2025 Lean 与数学形式化讲义(3A)

上海交通大学 AI4MATH 团队

1 Term Construction

1.1 The type system

Definition 1.1.1 Judgement

The statement "the mathematical object a has type A" is called a **judgment**, written as:

$$a : A$$

For convenience, when no ambiguity arises, we abbreviate $(a : A, b : A)$ as $(a, b : A)$.

Remark In Lean, we write `(a b : A)` instead of $(a, b : A)$.

Syntax 1.1.2 `#check`

`#check _TERM_`: to check the type of the `TERM`. The result is shown in InfoView as a message.

Example 1.1.1 Common terms and types

```
#check 1                    -- 1 : Nat
#check Nat                  -- Nat : Type
#check Type                 -- Type : Type 1
#check Type 1               -- Type 1 : Type 2

#check true                 -- Bool.true : Bool
#check (1 : Int)            -- 1 : Int
#check 1.0                  -- 1.0 : Float

#check 1 + 1                -- 1 + 1 : Nat
#check 1 + 1.0              -- 1 + 1.0 : Float
#check 1 + 1 == 2           -- 1 + 1 == 2 : Bool

#check 1 + 1 = 2            -- 1 + 1 = 2 : Prop
#check True                 -- True : Prop
#check False                -- False : Prop
#check Prop                 -- Prop : Type

#check Sort 0               -- Prop : Type
#check Sort 1               -- Type : Type 1
```

1.2 Defining simple terms

Syntax 1.2.1 `def`

- `def _TERM_ : _TYPE_ := ...`: to define a term.

Example 1.2.1 `def`

```
def this_year : Nat := 2025
def next_year : Nat := this_year + 1
```

Syntax 1.2.2 `example`

- `example : _TYPE_ := ...`: to define an anonymous and temporary term.

Example 1.2.2 `example`

```
example : Nat := 233
```

Syntax 1.2.3 `#eval`

`#eval _TERM_`: to evaluate a term (that can be evaluated)

Example 1.2.3 `#eval`

```
#eval this_year -- 2025
#eval next_year -- 2026
```

1.3 Functions

In this section, we first introduce the basic axioms about functions in type theory, and then demonstrate how to write them in Lean. In order to introduce these axioms as clearly as possible, we do not distinguish different layers of types, called **type universes**, before the last subsection, where type universes are systematically introduced.

1.3.1 Axioms about simple functions

Axiom 1.1 Axiom of Simple Function Type Construction

Given types A and B, we can construct the function type $A \to B$:

$$\frac{\Gamma \vdash (A, B : \mathtt{Type})}{\Gamma \vdash (A \to B : \mathtt{Type})}$$

Axiom 1.2 Axiom of Simple Function Application
Given types A and B, an element $a : A$, and a function $f : A \to B$, we can apply f to the element a to obtain the value $f(a)$, which has type B:

$$\frac{\Gamma \vdash (A, B : \texttt{Type}), (a : A), (f : A \to B)}{\Gamma \vdash (f(a) : B)}$$

Axiom 1.3 Axiom of Simple Function Construction (λ-abstraction)
For given types A and B, if we have an indeterminate term x of type A and an expression $\Phi[x]$ containing x such that substituting any element $a : A$ yields a well-defined element $\Phi(a) : B$, we can construct a simple function $[x \mapsto \Phi(x)]$ that maps x to $\Phi(x)$:

$$\frac{\Gamma, (x : A) \vdash (\Phi(x) : B)}{\Gamma \vdash ([x \mapsto \Phi(x)] : A \to B)}$$

Sometimes we do not explicitly name the constructed function, but instead directly use $[x \mapsto \Phi(x)]$ to denote the function. Historically, this method of function construction is called **λ-abstraction**. [a]

[a] In standard type theory textbooks, $[x \mapsto \Phi(x)]$ is usually written as $\lambda(x : A).\Phi[x]$. The $\mapsto$ notation is adopted here to align with Lean syntax and conventional mathematical notation.

