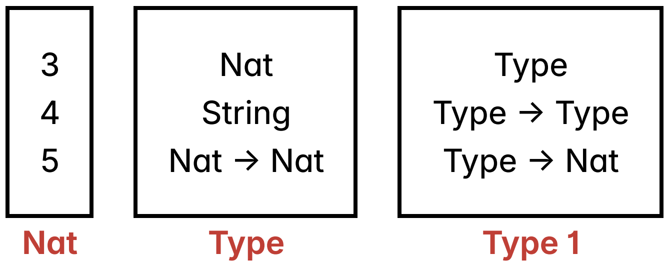
Type theory in Lean

Jan 31 2026

In many programming languages, there is a difference between "values" and "types". For instance, `3` is a value, `Nat` a type. In Lean, it is more complicated. Types such as `Nat` are also entities that have a type. In this case, `Nat` is an entity with type called `Type`. In Lean, this recurses: entities like `Type` are entities with type `Type 1`.

Since we have a situation more complicated than just values and types, we are going to refer the Lean entities as *objects*. (They are officially called "terms"; see here, but we use the word "object" here to make things more intuitive.) We list some objects and their types below in the format `<object> : <type>`. Remember that types are also objects.

```
3: Nat
Nat: Type
Type: Type 1
```

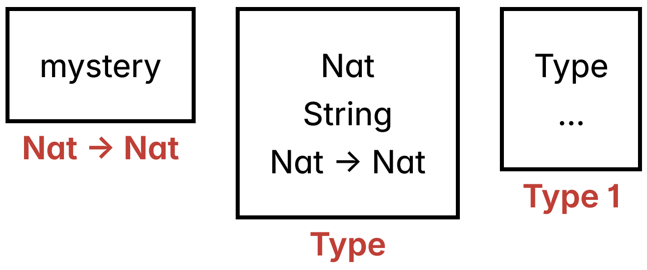


In Lean, there is no Type of all types. It instead uses a hierarchy, where any object using `Type n` has type `Type n+1`.

```
Type 3: Type 4
Type 3 → Type 3: Type 4
```

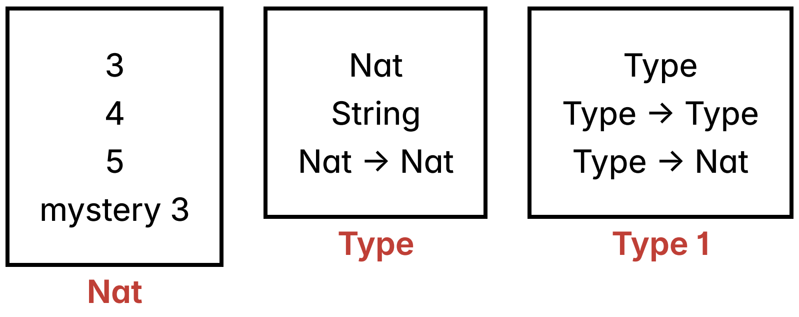
If you *construct* a new object (i.e. using `def`), it will also have a type. (Some people call this "defining a term", but I personally find this wording confusing.)

```
def mystery : Nat → Nat := fun x ⇒ 2*x
```



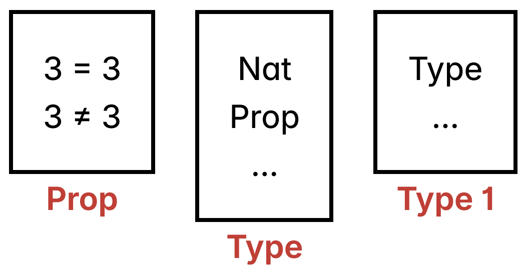
`mystery` is a function that inputs objects of type `Nat` and outputs objects of type `Nat`.

```
mystery 3: Nat
```



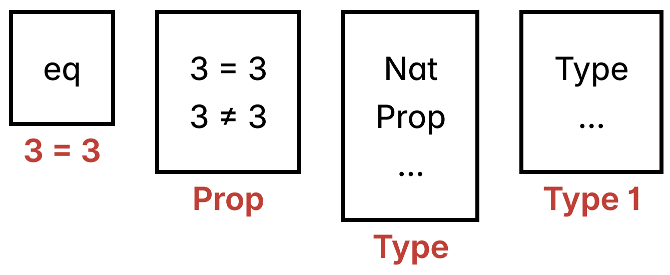
Prop is also a type. Math statements in Lean have the type **Prop**.

```
3 = 3: Prop
3 ≠ 3: Prop
Prop: Type
```



