Karlsruhe Institute of Technology

Theorembeweiserpraktikum

Tactic Proofs

Jakob von Raumer, Sebastian Ullrich | SS 2021

LEAN THEOREM PROVER

Why Tactics

Term proofs can be very compact

```
example : (∃ x, p x ∧ q x) → (∃ x, p x) ∧ (∃ x, q x) :=
  fun ⟨x, hpx, hqx⟩ => ⟨⟨x, hpx⟩, ⟨x, hqx⟩⟩
```

Why Tactics

Term proofs can be very compact

```
example : (∃ x, p x ∧ q x) → (∃ x, p x) ∧ (∃ x, q x) :=
  fun ⟨x, hpx, hqx⟩ => ⟨⟨x, hpx⟩, ⟨x, hqx⟩⟩
```

... but also be very tedious

```
example (d : Weekday) : next (previous d) = d :=
  match d with
  | monday    => rfl
  | tuesday   => rfl
  | wednesday => rfl
  ...
```

```
example : (p x → f x = y) → (if p x then f x else y) = y :=
  fun hfxy =>
    iteCongr rfl (fun hpx => hfxy hpx) (fun _ => rfl) ▸ ite_self (p x) y
```

Why Tactics

Tactics enable an *imperative*, step-by-step proof style

```
example : (∃ x, p x ∧ q x) → (∃ x, p x) ∧ (∃ x, q x) := by
  intro ⟨x, hpx, hqx⟩  -- ⊢ (∃ x, p x) ∧ (∃ x, q x)
  apply And.intro      -- ⊢ ∃ x, p x,   ⊢ ∃ x, q x
  focus                -- ⊢ ∃ x, p x
    exact ⟨x, hpx⟩
  focus                -- ⊢ ∃ x, q x
    exact ⟨x, hqx⟩
```

Why Tactics

Tactics enable an *imperative*, step-by-step proof style

```
example : (∃ x, p x ∧ q x) → (∃ x, p x) ∧ (∃ x, q x) := by
  intro ⟨x, hpx, hqx⟩  -- ⊢ (∃ x, p x) ∧ (∃ x, q x)
  apply And.intro      -- ⊢ ∃ x, p x,   ⊢ ∃ x, q x
  focus                -- ⊢ ∃ x, p x
    exact ⟨x, hpx⟩
  focus                -- ⊢ ∃ x, q x
    exact ⟨x, hqx⟩
```

... where a proof step can also *automate* away many term steps

```
example (d : Weekday) : next (previous d) = d := by
  cases d <;> rfl
```

```
example : (p x → f x = y) → (if p x then f x else y) = y := by
  simp_all
```

Running Tactics

At any point, instead of specifying a term we can use `by` to execute one or more tactics, separated by `;` or line breaks

```
example : (∃ x, p x ∧ q x) → (∃ x, p x) ∧ (∃ x, q x) :=
  fun ⟨x, hpx, hqx⟩ => by apply And.intro ⟨x, hpx⟩; exact ⟨x, hqx⟩
```

The expected type at the position of `by` becomes the proof *goal*, displayed after `⊢`

Basic Tactics

`intro x`	introduce variables/hypotheses, same syntax as `fun`
`exact e`	solve first goal with `e`
`apply e`	solve first goal with `e`, add missing arguments as new goals
`assumption`	solve first goal using any hypothesis of the same type
`contradiction`	solve first goal if "obviously" contradictory, e.g. with hypothesis `x ≠ x` or `none = some a`
`cases e`	split first goal into one case for each constructor of type of `e`
`byCases p`	split first goal into cases `p` and `¬ p` for a (decidable) proposition `p`
`induction e`	like `cases`, but also introduce induction hypotheses
`rfl`	abbreviation for `exact rfl`
`have e :=/by`	
`let x :=`	like in term mode
`show e`	

Basic Combinators

`focus t`	run tactic(s) on first goal only, which must be closed by the last tactic
`t <;> t'`	run `t'` on every goal (which must be closed) produced by `t`
`allGoals t`	run `t` on every goal

Equational Reasoning

`rw [e, ...]`	if `e : el = er`, replace every `el` in the first goal with `er`
`rw [e, ...] at h`	do so at hypothesis `h` instead
`rw [←e]`	invert equality before rewriting

Equational Reasoning

`rw [e, ...]`	if `e : el = er` , replace every `el` in the first goal with `er`
`rw [e, ...] at h`	do so at hypothesis `h` instead
`rw [←e]`	invert equality before rewriting

Arguments are inferred (once) where possible

```
example (n m k : Nat) : (n + m) * k = (m + n) * k := by rw [Nat.add_comm n]
```

Equational Reasoning

`rw [e, ...]`	if `e : e₁ = eᵣ`, replace every `e₁` in the first goal with `eᵣ`
`rw [e, ...] at h`	do so at hypothesis `h` instead
`rw [←e]`	invert equality before rewriting

Arguments are inferred (once) where possible

```
example (n m k : Nat) : (n + m) * k = (m + n) * k := by rw [Nat.add_comm n]
```

