2025 Lean 与数学形式化讲义(3B)

上海交通大学 AI4MATH 团队

1 Tactic Construction

If you want to define a function `f _PARAMETERS_ : _TYPE_`, what you do is fundamentally to use these `PARAMETERS` to construct an term of the `TYPE`.

If you want to prove a proposition `theorem _NAME_ _PARAMETERS_ : _PROPOSITION_` or `example _PARAMETERS_ : _PROPOSITION_`, where `_PROPOSITION_ : Prop`, you need to construct a term, of which the type is `_PROPOSITION_`, i.e., to provide a proof of this `PROPOSITION`.

To define a term using `... : _TYPE_ := _TERM_` is called a **term construction**. However, we can also use tactic to construct a term, called a **tactic construction**. Particularly, if we are constructing a term `h : p` of a type `p : Prop`, we say that we are providing a proof, or more specifically, a **term proof** or a **tactic proof**.

In tactic constructions, we focus on the context and the goals, both of which can be seen in Lean InfoView in VS Code. The context includes all the terms that are available to us for construction. In tactic proofs called the hypotheses. Among all the hypotheses, the temporary terms created during the current construction are shown in InfoView before ⊢, while previously defined terms are not shown but can also be used for construction. Every goal is shown in InfoView after ⊢. Each goal is a type, where a term of this type is to be constructed.

1.1 Basic Tactics - `by`, `exact`, `apply`, `intro` and `rfl`

Syntax 1.1.1 `by`

`by`: to start a tactic construction.

- `... : _TYPE_ := by _TACTIC_CONSTRUCTION_`

Syntax 1.1.2 `exact`

`exact`: to start a term construction

- `exact _TERM_`: to complete the construction by providing a term whose type is the goal.

Remark A single `by exact` means nothing, because `by` switchs to tactic construction, while `exact` immediately switchs back to term construction.

Example 1.1.1 `exact`

```
def example_exact : Nat := by
  exact 2048
#eval example_exact -- 2048
```

Syntax 1.1.3 `apply`

`apply _TERM_`: to use a function to construct a term, maybe with parameters remaining to be passed in

– remark:

- If the current goal is the type `GOAL_TYPE`, then the type of the `TERM` should be

  ```
  _GOAL_TYPE_
  ```

  or

  ```
  _TYPE_1_ → ... → _TYPE_K_ → _GOAL_TYPE_
  ```

  ("`_TYPE_I_ →`" can be replaced by "`∀ _TERM_I_ : _TYPE_I_, `".) Under the former case, the goal is solved, while under the latter case, K new goals `TYPE_1`, ..., `TYPE_K` are created.

Example 1.1.2 `apply`

```
def example_apply : Nat := by
  apply Nat.pow
  exact 2048
  exact 2
#eval example_apply -- 2048 ^ 2
```

Syntax 1.1.4 Colon (;)

`_SENTENCE_1_; ...; _SENTENCE_K_`: to connect several sentences in tactic construction

Example 1.1.3 Colon (;)

```
def example_colon : Nat := by
  apply Nat.pow; exact 2048; exact 2
#eval example_colon -- 2048 ^ 2
```

Syntax 1.1.5 `intro`

`intro`: to introduce a term to "eliminate" the first type in the Curry chain of the current goal

- `intro _TERM_NAME_`
  If you want to define (or prove) something of type `_TYPE_ → ...` or `∀ _TERM_ : _TYPE_, ...`, `intro` creates a term (named after `TERM_NAME`) of the `TYPE`.

