Doing math the Lean way

Some things I’ve found to be useful to know

Kyle Miller — University of California, Berkeley

Berkeley Lean Seminar, June 26, 2020

Motivation

There is *a lot* of stuff going on in Lean and mathlib.

Much of it is undocumented.

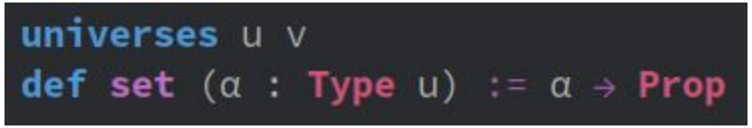
It merely *resembles* math as we know it (type theory ≠ set theory).

So: here are some notes from one fellow traveler to another.

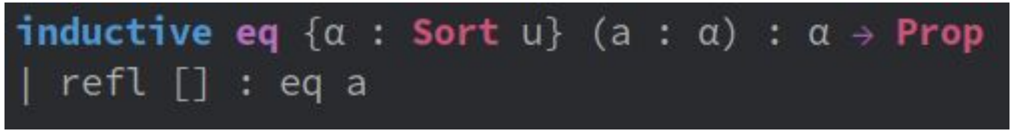
Resources

- The Lean Reference Manual
- Theorem Proving in Lean
- the Lean community Zulip
  - both by asking questions and the chat history
- **Following the definitions deep into mathlib**

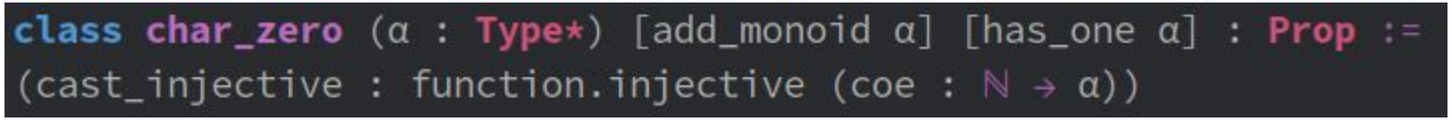
Universes



```
universes u v
def set (α : Type u) := α → Prop
```



```
inductive eq {α : Sort u} (a : α) : α → Prop
| refl [] : eq a
```



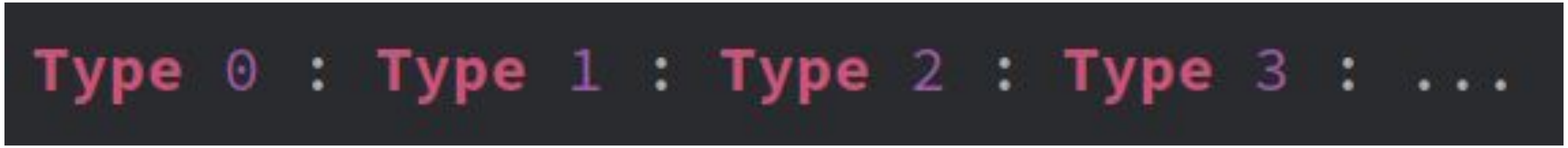
```
class char_zero (α : Type*) [add_monoid α] [has_one α] : Prop :=
(cast_injective : function.injective (coe : ℕ → α))
```

Universes

Everything in Lean is a term of precisely one type.

Types, too!

The *type universes* are an ℕ-indexed family of types



```
Type 0 : Type 1 : Type 2 : Type 3 : ...
```

Type 0 has the synonym **Type** for convenience.