1.3.2 Axioms about dependent functions

Axiom 1.4 Axiom of Dependent Function Type Construction
Given a type A and a type family $B : A \to \texttt{Type}$ indexed by A, we can construct the dependent function type $\Pi_{(x:A)}B(x)$:

$$\frac{\Gamma \vdash (A : \texttt{Type}), (B : A \to \texttt{Type})}{\Gamma \vdash (\Pi_{(x:A)}B(x) : \texttt{Type})}$$

Axiom 1.5 Axiom of Dependent Function Application
Given a given type A, a type family $B : A \to \texttt{Type}$, an element $a : A$ and a dependent function $f : \Pi_{(x:A)}B(x)$, we can apply f to the element a to obtain the value $f(a)$ of type $B(a)$:

$$\frac{\Gamma \vdash (A : \texttt{Type}), (B : A \to \texttt{Type}), (a : A), (f : \Pi_{(x:A)}B(x))}{\Gamma \vdash (f(a) : B(a))}$$

Axiom 1.6 Axiom of Dependent Function Construction
Given a type A, a type family $B : A \to \texttt{Type}$, and an expression $\Phi[x]$ with a indeterminate term $x : A$ such that substituting any element $a : A$ yields $\Phi(a)$ of type $B(a)$, we can construct a function $[x \mapsto \Phi(x)] : \Pi_{(x:A)}, B(x)$:

$$\frac{\Gamma, (x : A) \vdash (\Phi[x] : B(x))}{\Gamma \vdash ([x \mapsto \Phi(x)] : \Pi_{(x:A)}, B(x))}$$

1.3.3 Currying

Theorem 1.3.1 Currying (Simple Function Version)
For given types A, B, C, with indeterminate terms $x : A$ and $y : B$, if $\Phi[x, y]$ is an expression containing x and y that always yields elements of type C, then we can construct the function $[x \mapsto [y \mapsto \Phi(x, y)]]$ that maps x to the function $[y \mapsto \Phi(x, y)]$:

$$\frac{\Gamma, (x : A), (y : B) \vdash (\Phi[x, y] : C)}{\Gamma \vdash ([x \mapsto [y \mapsto \Phi(x, y)]] : A \to (B \to C))}$$

For convenience, we stipulate that the function type operator "$\to$" is right-associative, meaning that $A \to B \to C$ denotes the function type $A \to (B \to C)$.
Analogously, given finitely many types $A_1, A_2, \ldots, A_n$, we can construct the multi-argument function type

$$A_1 \to A_2 \to \cdots \to A_n$$

The practice of expressing multi-argument function types as nested single-argument functions is called **Currying**, and such function types are often referred to as **Curried function types**.

Theorem 1.3.2 Currying (Dependent Function Version)

$$\frac{\Gamma, (A : \mathtt{Type})(B : A \to \mathtt{Type})(C : \Pi_{(x:A)}, (B \to \mathtt{Type}))(a : A), (b : B(a)) \vdash (\Phi(a, b) : C(a)(b))}{\Gamma \vdash ([x \mapsto [y \mapsto \Phi(x, y)]] : \Pi_{(x:A)}, \Pi_{(y:B(x))}, C(x, y)}$$

Analogously, given finitely many type families $A_1, A_2, \ldots, A_n$, where

$$A_k : \Pi_{(x_1:A_1)}, \ldots, \Pi_{(x_{k-1}:A_{k-1}(x_1)\ldots(x_{k-1}))}, \mathtt{Type}$$

we can construct the multi-argument function type

$$\Pi_{(x_1:A_1)}, \ldots, \Pi_{(x_n:A_{n-1}(x_1)\ldots(x_{n-1}))}, A_n(x_1)\ldots(x_n)$$

1.3.4 Declaring a function

Syntax 1.3.3 Defining a function

```
def _FUNCTION_NAME_
  ...
  (_PARAMETER_K_ : _TYPE_K_)
  ...
: _GOAL_TYPE_ :=
  _CONSTRUCTION_
```

Remark One can also use `example : ...` to define an anonymous and temporary function, since a function is also a term.