- `intro _TERM_NAME_1_ ... _TERM_NAME_K_`
  a shorthand for `intro _TERM_NAME_1_; ...; intro _TERM_NAME_K_`

Example 1.1.4 `intro`

```
example {p q : Prop} : p → q → (p ∧ q) := by
  intro hp hq
  exact ⟨hp, hq⟩ -- also: exact And.intro hp hq
```

The corresponding term proof:

```
example {p q : Prop} : p → q → (p ∧ q) :=
  fun (hp : p) (hq : q) => ⟨hp, hq⟩
```

Example 1.1.5 `intro`

```
example : ∀ p q : Prop, p → q → p ∧ q ∧ p := by
  intro _ _ hp hq
  apply And.intro hp
  exact And.intro hq hp
```

The corresponding term proof:

```
example : ∀ p q : Prop, p → q → p ∧ q ∧ p :=
  fun _ _ hp hq => ⟨hp, hq, hp⟩
  -- `⟨hp, hq, hp⟩` is short for `⟨hp, ⟨hq, hp⟩⟩`
```

Syntax 1.1.6 `apply?`

`apply?`: to ask Lean Language Server for suggestion on which tactic to use in the next step

Remark This should be used in the process of writing a tactic construction, but shall not appear in a complete construction.

1.1.1 Reflexivity - `rfl`

Syntax 1.1.7 `rfl`

`rfl`: to prove an equivalence, e.g., two elements of the same type are equal.

- `rfl` If the construction can be completed by a reflexivity lemma tagged with the attribute `@[refl]`, then `rfl` completes the construction.

Remark

- This tactic applies to a goal whose target has the form $x \sim x$, where $\sim$ is equality, heterogeneous equality or any relation that has a reflexivity lemma tagged with the attribute `@[refl]`.
- `rfl` can be used both in tactic construction and in term construction.

Example 1.1.6 Using `rfl` in a tactic construction

```
example : ∀ (n : Nat), n = n := by
  intro _
  rfl
```

1.2 Basic Tactics for Calculation - `rw`, `calc` and `simp` together with `at`

Syntax 1.2.1 `rw`

`rw`: to use a transitive relation to rewrite a goal or a term in the context

- `rw [_EQUIV_]` With `_EQUIV_ : _TERM_LEFT_ = _TERM_RIGHT_` provided, this tactic replaces the subsentence that matches `TERM_LEFT` and appears the first by the corresponding `TERM_RIGHT`. Like `apply`, the EQUIV can be provided roughly by omitting arguments.
- `rw [← _EQUIV_]` to replaces the first appearing `TERM_RIGHT` by `TERM_LEFT`
- `rw [_EQUIV_1_, ...]` to executes `rw [_EQUIV_1_]`, ... one by one

Example 1.2.1 `rw`

```
example {tp : Sort u} : ∀ a b c : tp, a = b → a = c → c = b := by
  intro a b c h₁ h₂
  rw [← h₂] -- to replace the (first) `c` in the goal `c = b` by `a`
  exact h₁
-- the corrsponding term proof
example {tp : Sort u} : ∀ a b c : tp, a = b → a = c → c = b :=
  fun _ _ _ h₁ h₂ => Eq.trans (Eq.symm h₂) h₁
```

Syntax 1.2.2 `at`

`at`: to use a tactic on the terms

- `_TACTIC_ at _TERM_`
  to execute the `TACTIC` on the type of the `TERM` instead of the current goal
- `_TACTIC_ at _TERM_1_ ...`
  to execute the `TACTIC` on the type of each `TERM_I`
- `_TACTIC_ at *`
  to execute the `TACTIC` on all the terms in the context and the current goal

Remark Here the `TACTIC` can be `rw`, `simp` and some others.

Example 1.2.2 `rw [...] at ...`

```
example {tp : Sort u} : ∀ a b c : tp, a = b → a = c → c = b := by
  intro a b c h₁ h₂
  rw [h₂] at h₁ -- to replace the (first) `a` in `h₁ : a = b` by `c`
  exact h₁
```

Syntax 1.2.3 `calc`

`calc` is a tactic used

- to prove an equivalence by a sequence of equivalences, or
- to prove any other transitional relationship, including `LE` (less or equal) and `LT` (less than).
-
```
calc
  _LHS_ = _STEP_1_ := ...
  _ = _STEP_2_ := ...
  ...
  _ = _STEP_K_ := ...
  _ = _RHS_ := ...
```
  This constructs a proof of `_LHS_ = _RHS_` by providing a sequence of equivalences.
-
```
calc
  _LHS_ _R_1_ _STEP_1_ := ...
  _ _R_2_ _STEP_2_ := ...
  ...
  _ _R_K_ _STEP_K_ := ...
  _ _R_SUCC_K_ _RHS_ := ...
```
  This constructs a proof of `_LHS_ _R_ _RHS_`, where `R` is a transitional relationship, by proving a sequence of relationships `... _R_I_ ...`, where `R_I` is a relationship stronger than or equivalent to `R`.

