03-logic.lean

```
import Mathlib
```

Logic (Part II)

- `And` and `Or`
- `Forall` and `Exists`

1 `And` and `Or`

In Lean's dependent type theory, `∧` and `∨` serve as the direct product and the direct sum in the universe of `Prop`.

Eagle-eyed readers may notice that `∧` and `∨` act similarly to Cartesian product and disjoint union in set theory.

They are also constructed as inductive types.

```
section

variable (p q r : Prop)
```

1.1 `And` (∧)

1.1.1 Introducing `And`

The only constructor of `And` is `And.intro`, which takes a proof of `p` and a proof of `q` to produce a proof of `p ∧ q`.

It is self-evident. Regard this as the universal property of the direct product if you like.

```
#check And.intro

example (hp : p) (hq : q) : p ∧ q := And.intro hp hq
```

`And.intro hp hq` can be abbreviated as `⟨hp, hq⟩`, called the anonymous constructor.

```
example (hp : p) (hq : q) : p ∧ q := ⟨hp, hq⟩
```

introducing nested `And`

```
example (hp : p) (hq : q) (hr : r) : p ∧ q ∧ r := by
  exact ⟨hp, hq, hr⟩ -- equivalent to `⟨hp, ⟨hq, hr⟩⟩`
```

`constructor` tactic applies `And.intro` to split the goal `p ∧ q` into subgoals `p` and `q`. You may also use the anonymous constructor notation `⟨hp, hq⟩` to mean `And.intro hp hq`.

use `·` to focus on the first goal in your goal list.

```
example (hp : p) (hq : q) : p ∧ q := by
  constructor
  · exact hp
  · exact hq
```

`on_goal` tactic can be used to focus on a specific goal.

```
example (hp : p) (hq : q) : p ∧ q := by
  constructor
  on_goal 2 => exact hq
  exact hp
```

`all_goals` tactic can be used to simultaneously perform tactics on all goals.

```
example (hp : p) : p ∧ p := by
  constructor
  all_goals exact hp
```

`assumption` tactic tries to close goals using existing hypotheses in the context. Can be useful when there are many goals.

```
example (hp : p) (hq : q) : p ∧ q := by
  constructor
  all_goals assumption
```

`split_ands` tactic is like `constructor` but works for nested `And`s.

```
example (hp : p) (hq : q) (hr : r) : p ∧ q ∧ r := by
  split_ands
  · exact hp
  · exact hq
  · exact hr
```

[EXR] →–∨ distribution. Universal property of the direct product.

```
example (hrp : r → p) (hrq : r → q) : r → p ∧ q := by
  intro hr
  exact ⟨hrp hr, hrq hr⟩
```

1.1.2 Eliminating `And`

`And.left` and `And.right` are among the elimination rules of `And`, which extract the proofs of `p` and `q`.

```
#check And.left
#check And.right
example (hpq : p ∧ q) : p := hpq.left
example (hpqr : p ∧ q ∧ r) : r := hpqr.right.right
```

`rcases hpq with ⟨hp, hq⟩` is a tactic that breaks down the hypothesis `hpq : p ∧ q` into `hp : p` and `hq : q`. Equivalently you can use `have ⟨hp, hq⟩ := hpq`.

```
example (hpq : p ∧ q) : p := by
  rcases hpq with ⟨hp, _⟩
  exact hp
```

implicit break-down in `intro`

```
example : p ∧ q → p := by
  intro ⟨hp, _⟩
  exact hp
```

nested `And` elimination

```
example (hpqr : p ∧ q ∧ r) : r := by
  rcases hpqr with ⟨_, _, hr⟩
  exact hr
```

[EXR] `And` is symmetric

```
example : p ∧ q → q ∧ p := by
  intro ⟨hp, hq⟩
  exact ⟨hq, hp⟩
#check And.comm -- above has a name
```

[EXR] `→`–`∨` distribution, in another direction.

```
example (hrpq : r → p ∧ q) : (r → p) ∧ (r → q) := by
  constructor
  · intro hr
    exact (hrpq hr).left
  · intro hr
    exact (hrpq hr).right
```

1.1.3 Currification

The actual universal elimination rule of `And` is the so-called decurrification: From `(p → q → r)` we may deduce `(p ∧ q → r)`. This is actually a logical equivalence.

