04-logic.lean

```
import Mathlib
```

Logic (Part III)

1. `True`, `False` and `Not`
2. classical logic tactics, e.g. proof by contradiction
3. negation-pushing techniques
4. the difference between classical and intuitionistic logic
5. `Decidable`

3 is recommended for those who wants to have some exercises. For lazy ones, you may only remember the tactics introduced there.

4, 5 are optional and left for logical lunatics.

1 `True`, `False` and `Not`

In Lean's dependent type theory, `True` and `False` are propositions serving as the terminal and initial objects in the universe of `Prop`.

Eagle-eyed readers may notice that `True` and `False` act similarly to singleton sets and empty sets in set theory.

They are constructed as inductive types.

```
section

variable (p q : Prop)
```

1.1 `True` (`⊤`)

`True` has a single constructor `True.intro`, which produces the unique proof of `True`. `True` is self-evidently true by `True.intro`.

```
#check True.intro
```

`True` as the terminal object

```
example : p → True := by
  intro _
  exact True.intro
```

The following examples shows that `True → p` is logically equivalent to `p`.

```
example (hp : p) : True → p := by
  intro _
  exact hp
```

[IGNORE] Above is actually the elimination law of `True`.

```
example (hp : p) : True → p := True.rec hp

example (htp : True → p) : p := htp True.intro
```

`trivial` is a tactic that solves goals of type `True` using `True.intro`, though it's power does not stop here.

```
example (htp : True → p) : p := by
  apply htp
  trivial
```

1.2 `False` (⊥)

`False` has no constructors, meaning that there is no way to construct a proof of `False`. This means that `False` is always false.

`False.elim` is the eliminator of `False`, serve as the "principle of explosion", which allows us to derive anything from a falsehood. `False.elim` is self-evidently true in Lean's dependent type theory.

```
#check False.elim
#check False.rec -- [IGNORE] `False.elim` is actually defined as `False.rec`
```

eliminating `False`

```
example (hf : False) : p := False.elim hf
```

`exfalso` is a tactic that applys `False.elim` to the current goal, changing it to `False`.

```
example (hf : False) : p := by
  exfalso
  exact hf
```

`contradiction` is a tactic that proves the current goal by finding a trivial contradiction in the context.

```
example (hf : False) : p := by
  contradiction

-- [EXR]
example (h : 1 + 1 = 3) : RiemannHypothesis := by
  contradiction
```

On how to actually obtain a proof of `False` from a trivially false hypothesis via term-style proof [TODO], see here

[IGNORE] Experienced audiences may question why `False.elim` lands in `Sort*` universe instead of `Prop`. This is because `False` is a subsingleton. See the manual to understand how the universe of a recursor is determined.

```
end
```

2 Not (¬)

In Lean's dependent type theory, negation `¬p` is realized as `p → False`

You may understand `¬p` as "if `p` then absurd", indicating that `p` cannot be true.

```
section

variable (p q : Prop)

#print Not
```

this has a name `absurd` in Lean

```
#check absurd
example (hp : p) (hnp : ¬p) : False := hnp hp
```

[EXR] contraposition

```
example : (p → q) → (¬q → ¬p) := by
  intro hpq hnq hp
  exact hnq (hpq hp)
```

`contrapose!` is a tactic that does exactly this. We shall discuss this later.

[EXR]

```
example : ¬True → False := by
  intro h
  exact h True.intro
```

[EXR]

```
example : ¬False := by
  intro h
  exact h
```

[EXR] double negation introduction

```
example : p → ¬¬p := by
  intro hp hnp
  exact hnp hp
```

Double negation elimination is not valid in intuitionistic logic. You'll need proof by contradiction `Classical.byContradiction` to prove it. The tactic `by_contra` is created for this purpose. If the goal is `p`, then `by_contra hnp` changes the goal to `False`, and adds the hypothesis `hnp : ¬p` into the context.

```
#check Classical.byContradiction
```

double negation elimination

```
example : ¬¬p → p := by
  intro hnnp
  by_contra hnp
  exact hnnp hnp
```

You can use the following command to check what axioms are used in the proof

```
#print axioms Classical.not_not -- above has a name
```

For logical lunatics:

In Lean, `Classical.byContradiction` is proved by the fact that all propositions are `Decidable` in classical logic, which is a result of - the axiom of choice `Classical.choice` - the law of excluded middle `Classical.em`, which is a result of - the axiom of choice `Classical.choice` - function extensionality `funext`, which is a result of - the quotient axiom `Quot.sound` - propositional extensionality `propext`

You can always trace back like this in Lean, by ctrl-clicking the names. This is a reason why Lean is awesome for learning logic and mathematics.

