Lean cheat sheet

Maris Ozols

March 24, 2026

Contents

1 Logic **2**

2 Sets and functions **3**

3 Core tactics **4**

4 Spliting into cases **7**

5 Reals **8**

6 Functions **9**

7 Creating new hypotheses **10**

8 Structures **10**
8.1 Groups 10
8.2 Lattices 11
8.3 Linear algebra 11

9 Tactic combinators **11**

10 Arithmetic **12**

11 Simplification **13**

12 Other **13**

13 Metacommands **14**

14 Rules of thumb **14**

1 Logic

Notation

For a complete list of notation, click on the ∀ icon in "Lean Infoview", select "Documentation…" and then "Docs: Show Unicode Input Abbreviations".

Implication	$\to$	`\r`, `\to`, `\imp`
Left arrow	$\leftarrow$	`\l`, `\gets`
Equivalence	$\leftrightarrow$	`\iff`, `\leftrightarrow`, `\lr`, `\<->`
Logical AND	$\wedge$	`\and`, `\wedge`
Logical OR	$\vee$	`\or`, `\v`, `\vee`
Negation	$\neg$	`\n`, `\not`
Universal quantifier	$\forall$	`\all`, `\forall`
Existential quantifier	$\exists$	`\ex`, `\exists`
Not equal	$\neq$	`\neq`, `\ne`
Turnstile	$\vdash$	`\vdash`, `\\|-`, `\goal`

Conventions

- $A \to B \to C$ means $A \to (B \to C)$
- $\neg A$ means $A \to$ `False`
- $x \neq y$ means $(x = y) \to$ `False`
- $\forall x < 42,\ P\ x$ is a shorthand for $\forall x,\ x < 42 \to P\ x$

Accessors

- Modus ponens: h`.mp` and h`.mpr` turn a hypothesis $h : p \leftrightarrow q$ into $p \to q$ and $q \to p$, respectively.
- Modus tollens: h`.mt` turns a hypothesis $h : p \to q$ into $\neg q \to \neg p$.
- Left and right injection into `Or`:
  - `Or.inl` hl proves $hl \vee hr$ using hl,
  - `Or.inr` hr proves $hl \vee hr$ using hr.

Tactics

`left`

- Reduces $\vdash a \vee b$ to $\vdash a$.

`right`

- Reduces $\vdash a \vee b$ to $\vdash b$.

`tauto`

- `tauto` concludes the proof by automatically establishing logical statements.

`push_neg`

- `push_neg` rewrites the goal by pushing negations deeper. For example, $\vdash \neg\forall x,\ \exists y,\ x \leq y$ transforms into $\vdash \exists x,\ \forall y,\ y < x$.

contrapose

- `contrapose` turns a goal $P \to Q$ into $\neg Q \to \neg P$
- `contrapose` h changes the goal to $\neg h$ and replaces h by the negation of the original goal

contrapose!

- `contrapose!` h first does `contrapose` and then applies `push_neg` to both the goal and h

Contradiction tactics

exfalso

- `exfalso` changes the goal to `False`. Use it to start a proof by contradiction.

by_contra

- `by_contra` h replaces the goal g with `False` and introduces a new hypothesis $h : \neg g$. It can be used to prove the goal by contradiction.

by_contra!

- `by_contra!` is the same as `by_contra` except that it also automatically normalizes negations by using `push_neg`.

contradiction

- `contradiction` closes the goal if its hypotheses are "trivially contradictory". For example, if there is a hypothesis h : `False`, or two contradictory hypotheses $h : P$ and $h' : \neg P$, or a decideably false hypothesis such as h : `2 + 2 = 3`.

2 Sets and functions

Notation

Element	$\in$	`\in`
Not element	$\notin$	`\notin`
Subset	$\subseteq$	`\ss`, `\sub`, `\subset`
Intersection	$\cap$	`\i`, `\inter`, `\cap`
Union	$\cup$	`\union`, `\un`, `\cup`
Set difference	$\setminus$	`\\`
Empty set	$\varnothing$	`\emptyset`, `\empty`, `\varnothing`

Some further notation and conventions:

- Universal set: `Set.univ` is the set of all elements of a given type
- `{`y `|` P y`}` unfolds to `(fun` $y \mapsto P$ y`)`, so $x \in$ `{`y `|` P y`}` reduces to $P\ x$.
- The preimage f `⁻¹'` s is defined as `{`x `|` f x $\in$ s`}`, so $x \in f$ `⁻¹'` s reduces to $f\ x \in s$.
- `Injective` f is defined as $\forall\ x_1\ x_2,\ f\ x_1 = f\ x_2 \to x_1 = x_2$.

