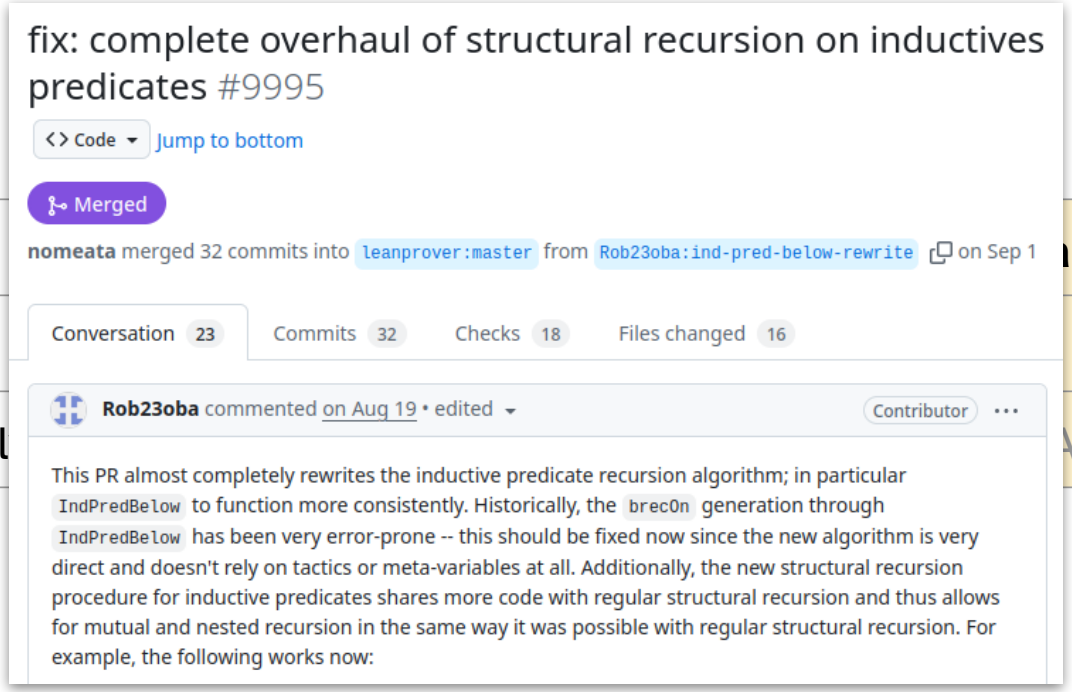
a mutual induction tactic for Lean 4

Jonathan Chan
24 October 2025

	Lean	**Rocq**	**Isabelle/HOL**
mutual induction **principles**	✅	✅	✅
mutually **recursive proofs**	✅ (v4.24.0)	✅	N/A

tactic-based proof assistants

		belle/HOL
mutual		
mutuall		A



tactic-based proof assistants

	Lean	Rocq	Isabelle/HOL
mutual induction **principles**	✅	✅	✅
mutually **recursive proofs**	✅ (v4.24.0)	✅	N/A
mutual induction **tactic**	🆕 (this talk)	❌	✅

tactic-based proof assistants

simple induction

inductive type

```
inductive Nat : Type where
  zero : Nat
  succ : Nat → Nat
```

induction principle (“recursor”)

```
Nat.rec :
  (motive : Nat → Prop) →
  -- case zero
  motive zero →
  -- case succ
  (∀n, motive n
     → motive (succ n)) →
  -- target, goal
  ∀n, motive n
```

simple induction

proof script

```
theorem mythm (n : Nat) :
  mygoal n := by
```

proof state

```
n : Nat
⊢ mygoal n
```

simple induction

proof script

```
theorem mythm (n : Nat) :
  mygoal n := by
  induction n
```

proof state

```
▼ case zero
⊢ mygoal zero

▼ case succ
n : Nat
IH : mygoal n
⊢ mygoal (succ n)
```

mutual induction

mutual inductive types

```
mutual

inductive Odd : Type where


inductive Even : Type where



end
```