Intuitively, requiring both `p` and `q` to deduce `r` is nothing but requiring `p` to deduce that `q` is sufficient to deduce `r`.

[IGNORE] Decurrification is also self-evidently true in Lean's dependent type theory.

Currification is heavily used in functional programming for its convenience, Lean is no exception.

You are no stranger to decurrification even if you are not a functional programmer: The universal property of the tensor product of modules says exactly the same.

$$\mathrm{Hom}(M \otimes N, P) \cong \mathrm{Hom}(M, \mathrm{Hom}(N, P))$$

[EXR] currification

```
example (h : p ∧ q → r) : (p → q → r) := by
  intro hp hq
  exact h ⟨hp, hq⟩
```

[EXR] decurrification

```
example (h : p → q → r) : (p ∧ q → r) := by
  intro hpq
  exact h hpq.left hpq.right

example (h : p → q → r) : (p ∧ q → r) := by
  intro ⟨hp, hq⟩ -- `intro` is smart enough to destructure `And`
  exact h hp hq

example (h : p → q → r) : (p ∧ q → r) := by
  intro ⟨hp, hq⟩
  apply h -- `apply` is smart enough to auto-decurrify and generate two subgoals
  · exact hp
  · exact hq
```

[IGNORE] decurrification actually originates from `And.rec`, which is self-evident

```
#check And.rec
theorem decurrify (h : p → q → r) : (p ∧ q → r) := And.rec h
```

`And.left` is actually a consequence of decurrification

```
example : p ∧ q → p := by
  apply decurrify
  intro hp _
  exact hp
```

1.2 `Iff` (↔), first visit

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

`Iff` (↔) contains two side of implications: `Iff.mp` and `Iff.mpr`.

Though it is defined as a distinct inductive type, `Iff` may be seen as a bundled version of `(p → q) ∧ (q → p)`. you may, somehow, even use it like a `(p → q) ∧ (q → p)`. The only major difference is the name of the two components.

```
#check Iff.intro
#check Iff.mp
#check Iff.mpr

example : (p ↔ q) ↔ (p → q) ∧ (q → p) := by
  constructor
  · intro h
    exact ⟨h.mp, h.mpr⟩
  · intro ⟨hpq, hqp⟩
    exact ⟨hpq, hqp⟩
```

1.3 `Or` (∨)

1.3.1 Introducing `Or`

`Or` has two constructors, `Or.inl` and `Or.inr`. Either a proof of `p` or a proof of `q` produces a proof of `p ∨ q`.

```
#check Or.inl
#check Or.inr

example (hp : p) : p ∨ q := Or.inl hp
```

`left` (resp. `right`) tactic reduce `Or` goals to `p` (resp. `q`)

```
example (hq : q) : p ∨ q := by
  right
  exact hq
```

1.3.2 Eliminating `Or`

To prove `r` from `p ∨ q`, it suffices to prove both `p → r` and `q → r`. This is the elimination rule of `Or`, or the universal property of the direct sum.

```
#check Or.elim
#check Or.rec -- [IGNORE]

example (hpr : p → r) (hqr : q → r) : (p ∨ q → r) := fun hpq ↦ (Or.elim hpq hpr hqr)
example (hpr : p → r) (hqr : q → r) : (p ∨ q → r) := (Or.elim · hpr hqr) -- note the use of `·`
example (hpr : p → r) (hqr : q → r) (hpq : p ∨ q) : r := by
  apply Or.elim hpq
  · exact hpr
  · exact hqr
```