We write $P\ a$ to mean that proposition P holds for $a : \alpha$, so under the hood a property or predicate on a type α is just a function P `:` $\alpha \to$ `Prop`. In *Mathlib*, `Set` α is defined to be $\alpha \to$ `Prop` and $x \in S$ is defined to be $S\ x$.

Set-builder notation is used to define

- $s \cap t$ as {x | $x \in s \ \wedge \ x \in t$},
- $s \cup t$ as {x | $x \in s \ \vee \ x \in t$},
- $\varnothing$ as {x | `False`},
- `univ` as {x | `True`}.

Tactics

`tauto_set`

- Proves any relation between sets by first reformulating it as a logical statement and then applying `tauto`.

3 Core tactics

Tactics

`intro`

- If the goal is $\vdash A \to B$ then `intro` h replaces it with $\vdash B$ and introduces a new hypothesis $h : A$. Omitting h or replacing it by `_` leaves the hypothesis unnamed.
- If the goal is $\vdash A \to B \to C$ then `intro` ha hb replaces it with C and introduces two new hypotheses, $ha : A$ and $hb : B$.
- If the goal is $\vdash \neg A$ then `intro` h replaces it with $\vdash$ `False` and introduces a hypothesis $h : A$.
- If the goal is $\vdash A \wedge B \to C$ then `intro` $\langle ha, hb \rangle$ introduces hypotheses $ha : A$ and $hb : B$, and replaces the goal with C.
- If the goal is $\vdash \forall (a : \mathbb{N}), a + a = 2 * a$ then `intro` x introduces a hypothesis $x : \mathbb{N}$ and changes the goal to $\vdash x + x = 2 * x$.

`revert`

- `revert` x is the inverse of `intro` x.

`exact`

- `exact` h concludes the proof if the goal is $\vdash A$ and there is a hypothesis $h : A$.
- `exact` $\langle hp, hq \rangle$ closes the goal $\vdash P \wedge Q$ using hypotheses $hp : P$ and $hq : Q$.
- `exact` $h1$ $h2$ is a shorthand for `apply` $h1$`;` `exact` $h2$.
- `exact` h a closes the goal by using hypothesis $h : (\forall y : Y), P(y)$ with $y = a$. It seems that `apply` h has the same effect.
- `exact?` performs library search.

`assumption`

- `assumption` closes the goal if there is a matching hypothesis.

refine

- `refine ⟨?_, h, ?_⟩` uses hypothesis $h : Q$ to replace the goal $\vdash P \wedge Q \wedge R$ with two new goals: $\vdash P$ and $\vdash R$.
- `refine h ?_` is equivalent to `apply h` since it uses hypothesis $h : P \to Q$ to change the goal from $\vdash Q$ to $\vdash P$.
- `refine ⟨2, ?_⟩` is equivalent to `use 2` since it changes an existential goal $\vdash \exists$ `(`$n : \mathbb{N}$`),` $n^4 = 16$ to $\vdash 2^4 = 16$.
- The cases can be given names: `refine` h `?`a `?`b `; case` a `=> …; case` b `=> …`.

refine'

apply

- If the goal is $\vdash B$ and $h : A \to B$ is a hypothesis, then `apply` h replaces the goal by $\vdash A$. This is *backwards reasoning* from the goal.
- If $h1 : A \to B$ and $h2 : A$ are hypotheses then `apply` $h1$ `at` $h2$ transforms the second hypothesis into $h2 : B$. This is *forwards reasoning*.
- If $h : A \to B \to C$ is a hypothesis and the goal is $\vdash C$ then `apply` h creates two cases with goals $\vdash A$ and $\vdash B$. Use `\.` to address the two cases. Can use `<;>` if both cases can be address in the same way.
- If $h : \neg P$ is a hypothesis and the goal is $\vdash$ `False` then `apply` h changes the goal to P since $\neg P$ is definitionally equivalent to $P \to$ `False`.
- If $h : \forall a, \varphi a = \psi a$ is a hypothesis and the goal is $\vdash \varphi x = \psi x$ then `apply` h will conclude the proof.
- `apply` h `at` $h1$ $\vdash$ applies hypothesis h at both $h1$ and the goal.
- `apply?` performs library search.