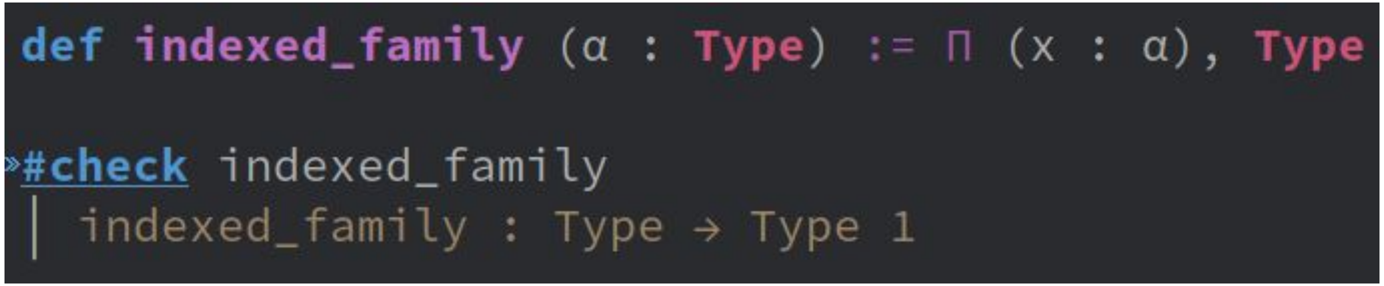
Universes

Universes solve Russell-style paradoxes:

Objects that contain stuff in a particular universe live in a higher universe.

The universe **Type** is where most of math happens (sorry logicians).

You might use higher universes for indexed families.



```
def indexed_family (α : Type) := Π (x : α), Type

#check indexed_family
 indexed_family : Type → Type 1
```

Universes in mathlib

Universe polymorphism is pervasive in mathlib

- If a statement generalizes to higher universes, why not?
- But higher universes are of limited practical use (for a mathematician)

Declare universe indices with

```
universes u v w
```

Use them like

```
{α : Type u}
```

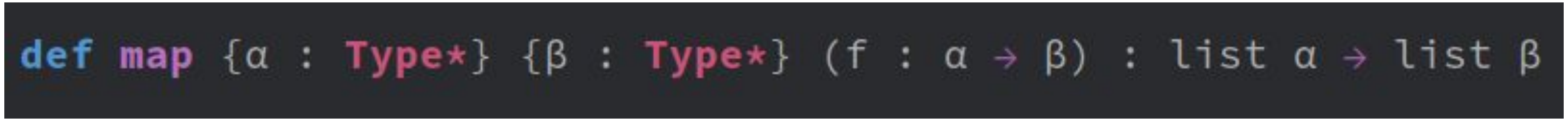
```
Type (u + 1)
```

```
Type (max u v)
```

The universe variables represent natural numbers, *not* the universe itself.

Universes in mathlib

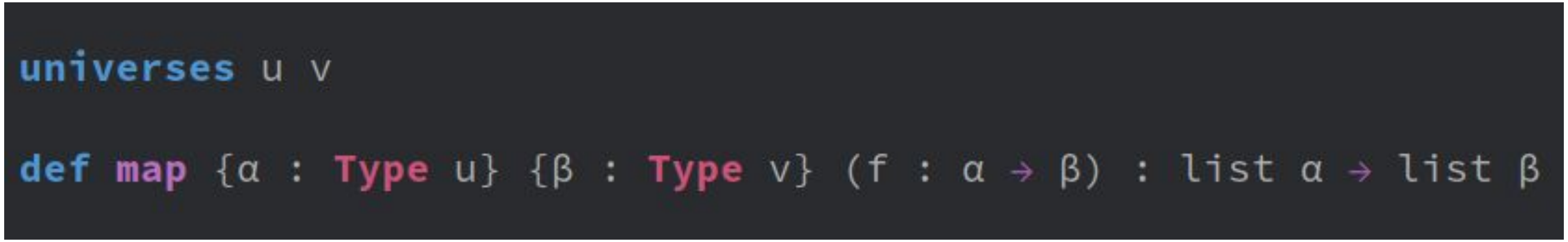
Another notation you'll see is



```
def map {α : Type*} {β : Type*} (f : α → β) : list α → list β
```

The star indicates Lean should create a fresh universe variable for you.

Equivalently,

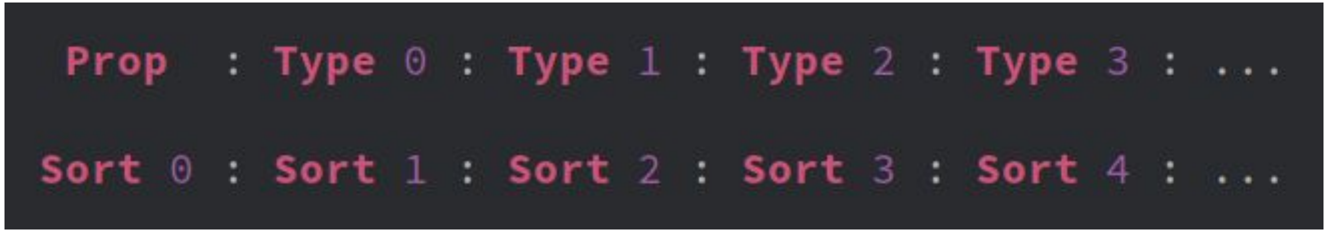


```
universes u v

def map {α : Type u} {β : Type v} (f : α → β) : list α → list β
```

