More on Types

So far, we've seen how to *work* with types, and we want to give a somewhat more robust account of their theory.

The hierarchy

We've seen that propositions are all types of a certain kind `Prop`; and that ℕ or ℝ are types of a different kind (indeed, both have more than one term!), called `Type`.

There is actually a whole hierarchy of kinds of types

```
Prop : Type 0 : Type 1 : ... : Type n : ...
```

So, `Prop` is a *term* of the *type* `Type 0`, itself a *term* of the type `Type 1`, etc.

Lean shortens `Type 0` to `Type`, omitting the index. It is where most known mathematical objects (like ℕ, ℤ, ℂ, etc) live. `Sort *` is either `Type (*+1)` or `Prop` in the sense that `Sort 0 = Prop`, `Sort 1 = Type 0`, `Sort 37 = Type 36`...

⌘

Dependent types

The type theory that Lean is built upon axiomatises the existence of certain constructions that allow to construct new types out of known ones.

+++ Function types
Given two types `X` and `Y`, it exists the type `X → Y`. Its terms are written

```
λ (x : X) ↦ (f x) : Y
```

or

```
fun (x : X) ↦ f x
```

These terms can be interpreted as functions from `X` to `Y`, in the sense that if x_0 `: X` and `f : X → Y` then `f x` is a term in `Y`.

⌘
+++

+++ Π-types and Σ-types

What is the type of

```
fun (α : Type) ↦ (fun x : α ↦ x)
```

namely the assignment sending a type to its identity function?

The "problem" is that for every element in the domain, the image lies in a different place: there is no "codomain".

It belongs to the Π-type (called pi-type, or forall-type, or *dependent* product)

```
Π (α : Type), α → α
∀ (α : Type), α → α
(α : Type) → (α → α)
```

More generally, given a type `A` (where `A = Sort u` is allowed), seen as an index set, and a function `I : A → Sort v`, seen as an "indexing family",

```
Π (a : A), I a
∀ (a : A), I a
(a : A) → I a
```

is the type whose terms are collections `(a, xₐ)` for `a` spanning `A` and where `xₐ : I a`. These are written `λ a : A ↦ xₐ`, or `fun a : A ↦ xₐ`.

- If you've got a geometric intuition, this looks very much like a fibration, where `A` is the base and `I a` is the fiber above `a : A`.
- As the `λ` or `fun` notation suggest, `X → Y` is a special case of a Π-type, where `I : X → Sort v` is the constant function `fun x ↦ I x = Y`.

Similarly, terms of the Σ-type

```
Σ (a : A), I a
(a : A) × I a
```

are pairs `⟨a, xₐ⟩` where `xₐ : I a` (for technical reasons, we need here that `A : Type u` and that `I : A → Type v`: if you really want to use `Sort` use Σ', or ×'). Type `×` (*resp.* `×'`) as `\x` (*resp.* `\x'`).

- These constructions of types that depend on terms give the name "dependent type theory" (or "dependent λ-calculus") to the underlying theory.
- From the hierarchy point of view, if `A : Sort u` and `I : A → Sort v`, then `(a : A) → I a` and `(a : A) ×' I a` are types in `Sort (max u v)` *except* when `v = 0` in which case both are still in

`Prop`. This is the "impredicativity" of the underlying type-theory.

⌘

+++

+++ ∀ and ∃

Universal quantifier

Consider the type

```
∀\; n ∈ ℕ, ∃\; p\, \text{ prime such that } n < p
```

It is a Π-type, with `I : ℕ → Prop` being `I := fun n ↦ ∃ p prime, n < p`.

Euclid's proof is a *term* of the above type.

- You can *prove* a ∀ by `intro`ducing a variable via `intro x`, and by proving `P x`.
- If you have `H : ∀ x : α, P x` and also a term `y : α`, you can specialise `H` to `y`:

  ```
  specialize H y (:= P y)
  ```

- If the goal is `⊢ P y`, you might simply want to do `exact H y`, remembering that implications, `∀` and functions are all the same thing.

Existential quantifier

A statement

```
∃\; n ∈ ℕ, n^2+37 · n < 2 ^ n
```

lives in `Prop`, so you *cannot extract a witness from an existential proof*, unless you want to extract a term in some other `P : Prop`.

But,

- To prove `∃ x, P x`, you first produce `x`, and then prove it satisfies `P x`: once you have constructed `x`, do `use x` to have Lean ask you for `⊢ P x`.
- If you have `H : ∃ x, P x`, do `obtain ⟨x, hx⟩ := H` to obtain the term `x` together with a proof that `P x`.

⌘

+++

+++ `¬` and Proofs by contradiction

- Given `P : Prop`, the *definition* of `¬ P` is

  P → `False`

- The `exfalso` tactic changes *any* goal to proving `False` (useful if you have an *assumption* `... → False`).

- Proofs by contradiction, introduced using the `by_contra` tactic, require you to prove `False` assuming `not (the goal)`: if your goal is `⊢ P`, typing `by_contra h` creates

  ```
  h : ¬ P
  ⊢ False
  ```

The difference between `exfalso` and `by_contra` is that the first does not introduce anything, and forgets the actual goal; the second negates the goal and asks for `False`.