change

- `change` g will change the current goal to g if they are definitionally equivalent.
- `change` t' `at` h will change the hypothesis $h : t$ to $h : t'$, assuming t and t' are definitionally equivalent.
- `change` a `with` b will change occurances of a with b in the goal, assuming a and b are definitionally equivalent.
- `change` a `with` b `at` h does the same for hypothesis h.

show

- `show` g selects which of the goals to prove and unifies it with g.

rename_i

???????? rename inaccessible variable (one with dagger)

clear

- `clear` h removes the hypothesis h.

triv

- `triv` concludes the proof if the goal is `True`.

rfl

- `rfl` tries to close the goal by using reflexivity, e.g., when the goal is $\vdash A \leftrightarrow B$.

rw

- `rw [`h`]` rewrites the goal using hypothesis h.
- `rw [`$\leftarrow h$`]` rewrites the goal using hypothesis h in the backwards direction.
- `rw [`h`] at` $h1$ rewrites the hypothesis $h1$.
- `rw?` searches for possible rewrites through the libarary.
- `rw? at` h does the same for hypothesis h.

erw

- Like `rw` but tries a little harder.

nth_rw

- `nth_rw` i j `[`h`]` rewrites i-th to j-th occurances of hypothesis h.

rwa

- `rwa` is the same as `rw` but afterwards closes any remaining goals using `assumption`.

unfold

- `unfold` def unfolds the definition def in the goal. Its action is similar to `rw [`def`]`.
- `unfold?` followed by shift-clicking an expression in the tactic state gives a list of rewrite suggestions for the selected expression. Clicking on a suggestion replaces `unfold?` by a tactic that performs this rewrite.

unfold_let

use

- `use` 42 chooses the variable under $\exists$ in the goal to have value 42.
- `use` $42, h1, h2$ chooses the value of the variable and uses hypotheses $h1$ and $h2$ to close the goal.
- `use` ha turns $ha : a, \vdash a \wedge b$ into $ha : a, \vdash b$.
- `⟨`$42, h1, h2$`⟩` does the same but using Lean's "anonymous constructor" notation.
- `example :` $\exists x : \mathbb{R}, 41 < x \wedge x < 43$ `:= ⟨`42`, by norm_num ⟩`.

delta

decide

?????????? brute force trial for finite types

fin_cases

????????? consider finite number of cases

done

- `done` succeeds if and only if there are no remaining goals.

sorry

admit

4 Spliting into cases

Notation

- Tactic combinator: `<;>` applies the subsequent tactic to all subgoals generated by `cases`. For example, `cases` h `; repeat assumption` tries to close all cases by using existing hypotheses. This can be abbreviated as `cases` h `<;> assumption`.

Tactics

by_cases

- `by_cases` $h : P$ splits the goal into two cases, assuming $h : P$ in the first branch and $h : \neg P$ in the second.

if ...then ...else ...

- `if` $h : t$ `then` $tac1$ `else` $tac2$ is equivalent to `by_cases` $h : t$ $\cdot$ $tac1$ `;` $\cdot$ $tac2$.

cases'

- `cases'` h `with` hp hq breaks up the hypothesis
  - $h : P \wedge Q$ into two new hypotheses: $hp : P$ and $hq : Q$,
  - $h : P \vee Q$ into two goals: one with hypothesis $hp : P$ and another with $hq : Q$.
- `cases'` h `with` z hf replaces hypothesis $h : \exists x : X, Px$ with $hf : Pz$ and introduces a new hypothesis $z : X$.
- `cases'` h z `with` x hx uses hypothesis $h : \forall b : Z, \exists a : X, fa = b$ to create two new hypotheses: $x : X$ and $hx : fx = z$.
- If $n : \mathbb{N}$ then `cases'` n `with` m gives two goals: one where n is replaced by 0 and another where it is replaced by `succ` m.

cases

- `cases` is the structured version of `cases'` that uses `cases` h `with` followed by `| constructor_name => tactic` branches.
- `cases` h `with | inl` hp `=> ... | inr` hq `=> ...` on $h : P \vee Q$ creates two goals:
  - `| inl` hp `=>` with hypothesis $hp : P$,
  - `| inr` hq `=>` with hypothesis $hq : Q$.
- `cases` n `with | zero => ... | succ` m `=> ...` on $n : \mathbb{N}$ creates two goals:
  - `| zero =>` where n is replaced by 0,
  - `| succ` m `=>` where n is replaced by `succ` m.
- Unlike `cases'`, the named branches can appear in any order, making proofs more robust to upstream changes.