Remark

- `...` after `:=` is by default a term proof. One need to use `by` to start a tactic proof.
- `calc` only works on transitional relationships, i.e. `_R_.Trans` should be defined.
- `_` in `_ _R_I_ _STEP_I_` represents the right hand side of the last step.
- The transitivity of the relationships `GE` (greater or equal) and `GT` (greater than) are not provided in the basic Lean (i.e., without any packages imported).
- In the basic Lean, `<` does not imply $\leq$.

Example 1.2.3 Using `calc` to prove an equivalence

```
example {tp : Sort u} : ∀ a b c : tp, a = b → a = c → c = b := by
  intro a b c h₁ h₂
  calc
    c = a := by rw [← h₂]
    _ = b := h₁
```

Example 1.2.4 Using `calc` to prove an inequality

```
example {tp : Type u} [LT tp]: ∀ a b c : tp, a = b → a > c → c < b
   := by
-- [LT tp] required by `a > c`, which in Lean is almost just an
   alternative form of `c < a`
  intro a b c h₁ h₂
  calc
    c < a := h₂
    _ = b := h₁
```

Counterexample 1.2.1 `calc` fails to deal with > directly.

```
-- Error: invalid 'calc' step, failed to synthesize `Trans` instance
example : ∀ a b c : Nat, a > b → a < c → c > b := by
  intro a b c h₁ h₂
  calc
    c > a := by sorry
    _ > b := by sorry
```

Syntax 1.2.4 `simp`

`simp`: to simplify

- `simp` use `[simp]` lemmas to simplify the goal
- `simp only [_TERM_1_, ...]` to execute simplification using only these `TERM_I`
- `simp [_TERM_1_, ...]` to execute simplification using only these `TERM_I` together with `[simp]` lemmas
- (many other uses)

Remark More `[simp]` lemmas are added to the Mathlib package. In other words, `simp` is stronger after importing Mathlib.

Example 1.2.5 `simp`

```
example {tp : Sort u} : ∀ a b c : tp, a = b → a = c → c = b := by
  simp
```

Example 1.2.6 To add customized `[simp]` lemmas

```
section How_to_extend_simp

attribute [local simp] Nat.mul_comm Nat.mul_assoc Nat.mul_left_comm
attribute [local simp] Nat.add_assoc Nat.add_comm Nat.add_left_comm

example (w x y z : Nat) (p : Nat → Prop)
        (h : p (x * y + z * w * x)) : p (x * w * z + y * x) := by
  simp at *; assumption

end How_to_extend_simp
```

1.3 Tackling multiple subgoals one by one - `case`

After a multi-variable function is applied, one goal may be decomposed to several goals. If we want to tackle the subgoals one by one, we can use the `case` tactic, or its abbreviation, i.e., a dot (`.`).

Multi-variable function is almost everywhere, including multi-parameter constructors (e.g. `And.intro`), multi-parameter eliminators (e.g. `Or.elim`).

The `case` tactic is designed to give proof "case by case", so that propositions `p ∧ q` or `p ∨ q` can be handled more easily. However, as can be seen from the syntax, this tactic can be used far beyond dealing with propositions.

Syntax 1.3.1 `case` **and** `.`

`case _SUBGOAL_K_ => _CONSTRUCTION_` : to focus on the `SUBGOAL_K` and ask for a construction.

Remark

- `_CONSTRUCTION_` after `=>` is a tactic construction by default. One may use `exact` to start a term proof.
- If one wants to achieve the first subgoal, `case _SUBGOAL_K_ =>` can be replaced by `.`.