Math statements in Lean are also types. Sometimes you can construct objects of this type. An object with a math statement as its type is called a *proof* of that math statement (for reasons we will describe later on).

```
def eq : 3 = 3 := rfl
```

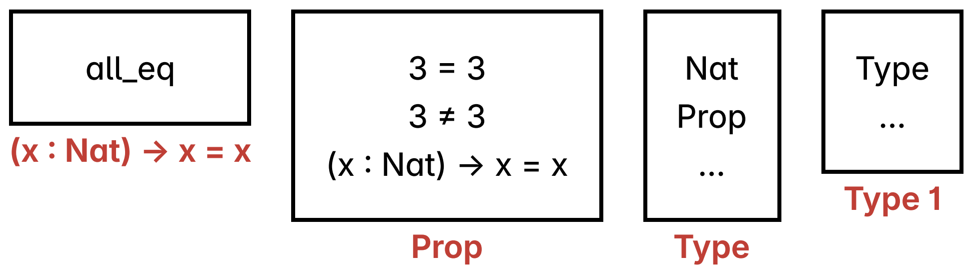


Math statements can be implications, like `P → Q`. In fact this is how ∀ is defined; i.e. `∀ (x: Nat), T x` is equivalent to `(x: Nat) → T x`.

```
∀x, x = x : Prop
(x : Nat) → x = x : Prop
```

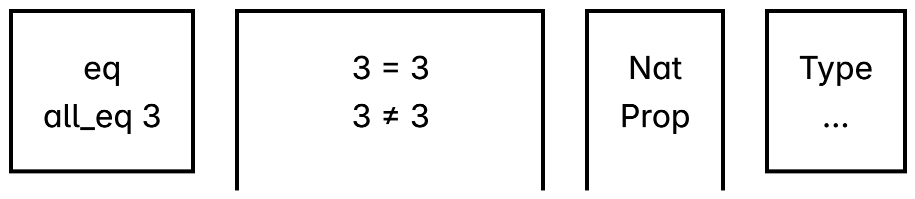
An object with this math statement as its type, is a *function*.

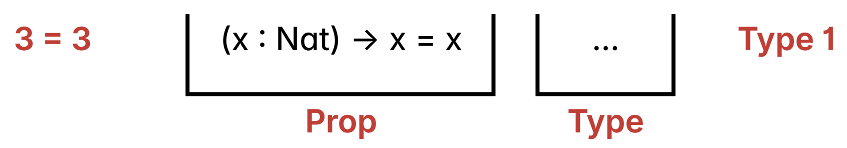
```
def all_eq : (x : Nat) → x = x := fun (y: Nat) ⇒ rfl
```



Here, `all_eq` is a function, that inputs an object of type `Nat`, and outputs an object of type `x = x`. We can use `all_eq` to construct other objects which have a math statement as its type.

```
all_eq 3 : Prop
```



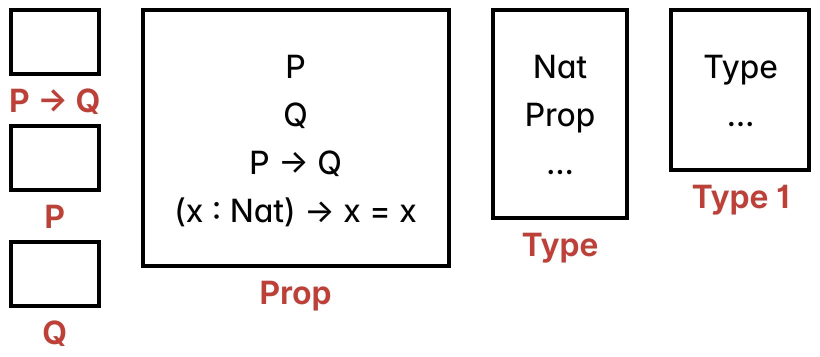


Implications work the same way. Suppose we have math statements `P` and `Q`:

```
P: Prop
Q: Prop
```

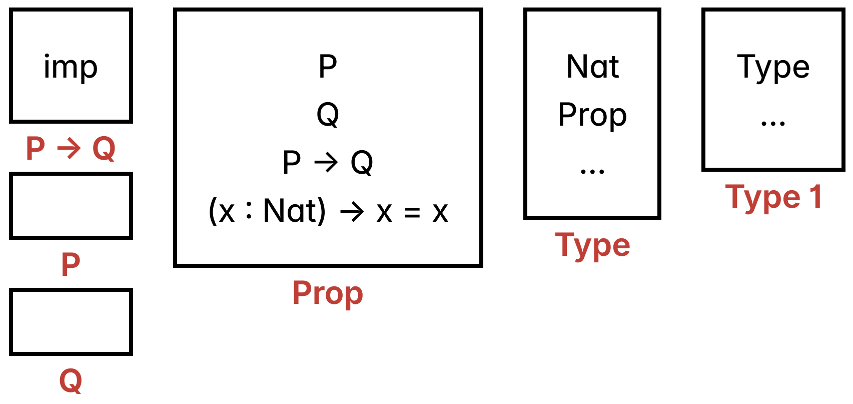
Of course, `P → Q` is also a math statement:

```
P → Q: Prop
```