An additional *very* special universe

There is a universe below Type, called **Prop**, the type of all propositions.

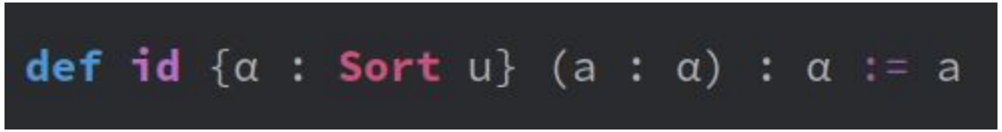
It behaves differently from other universes, so it is not included in **Type***



```
 Prop   : Type 0 : Type 1 : Type 2 : Type 3 : ...
Sort 0 : Sort 1 : Sort 2 : Sort 3 : Sort 4 : ...
```

Type u = **Sort** (u + 1).

There is a **Sort***, too.



```
def id {α : Sort u} (a : α) : α := a
```

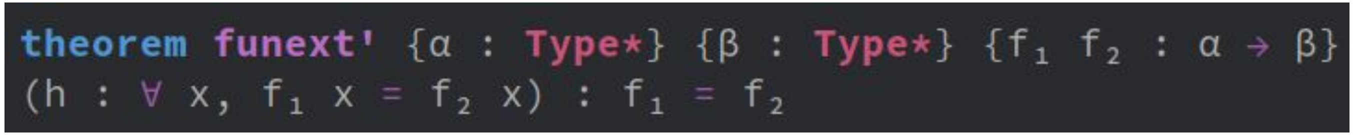
What is special about **Prop**?

Recall: in math equality tends to be *extensional*

this means a thing ***is*** its observable properties

ex. sets by their elements

ex. functions by their evaluations

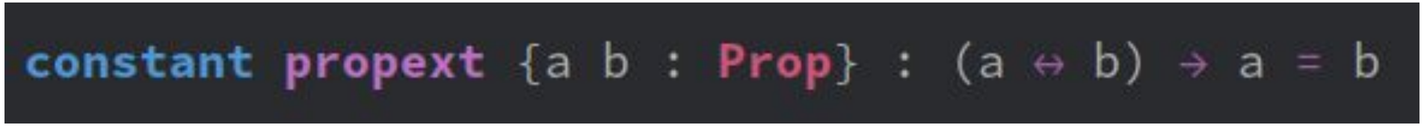


```
theorem funext' {α : Type*} {β : Type*} {f₁ f₂ : α → β}
(h : ∀ x, f₁ x = f₂ x) : f₁ = f₂
```

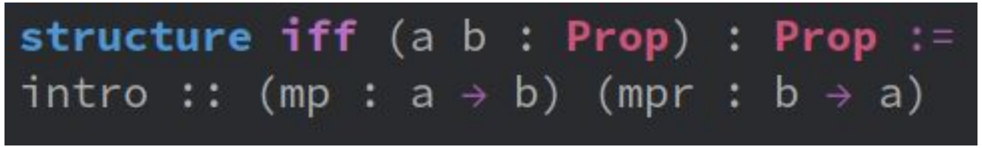
What is special about **Prop**?