Example 1.3.1 Using `case` to introduce `And`

```
example (p q : Prop) (hp : p) (hq : q) : q ∧ p := by
  apply And.intro
  case left => exact hq
  case right => exact hp
```

```
-- It is allowed to change the order of subgoals.
example (p q : Prop) (hp : p) (hq : q) : q ∧ p := by
  apply And.intro
  case right => exact hp
  case left => exact hq
-- dot (.) focus on proving the first uncompleted subgoal
example (p q : Prop) (hp : p) (hq : q) : q ∧ p := by
  apply And.intro
  . exact hq -- equivalent to `case left => ...`
  . exact hp -- equivalent to `case right => ...`
```

Example 1.3.2 **Using `case` to eliminate `Or`**

Remark Here is the definition of `Or.elim`:

```
theorem Or.elim {c : Prop} (h : Or a b) (left : a → c) (right : b →
    c) : c :=
  match h with
  | Or.inl h => left h
  | Or.inr h => right h
```

Therefore, to prove `p ∨ q → r`, we can apply `Or.elim`, and then provide proofs for `p → r` and `q → r`.

```
example {p q : Prop} : (p ∨ q) → (q ∨ p) := by
  intro h
  apply Or.elim h
  case left =>
    intro hp
    exact Or.inr hp
  case right =>
    intro hq
    exact Or.inl hq
-- use dot:
example {p q : Prop} : (p ∨ q) → (q ∨ p) := by
  intro h
  apply Or.elim h
  . intro hp
    exact Or.inr hp
  . intro hq
    exact Or.inl hq
-- the corresponding term proof
example {p q : Prop} : (p ∨ q) → (q ∨ p) :=
  fun h => Or.elim h Or.inr Or.inl
```

Example 1.3.3 A larger example for using `case`

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  apply Iff.intro
  . intro h -- equivalent to `case mp => ...`
    apply Or.elim (And.right h)
    . intro hq
      apply Or.inl
      apply And.intro
      . exact And.left h
      . exact hq
    . intro hr
      apply Or.inr
      apply And.intro
      . exact And.left h
      . exact hr
  . intro h -- equivalent to `case mpr => ...`
    apply Or.elim h
    . intro hpq
      apply And.intro
      . exact And.left hpq
      . apply Or.inl
        exact And.right hpq
    . intro hpr
      apply And.intro
      . exact And.left hpr
      . apply Or.inr
        exact And.right hpr
```

1.4 Eliminating inductive types - `cases`, `match` and `rcases`

Syntax 1.4.1 `cases`

`cases`: to break down an inductive type term by considering each of its possible constructors one by one.

```
cases _TERM_ with
...
| _CONSTRUCTOR_K_ _PARAMETERS_ => ...
...
```

In the case that `TERM` is constructed by `CONSTRUCTOR_K`, we can complete the tactic construction by `...` after `=>`.

Example 1.4.1 Using `cases` to eliminate `Or`

Among three propositions, if at least one of every two is true, then there exists two true propositions.

```
example
  (p q r : Prop)
  (hporq : p ∨ q)
  (hqorr : q ∨ r)
  (hrorp : r ∨ p)
: (p ∧ q) ∨ (q ∧ r) ∨ (r ∧ p) := by
  cases hporq with
  | inl hp =>
    cases hqorr with
    | inl hq => exact Or.inl ⟨hp, hq⟩
    | inr hr => exact Or.inr (Or.inr ⟨hr, hp⟩)
  | inr hq =>
    cases hrorp with
    | inl hr => exact Or.inr (Or.inl ⟨hq, hr⟩)
    | inr hp => exact Or.inl ⟨hp, hq⟩
```

Syntax 1.4.2 `match`

`match`: similar to `cases`, but used in term construction!!!