For performance reasons, subterms must match the rewrite rule *structurally*

```
example (h : succ n = m) : n + 1 = m := by
  rw [h]  -- tactic 'rewrite' failed, did not find instance of the pattern in the target expression
```

simp

`simp` is a supercharged `rw`:

- exhaustively applies all given equations

```
example
  (h1 : ∀ x, f (f x) = f x)
  (h2 : ∀ x, f' x = f x) :
    f' (f' (f' x)) = f' x := by
  --rw [h2, h2, h2, h1, h1]
  simp [h1, h2]
```

simp

`simp` is a supercharged `rw`:

- exhaustively applies all given equations
- ...including all theorems marked with `@[simp]`

```
@[simp] theorem zero_add : 0 + n = n := ...
@[simp] theorem zero_mul : 0 * n = 0 := ...

example : 0 * n + (0 + n) = n := by simp
```

`simp`

`simp` is a supercharged `rw`:

- exhaustively applies all given equations
- ...including all theorems marked with `@[simp]`
- ...unfolding given definitions

```
by ...          -- ... ⊢ add n (succ m) = k
   simp [add]  -- ... ⊢ succ (add n m) = k
   ...
```

`simp`

`simp` is a supercharged `rw`:

- exhaustively applies all given equations
- ...including all theorems marked with `@[simp]`
- ...unfolding given definitions
- ...recursively solving hypotheses

```
example (h1 : y = 0 → x = 0) (h2 : p → 0 = y) (h3 : p) : x = 0 := by simp [h1, h2, h3]
```

`simp`

`simp` is a supercharged `rw`:

- exhaustively applies all given equations
- ...including all theorems marked with `@[simp]`
- ...unfolding given definitions
- ...recursively solving hypotheses
- ...preprocessing theorems not yet in equation form

```
by simp [
     show p x from ...,           -- interpreted as `p x = True`
     show p x ∧ ¬ p y from ...,  -- interpreted as rules `p x = True` and `p y = False`
     show p a ↔ p b from ...,    -- interpreted as `p a = p b`
     ...]
```

simp

`simp` is a supercharged `rw`:

- exhaustively applies all given equations
- ...including all theorems marked with `@[simp]`
- ...unfolding given definitions
- ...recursively solving hypotheses
- ...preprocessing theorems not yet in equation form
- ...rewriting under binders

```
example (xs : List Nat) : xs.map (fun n => n + 1) = xs.map (fun n => 1 + n) := by simp [Nat.add_comm]
```

`simp`

`simp` is a supercharged `rw`:

- exhaustively applies all given equations
- ...including all theorems marked with `@[simp]`
- ...unfolding given definitions
- ...recursively solving hypotheses
- ...preprocessing theorems not yet in equation form
- ...rewriting under binders
- ...and finally tries to close goals with `True.intro`

simp_all

simp_all is a supercharged simp :

- iteratively simplifies all current hypotheses and the goal up to fixpoint

```
example (h1 : n + m = m) (h2 : m = n) : n + n = n := by simp_all
```

simp_all

`simp_all` is a supercharged `simp`:

- iteratively simplifies all current hypotheses and the goal up to fixpoint

```
example (h1 : n + m = m) (h2 : m = n) : n + n = n := by simp_all
```

- includes propositions it finds on the way

```
example : (p x → f x = y) → (if p x then f x else y) = y := by simp_all
```

Proof Structuring

How not to write tactic proofs:

```
induction n
simp [foo]
rw [←bar]
simp [baz]
```

Which tactics belong to which case...?
Repairing tactic proofs is hard, repairing unstructured ones is harder!

Proof Structuring

How to write maintainable tactic proofs:

```
induction n
focus
  simp [foo]
  rw [←bar]
focus
  simp [baz]
```

Better: clearly separate each case

Proof Structuring

How to write maintainable tactic proofs:

```
induction n
case zero =>
  simp [foo]
  rw [←bar]
case succ n' ih =>
  simp [baz]
```

Better: reference cases by name (see infoview for case names)
Also allows reordering cases, e.g. to eliminate trivial cases with a final `allGoals`

Proof Structuring

Karlsruhe Institute of Technology

How to write maintainable tactic proofs:

```
induction n with
| zero =>
  simp [foo]
  rw [←bar]
| succ n' ih =>
  simp [baz]
```

Better: use special `induction`/`cases` syntax that also allows naming new variables

Proof Structuring

How to write maintainable tactic proofs:

```
induction n with
| zero =>
  simp only [foo]  -- like `simp`, but ignores `@[simp]` theorems
  rw [←bar]
| succ n' ih => simp [baz]
```

Better: use extensible, *fragile* tactics like `simp` at the end of a branch only

Put it in a **`have`** side proof if necessary

Help, My Variables Are Dying?!

Lean marks *inaccessible* variable names with a ✝ in the output

```
example : zero + n = n := by
  induction n
```

```
case zero
⊢ zero + zero = zero

case succ
n✝ : Nat
n_ih✝ : zero + n✝ = n✝
⊢ zero + succ n✝ = succ n✝
```

Variable names become inaccessible when

- *shadowed*, e.g. `fun x => ... (fun x => ...)`, or
- generated by a tactic, as above, to avoid fragile proof scripts
  Give them explicit names as on the previous slide instead if you need to access them