mutual induction principle

```
???.rec :
  (motive_1 : Odd → Prop) →
  (motive_2 : Even → Prop) →


  -- target, goal
  ???
```

mutual induction

mutual inductive types

```
mutual

inductive Odd : Type where
  succ : Even → Odd

inductive Even : Type where



end
```

mutual induction principle

```
???.rec :
  (motive_1 : Odd → Prop) →
  (motive_2 : Even → Prop) →
  -- case Odd.succ
  (∀e, motive_1 e
     → motive_2 (succ e)) →


  -- target, goal
  ???
```

mutual induction

mutual inductive types

```
mutual

inductive Odd : Type where
  succ : Even → Odd

inductive Even : Type where
  zero : Even


end
```

mutual induction principle

```
???.rec :
  (motive_1 : Odd → Prop) →
  (motive_2 : Even → Prop) →
  -- case Odd.succ
  (∀e, motive_1 e
     → motive_2 (succ e)) →
  -- case Even.zero
  motive_2 zero →



  -- target, goal
  ???
```

mutual induction

mutual inductive types

```
mutual

inductive Odd : Type where
  succ : Even → Odd

inductive Even : Type where
  zero : Even
  succ : Odd → Even

end
```

mutual induction principle

```
???.rec :
  (motive_1 : Odd → Prop) →
  (motive_2 : Even → Prop) →
  -- case Odd.succ
  (∀e, motive_1 e
     → motive_2 (succ e)) →
  -- case Even.zero
  motive_2 zero →
  -- case Even.succ
  (∀o, motive_1 o
     → motive_2 (succ o)) →
  -- target, goal
  ???
```

mutual induction principles — two of them

```
Odd.rec :
  (motive_1 : Odd → Prop) →
  (motive_2 : Even → Prop) →
  -- case Odd.succ
  (∀e, motive_1 e
     → motive_2 (succ e)) →
  -- case Even.zero
  motive_2 zero →
  -- case Even.succ
  (∀o, motive_1 o
     → motive_2 (succ o)) →
  -- target, goal
  ∀o, motive_1 o
```

```
Even.rec :
  (motive_1 : Odd → Prop) →
  (motive_2 : Even → Prop) →
  -- case Odd.succ
  (∀e, motive_1 e
     → motive_2 (succ e)) →
  -- case Even.zero
  motive_2 zero →
  -- case Even.succ
  (∀o, motive_1 o
     → motive_2 (succ o)) →
  -- target, goal
  ∀e, motive_2 e
```

	Lean	**Rocq**
one motive, one goal	❌	`*_ind`
two motives, one goal	`*.rec`	`Scheme … := Induction …` `with … := Induction …`
two motives, ∧ goals	❌	`Combined Scheme … from …`

mutual induction principles in proof assistants

what I don’t like about applying conjoined eliminators

- order is fixed
- what if one motive is trivial?
  - *e.g.* $\vdash \Gamma$ and $\Gamma \vdash e : \tau$

what I don't like about applying conjoined eliminators

- order is fixed
- what if one motive is trivial? *e.g.* `⊢ Γ` and `Γ ⊢ e : τ`
- must generalize induction in theorem statement
  - looks ugly if the variables could instead be shared
    e.g. `(Γ : Ctxt) : (Γ ⊢ V : A → …) ∧ (Γ ⊢ M : B → …)`
  - sometimes idk what needs to be generalized until halfway thru proof

what I don't like about applying conjoined eliminators

- order is fixed
- what if one motive is trivial? *e.g.* `⊢ Γ` and `Γ ⊢ e : τ`
- must generalize induction in theorem statement
  - looks ugly if the variables could instead be shared
    e.g. `(Γ : Ctxt) : (Γ ⊢ V : A → …) ∧ (Γ ⊢ M : B → …)`
  - sometimes idk what needs to be generalized until halfway thru proof
- induction tactic (re)introduces variables *vs.* apply doesn't
  - generalized variables, inductive indices, induction hypotheses
    (and no more!)

```
                                    target these variables
mutual_induction x₁, …, xₙ -- in the next n goals (req)
  using r₁, …, rₙ          -- use these recursors (opt)
  generalizing y₁ … yₘ     -- generalizing these
                                    common variables   (opt)
```