```
match _TERM_ with
...
| _TYPE_._CONSTRUCTOR_K_ _PARAMETERS_ => ...
...
```

If the constructor of the `TERM` matches `CONSTRUCTOR_K`, we can complete the **term** construction by `...` after `=>`.

Example 1.4.2 Using `match` to eliminate `Or`

```
example (p q : Prop) : (p ∨ q) → (q ∨ p) := by
  intro h
  exact
    match h with
    | Or.inl hp => Or.inr hp
    | Or.inr hq => Or.inl hq
```

Syntax 1.4.3 `rcases`

`rcases`: to eliminate a term of an inductive type

- Given `_TERM_ : _STRUCTURE_TYPE_`, where the constructor has n parameters,

```
rcases _TERM_ with ⟨_PARAM_1_, ..., _PARAM_N_⟩
```

gets these parameters.

- `rcases` in

```
rcases _TERM_ with
  _TERM_BY_CONSTRUCTOR_1_ | ... | _TERM_BY_CONSTRUCTOR_N_
...
```

breaks down an inductive type, creates N tasks, and in the K-th task, the `TERM` is renamed as `_TERM_BY_CONSTRUCTOR_K_`.

- `rcases` can be used recursively, as shown in the following examples.

Example 1.4.3 Using `rcases` to eliminate `And`

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

Example 1.4.4 Using `rcases` to eliminate `Or`

```
example (h : p ∨ q) : q ∨ p := by
  rcases h with hp | hq
  case inl => exact Or.inr hp
  case inr => exact Or.inl hq
```

Example 1.4.5 Using `rcases` recursively

```
example (h : p ∨ (q ∧ r)) : (p ∨ q) ∧ (p ∨ r) := by
  rcases h with hp | ⟨hq, hr⟩
  . exact ⟨Or.inl hp, Or.inl hp⟩
  . exact ⟨Or.inr hq, Or.inr hr⟩
```

Example 1.4.6 Using `rcases` recursively (an additional example)

```
example (h : (p ∨ q) ∧ (p ∨ r)) : p ∨ (q ∧ r) := by
  rcases h with ⟨hp1 | hq, hp2 | hr⟩
  . exact Or.inl hp1
  . exact Or.inl hp1
  . exact Or.inl hp2
  . exact Or.inr ⟨hq, hr⟩
-- an alternative way
```

```
example (h : (p ∨ q) ∧ (p ∨ r)) : p ∨ (q ∧ r) := by
  rcases h with ⟨hp1 | hq, hpr⟩
  . exact Or.inl hp1
  . rcases hpr with hp2 | hr
    . exact Or.inl hp2
    . exact Or.inr ⟨hq, hr⟩
```

1.5 Manipulating the "exists" quantifier - `exists` and `rcases`

Syntax 1.5.1 `exists`

`exists`: to eliminate an "exists" quantifier by providing a term satisfying the condition

- `exists _TERM_`
  To prove `∃ (_TEMPORARY_TERM_ : _TYPE_), _PROPOSITION_`, one can provide a `TERM` of this `TYPE` such that `PROPOSITION` is true.
- `exists _TERM_1_, ...`
  to eliminate multiple "exists" quantifiers together

Example 1.5.1 Using `exists` to introduce an "exists" quantifiers

$2x = 6$ has a solution among natural numbers.

```
example : ∃ (x : Nat), 2 * x = 6 := by
  exists 3
```

$$\exists a, \forall b, P(a,b) \Longrightarrow \forall b, \exists a, P(a,b)$$

```
example {α β : Sort u} {P : α → β → Prop} : (∃ a, ∀ b, P a b) → (∀
   b, ∃ a, P a b) := by
  intro h b
  exists h.choose
  exact h.choose_spec b
```

One can merge consecutive `exists`.

```
example {α β : Sort u} {P : α → β → Prop}
: (a : α) → (b : β) → (P a b) → (∃ x y, P x y) := by
  intro a b h
  exists a, b
  -- One can also write two commands: `exists a; exists b`
```

Recall that `Exists` is an inductive type with a single constructor, i.e., a structure. Therefore one can use `rcases` to eliminate the exists quantifier.

Example 1.5.2 Using `rcases` to eliminate an "exists" quantifier

```
example {α β : Sort u} {P : α → β → Prop} : (∃ a, ∀ b, P a b) → (∀
    b, ∃ a, P a b) := by
  intro h b
  rcases h with ⟨a, h2⟩
  exists a
  exact h2 b
```

1.6 Managing subgoals - `let` and `have`

Syntax 1.6.1 `let`

`let _TERM_ : _TYPE_ := ...`: to create a `TERM` of this `TYPE` in the context, which can be used later.

Example 1.6.1 Let $x_0 = 3$.