In Lean, there is an axiom that propositions are extensional:

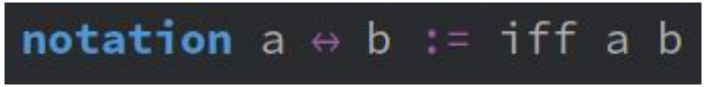
A proposition *is* what it’s equivalent to.



```
constant propext {a b : Prop} : (a ↔ b) → a = b
```



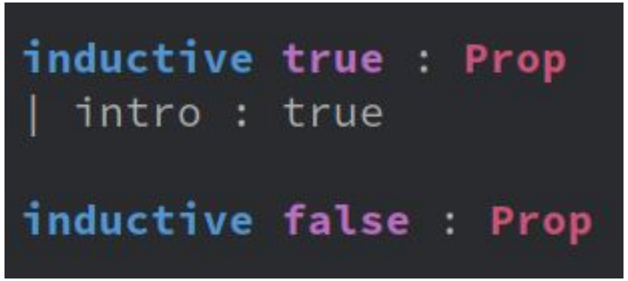
```
structure iff (a b : Prop) : Prop :=
intro :: (mp : a → b) (mpr : b → a)
```



```
notation a ↔ b := iff a b
```

What is special about **Prop**?

Consequence: every proposition equal to one of the following two:



```
inductive true : Prop
| intro : true

inductive false : Prop
```

← provable, has the proof `intro`

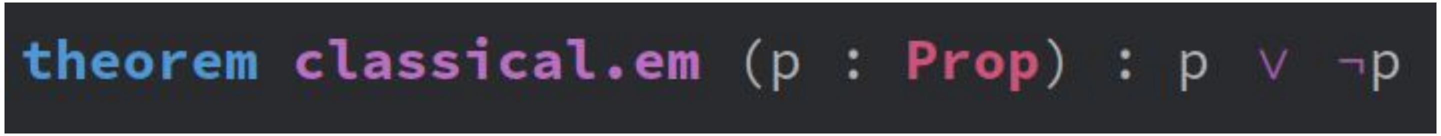
← not provable, has no terms by definition

But we don't necessarily know which it is! That's math!

Aside: the Law of the Excluded Middle

Lean has function extensionality and propositional extensionality.

Together, Diaconescu's theorem applies, giving



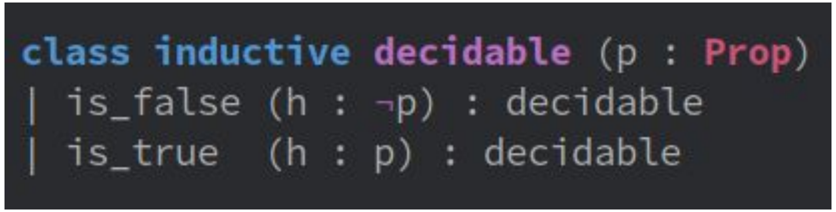
```
theorem classical.em (p : Prop) : p ∨ ¬p
```

This is what it means for every proposition to be equal to `true` or to `false`

Aside: decidability

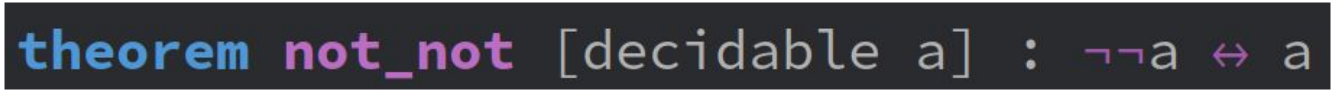
There is no function that can take an arbitrary **Prop** and tell you, definitively, whether it is true or false.

Lean has a facility for a partially defined function **Prop** → {true, false}, so to speak.



```
class inductive decidable (p : Prop)
| is_false (h : ¬p) : decidable
| is_true  (h : p) : decidable
```

Double negation elimination does *not* follow from LEM!

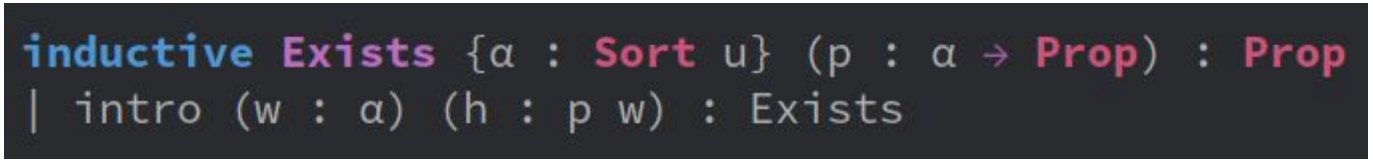


```
theorem not_not [decidable a] : ¬¬a ↔ a
```

