Tactics and automation

Jesse Michael Han
Hanoi Lean 2019
University of Pittsburgh

Outline

Outline

rw

- `rw [(h : a = b)]` attemps to find a in your goal and rewrite it to b

rw

- `rw [(h : a = b)]` attemps to find a in your goal and rewrite it to b
- To rewrite, `rw` computes a "motive", which basically means it will try to rewrite all instances of a simultaneously

rw

- `rw [(h : a = b)]` attemps to find a in your goal and rewrite it to b
- To rewrite, `rw` computes a "motive", which basically means it will try to rewrite all instances of a simultaneously
- To avoid this behavior, use `conv`

rw

- `rw [(h : a = b)]` attemps to find `a` in your goal and rewrite it to `b`
- To rewrite, `rw` computes a "motive", which basically means it will try to rewrite all instances of `a` simultaneously
- To avoid this behavior, use `conv`
- To rewrite backwards, use `rw [<-h]`

rw

- `rw [(h :  a = b)]` attemps to find `a` in your goal and rewrite it to `b`
- To rewrite, `rw` computes a "motive", which basically means it will try to rewrite all instances of `a` simultaneously
- To avoid this behavior, use `conv`
- To rewrite backwards, use `rw [<-h]`
- `rw` parses a `texpr`, so you can prove your own inequalities inside the square brackets

rw

- `rw [(h : a = b)]` attemps to find `a` in your goal and rewrite it to `b`
- To rewrite, `rw` computes a "motive", which basically means it will try to rewrite all instances of `a` simultaneously
- To avoid this behavior, use `conv`
- To rewrite backwards, use `rw [<-h]`
- `rw` parses a `texpr`, so you can prove your own inequalities inside the square brackets
- e.g. `rw[show a = b, by foo]`

rw

Use `rw` when:

rw

Use `rw` when:

- You need to rewrite something

rw

Use `rw` when:

- You need to rewrite something
- `simp` will rewrite too many things

rw

Use `rw` when:

- You need to rewrite something
- `simp` will rewrite too many things
- You don't need to use `conv`

rw

Use `rw` when:

- You need to rewrite something
- `simp` will rewrite too many things
- You don't need to use `conv`
- You want to unfold a definition

Outline

simp

- `simp` is an essential tool for writing proofs

simp

- `simp` is an essential tool for writing proofs
- `simp` has access to a library of lemmas tagged with the `@[simp]` attribute

simp

- `simp` is an essential tool for writing proofs
- `simp` has access to a library of lemmas tagged with the `@[simp]` attribute
- `simp` will attempt to rewrite using the given lemmas until no rewrites succeed.

simp

- `simp` is an essential tool for writing proofs
- `simp` has access to a library of lemmas tagged with the `@[simp]` attribute
- `simp` will attempt to rewrite using the given lemmas until no rewrites succeed.
- `simp` uses a different method than `rw` to perform rewrites.

simp

- `simp` is an essential tool for writing proofs
- `simp` has access to a library of lemmas tagged with the `@[simp]` attribute
- `simp` will attempt to rewrite using the given lemmas until no rewrites succeed.
- `simp` uses a different method than `rw` to perform rewrites.
- `simp` has an easier time rewriting under binders.

simp

Use `simp` when:

simp

Use `simp` when:

- You can close your goal by just applying previous simp lemmas.

simp

Use `simp` when:

- You can close your goal by just applying previous simp lemmas.
- You want to simplify your goal using simp lemmas

simp

Use `simp` when:

- You can close your goal by just applying previous simp lemmas.
- You want to simplify your goal using simp lemmas
- You want to rewrite repeatedly and don't need to rewrite backwards

simp

Usage tips:

simp

Usage tips:

- `simp[hyp1, hyp2, hyp3 ...]` will make the extra hypotheses available as simp lemmas.

simp

Usage tips:

- `simp[hyp1, hyp2, hyp3 ...]` will make the extra hypotheses available as simp lemmas.
- `simp at foo` will call `simp` at the location `foo`, where `foo` can be the target, a hypothesis, or a wildcard.

simp

Usage tips:

- `simp[hyp1, hyp2, hyp3 ...]` will make the extra hypotheses available as simp lemmas.
- `simp at foo` will call `simp` at the location `foo`, where `foo` can be the target, a hypothesis, or a wildcard.
- `simp only [hyp1, hyp2, hyp3...]` will only make the given hypotheses available as simp lemmas

simp

Usage tips:

- `simp[hyp1, hyp2, hyp3 ...]` will make the extra hypotheses available as simp lemmas.
- `simp at foo` will call `simp` at the location `foo`, where `foo` can be the target, a hypothesis, or a wildcard.
- `simp only [hyp1, hyp2, hyp3...]` will only make the given hypotheses available as simp lemmas
- replacing `simp` by `squeeze_simp` will give a list of `simp` lemmas that were used by `simp`

simp

Caveats:

simp

Caveats:

- The user must design libraries around `simp`.

simp

Caveats:

- The user must design libraries around `simp`.
- The user must ensure that `simp` lemmas constitute a confluent rewriting system

simp

Caveats:

- The user must design libraries around `simp`.
- The user must ensure that `simp` lemmas constitute a confluent rewriting system
- The user must ensure that `simp` lemmas do not throw `simp` into a loop.

simp

Caveats:

- The user must design libraries around `simp`.
- The user must ensure that `simp` lemmas constitute a confluent rewriting system
- The user must ensure that `simp` lemmas do not throw `simp` into a loop.
- The user must ensure that `simp` lemmas usually replace more complicated expressions by simpler ones.

Outline

conv

- `conv` is a special mode you can use to perform surgical rewrites
- To navigate inside a `conv` block, use:
    - `congr` to break an expression into subexpressions,
    - `skip` to move to the next subexpression, and
    - `funext` to move under binders.

conv

Use `conv` when:

conv

Use `conv` when:

- `rw` and `simp` are not smart enough to rewrite in only place in your expression

conv

Use `conv` when:

- `rw` and `simp` are not smart enough to rewrite in only place in your expression
- You need to control exactly what gets rewritten

conv

Use `conv` when:

- `rw` and `simp` are not smart enough to rewrite in only place in your expression
- You need to control exactly what gets rewritten
- `rw` fails to infer a motive, so you need to make the target simpler

Outline

tauto

- `tauto` implements a tableaux prover

tauto

- `tauto` implements a tableaux prover
- `tauto!` closes goals provable in classical propositional logic

tauto

Use `tauto` when:

tauto

Use `tauto` when:

- Your goal is a theorem of classical propositional logic

tauto

Use `tauto` when:

- Your goal is a theorem of classical propositional logic
- You don't feel like calling `finish`

Outline

linarith

- `linarith` is a tactic for linear (in)equalities

linarith

- `linarith` is a tactic for linear (in)equalities
- Implements Fouier-Motzkin elimination

linarith

- `linarith` is a tactic for linear (in)equalities
- Implements Fouier-Motzkin elimination
- Theoretically complete for $\mathbb{R}$ and $\mathbb{Q}$.

linarith

- `linarith` is a tactic for linear (in)equalities
- Implements Fouier-Motzkin elimination
- Theoretically complete for $\mathbb{R}$ and $\mathbb{Q}$.
- Written by Rob Lewis!

linarith

Use `linarith` when:

linarith

Use `linarith` when:

- Your goal is a linear inequality with coefficients in $\mathbb{R}$ and $\mathbb{Q}$

linarith

Use `linarith` when:

- Your goal is a linear inequality with coefficients in $\mathbb{R}$ and $\mathbb{Q}$
- You can reach a contradiction from linear inequalities in context.