We can try to construct objects of type `P → Q`, even if we have no objects of type `P` or of type `Q`.

```
def imp : P → Q := ...
```

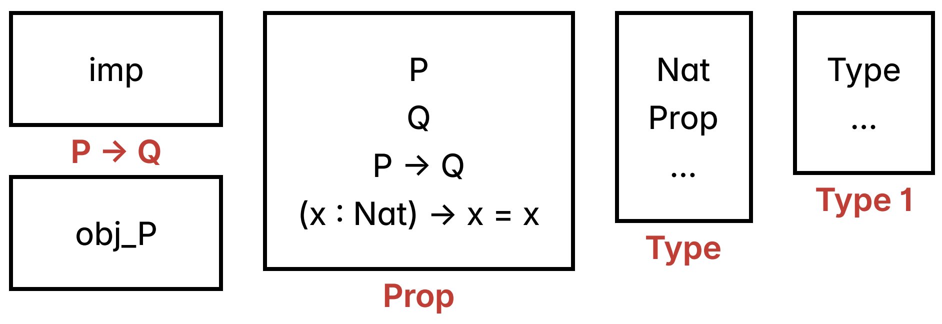


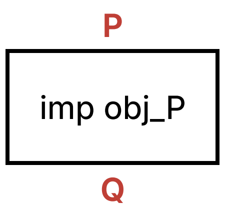
Then `imp` is a function, that inputs an object of type `P` and outputs an object of type `Q`. Note that `imp` can be constructed whether or not we have objects of type `P`.

Now, *if we had* an object `obj_P` of type `P`, then we can use `imp` to construct an object of type `Q`.

```
def obj_P : P := ...
```

```
obj_P : P
implication obj_P : Q
```





So what does it mean to "prove something" in Lean? Colloquially:

> Proving `P: Prop` in Lean means constructing an object with type `P`.

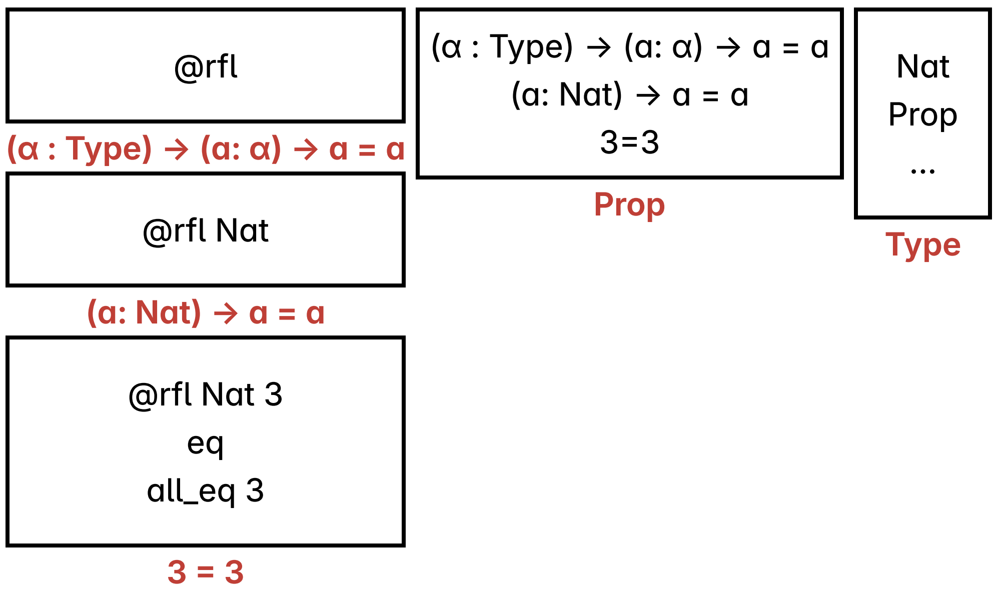
The axioms of Lean are the built-in objects of math statements. For Lean, this is basically the axioms of ZFC (precisely, it is `Calculus of Inductive Constructions`, which you can learn more about here). One example of a built-in object is `rfl`, which is roughly:

```
rfl : ∀ {α : Type} {a : α}, a = a
```

We can use `rfl` as a function to construct other objects which have a math statement as a type.

```
@rfl Nat : ∀ {a : Nat}, a = a
@rfl Nat 3 : 3 = 3
```

(`@` forces Lean to make all implicit arguments marked with `{}`, *explicit*.) Setting aside this detail, we are using `rfl` as a function to construct new objects.