Example 1.3.1 The square function defined on natural numbers.
Here is an example for defining and applying the square function on natural numbers.

```
def square (n : Nat) : Nat := n * n
#check square
#print square -- to print the definition of the term
#eval square this_year
#eval square (square this_year)
```

The type of the goal can be a (dependent) function type.

Syntax 1.3.4 Dependent function type

```
def _FUNCTION_NAME_
  ...
  (_LEFT_PARAMETER_K_ : _LEFT_TYPE_K_)
  ...
: ...
  (_RIGHT_PARAMETER_K_ : _RIGHT_TYPE_K_) →
  ...
  _GOAL_TYPE_ :=
  _CONSTRUCTION_
```

If no right parameters rely on some `RIGHT_PARAMETER_K`, we can simply write `_RIGHT_TYPE_K_` →instead of `(_RIGHT_PARAMETER_K_ : _RIGHT_TYPE_K_)` →.

Remark

- Any type in the definition of a function should only rely on the parameters before.
- `LEFT_TYPE_K` should only rely on `LEFT_PARAMETER_I`, where $1 \leq i < k$.
- `RIGHT_TYPE_K` should only rely on left parameters and `RIGHT_PARAMETER_I`, where $1 \leq i < k$.
- `GOAL_TYPE` should only rely on all these parameters.

Example 1.3.2

```
example : Nat → Nat := sorry
```

1.3.5 Constructing a function term

Syntax 1.3.5 `fun` **together with** `=>`

`fun (_TERM_1_ : _TYPE_1_ ) => ...`

Remark `=>` can be replaced by ↦.

Example 1.3.3 Define the square function using `fun`

```
def square_ : Nat → Nat :=
  fun (n : Nat) => n * n
```

Remark α-sequences are written as functions of type `Nat` $\to \alpha$. Therefore this function can be viewed as the sequence $(0, 1, 4, 9, 16, \ldots)$.

Another example is that the application of a function on a term is also a function.

Example 1.3.4 Defining a function that applys a function

```
def my_apply
  {α β : Sort u}
  (a : α)
  (f : α → β)
: β :=
  f a
-- an alternative way of definition:
def my_apply_ {α β : Sort u} : α → (α → β) → β :=
  fun (a : α) => fun (f : α → β) => f a

#check my_apply
#check my_apply_
-- They have the same ``expected type''.

#eval my_apply this_year square
#eval my_apply (square this_year) square
```

As an additional example, function composition is a function.

Example 1.3.5 Defining function composition as a function

```
def compose
  {α β γ : Sort u}
  (f : α → β)
  (g : β → γ)
: α → γ :=
  fun (a : α) => g (f a)

#check compose
```

`compose square square` is a function that squares a natural number twice.

```
#eval compose square square this_year -- 16815125390625
```

1.3.6 Types of function types

Definition 1.3.6 Type Universe

- In type theory, any type A has the type `Sort` u for some nonnegative integer u. It has to be pointed out that u here is just an index without a type, rather than an element of the natural number type.
- Every element that has type `Sort` u itself is a type.
- These `Sort` u are called **type universes**, with the following judgements given:

$$\texttt{Sort}\ u : \texttt{Sort}\ (u+1), \quad u = 0, 1, 2, \ldots$$

- `Prop` is an alias for `Sort` 0. `Type` u is an alias for `Sort` $(u+1)$. Particularly, `Type` is short for `Type` 0.
- Any proposition has type `Prop`. Most types have type `Type`.
- The type of a simple function type $A \to B$ is defined as

$$\Gamma, (A : \texttt{Sort}\ u), (B : \texttt{Sort}\ v) \vdash A \to B : \texttt{Sort}\ \operatorname{imax}(u, v)$$

where

$$\operatorname{imax}(u, v) := \begin{cases} \max(u, v), & v > 0 \\ 0, & v = 0 \end{cases}$$