rcases

- `rcases h with ⟨g, ⟨fg, gf⟩⟩` breaks up hypothesis $h : \exists g, f \circ g = id \wedge g \circ f = id$ into three hypotheses: $g : Y \to X$, $fg : f \circ g = id$, $gf : g \circ f = id$.
- `rcases h with ⟨a, rfl⟩` unpacks the hypothesis h into a and another unnamed hypothesis that is immediately used to rewrite the goal.
- `rcases h with hl | hr` breaks the hypothesis `h = (...) ∨ (...)` into two cases called `hl` and `hr`.

rintro

- `rintro` is a combination of `intro` and `rcases`.
- `rintro hP ⟨hQ, hR⟩` is equivalent to `intro hP` followed by `intro h` and `cases' h with hQ hR`.
- `rintro x y h` replaces goal $\vdash \forall (x\ y : \mathbb{R}), x = y \to y = x$ by $\vdash y = x$ and introduces hypotheses $x\ y : \mathbb{R}$ and $h : x = y$.
- `rintro x y rfl` is equivalent to `rintro x y h; subst h`, which speeds up the previous example by immediately changing the goal to $\vdash x = x$ with hypothesis $x : \mathbb{R}$.

interval_cases

- `interval_cases n` searches for upper and lower bounds on a variable n, and if bounds are found, splits into separate cases for each possible value of n.

constructor

- `constructor` breaks the goal $P \wedge Q$ into two cases: P and Q. Each of them can then be addressed using $\cdot$ (entered as `\.`). If both cases can be addressed the same way, use `constructor <;>`, followed by $\cdot$ and commands that address both cases simultaneously.
- `constructor` breaks the goal $A \leftrightarrow B$ into two cases: $A \to B$ and $B \to A$.

5 Reals

Notation

- Reals: $\mathbb{R}$ is `\R`

Tactics

norm_num

- `norm_num` normalizes the numerical expression in the goal.

subst

- `subst h` removes hypothesis $h : x = y$ and replaces all occurances of y by x throughout.
- `subst y` has the same effect, while `subst x` substitutes all occurances of x by y.
- `subst h1 h2` makes two substitutions simultaneously based on hypotheses $h1$ and $h2$.

subst_vars

- `subst_vars` performs all possible substitutions with `subst`.

specialize

- `specialize` h 42 replaces the variable under $\forall$ in hypothesis h with the value 42.

generalize ?????????? look for an expression and replace it by a variable

clean_value ??????????

6 Functions

Notation

- Function composition: $\circ$ is `\o`
- Map: $\mapsto$ is `\mapsto`
- Identity function: `id` (`id` x is definitionally equal to x)
- Subscripts: x_1 is `x\1` or `x\_1`
- Equivalence: $\simeq$ is `\equiv` or `\simeq`

Conventions

- $X \simeq Y$ is the type of functions from X to Y with a two-sided inverse.

Commands

fun

- `fun` x y `=>` x `+` y creates an anonymous fuction (lambda expression).
- λ is a shorthand for `fun`.

def

- `def` $f : \mathbb{N} \to \mathbb{R}$ `:=` `fun` $n \mapsto n^2 + 3$ defines a function.

Tactics

apply_fun

- `apply_fun` f turns goal $\vdash x = y$ into $\vdash f(x) = f(y)$.
- `apply_fun` f `at` h turns hypothesis $h : x = y$ into $h : f(x) = f(y)$.

choose

- `choose` g hg `using` *sur* creates a new hypothesis $hg : \forall (y : Y), f(g\ y) = y$ which asserts that $g : Y \to X$ is a one-sided inverse of function $f : X \to Y$ whose surjectivity is guaranteed by hypothesis *sur* `: Function.Surjective` f.

ext

- `ext` x changes the goal from $f = g$ to $f\ x = g\ x$ and introduces a new hypothesis $x : X$ where X is the domain of both functions f and g.

7 Creating new hypotheses

have

- have $h1$:= h creates a copy of hypothesis h called $h1$.
- have h:<statement> := by <proof> introduces a new hypothesis h defined as <statement> and justified by <proof>.

suffices

????????? similar to have

obtain

- obtain is a combination of have and rcases.
- obtain ⟨patt⟩ : type := proof is equivalent to have h : type := proof; rcases h with patt.
- obtain ⟨w, hw⟩ := h destructs a hypothesis $h : \exists\ x,\ P\ x$ into a witness x and proof $hx : P\ x$
- obtain ⟨hP, hQ⟩ := $P \wedge Q$ creates two new hypotheses, $hP : P$ and $hQ : Q$.

let

set

- set a := t with h is a variant of let a := t. It adds the hypothesis $h : a = t$ to the local context and replaces t with a everywhere it can.