Constructing an object of type `3 = 3`, colloquially, is a *Lean proof* of 3 = 3. In other words, `@rfl Nat 3` is a proof of the statement `3 = 3` (starting from the built-in axioms of Lean).

Implications work the same way.

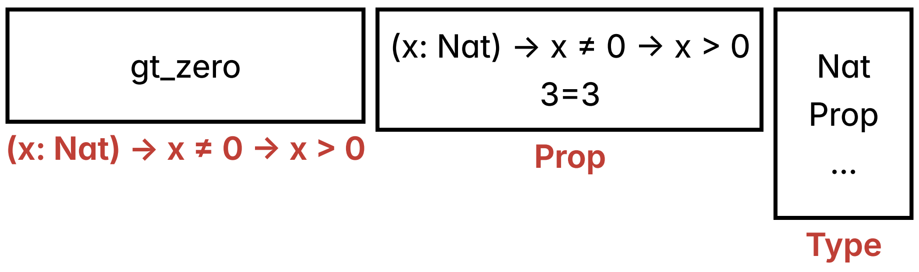
```
∀ (x : Nat) x ≠ 0 → x > 0 : Prop
```

This is equivalent to the statement

```
(x: Nat) → x ≠ 0 → x > 0 : Prop
```

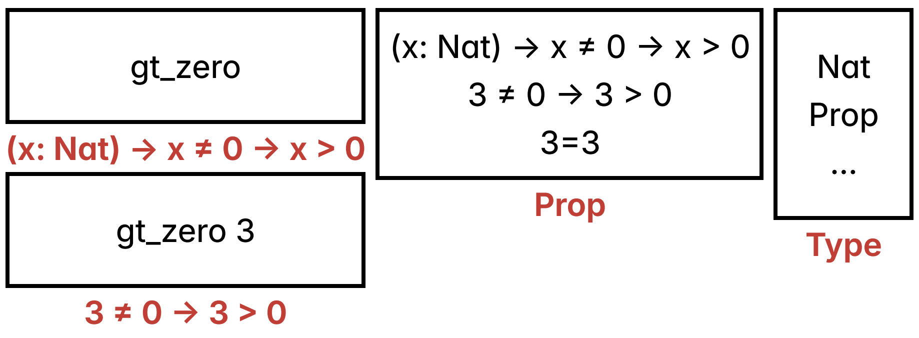
If we construct an object which has this statement as its type, then we have *proven* this statement.

```
def gt_zero : ∀ (x : Nat), x ≠ 0 → x > 0 :=
  fun x h ⇒ Nat.zero_lt_of_ne_zero h
```



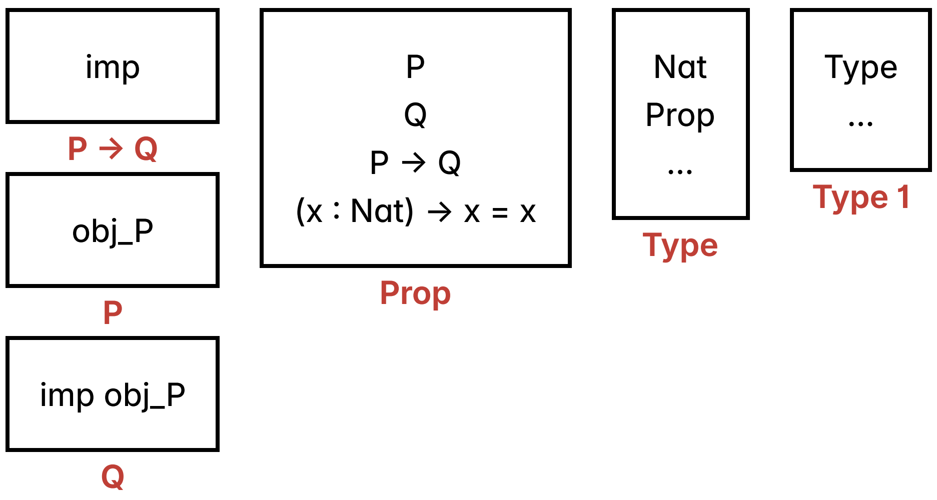
Notice that `gt_zero` is also a function; we can use it to *prove* other statements.

```
gt_zero 3 : Prop
```



Of course, the math statement `3 ≠ 0 → 3 > 0` is obvious to us. But in Lean, it's a proof **only** when you can *construct* an object who has that math statement as its type.