mutual induction on Even/Odd

proof script

```
theorem mythm (k : Nat) : …

  := by …
  mutual_induction o, e
    generalizing k
```

proof state

```
▼ case odd
k : Nat
o : Odd
⊢ mygoal1 o

▼ case even
k : Nat
e : Even
⊢ mygoal2 e
```

mutual induction on Even/Odd

proof script

```
theorem mythm (k : Nat) : …

  := by …
  mutual_induction o, e
    generalizing k
```

single tactic acts on *multiple* goals simultaneously!

```
▼ case odd.succ
k : Nat
e : Even
IH : ∀k, mygoal2 e
⊢ mygoal1 (Odd.succ e)

▼ case even.zero
k : Nat
⊢ mygoal2 Even.zero

▼ case even.succ
k : Nat
o : Odd
IH : ∀k, mygoal1 o
⊢ mygoal2 (Even.succ o)
```

syntax design: optional arguments

Lean

```
mutual_induction x₁, …, xₙ
  using r₁, …, rₙ
  generalizing y₁ … yₘ
```

all goals must generalize
the same variables

Isabelle

```
apply (
  induct x₁ and … and xₙ
  arbitrary: y₁… and … and yₙ…
  rule: rs
)
```

each goal generalizes
their own set of variables

syntax design: optional arguments

Lean

```
mutual_induction x₁, …, xₙ
  using r₁, …, rₙ
  generalizing y₁…, …, yₙ…
```

what if only the last goal generalizes any variables?

```
  generalizing ,,,,, z
```

Isabelle

```
apply (
  induct x₁ and … and xₙ
  arbitrary: y₁… and … and yₙ…
  rule: rs
)
```

syntax design: optional arguments

Lean

```
mutual_induction
| case g₁ ⇒ x₁ using r₁
  generalizing y₁…
…
| case gₙ ⇒ xₙ using rₙ
  generalizing yₙ…
```

pros:

- can be specified in any order
- optional arguments omittable per-case

cons:

- anon. goals don’t have names
- much wordier
- unconventional syntax among Lean tactics

syntax design: mutual theorems

proof script

```
theorem mythm (k : Nat) : …

  := by …
  mutual_induction o, e
    generalizing k
```

proof state

```
▼ case odd
k : Nat
o : Odd
⊢ mygoal1 o

▼ case even
k : Nat
e : Even
⊢ mygoal2 e
```

syntax design: mutual theorems

proof script

```
theorem mythm (k : Nat) :
  (∀ o, mygoal1 o) ∧
  (∀ e, mygoal2 e) := by
  refine
    ⟨λo ↦ ?odd, λe ↦ ?even⟩
  mutual_induction o, e
    generalizing k
```

proof state

```
▼ case odd
k : Nat
o : Odd
⊢ mygoal1 o

▼ case even
k : Nat
e : Even
⊢ mygoal2 e
```

syntax design: mutual theorems

Lean (future work?)

```
joint (k : Nat)
theorem mythm1 o : mygoal1 o
theorem mythm2 e : mygoal2 e
by mutual_induction o, e
     generalizing k
```

Rocq

```
Lemma mythm1 k o : mygoal1 o
 with mythm2 k e : mygoal2 e.
Proof.
```

how does mutual_induction work?

1. compute for each target:
    a. identify inductive type & correct recursor
    b. collect variables to generalize over
    c. construct motive from goal + genvars
2. apply recursors to each goal
    a. instantiate missing motives with `True`
    b. retrieve new metavars for each case
3. deduplicate cases:
    a. pick canonical case and assign it to all other metavars
    b. solve trivial `True` cases
4. add metavars to list of goals
    a. intros genvars + constructor args + IHs

how does mutual_induction work?