Outline

omega

- `omega` is a tactic for discharging linear integer and natural number arithmetic goals.

omega

- `omega` is a tactic for discharging linear integer and natural number arithmetic goals.
- Should finish any goal which is a quantifier-free formula in Presburger arithmetic

omega

- `omega` is a tactic for discharging linear integer and natural number arithmetic goals.
- Should finish any goal which is a quantifier-free formula in Presburger arithmetic
- Written by Seul Baek!

omega

Use `omega` when:

omega

Use `omega` when:

- You have a linear integer or natural number arithmetic goal

omega

Use `omega` when:

- You have a linear integer or natural number arithmetic goal
- You're feeling lazy

Outline

norm_num

- `norm_num` normalizes numerical expressions in ordered fields

norm_num

- `norm_num` normalizes numerical expressions in ordered fields
- `norm_num` calls `simp`, and accepts a list of `simp` lemmas

norm_num

Use `norm_num` when:

norm_num

Use `norm_num` when:

- Your goal is just a matter of arithmetic, e.g. $1 \neq 2$

norm_num

Use `norm_num` when:

- Your goal is just a matter of arithmetic, e.g. $1 \neq 2$
- Your goal is just a matter of arithmetic modulo rewriting with `simp` lemmas

Outline

solve_by_elim

- `solve_by_elim` will automatically find a hypothesis whose target matches the main goal and apply it, and does so recursively.

solve_by_elim

- `solve_by_elim` will automatically find a hypothesis whose target matches the main goal and apply it, and does so recursively.
- Automated version of `apply`

solve_by_elim

Use `solve_by_elim` when:

- You're pretty sure the goal is true and reachable through applications of hypotheses

solve_by_elim

Use `solve_by_elim` when:

- You're pretty sure the goal is true and reachable through applications of hypotheses
- you're too lazy to write out `exact blah blah blah`

Outline

eblast

- `eblast` repeated tries to use `ematch`, followed by `close`.

eblast

- `eblast` repeated tries to use `ematch`, followed by `close`.
- It's powerful, but slow.

eblast

- `eblast` repeated tries to use `ematch`, followed by `close`.
- It's powerful, but slow.
- Useful as a rewrite search. It will generally close a goal which is accessible by rewriting with equalities in context

eblast

- `eblast` repeated tries to use `ematch`, followed by `close`.
- It's powerful, but slow.
- Useful as a rewrite search. It will generally close a goal which is accessible by rewriting with equalities in context
- More intelligent than `simp`, because it can rewrite backwards as necessary

eblast

Use `eblast` when:

- `eblast_using` can be useful to perform a rewrite search

eblast

Use `eblast` when:

- `eblast_using` can be useful to perform a rewrite search
- You're feeling lazy

eblast

Use `eblast` when:

- `eblast_using` can be useful to perform a rewrite search
- You're feeling lazy
- You're pretty sure the goal is true and reachable through rewrites and applications of hypotheses

Outline

tidy

- `tidy` tries all the obvious things until none of them work anymore.

tidy

- `tidy` tries all the obvious things until none of them work anymore.
- `tidy` combines `chain` with the ability to emit tactic proof scripts

tidy

- `tidy` tries all the obvious things until none of them work anymore.
- `tidy` combines `chain` with the ability to emit tactic proof scripts
- `tidy` will invoke `auto_cases` and `solve_by_elim`

tidy

- `tidy` tries all the obvious things until none of them work anymore.
- `tidy` combines `chain` with the ability to emit tactic proof scripts
- `tidy` will invoke `auto_cases` and `solve_by_elim`
- You can also tell `tidy` which tactics to try.

tidy

Use `tidy` when:

tidy

Use `tidy` when:

- The goal is "trivial" modulo unfolding, casing, and calls to `simp`

tidy

Use `tidy` when:

- The goal is "trivial" modulo unfolding, casing, and calls to `simp`
- The goal follows from unwinding definitions and applying hypotheses

tidy

Use `tidy` when:

- The goal is "trivial" modulo unfolding, casing, and calls to `simp`
- The goal follows from unwinding definitions and applying hypotheses
- You're feeling lazy

Outline

finish

- Along with `tidy`, `finish` is currently the most advanced tool for automated reasoning in Lean.

finish

- Along with `tidy`, `finish` is currently the most advanced tool for automated reasoning in Lean.
- `finish` combines three things:
  - a tableaux prover
  - `simp * at *`
  - `eblast`

finish

- Along with `tidy`, `finish` is currently the most advanced tool for automated reasoning in Lean.
- `finish` combines three things:
  - a tableaux prover
  - `simp * at *`
  - `eblast`
- Related variants: `safe` and `clarify`.

finish

Tableaux prover

The first part of `finish` preprocesses the goal and hypotheses, and is complete for propositional logic.

finish

Tableaux prover

The first part of `finish` preprocesses the goal and hypotheses, and is complete for propositional logic.

1. negate the assumption

finish

Tableaux prover

The first part of `finish` preprocesses the goal and hypotheses, and is complete for propositional logic.

1. negate the assumption

1. push negations inwards

finish

Tableaux prover

The first part of `finish` preprocesses the goal and hypotheses, and is complete for propositional logic.

1. negate the assumption

1. push negations inwards

1. split conjunctions

finish

Tableaux prover

The first part of `finish` preprocesses the goal and hypotheses, and is complete for propositional logic.

1. negate the assumption
1. push negations inwards
1. split conjunctions
1. try `by contradiction`

finish

- `finish` will invoke `simp` on all hypotheses.

finish

- `finish` will invoke `simp` on all hypotheses.
- `finish[h₁, h₂, h₃, ...]` will parse simp lemmas $\mathtt{h}_1$, $\mathtt{h}_2$, $\mathtt{h}_3$, ... and make these lemmas available to `simp` when invoking `simp` on all hypotheses.

finish

- `finish` will invoke `simp` on all hypotheses.
- `finish[`$\mathtt{h}_1$`, `$\mathtt{h}_2$`, `$\mathtt{h}_3$`, ...]` will parse simp lemmas $\mathtt{h}_1$, $\mathtt{h}_2$, $\mathtt{h}_3$, ... and make these lemmas available to `simp` when invoking `simp` on all hypotheses.
- `finish[`$\mathtt{h}_1$`, `$\mathtt{h}_2$`, `$\mathtt{h}_3$`,...]` should be able to close any goals that `simp[`$\mathtt{h}_1$`, `$\mathtt{h}_2$`, `$\mathtt{h}_3$`...]` can close.

finish

- Finally, `finish` (essentially) invokes `eblast` (limiting itself to 20 iterations of `ematch`).
- This can sometimes make `finish` very slow.
- `finish` can only parse a list of `simp` lemmas. It does not subsume `eblast_using`[$\mathtt{h}_1$,$\mathtt{h}_2$,$\mathtt{h}_3$...], but it will use `@[ematch]` lemmas available in the environment.

finish

Use `finish` when:

finish

Use `finish` when:

- The goal is "obvious", but the proof requires:
  - unfolding reducible definitions

finish

Use `finish` when:

- The goal is "obvious", but the proof requires:
  - unfolding reducible definitions
- case-splits

finish

Use `finish` when:

- The goal is "obvious", but the proof requires:
  - unfolding reducible definitions
- case-splits
- applying some hypotheses

finish

Use `finish` when:

- The goal is "obvious", but the proof requires:
  - unfolding reducible definitions
- case-splits
- applying some hypotheses
- some simple rewrites by equalities either in context or which are available as `@[ematch]` lemmas.

finish

Use `finish` when:

- The goal is "obvious", but the proof requires:
  - unfolding reducible definitions
- case-splits
- applying some hypotheses
- some simple rewrites by equalities either in context or which are available as `@[ematch]` lemmas.
- Or, just use it when you're pretty sure the goal is true and easy to prove