At the same time, *proofs* (i.e. objects) of type `P → Q` are functions. They input objects of type `P`, and output objects of type `Q`. In our earlier example, if we have a *proof* (i.e. object) `obj_P` of type `P`, and a *proof* (i.e. object) `imp` of type `P → Q`, then `imp obj_P` is a *proof* (i.e. object) of type `Q`.



This equivalence between *functions* and *implications*, and *proofs of P* and *objects of type P* at the core of the Curry-Howard correspondence.

So, what we can prove in Lean? Valid proofs in Lean are objects we can construct who have math statements as their type, given Lean's built-in objects (i.e. axioms). For example, if we can construct an object of the type

```
∀(n: Nat), ∀(a: Nat), ∀(b: Nat), ∀(c: Nat), a^(n+3) + b^(n+3) ≠
c^(n+3)
```

then we have *proven* Fermat's last theorem.

Frequently asked questions

Can you ever prove `3 ≠ 3`?

The answer is a bit scary. The answer *should* be "no". But to be honest, we don't know for sure. In a proof system, this is called "consistency": You should never be able to prove both the statements `P: Prop` and `¬P: Prop`. But in fact, we don't know if the axioms that are built-in to Lean are consistent. And by a result of Godel, in any proof system, we can never prove that our system is consistent. But so far, we haven't found any contradictions... so it shouldn't be possible to construct an object of type `3 ≠ 3`. However, if we can, then the proof system that Lean is based on is inconsistent, which would surprise many mathematicians.

Are there things you can't prove in Lean?

For some `P: Prop`, if it is impossible to construct an object of type `P`, and *also* impossible to construct an object of type `P`, then we say `P` is undecidable in this proof system. It might be that statements like "BQP ≠ NP" and the Riemann hypothesis are not only hard to prove, but simply not possible to prove with the axioms that are built into Lean. From Godel's incompleteness theorem, we *know* there are undecidable statements in Lean. We can only hope that we aren't wasting our time trying to prove an undecidable statement.

What is `tactic mode`?

When constructing proofs (i.e. objects who have a math statement as their type), we can use "tactic" mode. This allows you to prove theorems in a way more similar to how mathematicians prove things: step-by-step, from hypotheses to goal. Use `by` to initialize tactic mode.

```
def all_eq_tactic_mode : ∀ (x : Nat), x = x := by
  intro _
  rfl
```

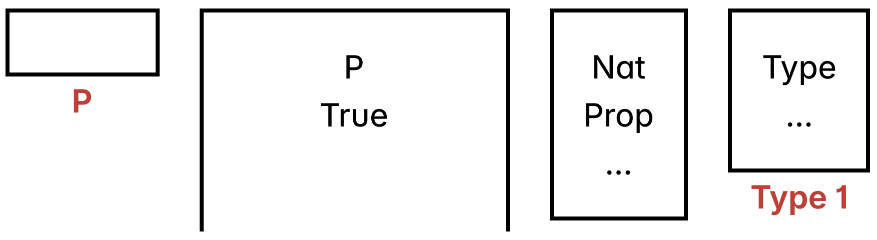
Lean will convert this tactic-style proof into an object of type `∀ (x : Nat), x = x`; i.e. proofs of the statement `∀ (x : Nat), x = x`. You can use whichever mode you prefer. In general I find tactic mode much easier to reason about. But you may feel differently, and that's ok.

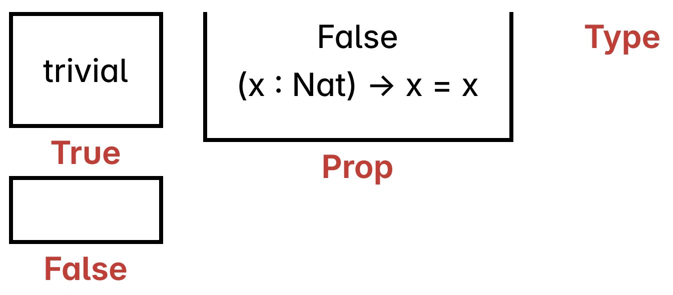
If you put your cursor on different parts of the proof, the Lean compiler will tell you the "state" of the proof: what are the current hypotheses, and what is the goal.

Isn't there a `True` and `False` in Lean?

Yes, they are defined in the language for convenience. *However, be warned, I think they are a bit of a red herring.* We colloquially say a math statement `P: Prop` is proven in Lean if there is an object of type `P`. In Lean, `True` and `False` are both math statements:

```
True: Prop
False: Prop
```





There is a built-in object `trivial` which has type `True`. (A proof of `True` is a tautology...) There are no objects with type `False`, and it *should* be impossible to construct one.