Example 1.3.6 Types of function types

```
section types_of_function_types

variable (p : Prop) (α : Type) (β : Type 6)

#check p → p -- Prop
#check α → p -- Prop
#check β → p -- Prop

#check p → α -- Type
#check α → α -- Type
#check β → α -- Type 6

#check p → β -- Type 6
#check α → β -- Type 6
#check β → β -- Type 6

end types_of_function_types
```

1.4 Inductive types

Inductive types are one of the most important concepts in the dependent type theory. We will immediately give a brief definition of them. However, for readers not familiar with imductive types, we strongly recommend skipping to the examples following the definition and the syntax to get a general sense of inductive types.

Axiom 1.7 Inductive types and constructors
T is called a **non-parameterized inductive type**, if T is equipped with a list of (dependent) functions, called the **constructors** of T, and a term of type T can be constructed from any constructor

$$c_k : \Pi(a_1 : \alpha_{k,1}), \ldots, \Pi(a_{n_k} : \alpha_{k,n_k}(a_1, \ldots, a_{n_k-1})), T$$

T is called a **parameterized inductive type** if T is a dependent function, the "final" image is a non-parameterized inductive type. Formally,

$$T : \Pi(b_1 : \beta_1), \ldots, \Pi(b_m : \beta_m(b_1, \ldots, b_{m-1})), \texttt{Sort}\ u$$

is equipped with a list of constructors

$$\begin{aligned} c_k : &\Pi(b_1 : \beta_1), \ldots, \Pi(b_m : \beta_m(b_1, \ldots, b_{m-1})), \\ &\Pi(a_1 : \alpha_{k,1}(b_1, \ldots, b_m)), \\ &\cdots, \\ &\Pi(a_{n_k} : \alpha_{k,n_k}(b_1, \ldots, b_m, a_1, \ldots, a_{n_k-1})), \\ &T(b_1, \ldots, b_m) \end{aligned}$$

Both non-parameterized inductive types and parameterized inductive types are called **inductive types**. Non-parameterized inductive types can be viewed as special parameterized inductive types with zero parameters.
`Sort` u should be either `Prop` or the least upper bound of the types of all these β_j and $\alpha_{k,i}$.

Syntax 1.4.1 Defining an inductive type

```
inductive ... (_PARAM_NAME_0_J_ : _PARAM_TYPE_0_J_) ... : _TYPE_
   where
  ...
  | _CONSTRUCTOR_NAME_K_ (_PARAM_NAME_K_I_: _PARAM_TYPE_K_I_) : ...
     → _TYPE_
  ...
```

1.4.1 Example - Nat

Example 1.4.1 Inductive type - natural numbers

```
-- Standard definition in Lean
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

Remark `Nat : Type` is omitted in the definition, because Lean can calculate it.

Syntax 1.4.2 `match`

```
match _TERM_ with
| _INDUCTIVE_TYPE_._CONSTRUCTOR_1_ => ...
...
| _INDUCTIVE_TYPE_._CONSTRUCTOR_K_ => ...
...
| _INDUCTIVE_TYPE_._CONSTRUCTOR_N_ => ...
```

The TERM of the `INDUCTIVE_TYPE` is constructed by exactly one of the constructors. This tactic gives a term construction according to which constructor the `TERM` is constructed by.
Remark `_INDUCTIVE_TYPE_._CONSTRUCTOR_K_` can be written as `._CONSTRUCTOR_K_` without ambiguity.

Example 1.4.2 Factorial

```
def fact (n : Nat) : Nat :=
  match n with
  | Nat.zero => 1
  | Nat.succ m => (fact m) * n
-- or simply:
example (n : Nat) : Nat :=
  match n with
  | .zero => 1
  | .succ m => (_example m) * n
```

Remark `_example` is the temporary name for an anonymous term If you forget it, move the cursor to `example` and the message box will tell you how to refer to it.

1.4.2 Example - binary trees

Example 1.4.3 Inductive type - binary trees