A consequence of propext

Let's look at proofs of existence: `∃ x, p x`

This is shorthand for `Exists p`, with



```
inductive Exists {α : Sort u} (p : α → Prop) : Prop
| intro (w : α) (h : p w) : Exists
```

So, `Exists.intro w h` would be a proof if `h` is a proof of `p w`

Note. Exists is a **Prop** — if it's true, it is equal to true, so has exactly one proof.
Hence, `Exists.intro w h = Exists.intro w' h'`

A consequence of propext

This means you cannot define a function

```
def some {α : Sort u} {p : α → Prop} (h : ∃ x, p x) : α
```

by reaching into a proof of existence.

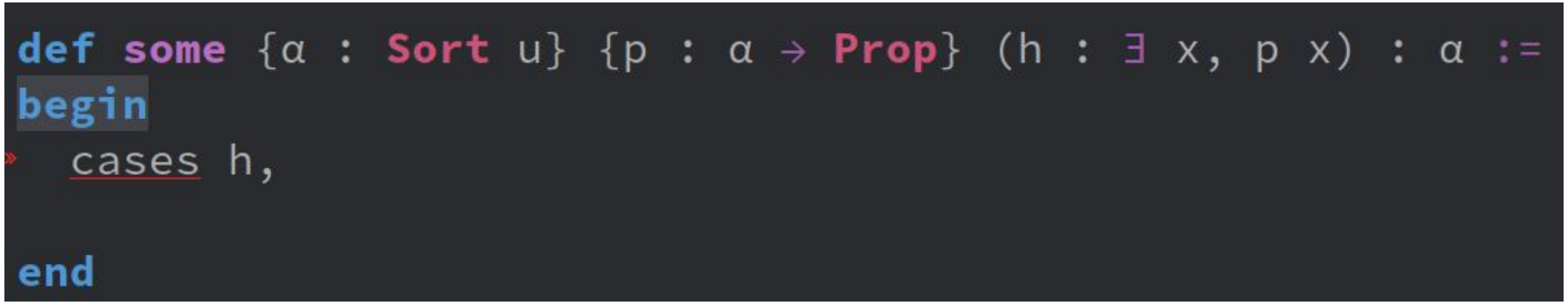
This is because of the substitution principle:

> *Substituting in equal things yields equal things.*

We would need this to be true:

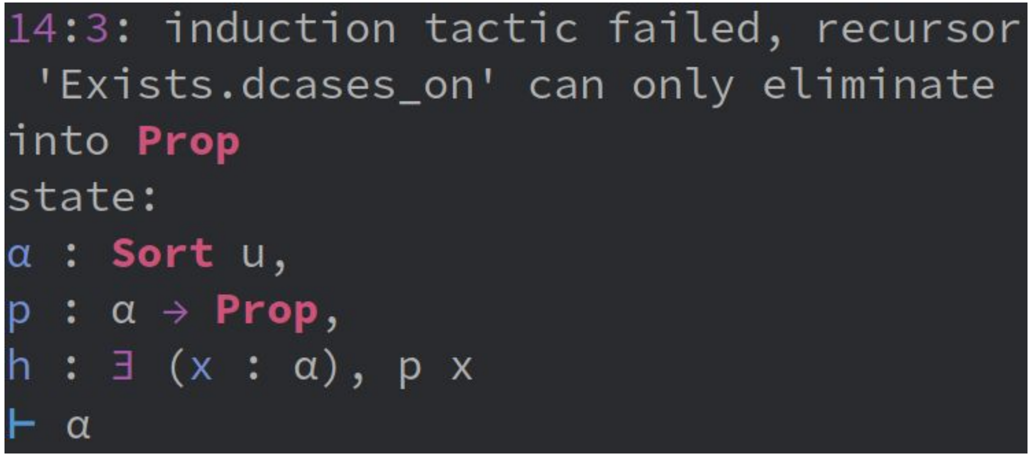
```
example {α : Sort u} (p : α → Prop) (h₁ h₂ : ∃ x, p x) :
some h₁ = some h₂
```