Of course, for any proposition `Q`, the statement `Q → True` seems logically correct. The relevant math statement is:

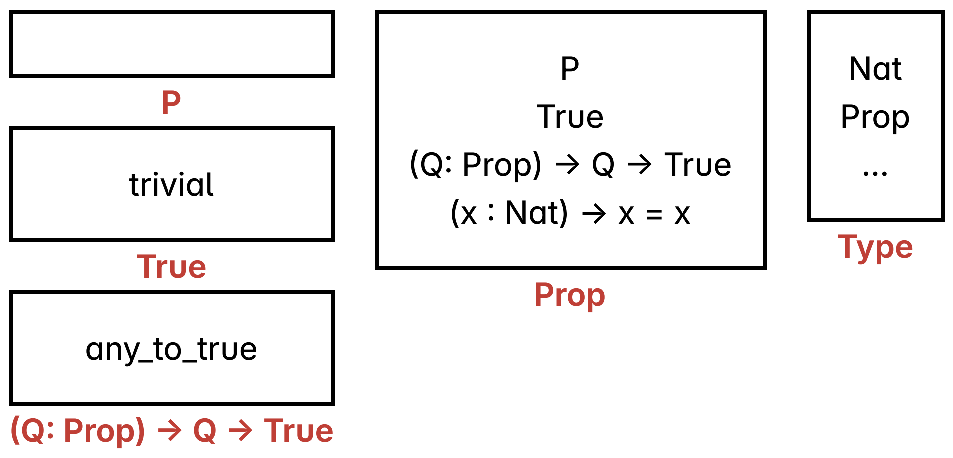
```
∀ (Q : Prop) : Q → True : Prop
```

which is equivalent to

```
(Q: Prop) → Q → True : Prop
```

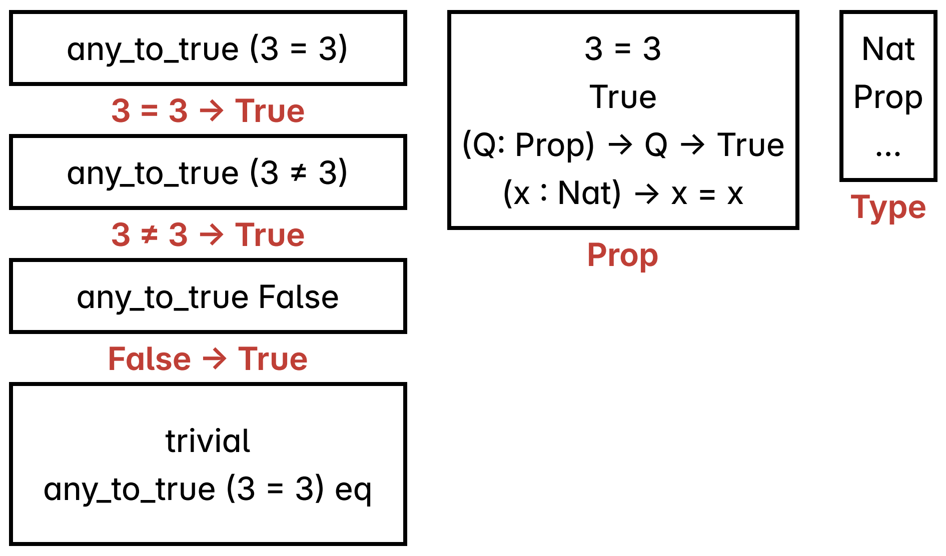
We *prove* this statement by constructing an object with that type:

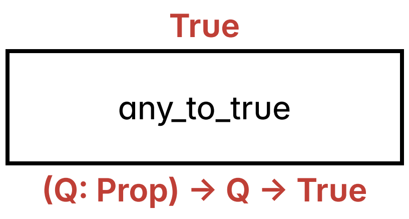
```
def any_to_true : ∀ (Q : Prop), Q → True :=
  fun (Q: Prop) ⇒ (fun (q: Q) ⇒ trivial)
```



As usual, `any_to_true` is a function that inputs an object of type `Prop`. We can use it to *prove* other statements (i.e. make objects that have a math statement as their type.)

```
any_to_true (3 = 3) : 3 = 3 → True
any_to_true (3 = 3) eq : True
any_to_true (3 ≠ 3) : 3 ≠ 3 → True any_to_true False : False → True
```