```
inductive BT where
  | nil : BT
  | succ : BT → BT → BT
```

```
def BT.size (root : BT) : Nat :=
  match root with
  | nil => 0
  | succ lson rson => lson.size + rson.size + 1
#eval (BT.succ (BT.succ BT.nil BT.nil) BT.nil).size -- 2

/-------
    0
   / \
  0  nil
 / \
nil nil
-------/
```

1.4.3 Example - Inductive types with no terms

Question. Is there any type with no terms?
Answer. Yes!

Example 1.4.4 The empty type

```
-- Standard definition in Lean
inductive Empty : Type
```

Remark `Empty` is the type with no constructors.

Here is another example.

Example 1.4.5 An strange inductive type of which no term can be constructed

```
inductive strange where
  | c : strange → strange

def strange.cc : strange → strange :=
  fun x => .c (.c x)
```

Remark One can not construct a term of this type. However, there exists a function defined on this type, though maybe meaningless.

1.4.4 (*) Construction by induction

Example 1.4.6 `.rec`

```
example : ∀ (n : Nat), (n = 0 + n) :=
  Nat.rec
    rfl
    (fun _ ih => Eq.symm (congrArg Nat.succ (Eq.symm ih)))
```

Example 1.4.7 **Construction by induction using `|`**

```
example : ∀ (n : Nat), 0 + n = n
  | Nat.zero   => rfl
  | Nat.succ n => congrArg Nat.succ (_example n)
```

Example 1.4.8 **Subtree relation**

```
inductive BT.subtree (a : BT) : BT → Prop where
  | refl : BT.subtree a a
  | ls {l r: BT} : (BT.subtree a l) → BT.subtree a (BT.succ l r)
  | rs {l r: BT} : (BT.subtree a r) → BT.subtree a (BT.succ l r)
```

Example 1.4.9 `BT.subtree.rec`

```
#print BT.subtree.rec

theorem BT.subtree.trans
  {a b c: BT}
  (hab : a.subtree b)
  (hbc : b.subtree c)
: a.subtree c :=
  subtree.rec
    hab
    (fun _ a_ih => ls a_ih)
    (fun _ a_ih => rs a_ih)
  hbc
```

One can use `match` to create a more readable, yet a bit longer, construction.

Example 1.4.10 **Proof by induction using `match`**

```
example
  {a b c: BT}
```

```
  (hab : a.subtree b)
  (hbc : b.subtree c)
: a.subtree c :=
  match hbc with
  | .refl => hab
  | .ls hblc =>
    .ls (_example hab hblc)
  | .rs hbrc =>
    .rs (_example hab hbrc)
```

1.5 Structures

Definition 1.5.1 Structure type
A **structure type** is an inductive type with a single constructor.

Syntax 1.5.2 A basic way to define a structure type

```
structure _STRUCTURE_NAME_
  ...
  (_NAME_OF_PARAMETER_K_ : _TYPE_OF_PARAMETER_K_)
  ...
: _GOAL_TYPE_ where
  _CONSTRUCTOR_NAME_ ::
  ...
  (_NAME_OF_MEMBER_K_ : _TYPE_OF_MEMBER_K_)
  ...
```

Remark

- Like dependent functions, the type of some term in the definition of a structure type can be dependent on previous terms.
- `_STRUCTURE_NAME_` is a function of the type:

  ```
  ... → (_NAME_OF_PARAMETER_K_ : _TYPE_OF_PARAMETER_K_) → ... →
    _GOAL_TYPE_
  ```

- The complete name of the constructor is `_STRUCTURE_NAME_._CONSTRUCTOR_NAME_`. The type of the constructor is:

  ```
  ... → {_NAME_OF_PARAMETER_K_ : _TYPE_OF_PARAMETER_K_} → ...
    → ... → (_NAME_OF_MEMBER_K_ : _TYPE_OF_MEMBER_K_) → ...
    → _STRUCTURE_NAME_ ... _NAME_OF_PARAMETER_K_ ...
  ```

- The member access function `_STRUCTURE_NAME_._NAME_OF_MEMBER_K_` has the type