1. compute for each target:
   a. recursor
   b. er
   c. construct motive from goal + genvars
2. app
   a. `True`
   b. retrieve new metavars for each case
3. dec
   a. to all other metavars
   b. solve trivial `True` cases
4. add
   a. intros genvars + constructor args + IHs

compute motives

apply recursors

combine cases

add as goals

1. compute motives

```
theorem mythm (m k : Nat) : … := by …
  mutual_induction o, e generalizing k
```

```
▼ case odd
n m k : Nat
eq : n + m = k
o : Odd
⊢ mygoal1 eq o

motive_1 o ≔ mygoal1 eq o
```

```
▼ case even
m k : Nat
e : Even
⊢ mygoal2 m e

motive_2 e ≔ mygoal2 m e
```

1. compute motives — close over unshared vars

```
theorem mythm (m k : Nat) : … := by …
  mutual_induction o, e generalizing k
```

```
▼ case odd
n m k : Nat
eq : n + m = k
o : Odd
⊢ mygoal1 eq o

motive_1 o :=
  ∀ (eq : n + m = k),
  mygoal1 eq o
```

```
▼ case even
m k : Nat
e : Even
⊢ mygoal2 m e

motive_2 e := mygoal2 m e
```

1. compute motives — close over unshared vars

```
theorem mythm (m k : Nat) : … := by …
  mutual_induction o, e generating k
```

```
▼ case odd
n m k : Nat
eq : n + m = k
o : Odd
⊢ mygoal1 eq o
```

```
motive_1 o :=
  ∀ n (eq : n + m = k),
  mygoal1 eq o
```

```
▼ case even
m k : Nat
e : Even
⊢ mygoal2 m e
```

```
motive_2 e := mygoal2 m e
```

1. compute motives — close over unshared vars + genvars

```
theorem mythm (m k : Nat) : … := by …
  mutual_induction o, e generalizing k
```

```
▼ case odd
n m k : Nat
eq : n + m = k
o : Odd
⊢ mygoal1 eq o

motive_1 o ≔
  ∀ n k (eq : n + m = k),
  mygoal1 eq o
```

```
▼ case even
m k : Nat
e : Even
⊢ mygoal2 m e

motive_2 e ≔ ∀ k, mygoal2 m e
```

2. apply recursors

```
▼ case odd
n m k : Nat
eq : n + m = k
o : Odd
⊢ mygoal1 eq o
```

```
▼ case even
m k : Nat
e : Even
⊢ mygoal2 m e
```

```
?odd ≔ Odd.rec
      (motive_1 ≔ λo. ∀ n k (eq : n + m = k), mygoal1 eq o)
      (motive_2 ≔ λe. ∀ k, mygoal2 m e)
```

```
?odd.Odd.succ
?odd.Even.zero
?odd.Even.zero
o n k eq
```

```
?even ≔ Even.rec
```

```
?even.Odd.succ
?even.Even.zero
?even.Even.zero
e k
```

2. apply recursors — trivialize missing motives

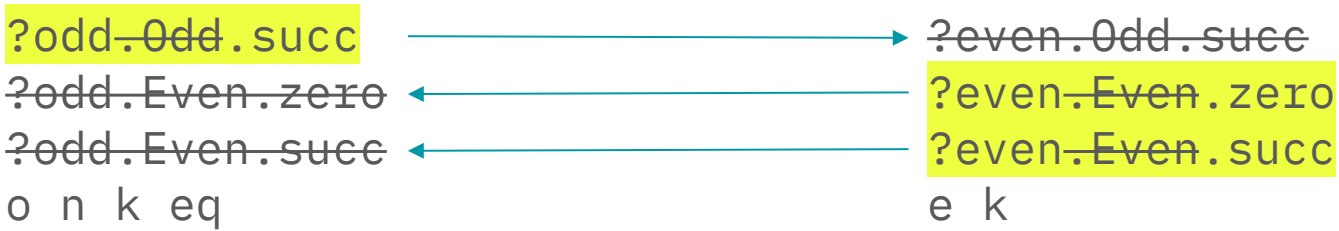
```
▼ case odd
n m k : Nat
eq : n + m = k
o : Odd
⊢ mygoal1 eq o

?odd ≔ Odd.rec
      (motive_1 ≔ λo. ∀ n k (eq : n + m = k), mygoal1 eq o)
      (motive_2 ≔ λ_. True)

      ?odd.Odd.succ
      ?odd.Even.zero
      ?odd.Even.zero
      o n k eq
```

