Tactics and Types

Welcome!

Before discussing how to do things, let's have a look at what we're speaking about.

⌘

Tactics

Lean relies on the *Curry–Howard isomorphism*, sometimes called the

> Propositions-as-Types and Proofs as-Terms Correspondence

(more on this later).
+++In a nutshell
Each statement (or proposition) `P` is seen as a one-slot drawer: it either contains *one* gadget (a/the proof of `P`) or nothing. In the first case `P` is true, in the second it is "false" (or unprovable...).
+++
Proving `P : Prop` boils down to producing a/the term `hp : P`.

This is typically done by

1. Finding somewhere a *true* proposition `Q` and a term `hq : Q`;
2. Producing a function `f : Q → P` ("an implication");
3. Defining `hp := f hq`.

This is often painful: to simplify our life, or to build more convoluted implications, we use *tactics*.

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

- 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

```
hq : Q
f : Q → P
⊢ P
```

we are done because we can use `f` to reduce, or back-track, the truth of `P` to that of `Q`, that we know (it is `hq`).

- When wanting to prove an implication `P → Q`, the tactic `intro hp` introduces a term `hp : P`: after all, an implication *is* a function, and to define it you give yourself a "generic element in the domain".

- 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 with `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!**

⌘

+++

+++ 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 one assuming its negation.

⌘

+++

Types

Lean is based on (dependent) type theory. It is a very deep foundational theory, and we won't dig into all its details.

We'll use it as a replacement for the foundational theory, **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`".

Given some term `t` we can ask Lean what its type is with the command

```
#check t
```

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

So, if you encounter

```
t : T, t : S
```

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

⌘

+++

`Prop`

There is one kind of particular types, called *propositions* and denoted `Prop`.

Types in it contain *either one term, or no term at all*.

Types of kind `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)`.

+++ `True`, `False` and `Bool`
Fundamentally, `Prop` contains only two types: `True` and `False`.

- `True : Prop` is a type whose only term is called `trivial`. To prove `True`, you can do `exact trivial`.
- `False` has no term. Typically, you do not want to construct terms there...
- `Bool` is a different type, that contains two *terms*: `true` and `false` (beware the capitalization!).

> `Bool` ≠ `Prop`, and we'll ignore `Bool` most of the time.

+++

⌘

Key points to keep in mind

- If `P : Prop` then either `P` has not term at all ("`P` is false"), or `P` has a unique term `hp` (`hp` is "a witness that `P` is true"; or a **proof** of `P`).
- Both `ℕ` and `3 < -1` and `ℝℙ²` and `(a+b)² = a² + 2ab + b²` are types, although of different flavour.

⌘ → `Exercises`