```
... → {_NAME_OF_PARAMETER_K_ : _TYPE_OF_PARAMETER_K_} → ...
  → _STRUCTURE_NAME_ ... _NAME_OF_PARAMETER_K_ ...
  → _TYPE_OF_MEMBER_K_
```

- `_CONSTRUCTOR_NAME_ ::` can be omitted, and if so, `mk` is the default name of the constructor.
- One can use `⟨..., _TERM_K_, ...⟩` to construct a term of a structure type, which is short for

```
_STRUCTURE_NAME_._CONSTRUCTOR_NAME_
  ...
  _TERM_K_
  ...
```

- Inferrable parameters and members are allowed.

Example 1.5.1 3D vector

```
structure float_3d_vector : Type where
  locate ::
  x : Float
  y : Float
  z : Float

def inner_product (u v : float_3d_vector) : Float :=
  u.x * v.x + u.y * v.y + u.z * v.z

def outer_product (u v : float_3d_vector) : float_3d_vector :=
  float_3d_vector.locate
    (u.y * v.z - v.y * u.z)
    (u.z * v.x - v.z * u.x)
    (u.x * v.y - v.x * u.y)
-- use `⟨...⟩`
example (u v : float_3d_vector) : float_3d_vector :=
  ⟨
    u.y * v.z - v.y * u.z,
    u.z * v.x - v.z * u.x,
    u.x * v.y - v.x * u.y
  ⟩
```

1.6 Propositions

Recall: Elements related to `Prop` in the type system:

- `True : Prop`
- `False : Prop`
- `Prop : Type`
- `_ANYTYPE_ → p : Prop` (with any `p : Prop` given)

  Question. Given `p : Prop`, what does `h : p` mean?
  Answer. `h` is a proof of the proposition `p`.
- Any element whose type is a proposition is a proof of the proposition.
- Any proof of a proposition is an element whose type is the proposition.
- Each proof only shows that the proposition is proven, carrying no extra information.
- Proofs of a proposition is seen as equal to each other.

1.6.1 Propositional logic

And

Definition 1.6.1 `And`

```
-- Standard definition in Lean
structure And (a b : Prop) : Prop where
  intro ::
  left : a
  right : b
```

Remark

- The constructor `And.intro` is to construct a proof of `And a b` via a proof of `a` (called `left`) and a proof of `b` (called `right`).
- `And.intro` is a function that gives a proof of `And a b` with `left : a` and `right : b` given.
- `a ∧ b` is short for `And a b`.

```
#check And -- And (a b : Prop) : Prop
#check And.intro -- And.intro {a b : Prop} (left : a) (right : b) :
   a ∧ b
```

Example 1.6.1 **Applying the constructor of** `And`

```
example {p q : Prop} (hp : p) (hq : q) : And p q :=
  And.intro hp hq
-- use symbols
example {p q : Prop} (hp : p) (hq : q) : p ∧ q :=
```

```
  ⟨hp, hq⟩
```

Example 1.6.2 Applying the member access functions of `And`

```
example {p q : Prop} : p ∧ q → q ∧ p :=
  fun h => ⟨h.right, h.left⟩
-- use `match`
example {p q : Prop} : p ∧ q → q ∧ p :=
  fun h =>
    match h with
    | ⟨hp, hq⟩ => ⟨hq, hp⟩
```

Or

Definition 1.6.2 `Or`

```
-- Standard definition in Lean
inductive Or (a b : Prop) : Prop where
  | inl (h : a) : Or a b
  | inr (h : b) : Or a b

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

Remark

- If you have a proof of `a` (called `h`), then you can prove `Or a b`. `Or.inl` is such a function describing this proving method.
- If you have a proof of `b`, similarly you can prove `Or a b`, and `Or.inr` is the function you need to construct the new proof.
- `a ∨ b` is short for `Or a b`.