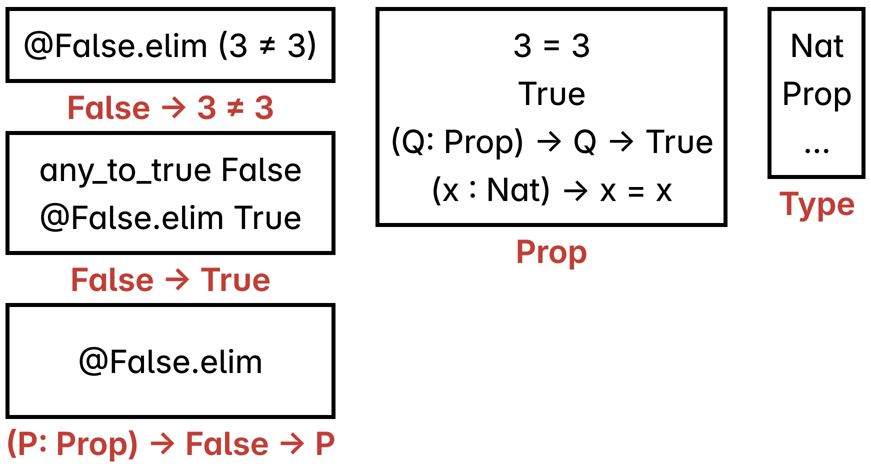


Remember, a statement `P: Prop` is *proven* in Lean if there is an object of type `P`. By contrast, objects of type `True` in Lean are only tautologies.

Notice that we can prove statements of the form `P → Q` even if there is no proof of `P`. For example, `any_to_true (3 ≠ 3)` is a *function* that, if there was a proof of `3 ≠ 3`, it would output a proof of `True`. The object `any_to_true (3 ≠ 3)` is a proof of the implication `(3 ≠ 3) → True`, which we know holds vacuously.

Similarly, for any proposition `P: Prop`, `@False.elim P` is an object of type `False → P`.

```
@False.elim : (P: Prop) → (False → P)
@False.elim (3 ≠ 3) : False → 3 ≠ 3
```

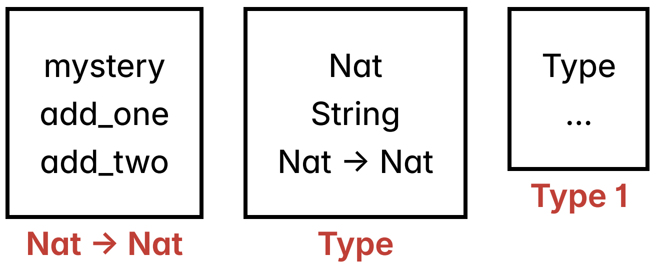


Do we need more than one proof of a statement?

This hints at something important that makes Lean run quickly.

Functions with different behavior might have the same type; say, `Nat → Nat`.

```
-- these do different things, but have the same type
def add_one : Nat → Nat := fun x ⇒ x + 1
def add_two : Nat → Nat := fun x ⇒ x + 2
```



However, we only need one *proof* of a statement (i.e. object with type) `P: Prop` to know that `P` is proven in Lean. With respect to *proving* `P`, Lean can *treat all objects of type `P` in the same way.* So, for math statements in particular, Lean can remove inessential details of proofs (i.e. objects who have a math statement as a type). This is an axiom of Lean:

```
axiom propext : ∀ {a b : Prop}, (a ↔ b) → a = b
```

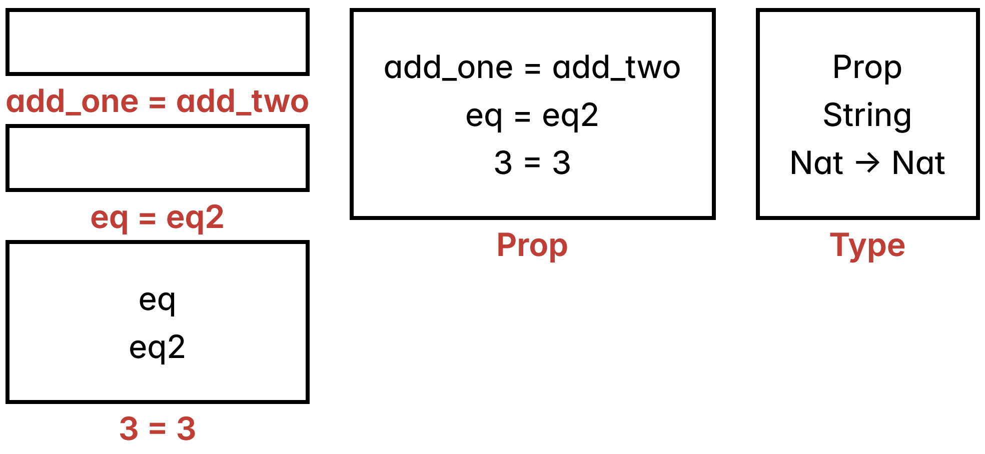
Let's construct a new proof of `3 = 3` (here I am using tactic mode):

```
def eq2 : 3 = 3 := by
  symm
  apply eq
```