```
example : ∃ (x : Nat), 2 * x = 6 := by
  let x₀ : Nat := 3
  exists x₀
```

Syntax 1.6.2 `have`

`have _TERM_ : _TYPE_ := ...`: to create a subgoal `_TYPE_` and to achieve this goal by constructing a `TERM` of this `TYPE`.

Remark

- The construction after `:=` will be forgotten!
- The `TYPE` can be dependent of existing terms in the context
- One can start a tactic construction of this `TERM` using the `by` tactic.
- Since the proof of a proposition need not be remembered, the `TYPE` is often a proposition. In this case, we call the `TYPE` a **lemma**.

Example 1.6.2 `have`

```
example (p q r : Prop) : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) := by
  intro h
  have hp : p := h.left
  have hqr : q ∨ r := h.right
  cases hqr with
  | inl hq => exact Or.inl ⟨hp, hq⟩
  | inr hr => exact Or.inr ⟨hp, hr⟩
```

Example 1.6.3 Properly using `let` and `have`

```
example (a : Nat) : a + 1 > 0 := by
  let c : Nat := a + 1
  have h : c > 0 := by exact Nat.zero_lt_succ a -- remembered
  exact h
```

Counterexample 1.6.1 Improperly using `have`

```
example (a : Nat) : a + 1 > 0 := by
  have b : Nat := a + 1
  have h : b = a + 1 := rfl
  -- error: the definition of `b` is forgotten
  exact h
```

1.7 Contradiction - `contradiction`, `by_contra` and `contrapose`

Syntax 1.7.1 `contradiction`

`contradiction`: to complete a tactic construction with two contradicting propositions.
If there is a proof of a `PROPOSITION` in the context, and meanwhile there is a proof of ¬ `_PROPOSITION_`, then this tactic asserts that the tactic construction is complete.

Example 1.7.1 `contradiction`

```
example : ∀ (p: Prop), p → ¬ p → q := by
  intro _ _ _
  contradiction
```

Syntax 1.7.2 `by_contra`

`by_contra`: proof by contradiction
Requirement: Mathlib

- `by_contra _PROOF_`
  If the current goal is to prove a `PROPOSITION`, this tactic introduces a `PROOF` of ¬ `_PROPOSITION_`, and changes the goal into `False`.

Example 1.7.2 `by_contra`

```
example : ∀ (n : Nat), n ≥ 1 → n ≠ 0 := by
  intro n h_ngeq1
  by_contra h_neq0
  have h_nle1 : n < 1 := by
```

```
    rw [h_neq0]
    exact Nat.one_pos
  have h_not_nle1 : ¬ (n < 1) := by
    exact Nat.not_lt.mpr h_ngeq1
  contradiction
```

Syntax 1.7.3 `contrapose`

`contrapose`: to negate and exchange the goal and a hypothesis (both of type `Prop`)

Requirement: Mathlib

- `contrapose _HYPOTHESIS_`
  This changes the goal into ¬ `_HYPOTHESIS_` and replaces `_HYPOTHESIS_` by the negation of the previous goal, keeping the name of the `HYPOTHESIS` unchanged.

Remark It is sometimes misleading to maintain the name of the `HYPOTHESIS`. Often it is useful to use the `rename` tactic to rename the `HYOPTHESIS`.

Example 1.7.3 `contrapose`

```
example : ∀ (n : Nat), n ≥ 1 → n ≠ 0 := by
  intro n h_n_geq_1
  contrapose h_n_geq_1
  rename ¬(n ≠ 0) => h_n_not_not_eq_0
  have h_n_eq_0 : n = 0 := by
    simp at h_n_not_not_eq_0
    assumption
  simp
  assumption
```

1.8 Induction - `induction`

Syntax 1.8.1 `induction`

