Propositional Calculus

For this and all other classes, excellent material can be found in

- Theorem Proving in Lean 4, by J. Avigad, L. de Moura, S. Kong, S. Ullrich
- Mathematics in Lean, by J. Avigad and P. Massot

Let's see an example before moving on

⌘

Types

Lean is based on (dependent) type theory. It is a very deep foundational theory, and we will not dig into the details of this (which I am certainly *not* an expert of).

We'll content ourselves by using it as a replacement for the foundational theory underneath "usual" mathematics, replacing sets by **types as fundamental objects**.

- We do not define *what* types are. They *are*.

Types contain *terms*: we do not call them elements. The notation `x ∈ A` is **not** used, and reserved for sets (that will appear, at a certain point). The syntax to say that `t` is a term of the type `T` is

```
t : T
```

and reads "the type of `t` is `T`".

+++ Sets = Types?
No! Of course, you can bring over some intuition from basic set-theory, but the crucial difference is that **every term has a unique type**.

So,

```
t : T ∧ t : S
```

is certainly *false*, unless `T = S`. In particular, `1 : ℕ` and `1 : ℤ` shows that the two `1`'s above are **different**.
+++

Prop and the hierarchy

There is a class of particular types, called *propositions*. This class is denoted `Prop`.

- Types in the class `Prop` represent propositions (that can be either true or false). So, `(2 < 3) : Prop` and `(37 < 1) : Prop` are two *types* in this class, as is `(A finite group of order 11 is cyclic)`.

+++ Two crucial examples
`True : Prop` and `False : Prop`.

+++

- Key point: if `p : P` then either `p` has not term at all ("`p` is false"), or `p` has a unique term `h` (`h` is "a witness that `p` is true"; or a **proof** of `p`).

+++ Other types
There is actually a whole hierarchy 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: they are terms of this type.

⌘
+++

Tactics

To prove a proposition `p : Prop` boils down to producing a/the term `hp : p`.

This is typically done by

1. Producing another type `q : Prop` that we know to be true, so that we have a term `hq : q`.
2. Producing a function `f : p → q`("an implication").
3. Defining `hp := f hq`.

Of course, this is too painful: to simplify our life, or to build more convoluted implications, we use *tactics*.

+++ `intro`, `exact`, `apply` and `rfl`

- Given an implication `p → q`, the tactic `intro hp` introduces a term `hp : p`.

- On the other hand, given a term `hq : q` and a goal `⊢ q`, the tactic `exact hq` closes the goal, instructing Lean to use `hq` as the sought-for term in `q`.

- `apply` is the crucial swiss-knife for *backwards reasoning*: in a situation like

  | hpq : p → q
  ⊢ q

the tactic `apply hpq` changes the goal to `⊢ p`: it tells Lean that, granted `hpq` it suffices to construct a term in `p` to deduce a term in `q`.

- If your goal is `a = a`, the tactic `rfl` closes it.

⌘
+++

+++ `rw`
This tactic takes an assumption `h : a = b` and replaces all occurrences of `a` in the goal to `b`. Its variant

```
rw [h] at h1
```

replaces all occurrences of `a` in `h1` with `b`.

- Unfortunately, `rw` is not symmetric: if you want to change `b` to `a` use `rw [← h]` (type `←` using `\l`): **beware the square brackets!**

⌘
+++

+++ `True`, `False`, `¬` and proofs by contradiction

- `True : Prop` is a type whose only term is called `trivial`. To prove `True`, do `exact trivial`, for instance.
- `False` has no term. Typically, you do not want to construct terms there...
- The `exfalso` tactic changes *any* goal to proving `False` (useful if you have an assumption `... → False`).
- The *definition* of `¬ P` is

  P → False

and 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`.

⌘
+++

+++ Conjunction ("And") and Disjunction ("Or")
For both logical connectors, there are two use-cases: we might want to *prove* a statement of that form, or we might want to *use* an assumption of that form.

And

- `constructor` transforms a goal `⊢ p ∧ q` into the two goals `⊢ p` and `⊢ q`.

- `.left` and `.right` (or `.1` and `.2`) are the projections from `p ∧ q` to `p` and `q`.

Or

- `right` and `left` transform a goal `p ∨ q` in `p` and in `q`, respectively.
- `cases p ∨ q` creates two goals: one assuming `p` and the other assuming `q`.
  +++

+++ `by_cases`
The `by_cases` tactic, **not to be confused with** `cases`, creates two subgoals: one assuming a premise, and the one assuming its negation.

⌘
+++

+++ Equivalences
As above, an equivalence can be either *proved* or *used*.

- A goal `⊢ P ↔ Q` can broken into the goals `⊢ P → Q` and `⊢ Q → P` using `constructor`.
- The projections `(P ↔ Q).1` (or `(P ↔ Q).mp`) and `(P ↔ Q).2` (or `(P ↔ Q).mpr`) are the implications `P → Q` and `Q → P`, respectively.

⌘
+++

Quantifiers

Again, the two quantifiers `∀ ...` and `∃ ...` can either occur in assumptions or in goals.

+++ `∀`

- Internally, the `∀` construction is a generalization of an implication.

- You can the prove it by `intro`ducing a variable (thought of as a "generic element", do `intro x` to call this element `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.
+++

+++ `∃`
Once more,

- 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`.

`⌘`

+++