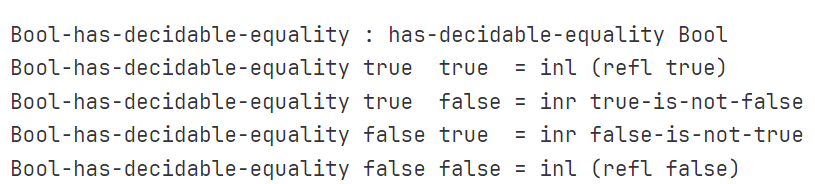
* Similarly to predicates, we can define decidability on relations:
  + is-decidable-relation : {A : Type} (R : A -> A -> Type) -> Type
  + is-decidable-relation {A} R = (a : A) (b : A) -> is-decidable (R a b)
* In the lecture, Eric introduced a type for <=, including base case 0 <= everything, and successive case m <= n.
* data \_ <= \_ : Nat -> Nat -> Type where
  + 0<= : {n : Nat} -> 0 <= n
  + S<= : {m n : Nat} -> m <= n -> suc m -> suc n
* 2 <= 4 = S<= (S<= 0<=)
  + The actual nats are implicit.
* We can use this with our is-decidable-relation using:
  + <=-is-decidable : is-decidable-relation \_<=\_
  + <=-is-decidable zero n = inl 0<=
  + <=-is-decidable (suc m) zero = inr (\ { () }) -- impossible argument proof
  + <=-is-decidable (suc m) (suc n) = ….recursive call on <=-is-decidable

# Decidable Equality

* The lecture notes also define **decidable equality:**

****

* To define this, we have to select which type of decidability we are showing (exists, or not exists), then provide the proof. E.G: for bools:



* The [lecture notes](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/decidability.lagda.md) also feature a proof of equality of functions.
  + While we cannot generally decide equality of functions, in specific cases we can prove or disprove it.
  + The example in the lecture notes provides two functions, f and g that are equal, as well as a third function h that is not equal, and shows proving f = g and f != h.