Let’s try anyway



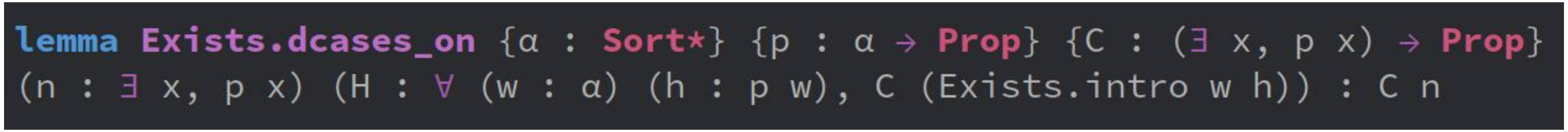
```
def some {α : Sort u} {p : α → Prop} (h : ∃ x, p x) : α :=
begin
  cases h,

end
```

Lean knows the target goal is any **Sort**, which includes things in **Type**. Since Exists is a **Prop**, the generated *recursor* is only allowed to construct a **Prop**.



```
14:3: induction tactic failed, recursor
 'Exists.dcases_on' can only eliminate
into Prop
state:
α : Sort u,
p : α → Prop,
h : ∃ (x : α), p x
⊢ α
```

Let's try anyway



```
lemma Exists.dcases_on {α : Sort*} {p : α → Prop} {C : (∃ x, p x) → Prop}
(n : ∃ x, p x) (H : ∀ (w : α) (h : p w), C (Exists.intro w h)) : C n
```

Lean knows the target goal is any **Sort**, which includes things in **Type**. Since Exists is a **Prop**, the generated *recursor* is only allowed to construct a **Prop**.

This is generated by the "equation compiler" from the inductive definition.

Prop vs **Type**

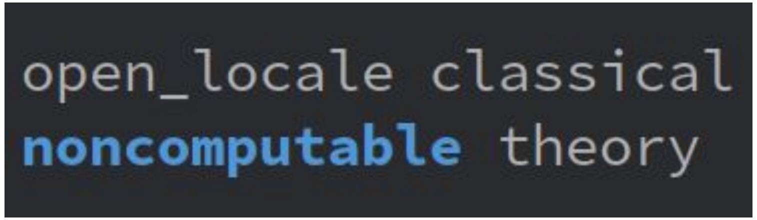
By default, everything in **Type*** that you refer to must be constructible.

Prop is sort of a escape hatch where we can work with the truth of a statement, even if there might be no way of computing it.

Lean takes great pains to keep you from using “illegally obtained” data, but you are free to use it in proofs supporting a construction.

Doing classical logic

That said, you can avoid worrying about any of this by declaring

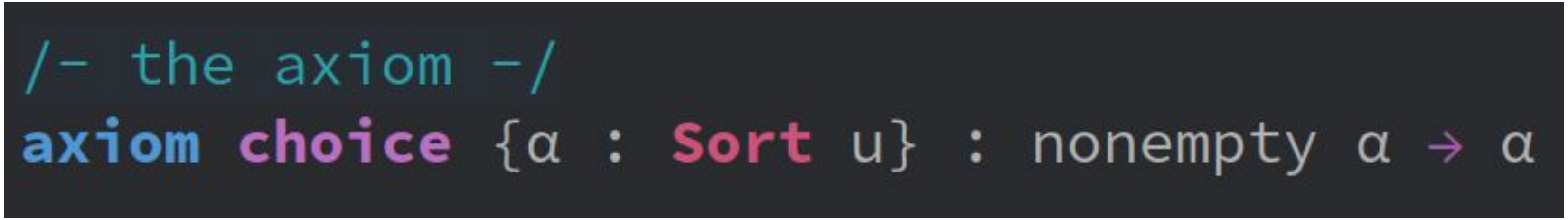


```
open_locale classical
noncomputable theory
```

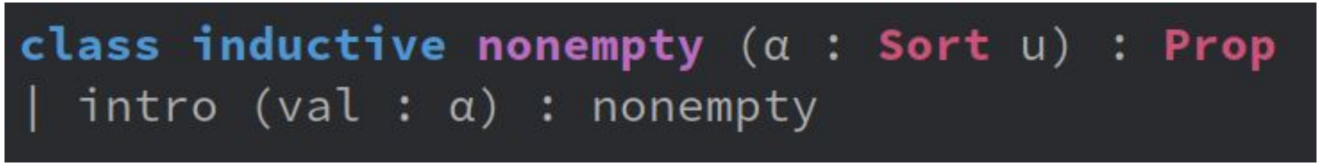
This causes Lean to pretend everything is decidable and to not worry if constructions are actually constructible by automatically tagging them with **noncomputable** for you.