[EXR] another side of contraposition

```
example : (¬q → ¬p) → (p → q) := by
  intro hnqnp hp
  by_contra hnq
  exact hnqnp hnq hp

end
```

[IGNORE] In fact above is equivalent to double negation elimination. This one use the `have` tactic, which allows us to state and prove a lemma in the middle of a proof.

```
example (hctp : (p q : Prop) → (¬q → ¬p) → (p → q)) : (p : Prop) → (¬¬p → p) := by
  intro p hnnp
  have h : (¬p → ¬True) := by
    intro hnp _
    exact hnnp hnp
  apply hctp True p h
  trivial
```

3 Pushing negations

Some negation can be pushed within intuitionistic logic. Some cannot.

3.1 Negation with ∧ and ∨

```
section

variable (p q r : Prop)
```

Classical logic: case analysis

```
example (hpq : p → q) (hnpq : ¬p → q) : q := Or.elim (Classical.em p) hpq hnpq
#check Classical.byCases -- above has a name
```

We have a corresponding tactic: `by_cases`

```
example (hpq : p → q) (hnpq : ¬p → q) : q := by
  by_cases hp : p
  · exact hpq hp
  · exact hnpq hp
```

Proof by cases would help us to obtain an equivalent characterization of `Or`.

```
example : (p ∨ q) ↔ (¬p → q) := by
  constructor
  · rintro (hp | hq)
    · intro hnp
      exfalso
      exact hnp hp
    · intro _
      exact hq
  · intro hnpq  -- the direction of constructing `Or` needs classical logic
    by_cases h?p : p
    · left; exact h?p
    · right; exact hnpq h?p
```

Note that this vividly illustrates the difference between classical logic and intuitionistic logic.

In intuitionistic logic, `Or` means slightly stronger than in classical logic: by `p ∨ q` we mean that we know explicitly which one of `p` and `q` is true. We cannot do implications like `¬p → q` implying `p ∨ q`, because we don't know exactly which one of `p` and `¬p` is true, and the introduction rules of `Or` are asking us to provide it explicitly. This is a reason why intuitionistic logic is considered to be computable.

We also have an equivalent characterization of `And`. This is also done in classical logic.

```
example : (p ∧ q) ↔ ¬(p → ¬q) := by
  constructor
  · intro ⟨hp, hnq⟩ hpnq
    exact hpnq hp hnq
  · intro hnpnq -- the direction of constructing `And` needs classical logic
    contrapose hnpnq
    rw [Classical.not_not]
    intro hp hq
    exact hnpnq ⟨hp, hq⟩
```

[EXR] →–∨ distribution

```
example : (r → p ∨ q) ↔ ((r → p) ∨ (r → q)) := by
  constructor
  · intro hrpq -- this direction needs classical logic
    by_cases h?r : r
    · rcases hrpq h?r with (hp | hq)
      · left; intro _; exact hp
```

```
       · right; intro _; exact hq
     · left
      intro hr
      exfalso; exact h?r hr
   · rintro (hrp | hrq)
     · intro hr
      left; exact hrp hr
     · intro hr
      right; exact hrq hr
#check imp_or -- above has a name
```

[EXR] De Morgan's laws

```
example : ¬(p ∨ q) ↔ ¬p ∧ ¬q := by
  constructor
   · intro hnq
    constructor
     · intro hp
      apply hnq
      left; exact hp
     · intro hq
      apply hnq
      right
      exact hq
   · rintro ⟨hnp, hnq⟩ (hp | hq)
     · exact hnp hp
     · exact hnq hq
#check not_or -- above has a name
```