Example 1.6.3 Proving the commutativity of `Or`

```
example {p q : Prop} : Or p q → Or q p :=
  fun h =>
    match h with
    | Or.inl hp => Or.inr hp
    | Or.inr hq => Or.inl hq
-- use symbols
example {p q : Prop} : p ∨ q → q ∨ p :=
  fun h =>
```

```
    match h with
    | Or.inl hp => Or.inr hp
    | Or.inr hq => Or.inl hq
```

Iff

For the definition of other logical types, such as `Iff` and `Not`, one can use the "`#check` and Ctrl+Click" trick.

Definition 1.6.3 `Iff`

```
-- Standard definition in Lean
structure Iff (a b : Prop) : Prop where
  intro ::
  mp : a → b
  mpr : b → a
```

Remark

- `a ↔ b` is short for `Iff a b`.

Example 1.6.4 `Iff`

```
example {p q : Prop} : (¬p ∨ q) ↔ (p → q) :=
  Iff.intro
    fun hnporq =>
      match hnporq with
      | Or.inl hnp => fun hp => False.elim (hnp hp)
      | Or.inr hq => fun _ => hq
    fun hptoq =>
      match Classical.em p with
      | Or.inl hp => Or.inr (hptoq hp)
      | Or.inr hnp => Or.inl hnp
```

Not

Definition 1.6.4 `Not`

```
-- Standard definition in Lean
def Not (a : Prop) : Prop := a → False
```

Remark

- * `¬p` is short for `Not p`, which means that `p` implies a contradiction.

Example 1.6.5 Contradiction gives anything.

```
example {a : Prop} {b : Sort v} (h₁ : a) (h₂ : Not a) : b :=
  absurd h₁ h₂
```

```
section open_classical
```

Some theorems in classical logic can not be proved without some basic axioms, such as the Law of Excluded Middle (LEM) and the Axiom of Choice (AC).

One need to write:

```
open Classical
```

in order to allow AC and its corollaries.

Definition 1.6.5 Axiom of Choice

```
-- Standard definition in Lean
axiom Classical.choice {α : Sort u} : Nonempty α → α
```

Theorem 1.6.6 Diaconescu's theorem
AC is not weaker than LEM.

Proof.

```
example (p : Prop) : p ∨ ¬p := Decidable.em p
```

Example 1.6.6

```
example (p : Prop) : ¬¬p ↔ p :=
  @not_not p
```

```
end open_classical
```

1.6.2 Equality

Definition 1.6.7 `Eq`

```
-- Standard definition in Lean
inductive Eq : α → α → Prop where
  | refl (a : α) : Eq a a
```

Remark `a = b` is short for `Eq a b`.

Example 1.6.7 Transitivity of =

```
example (hab : a = b) (hbc : b = c) : a = c :=
  match hab with
  | Eq.refl a => hbc
```

Remark Here, the common type of `a`, `b` and `c` and the type of this type are omitted. You can check the type of the function defined above by moving the cursor to the keyword `example`. In fact, this type is the same as the type of `Eq.trans`.

Syntax 1.6.8 `rfl`

`rfl` gives a proof of the equality of two equal terms.

Example 1.6.8 `rfl`

```
example : 1 + 1 = 2 := rfl
```

1.6.3 Quantifiers

Forall

In type theory, propositions starting with the quantifier $\forall$ is seen as a dependent function type.

Example 1.6.9 How to prove a "forall" proposition

```
example : ∀ (a b : Nat), a + b = b + a :=
  fun a => fun b => Nat.add_comm a b
```

Remark Given `a b : Nat`, one can prove that `a + b = b + a`, which is a proposition dependent of `a` and `b`. Hence the proof is a dependent function.