8 Structures

8.1 Groups

Notation

- Inverse: $^{-1}$ is \- or \inv
- Quotient: / is \quot or \/ (not the same as the "/" character)
- Group isomorphisms: $\simeq *$ where $\simeq$ is \simeq or \equiv (short for MulEquiv)
- Ring isomorphisms: $\simeq +*$ (short for RingEquiv)

Conventions

- a * b * c means (a * b) * c

Tactics

group

- Automatically proves any group identity that does not require the hypotheses.

abel

- Similar but for abelian groups.

8.2 Lattices

Notation

- Less than or equal: $\le$ is `\le`
- Greater than or equal: $\ge$ is `\ge`
- Greatest lower bound, infimum, or meet: $\sqcap$ is `\inf`, `\meet`, `\glb`, or `\sqcap`
- Least upper bound, supremum, or join: $\sqcup$ is `\sup`, `\join`, `\lub`, or `\sqcup`
- The bottom element: $\bot$ is `\bot`
- The top element: $\top$ is `\top`

8.3 Linear algebra

Notation

- Linear maps: $V \ {}_{\to l}[k] \ W$ where ${}_{\to l}$ is `\tol`, `\rl`, `\to_l`, `\r_l`, or `\linearmap`
- Vector (neither row nor column): `![1, 2, 3]`
- Vector dot product: `![1, 2]` $\cdot_v$ `![3, 4]` where $_v$ is `\_v`
- Matrix: `!![1, 2; 3, 4]`
- Matrix multiplication: `!![1, 2; 3, 4] * !![3, 4; 5, 6]`
- Matrix-vector multiplication: `!![1, 2; 3, 4]` $*_v$ `![1, 1]`
- Vector-matrix multiplication: `![1, 1]` ${}_v*$ `!![1, 2; 3, 4]`
- Transpose: `!![1, 2; 3, 4]`$^{\mathsf{T}}$ where $^{\mathsf{T}}$ is `\^T`
- Inverse: `!![1, 2; 3, 4]`$^{-1}$ where $^{-1}$ is `\-`, `\inv`, `\-1`, or `\^-1`

9 Tactic combinators

;

- t_1 `;` t_2 means tactic t_1 followed by t_2.

<;>

- `cases <;> assumption` splits the proof into cases and uses `assumption` in each case.

▶

- ▶ is entered as `\t` or `\blacktriangleright`.
- Given $h : a = b$ and $e : p\ a$, the term h ▶ e has type $p\ b$. You can also view h ▶ e as a "type casting" operation where you change the type of e by using h.
- `example {`$f : \mathbb{R} \to \mathbb{R}$`}` $(h :$ `Surjective` $f) : \exists x, fx^2 = 4 :=$
  `let` $\langle x, hx \rangle := h\ 2$
  $\langle x, hx$ ▶ `by norm_num` $\rangle$

<|

try

- `try assumption` tries to close the goal by using some hypothesis. If no hypothesis applies then no progress is made.

repeat

- `repeat` t keeps applying tactic t until it fails.

repeat'

- Repeat the given tactic as many times as possible. Subgoals where it fails are left over.

all_goals

- `all_goals t` applies the tactic `t` to every goal, and succeeds if all applications succeed.

any_goals

- `any_goals t` applies the tactic `t` to every goal. If `t` fails on all of the goals, the entire tactic fails.

first

- `first` | t_1 | t_2 | ... | t_n tries each tactic t_i in order until one of them succeeds, or else fails.

@

- `@x` disables automatic insertion of implicit parameters of the constant `x`. `@e` for any term `e` also disables the insertion of implicit lambdas at this position.