```
induction _TERM_ with
| ...
| _CONSTRUCTOR_K_ _PARAMETERS_ => ...
| ...
```

This tactic not only provides a template for construction by cases, but also provides the base hypotheses.

Remark Base hypotheses are anonymous / hidden. One can use `rename_i` to rename them.

Example 1.8.1 `induction`

```
example : (m n : Nat) → (m ≤ m + n) := by
  intro m n
  induction n with
  | zero => exact @Nat.le.refl m
  | succ k =>
    rename_i hk -- : m ≤ m + k
    exact Nat.le.step hk
```

Exercise Use the `induction` tactic to prove the following statement: For any natural numbers $m, n \in \mathbb{N}$, $m \leq n$ if and only if there exists $x \in \mathbb{N}$ such that $m + x = n$.

```
example : (m n : Nat) → (m ≤ n ↔ ∃ (x : Nat), m + x = n) := by sorry
```

1.9 Other tactics

1.9.1 Generalization - `revert` and `generalize`

Syntax 1.9.1 `revert`

`revert`: inverse to `intro`

- `revert _TERM_1_ ...`
  Suppose each `TERM_K` is a term in the context with `TYPE_K`. `DEPENDENT_TERM_J`'s are all the terms in the context that are dependent of some of these `TERM_K`'s. Then all these `TERM_K`'s together with these `DEPENDENT_TERM_J`'s are removed from the context, and the `GOAL` is changed into `∀ _TERM_1_ : _TYPE_1_, ..., _DEPENDENT_TERM_1_ → ... → _GOAL_`.

Example 1.9.1 `revert`

```
example
  (a : α)
  (b : β)
  (p : α → β → Prop)
  (h : ∀x, ∀y, p x y)
: p a b := by
  revert a b
  exact h
```

Syntax 1.9.2 `generalize`

- `generalize _SUB_EXPRESSION_ = _TERM_` to replace the `SUB_EXPRESSION` appearing in the goal by a new `TERM`, and add the `TERM` to the context.

- `generalize _HYPOTHESIS_ : _SUB_EXPRESSION_ = _TERM_` to replace the `SUB_EXPRESSION` appearing in the goal by a new `TERM`, and add the `TERM` as well as the new `HYPOTHESIS` to the context.

Example 1.9.2 `generalize`

```
example : 3 * 2 = 3 + 3 := by
  generalize 3 = x
  -- The state is changed to `x : Nat ⊢ x = x`.
  exact Nat.mul_two x
```

Example 1.9.3 `generalize` **with hypothesis named**

```
example : 2 * 2 = 2 + 2 := by
  generalize h : 2 = x
  -- The state is changed to `x : Nat, h : 2 = x ⊢ x = x`.
  calc
    x * x = x * 2 := by rw[← h]
    _ = x + x := by exact Nat.mul_two x
```

1.9.2 Combining `intro` with `rcases`

Example 1.9.4 **Angle brackets are right-associative.**

```
example (α : Type) (p q : α → Prop) : (∃ x, p x ∧ q x) → ∃ x, q x ∧
    p x := by
  intro ⟨w, hpw, hqw⟩
  exact ⟨w, hqw, hpw⟩
```

Example 1.9.5 **Combining** `intro` **with** `rcases`

```
example (α : Type) (p q : α → Prop) : (∃ x, p x ∨ q x) → ∃ x, q x ∨
    p x := by
  intro
  | ⟨w, Or.inl h⟩ => exact ⟨w, Or.inr h⟩
  | ⟨w, Or.inr h⟩ => exact ⟨w, Or.inl h⟩
```

1.9.3 `intros` and `rename_i`

Syntax 1.9.3 `intros`

`intros`: to introduce all the arguments without naming them.

Remark Each unnamed term is actually given an inaccessible name, shown in Lean InfoView with a dagger (†).

Example 1.9.6 `intros`

```
example : ∀ (a b c : Nat), a + b + c = a + (b + c) := by
  intros
  rw[Nat.add_assoc]
```

Syntax 1.9.4 `rename_i`

`rename_i`: to name the LAST unnamed term in the context

Example 1.9.7 `rename_i`