Example 1.6.10 how to use a proof of a "forall" proposition

```
example (h : ∀ (n : Nat), n + 1 > n) : 2 + 1 > 2 :=
  h 2 -- h is simply a (dependent) function
```

Exists

Definition 1.6.9 `Exists`

```
-- Standard definition in Lean
inductive Exists {α : Sort u} (p : α → Prop) : Prop where
  | intro (w : α) (h : p w) : Exists p
```

Remark

- To prove a proposition `p` dependent of a term of the type α, one need to find a term `w : α` and also give a proof of `p w`.

- For `p : α → Prop`, `∃ (a : α), p a` (or simply `∃ a, p a`) is short for the proposition `Exists p`.

Example 1.6.11 How to prove an "exists" proposition

```
example : ∃ (x : Nat), 3 * x = 9 :=
  ⟨3, rfl⟩ -- `rfl` proves that 3 * 3 = 9

example : ∃ (x y : Nat), 3 * x + 4 * y = 17 :=
  ⟨3, ⟨2, rfl⟩⟩

example (p : α → Prop) (a : α) : (∀ (x : α), p x) → ∃ (x : α), p x
   :=
  fun h_forall => ⟨a, h_forall a⟩
```

Example 1.6.12 How to use a proof of an "exists" proposition

```
example
  {α : Sort u}
  (p q : α → Prop)
: (∃ x, p x ∧ q x) → (∃ x, p x) ∧ (∃ x, q x) :=
  fun h => ⟨
    ⟨h.choose, h.choose_spec.left⟩,
    ⟨h.choose, h.choose_spec.right⟩
  ⟩

-- use `match`
example
  {α : Sort u}
  (p q : α → Prop)
: (∃ x, p x ∧ q x) → (∃ x, p x) ∧ (∃ x, q x) :=
  fun h =>
    match h with
    | ⟨x, ⟨hpx, hqx⟩⟩ => ⟨⟨x, hpx⟩, ⟨x, hqx⟩⟩
```

1.7 Type classes

```
#check 2 * 3
```

Ctrl+Click `*` and jump to

```
macro_rules | `($x * $y)   => `(binop% HMul.hMul $x $y)
```

This is the "definition" of the abbreviation symbol `*`.

```
#check HMul.hMul -- HMul.hMul.{u, v, w} {α : Type u} {β : Type v} {γ :
    outParam (Type w)} [self : HMul α β γ] : α → β → γ
```

Ctrl+Click `hMul` and jump to

```
class HMul (α : Type u) (β : Type v) (γ : outParam (Type w)) where
  hMul : α → β → γ
```

And you get nothing else except the fact that hMul is a two-variable function.

Questions.

- How is the multiplication of 2 and 3 defined?
- How to find the definition of the multiplication of two natural numbers?

```
-- to get explicit `#check` message
set_option pp.explicit true in #check 2 * 3
```

Message:

```
@HMul.hMul Nat Nat Nat
  (@instHMul Nat instMulNat)
  (@OfNat.ofNat Nat 2 (instOfNatNat 2))
  (@OfNat.ofNat Nat 3 (instOfNatNat 3)) : Nat

#check HMul.hMul
```

Message:

```
HMul.hMul.{u, v, w}
  {α : Type u} {β : Type v}
  {γ : outParam (Type w)}
  [self : HMul α β γ]
: α → β → γ
```

Correspondence:

- α, β and γ are `Nat`;
- `self` is `(@instHMul Nat instMulNat)`;
- `(@OfNat.ofNat Nat 2 (instOfNatNat 2))` is the complete expression of `2`.

```
#check instHMul
```

Ctrl+Click `instHMul`:

```
@[default_instance]
instance instHMul [Mul α] : HMul α α α where
  hMul a b := Mul.mul a b

#check instMulNat
```

Ctrl+Click `instMulNat`:

```
instance instMulNat : Mul Nat where
  mul := Nat.mul
```

Ctrl+Click `Nat.mul`:

```
@[extern ``lean_nat_mul'']
protected def Nat.mul : (@& Nat) → (@& Nat) → Nat
  | _, 0          => 0
  | a, Nat.succ b => Nat.add (Nat.mul a b) a
```