10 Arithmetic

ring

- `ring` evaluates the goal expression in commutative (semi)ring.

ring_nf

- `ring_nf` rewrites the ring expression in the goal into a normal form.

noncomm_ring

- Similar but for non-commutative rings.

linarith

- `linarith` uses linear arithmetic to prove the goal.

nlinarith

- `nlinarith` is extension of `linarith` with some preprocessing to allow solving nonlinear arithmetic problems.

linear_combination

polyrith

- `polyrith` finds the right coefficients for `linear_combination`.

calc

```
example {a b : C} (h1 : a - 5 * b = 4) (h2 : b + 2 = 3) : a = 9 :=
  calc
    a = a - 5 * b + 5 * b := by ring
    _ = 4 + 5 * b := by rw [h1]
    _ = -6 + 5 * (b + 2) := by ring
    _ = -6 + 5 * 3 := by rw [h2]
    _ = 9 := by ring
```

field_simp

- Reduce an expression in a field to an expression of the form n/d where neither n nor d contains any division symbol.

11 Simplification

dsimp

- `dsimp` stands for "definitional simplifier" since it simplifies the goal using only theorems that hold by reflexivity. For example, it can be used to substitute arguments into a function: `(fun` $n \mapsto n^2 + 3$`)` 37 simplifies into $37^2 + 3$.
- `dsimp only [...]` simplifies using only the given set of rules.
- `dsimp only` is equivalent to `dsimp only []`, meaning that it does not use any additional rules for simplification.

simp

- `simp` simplifies the goal using lemmas (tagged with `[simp]`) and hypotheses.
- `simp at` h simplifies the hypothesis h.
- `simp only [...]` simplifies using only the given set of rules.

simp_all

simpa

- `simpa [rules, …] using e`
- `simpa [rules, …]`
- `[…] asdf`

simp_rw

aesop

12 Other

congr

trans

- `trans` y changes the goal from $x \leq z$ to two new goals: $x \leq y$ and $y \leq z$. It can be used whenever the goal contains a transitive relation.

Tactics for general-purpose or domain-specific automation:

- `suggest` searches the library for an applicable lemma.
- `simp` is a general-purpose simplifier.
- `abel`, `ring`, `linarith`, `omega`, continuity: domain-specific automation.
- `tidy`, `finish`, `solve_by_elim`: general purpose automation.

13 Metacommands

- `#eval`
- `#check`
- `#print`
- `#print axioms theorem_name` – prints all the axioms used in the proof of `theorem_name`
- `#reduce`
- `#synth`
- `#version` – prints the Lean version

Search commands:

- `#leansearch`
- `#loogle`
- `#moogle`
- `#statesearch`

14 Rules of thumb

Goal

- $\vdash \exists\ a,\ \ldots$ – use `use`
- $\vdash \forall\ a,\ \ldots$ – use `intro`
- $\vdash a \to b$ – use `intro`
- $\vdash a \leftrightarrow b$ – use `constructor`
- $\vdash a \wedge b$ – use `constructor`
- $\vdash a \vee b$ – use `left` or `right`
- $\vdash s \subseteq t$ – `intro` $x\ xs$

Hypothesis

- $h : x \in f\texttt{''}s$ – use `rcases` h `with` $\langle x,\ xs,\ fxeq \rangle$

Table

$\vdash \exists\ x,\ f\ x$	`use` 42	$\vdash f\ 42$
$\vdash \forall\ x,\ f\ x$	`intro` x	$x : \alpha$ $\vdash f\ x$
$\vdash a \leftrightarrow b$	`constructor`	$\vdash a \to b$ $\vdash b \to a$
$\vdash a \to b$	`intro` h	$h : a$ $\vdash b$
$\vdash a \wedge b$	`constructor`	$\vdash a$ $\vdash b$
$\vdash a \vee b$	`left` `right`	$\vdash a$ $\vdash b$
$\vdash s \subseteq t$	`intro` $x\ xs$	$x : \alpha$ $xs : x \in s$ $\vdash x \in t$
$\vdash s = t$	`ext` x	$x : \alpha$ $\vdash x \in s \leftrightarrow x \in t$
$\vdash x \in s \cup t \to p$	`rintro (`xs `|` xt`)`	$xs : x \in s$ $\vdash p$ $xt : x \in t$ $\vdash p$
$\vdash y \in f\text{''}s$	`use` x	$\vdash x \in s \wedge f\ x = y$
$\vdash y \in f\text{''}s \to p$	`rintro ⟨`x`,` xs`,` fx`⟩`	$x : \alpha$ $xs : x \in s$ $fx : f\ x = y$ $\vdash p$
$h : y \in f\text{''}s$	`rcases` h `with ⟨`x`,` xs`,` fx`⟩`	$x : \alpha$ $xs : x \in \alpha$ $fx : f\ x = y$