- `contrapose` is the tactic that changes a goal `P → Q` to `¬ Q → ¬ P`.

`⌘`
+++

Inductive Types

So far, we

- met some abstract types `α, β, T : Type`, and variations like `α → T` or `β → Type`;
- also met a lot of types `P, Q, (1 = 2) ∧ (0 ≤ 5) : Prop`;
- struggled a bit with `h : (2 = 3)` *versus* `(2 = 3) : Prop`;
- also met `ℕ`, `ℝ`...

How can we *construct* new types, or how have these been constructed? For instance, `ℝ`, or `True : Prop` or the set of even numbers? Using **inductive types**.

+++ Perspectives

- *Theoretical*: this is (fun & interesting, but) beyond the scope of this course: it is very much discussed in the references.
- *Practical*: think of `ℕ` and surf the wave. It has two **constructors**: the constant `0 : ℕ` and the function `succ : ℕ → ℕ`, and every `n : ℕ` is of either form. Moreover, it satisfies **induction**/recursion.
  +++

For example

```
inductive NiceType
  | Tom : NiceType
  | Jerry : NiceType
  | f : NiceType → NiceType
  | g : ℕ → NiceType → NiceType → NiceType
```

constructs the "minimal/smallest" type `NiceType` whose terms are

1. Either `Tom`;
2. Or `Jerry`;
3. Or an application of `f` to some previously-defined term;
4. Or an application of `g` to a natural and a pair of previously-defined terms.

For example, `f (g 37 Tom Tom) : NiceType`.

- Every inductive type comes with its *recursor*, that is automatically constructed by Lean: it builds dependent functions *out of the inductive type being constructed* by declaring the value that should be assigned to every constructor.

- In order to

    1. construct terms of type `NiceType` you can use the ... *constructors*!;
    2. access terms of type `NiceType` (in a proof, say), use the tactic `cases` (or or `rcases`): the proofs for Tom and for Jerry might differ, so a case-splitting is natural.

⌘

Every type in Lean is either a function type, a quotient type or an inductive type

By *every* we mean `True`, `False`, `Bool`, `P ∧ Q`... every! And among those,

If and only if statements:

`↔ : Prop → Prop → Prop`, giving rise to `P ↔ Q`: it is an inductive type (of course), with

- two parameters (`P` and `Q`)
- one constructor, itself made of
    - two fields: `Iff.mp : P → Q` and `Iff.mpr : Q → P`

Calling `#print Iff` produces:

```
structure Iff (P Q : Prop) : Prop
    number of parameters: 2
    fields:
        Iff.mp : P → Q
        Iff.mpr : Q → P
    constructor: Iff.intro {P Q : Prop} (mp : P → Q) (mpr : Q → P) : P ↔ Q
```

An equivalence can be either *proved* or *used*. This amounts to saying that:

- A goal `⊢ P ↔ Q` can be broken into the goals `⊢ P → Q` and `⊢ Q → P` using `constructor`: indeed, to prove `⊢ P ↔ Q` amounts to creating *the unique term* of `P ↔ Q` which has two constructors;
- The projections `(P ↔ Q).mp` (or `(P ↔ Q).1`) and `(P ↔ Q).mpr` (or `(P ↔ Q).2`) are the implications `P → Q` and `Q → P`, respectively. These are the two "components" of the term in `P ↔ Q`.

⌘

Structures

+++ Why did `#print Iff` begun with `structure` rather than with `inductive`?
Because it is a *structure* (with two fields):

> **Definition**
>
> A structure is an inductive type with a unique constructor.
>
> +++

Indeed, among inductive types (*i. e.* all types...), some are remarkably useful for formalising mathematical objects: those that *bundle* objects and properties together. So, we give them a different name.

As an example, let's see what a Monoid is:

```
structure (M : Type*) Monoid where
    | mul : M → M → M                        -- denoted *
    | one : M                                -- denoted 1
    | mul_assoc (a b c : M) : a * b * c = a * (b * c)
    | one_mul (a : M) : 1 * a = a
    | mul_one (a : M) : 1 * 1 = a
```

- Two of these fields are terms in types of kind `Type *`;
- three of them are terms in types of kind `Prop`;
- we often call a structure having constructor fields both in `Type *` and in `Prop` a *mixin*.

So,

- a *monoid structure* on `M` is a collection `⟨*, 1, mul_assoc, one_mul, mul_one⟩`
- a term of a monoid is just a term of it! The monoid is a type, so it comes with its terms even if it has more structure, which is encoded in a term `str : Monoid M`.

Another extremely useful structure is the equivalence (thought of as an isomorphism):

```
structure (α β : Type*) : Equiv α β where
    | toFun : α → β
    | invFun : β → α
    | left_inv : LeftInverse self.invFun self.toFun
    | right_inv : RightInverse self.invFun self.toFun

infixl:25 " ≃ " => Equiv
```

⌘