The axiom of choice

Lean’s version of the axiom of choice is that there is a consistent way of choosing an term from every type that has *propositionally* been proven to have a term.



```
/- the axiom -/
axiom choice {α : Sort u} : nonempty α → α
```

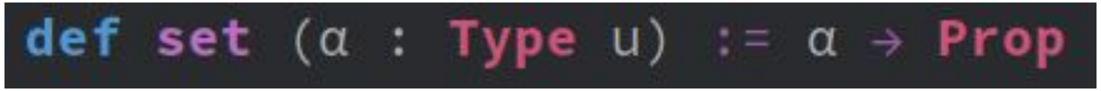


```
class inductive nonempty (α : Sort u) : Prop
| intro (val : α) : nonempty
```

With this, you can easily define the `some` function from before: every statement of nonemptiness has a canonical proof by `choice`.

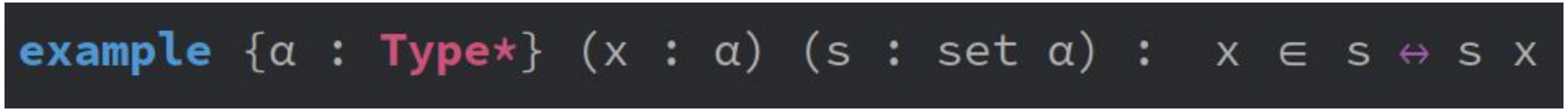
Sets and subtypes

A “set” in Lean is not what you would expect.



```
def set (α : Type u) := α → Prop
```

It is an indicator function on a type.



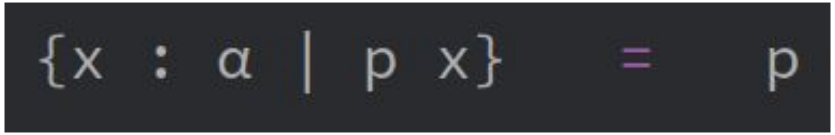
```
example {α : Type*} (x : α) (s : set α) :  x ∈ s ↔ s x
```

Hence: a set is a ***subset*** of a type

Subsets as indicator functions

We are used to describing subsets with indicator functions.

Lean gives us special syntax for this:



```
{x : α | p x}    =    p
```

Subsets as indicator functions

What problem is this solving?

In Lean, every term belongs to precisely one type.

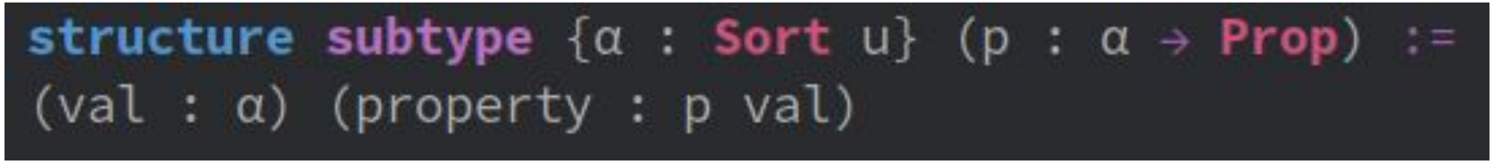
If a subset were a type, then its terms would not be terms of the total type.

Elements of a `set` are terms of the total type.

Downside: a set is ***not*** a type itself. It’s a term of a function type.