```
example : ∀ (a b c : Nat), a = b → a + c = b + c := by
  intros
  rename_i h -- h : a = b
  rw[h]
```

1.9.4 `repeat`

Syntax 1.9.5 `repeat`

`repeat`: to repeat a tactic or a sequence of tactic as many times as possible

Remark Usually, it is better to repeat the code instead of using the `repeat` tactic, because the time of repetition may get out of control, and such tactic construction is not clear. However, when constructing a proof of a proposition, sometimes `repeat` indeed shortens the proof and enhances readability.

Example 1.9.8 `repeat`

```
example : ∀ (a b c d e : Nat), a + (b + (c + d) + e) = a + b + c + d
    + e := by
  intros
  repeat rw[←Nat.add_assoc]
```

1.9.5 `constructor` and `fconstructor`

Syntax 1.9.6 `constructor`

`constructor`: to apply the unique constructer of an structure

Remark In fact, when the goal is an inductive type, `constructor` always applies the first constructor of the inductive type. It is safer to use `apply` to specify which constructor to apply.

Example 1.9.9 `constructor`

```
example (p q : Prop) : p ∧ q → q ∧ p := by
  intro h
  cases h with
  | intro hp hq => constructor; exact hq; exact hp
```

Example 1.9.10 `constructor` **may change the order of goals**

```
example : ∃ (n : Nat), n + 3 = 5 := by
  constructor
  case w => exact 2
  rfl
```

Syntax 1.9.7 `fconstructor`

`fconstructor`: just like `constructor` without changing the order of goals

Requirement: Mathlib

Example 1.9.11 `fconstructor`

```
example : ∃ (n : Nat), n + 3 = 5 := by
  fconstructor
  . exact 2
  . exact rfl
```

1.10 Not Recommended Tactics or Uses

1.10.1 `assumption`

```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q := by
  constructor
  repeat assumption
--- Recommended: to explicitly use an assumption in context
```

1.10.2 `<;>`

```
--- Use this when the same tactic is used to prove all the subgoals
example (p q : Prop) (hp : p) (hq : q) : p ∧ q :=
  by constructor <;> assumption
```

1.10.3 `first`

The `first | t₁ | t₂ | ... | tₙ` applies each `tᵢ` until one succeeds

```
example (p q r : Prop) (hp : p) : p ∨ q ∨ r :=
  by repeat (first | apply Or.inl; assumption | apply Or.inr |
     assumption)

example (p q r : Prop) (hq : q) : p ∨ q ∨ r :=
  by repeat (first | apply Or.inl; assumption | apply Or.inr |
     assumption)

example (p q r : Prop) (hr : r) : p ∨ q ∨ r :=
  by repeat (first | apply Or.inl; assumption | apply Or.inr |
     assumption)

-- `all_goals`, `any_goals` and `focus`
example (p q r : Prop) (hp : p) (hq : q) (hr : r) :
      p ∧ ((p ∧ q) ∧ r) ∧ (q ∧ r ∧ p) := by
  repeat (any_goals constructor)
  all_goals assumption
example (p q r : Prop) (hp : p) (hq : q) (hr : r) :
      p ∧ ((p ∧ q) ∧ r) ∧ (q ∧ r ∧ p) := by
  repeat (any_goals (first | constructor | assumption))
```

1.10.4 `split`

```
def f (x y z : Nat) : Nat :=
  match x, y, z with
  | 5, _, _ => y
  | _, 5, _ => y
  | _, _, 5 => y
  | _, _, _ => 1

example (x y z : Nat) : x ≠ 5 → y ≠ 5 → z ≠ 5 → z = w → f x y w = 1 :=
   by
  intros
  simp [f]
  split
  . contradiction
```

```
. contradiction
. contradiction
. rfl
```

1.10.5 left and right

```
-- `left` (or `right`) is used to apply the first (or the second)
   constructor of an inductive type with exactly two constructors
example {p q : Prop} : p → q → p ∨ q := by
  intro hp _
  left -- apply Or.inl
  exact hp
```