`match`-style syntax is designed to make use of `Or.elim` to destructure `Or` to cases. [IGNORE] You may just skim through this syntax for now.

```
example (hpr : p → r) (hqr : q → r) : (p ∨ q → r) := fun
  | Or.inl hp => hpr hp
  | Or.inr hq => hqr hq
example (hpr : p → r) (hqr : q → r) (hpq : p ∨ q) : r :=
  match hpq with
  | Or.inl hp => hpr hp
  | Or.inr hq => hqr hq
example (hpr : p → r) (hqr : q → r) (hpq : p ∨ q) : r := by
  match hpq with
  | Or.inl hp => exact hpr hp
  | Or.inr hq => exact hqr hq
example (hpr : p → r) (hqr : q → r) (hpq : p ∨ q) : r := by
  cases hpq with
  | inl hp => exact hpr hp
  | inr hq => exact hqr hq
```

`rcases` may also serve as a tactic version of `match`, which is much more convenient.

```
example (hpr : p → r) (hqr : q → r) (hpq : p ∨ q) : r := by
  rcases hpq with (hp | hq) -- `rcases` can also destructure `Or`
  · exact hpr hp
  · exact hqr hq
```

```
example (hpr : p → r) (hqr : q → r) : p ∨ q → r := by
  rintro (hp | hq) -- `rintro` is a combination of `intro` and `rcases`
  · exact hpr hp
  · exact hqr hq
```

[EXR] distributive laws

```
example : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by sorry
example : p ∨ (q ∧ r) ↔ (p ∨ q) ∧ (p ∨ r) := by sorry

end
```

2 `Forall` and `Exists`

2.1 Forall (`∀`)

As you may have already noticed, `∀` is just an alternative way of writing `→`. Say `p` is a predicate on a type `X`, i.e. of type `X → Prop`, then `∀ x : X, p x` is exactly the same as `(x : X) → p x`.

Though `→` is primitive in Lean's dependent type theory, we may still (perhaps awkwardly) state the introduction and elimination rules of `∀`:

- Introduction: `fun (x : X) ↦ (h x : p x)` produces a proof of `∀ x : X, p x`.
- Elimination: Given a proof `h` of `∀ x : X, p x`, we can obtain a proof of `p a` for any specific `a : X`. It is exactly `h a`.

```
section

variable {X : Type} (p q : X → Prop) (r s : Prop) (a b : X)

#check ∀ x : X, p x
#check ∀ x, p x -- Lean is smart enough to infer the type of `x`

example : (∀ x : X, p x) → p a := by
  intro h
  exact h a
```

[IGNORE] Writing `∀` emphasizes that the arrow `→` is of dependent type, and the domain `X` is a type, not a proposition. But they are just purely psychological, as the following examples show.

```
example : (hrs : r → s) → (∀ _ : r, s) := by
  intro hrs
  exact hrs
```

2.2 `Exists` (`∃`)

`∃` is a bit more complicated.

Slogan: `∀` is a dependent `→`, `∃` is a dependent `×` (or `∧` in `Prop` universe)

```
#check ∃ x : X, p x
#check ∃ x, p x -- Lean is smart enough to infer the type of `x`
```

2.2.1 Introducting `Exists`

`∃ x : X, p x` means that we have the following data:

- an element `a : X`;
- a proof `h : p a`.

So a pair `(a, h)` would suffice to construct a proof of `∃ x : X, p x`. This is the defining introduction rule of `Exists` as an inductive type.

```
#check Exists.intro
example (a : X) (h : p a) : ∃ x, p x := Exists.intro a h
```

As like `And`, you may use the anonymous constructor notation `⟨a, h⟩` to mean `Exists.intro a h`.

```
example (a : X) (h : p a) : ∃ x, p x := ⟨a, h⟩
```

In tactic mode, `use a` make use of `Exists.intro a` to reduce the goal `∃ x : X, p x` to `p a`.

```
example (a : X) (h : p a) : ∃ x, p x := by use a

-- [EXR]
example (x y z : ℕ) (hxy : x < y) (hyz : y < z) : ∃ w, x < w ∧ w < z :=
  ⟨y, ⟨hxy, hyz⟩⟩
```

Note that in the defining pair `(a, h)`, `h` is a proof of `p a`, whose type depends on `a`. Thus psychologically, you may view `∃ x : X, p x` as a dependent pair type `(x : X) × (p x)`.