Then we can consider the following two propositions:

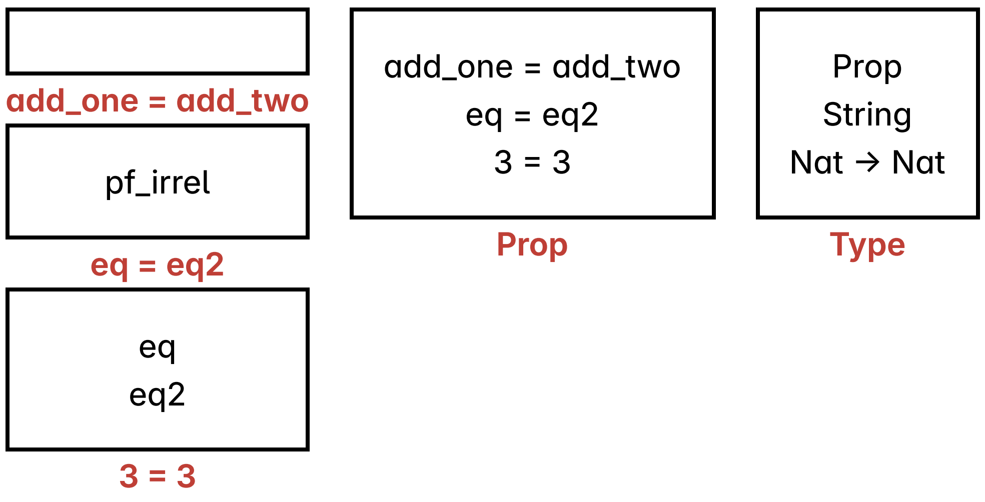
```
add_one = add_two : Prop
eq = eq2 : Prop
```

The first considers if the two functions are equal. The second considers if two objects of type `3 = 3` are equal.



The proposition `add_one = add_two` should not be true; they are different! (So, we shouldn't be able to create an object with this type.) But we *can* create an object of type `eq = eq2`:

```
def pf_irrel : eq = eq2 := rfl
```



This design decision (called "proof irrelevance") in Lean makes the compiler faster, since from the perspective of *proving*, one proof of a math statement is as good as the next, as long as they're both correct.

In other words, all proofs of `True` are equal in Lean. All proofs of the same math statement are equal in Lean.

Other people use the word `theorem` in Lean. What's that?

In this note, we created *proofs* with the keyword `def`, since proofs are objects who have a math statement as a type. However, you can also create *noncomputable* proofs with `theorem`:

```
theorem all_eq_thm : ∀ (x : Nat), x = x := fun (x: Nat) ⇒ rfl
```

This tells Lean that you don't need to use `all_eq_thm` just for proving, not for computation (hence, *noncomputable*). Because of the "proof irrelevance" concept (see the previous section), the Lean compiler can optimize , and avoid storing implementation details that are unnecessary once the object has been created.

You can also use `example` to create unnamed *noncomputable* proofs; some people find this more convenient.

```
example : ∀ (x : Nat), x = x := fun (x: Nat) ⇒ rfl
```

What about `sorry`?

`sorry` is a "cheat code" in Lean. With sorry, you can prove anything:

```
-- this shouldn't be possible!
def impossible : 3 ≠ 3 := sorry
```

But the Lean compiler *knows* you cheated. (It will underline `impossible` in yellow.) The Lean compiler keeps track of what objects are made with *sorry*. We think of the statement `P: Prop` as proven in Lean only if an object is constructed of type `P`, *and* the construction is sorry-free.

That being said, `sorry` can be useful when you want to prove something, but don't know how just yet. In mathematics, you might do this by saying "assume *this* is true; I'll fill in the details later..." In computer science, you might do this by saying "I want *this* method, but I'll implement it later".

```
-- Trust me, I have a proof, but it's too long for this note ...
def pf_of_flt : ∀(n: Nat), ∀(a: Nat), ∀(b: Nat), ∀(c: Nat),
                a^(n+3) + b^(n+3) ≠ c^(n+3)
    := sorry
```

So, it's a cheat code, but it can be useful during the process of formalizing a proof. And don't think you can fool the Lean compiler -- it knows you're using `sorry`.

If you have other questions, or feel like this document could be improved, please let Kunal know!