Subtypes

There is a way to convert a set into a bona fide type



```
structure subtype {α : Sort u} (p : α → Prop) :=
(val : α) (property : p val)
```

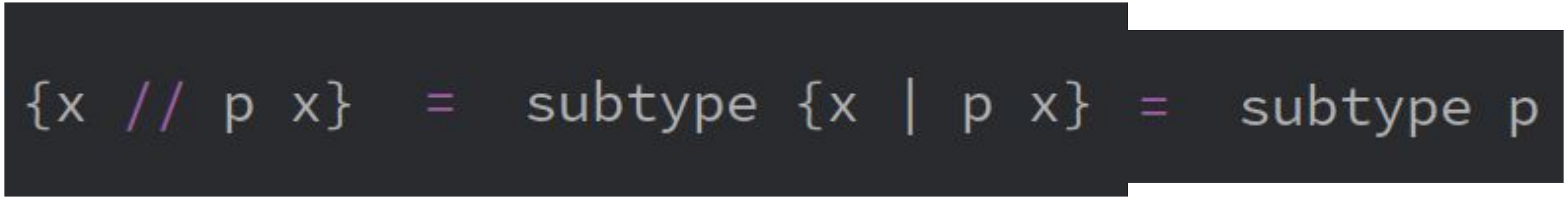
If `s` is a set, then `subtype s` is a type whose terms are elements of `s`

This relies on propositional extensionality to work out.

```
    subtype.val : subtype s → α
```

is an injection that gives precisely the elements of `s`.

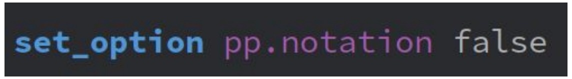
Subtypes



```
{x // p x}  =  subtype {x | p x} =  subtype p
```

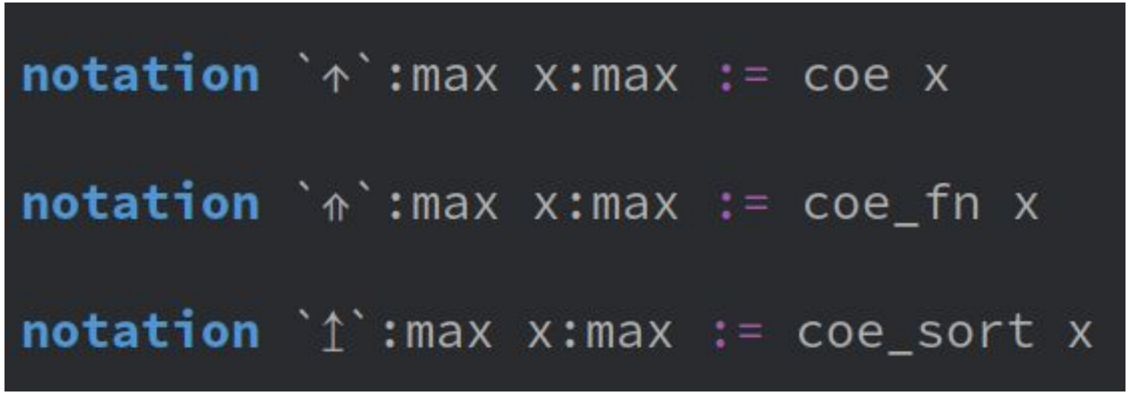
Lean has a concept of *coercions*, indicated by upward arrows. Lean will automatically coerce sets to subtypes in certain locations.

- As the domain of a function
- When assigning to a variable that is supposed to be a **Type**.



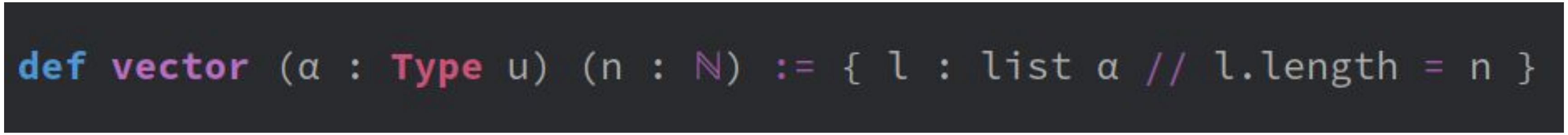
Use `set_option pp.notation false` to see the underlying function names for arrows.

Coercions



```
notation `↑`:max x:max := coe x

notation `⇑`:max x:max := coe_fn x

notation `↥`:max x:max := coe_sort x
```

Example: “vectors”



```
def vector (α : Type u) (n : ℕ) := { l : list α // l.length = n }
```

These are homogeneous lists of a particular length.

It is a list along with a proof that it is of a given length.