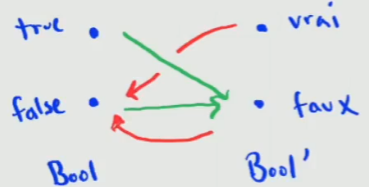
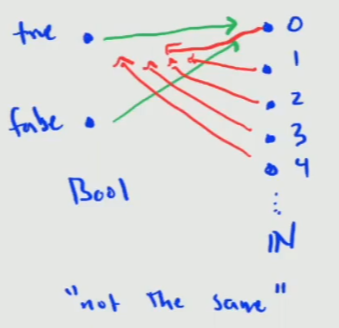
**Isomorphisms**

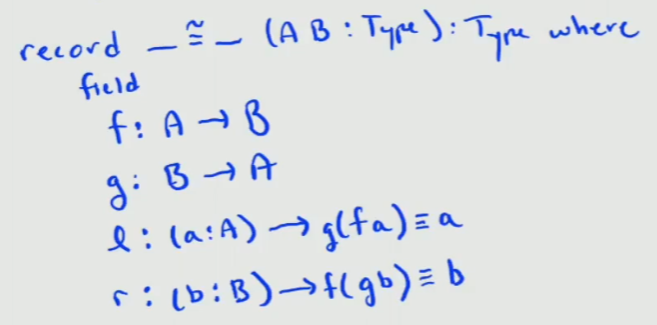
* Isomorphisms are also known as **bijections**.
* We have previously defined iff (if and only if) using A ⇔ B, and said that this is the case in Agda if we have:
  + f : A -> B and g : B -> A
* Isomorphisms are a refinement of this - we have **more information.**
* As an example, say we have Bool, and we define a Bool’ using the French for true and false. We can define Bool -> Bool’ and the reverse, but what about if we defined it wrong? The image below shows what this might look like:



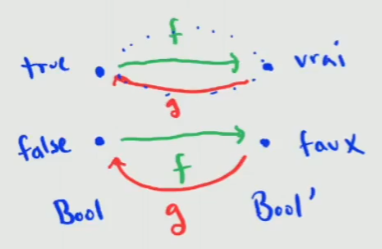
* + While we do have complete functions, we haven’t really captured the concept that the two should be the same (partially because our definition is wrong, and partially because our types don’t capture it right now).
* Another example from Bool -> Nat shows a similar situation. Intuitively, we know that the two are **not the same**, but we can still define functions to create that mapping:



* Therefore, to refine our iff to capture this, we define a **bijection** from A to B using a record type. The [notes](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/isomorphisms.lagda.md) provide a similar definition using more complex terminology:



* We define the two functions f and g to convert. W
* We also define two more, l and r, which capture the notion that, given any value, applying **both** functions gets us back to the value that we started with.
* To visualise this a little, we go back to our earlier example. We can define two functions that meet this bijection type for Bool and Bool’:



* We cannot do it for our Bool and Nat example, because while we can pick either true or false for **two** of the numbers, all of the rest of the infinitely many numbers will have nowhere to go - it’s not a 1-1 reversible mapping.
* [This video](https://bham.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=a1fd8904-7924-467a-8167-af9e010664be) works through in detail an example of writing a bijection for OR (A + B) and an equivalent for OR using dependent sums (exists).
* The [lecture videos](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/ActivityPlans/files/week4.md) include Agda specific examples of defining Bijections using the record type:
  + Mapping bool to the 2 type (type with two values)
  + Mapping Agda binary sums to binary sums defined using the 2 type.
  + Mapping Agda products to products defined using the 2 type
    - This uses function extensionality (as an assumption).
* In the repository, there is a **template** - isomorphism-template.lagda.md.
  + This template is something that can be inserted to prove any isomorphism, as it features all the goals required to prove it.
  + Prove f, then prove g, then prove gf and fg and the template will handle the boilerplate to construct the record type.

# Lecture Notes

* In the lecture, [Eric demonstrated](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/lecture4.agda#L45) constructing a number of isomorphisms:
  + Proving that or (+) associativity is an isomorphism:
    - {A B C : Type} -> A + (B + C) ≅ (A + B) + C
  + Proving that exists (𝝨)-or is distributive:
    - {A : Type} {B C : A -> Type} -> (𝝨 a : A, B a + C a) ≅ (𝝨 a : A, B a) + (𝝨 a : A, C a)
* In general, showing isomorphisms is generally a task of moving syntax around, rather than solving anything:
  + Lots of pattern matching, and using the goals to get the right output

# Motivation

* While it is helpful to have the definition of isomorphisms, why do we need them?
* When we have an isomorphism defined, anything we can prove about one type of an isomorphism is **by default** proved in the other type.
* Firstly, [Eric proved](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/lecture4.agda#L135) transitivity of **predicates (A -> Type)** with bijectivity.
* This means that, given a predicate on A, and an isomorphism between A and B, we can also get a predicate on B (trans-pred)
* This proof was then used to show invariance of isomorphisms:
  + {A B : Type} (P : A -> Type) -> (iso : A ≅ B) -> (a : A) -> P a iff trans-pred P iso (bijection iso a)
* What this proof essentially shows is that given a predicate on A, we can apply gf (or fg) from the isomorphism to get the same predicate on a.
  + Read more into this proof from the [live coding lecture](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/lecture4.agda#L141), uses symmetry

# Natural Numbers Isomorphism

* The recursive (zero/successor) representation of the natural numbers is great for proving (as you just have to define the zero and successive cases).
* However, it is awful for computation.
* It’s possible to define natural numbers [in Agda](https://git.cs.bham.ac.uk/afp/afp-learning-2022-2023/-/blob/master/files/LectureNotes/files/lecture4.agda#L158) in binary, using a base case, then a zero and one case (appending zeros and ones onto the base case).
  + This is much better for computations, because the information we are storing is exponential (squared) in the size of the number.
* Then, it’s possible to define an **isomorphism** between the two definitions:
  + To convert from Nats to binary numbers, we have a recursive definition on zero and suc n.
  + The zero definition is trivial, and the suc n case uses a secondary successor of a binary number function (which is itself recursive).
* Note that we cannot define an isomorphism properly however, because leading zeroes are possible in our binary definition (meaning there are multiple definitions of the same number).
  + We would have to define a reduced form first.