[EXR] De Morgan's laws

```
example : ¬(p ∧ q) ↔ ¬p ∨ ¬q := by
  constructor
   · intro hnpq -- this direction needs classical logic
    by_cases h?p : p
     · right
      intro hq
      apply hnpq
      exact ⟨h?p, hq⟩
     · left
      exact h?p
   · rintro (hnp | hnq) ⟨hp, hq⟩
     · exact hnp hp
     · exact hnq hq
#check not_and -- above has a name
```

Introducing `push_neg` tactic: automatically proves all the above. It works in classical logic where negation normal forms exist.

`by_contra!`, `contrapose!` are `push_neg`-enhanced version of their non-`!` counterparts.

For more exercises, see Propositions and Proofs - TPiL4

```
end
```

4 [IGNORE] `Decidable`

It's high time to introduce `Decidable` here for the first time.

Mathematicians are often aware of intuitionistic logic. They know classical logic is equipped with `Classical.em: p ∨ ¬p` for any proposition `p`. Though rarely do they know the concept of `Decidable`, which more often appears in the theory of computation.

For short, `Decidable p` means exactly the same as `p ∨ ¬p` in intuitionistic logic. It means that we know explicitly (or computationally) which one of `p` and `¬p` is true.

Though formally in Lean, `Decidable` is defined as a distinct inductive type, it is very similar to `Or` in that you may, somehow, even use it like a `p ∨ ¬p`. But there are major differences. They are:

- [IGNORE] `Decidable` lives in `Type` universe, instead of `Prop` universe.

  In Lean's dependent type theory, things in `Prop` universe are allowed to be non-constructive. This is because in `Prop` universe, proofs are proof-irrelevant: Lean forgets the exact proof of a proposition once it is proved. So when we have an `Or`, we actually have no idea which one of the two sides is true. Lean is designed so, probably because most of the mathematics is non-constructive.

  On the other hand, things in `Type` universe are required to be constructive, unless you have used `Classical.choice` (In such situation, Lean will require you to tag it as `noncomputable`).

  `Decidable` is designed to be constructive, because it is used to decide whether a proposition is true or false by computation. So `Decidable` must live in `Type` universe: To save whether `p` or `¬p` is true.

  In short, `Prop` is non-constructive and proof-irrelevant, while `Type` is constructive and saves data. This makes `Decidable` stronger than a pure proof of `p ∨ ¬p : Prop`.

- [IGNORE] It is tagged as a typeclass.

  This allows Lean to automatically find a proof of `Decidable p` so that you don't have to prove it yourself.

  So at many places `Decidable p` is implicitly deduced.

- The constructors of `Decidable` has different names: `isTrue` and `isFalse`

To wrap up, we have `Decidable` because:

- To mean exactly the same as `p ∨ ¬p` in intuitionistic logic, to make it computable.
- To allow you to just assume `p ∨ ¬p` for only some propositions, which is more flexible than a classical logic overkill.

```
section

variable (p q : Prop)

#print Decidable
#check Decidable.isTrue
#check Decidable.isFalse
```

`Decidable` enables computational reasoning to see if a proposition is true or false

```
#eval True
#eval True → False
#eval False → (1 + 1 = 3)
#synth Decidable (False → (1 + 1 = 3))
```

Manually proving `Decidable` to ensures a computable proof

```
instance : Decidable (p → p ∨ q) := by
  apply Decidable.isTrue -- explicit use of constructor
  intro hp
  left
  exact hp
#synth Decidable (p → p ∨ q)
#eval (p q : Prop) → (p → (p ∨ q))
```

`Decidable` enables partial classical logic

```
#check Classical.byContradiction -- we have done this before
```

proof by contradiction in intuitionistic logic with decidable hypothesis

```
example [dp : Decidable p] : (¬p → False) → p := by
  intro hnpn
  rcases dp with (hnp | hp)
  · exfalso; exact hnpn hnp
  · exact hp
#check Decidable.byContradiction -- above has a name

end
```