Have writing `Exists` as a dependent pair type reminded you of the currification process?

2.2.2 Eliminating `Exists`

To construct the implication `(∃ x : X, p x) → q`, it suffices to have a proof of `(∀ x : X, p x → q)`, i.e. `(x : X) → p x → q`. `Exists.elim` does exactly above.

```
#check Exists.elim

example : (∀ x, p x → r) → ((∃ x, p x) → r) := by
  intro hf he
  exact Exists.elim he hf
```

In tactic mode, `rcases h with ⟨a, ha⟩` make use of this elimination rule to break down a hypothesis `h : ∃ x : X, p x` into a witness `a : X` and a proof `ha : p a`.

```
example : (∀ x, p x → r) → ((∃ x, p x) → r) := by
  intro hf he
  rcases he with ⟨a, hpa⟩
  exact hf a hpa

example : (∀ x, p x → r) → ((∃ x, p x) → r) := by
  intro h ⟨a, hpa⟩ -- you may also `rcases` explicitly
  exact h a hpa
```

[EXR] reverse direction is also true

```
example :  ((∃ x, p x) → r) → (∀ x, p x → r) := by
  intro h a hpa
  apply h
  use a

-- [EXR]
example : (∃ x, r ∧ p x) → r ∧ (∃ x, r ∧ p x) := by
  intro ⟨a, ⟨hr, hpa⟩⟩
  exact ⟨hr, ⟨a, ⟨hr, hpa⟩⟩⟩

-- [EXR]
example : (∃ x, p x ∨ q x) ↔ (∃ x, p x) ∨ (∃ x, q x) := by
  constructor
   · rintro ⟨a, (hpa | hqa)⟩
     · left; use a
     · right; use a
   · rintro (⟨a, hpa⟩ | ⟨a, hqa⟩)
     · use a; left; exact hpa
     · use a; right; exact hqa

end
```

2.2.3 [IGNORE] A cosmological remark

The pair `(a, h)` actually do not have type `(x : X) × (p x)`. The latter notation is actually for the dependent pair type (or `Sigma` type), which lives in `Type*` universe.

But `Exists` should live in `Prop`, and in `Prop` universe we admit proof-irrelevance, i.e. we do not save data. So `Exists` forget the exact witness `a` once it is proved.

This "forgetfulness" is revealed by the fact that there is no elimination rule `Exists.fst` to extract the witness `a` from a proof of `∃ x : X, p x`, as long as `X` lives in the `Type*` universe. (Note that `Exists.elim` can only produce propositions in `Prop`)

But if `X` lives in `Prop` universe, then we do have `Exists.fst`:

```
section

#check Exists.fst
```

Wait, wait, we never worked with `X : Prop` before. Say `p : r → Prop` and `r s : Prop`, what does `∃ hr : r, p hr` mean? It means that `r` and `p hr` are both true? [TODO] I don't know how to explain this properly so far.

```
variable (r : Prop) (p : r → Prop)
#check ∃ hr : r, p hr

-- Prove `Exists.fst` and `Exists.snd` by `Exists.elim`
example (he : ∃ hr : r, p hr) : r ∧ p he.fst := by
  apply Exists.elim he
  intro hr hpr
  exact ⟨hr, hpr⟩

end
```

3 [IGNORE] A cosmological remark, continued

Same construction, different universes. Other examples are also shown below.

```
#print And -- `×` in `Prop`
#print Prod -- `×` in `Type*`

-- Forall `∀`: dependent `∏` in `Prop`
-- dependent function type: dependent `∏` in `Type*`

#print Or -- `⊕` in `Prop`
#print Sum -- `⊕` in `Type*`

#print Exists -- dependent `∑` in `Prop`
#print Sigma -- dependent `Σ` in `Type*`

#print Nonempty -- a proof of non-emptiness living in `Prop`
#print Inhabited -- an designated element living in `Sort*`
```

4 Remainder

`Iff` (↔), second visit, bundled and unbundled version
  we do it with `Eq`? `Eq` is hard. Maybe a second visit when touching inductive types.