3. combine cases

```
▼ case odd
n m k : Nat
eq : n + m = k
o : Odd
⊢ mygoal1 eq o
```

```
▼ case even
m k : Nat
e : Even
⊢ mygoal2 m e
```

```
?odd ≔ Odd.rec
      (motive_1 ≔ λo. ∀ n k (eq : n + m = k), mygoal1 eq o)
      (motive_2 ≔ λe. ∀ k, mygoal2 m e)
```



4. add as goals

```
▼ case odd.succ
m : Nat
⊢ ∀ e n k (eq : n + m = k),
  mygoal2 m e →
  mygoal1 eq (Odd.succ e)
```

```
▼ case even.zero
m : Nat
⊢ ∀ k, mygoal2 m zero
```

```
▼ case even.succ
m : Nat
⊢ ∀ o k,
  (∀ n k (eq : n + m = k),
     mygoal1 eq o) →
  mygoal2 m (Even.succ o)
```

4. add as goals — intro constructor arg

```
▼ case odd.succ
m : Nat
e : Even
⊢ ∀ n k (eq : n + m = k),
  mygoal2 m e →
  mygoal1 eq (Odd.succ e)
```

```
▼ case even.zero
m : Nat
⊢ ∀ k, mygoal2 m zero
```

```
▼ case even.zero
m : Nat
o : Odd
⊢ ∀ k,
  (∀ n k (eq : n + m = k),
     mygoal1 eq o) →
  mygoal2 m (Even.succ o)
```

4. add as goals — intro generalized vars

```
▼ case odd.succ
m n k : Nat
e : Even
eq : n + m = k
⊢ mygoal2 m e →
  mygoal1 eq (Odd.succ e)
```

```
▼ case even.zero
m k : Nat
⊢ mygoal2 m zero
```

```
▼ case even.zero
m k : Nat
o : Odd
⊢ (∀ n k (eq : n + m = k),
     mygoal1 eq o) →
  mygoal2 m (Even.succ o)
```

4. add as goals — intro IHs

```
▼ case odd.succ
m n k : Nat
e : Even
eq : n + m = k
IH : mygoal2 m e
⊢ mygoal1 eq (Odd.succ e)
```

```
▼ case even.zero
m k : Nat
⊢ mygoal2 m zero
```

```
▼ case even.zero
m k : Nat
o : Odd
IH : (∀ n k eq, mygoal1 eq o)
⊢ mygoal2 m (Even.succ o)
```

what about nested induction??

https://github.com/ionathanch/MutualInduction#towards-nested-induction-mk_all

Towards nested induction: `mk_all`

As an extra bonus, this repo also includes a `mk_all` attribute for inductive types that generates new definitions that lift predicates over the given parameters to predicates over that inductive type. For example, consider the following list type that we want to lift predicates over.

```
@[mk_all α]
inductive List (α : Type) where
  | nil : List α
  | cons : α → List α → List α
```

By default, `List` is in a `Type` universe. Then two definitions are generated: `List.all` and `List.iall`, which lift predicates over `α` to predicates over `List α` into `Prop` and into `Type`, respectively. Internally, the predicates are implemented using the list recursor, but are equivalent to the following recursive functions.

```
def List.all {α : Type} (P : α → Type) : List α → Type
  | nil => Unit
  | cons x xs => P x × List.all P xs

def List.iall {α : Type} (P : α → Prop) : List α → Prop
  | nil => True
  | cons x xs => P x ∧ List.iall P xs
```

If `List` is in the `Prop` universe, then only a `List.all` into `Prop` is generated as an inductive type.

try it yourself!

```
import Lake

open Lake DSL

require "ionathanch" / "MutualInduction" @ git "v0.1.0"
  from git "https://github.com/ionathanch/MutualInduction" @ "main"

package «your_package» where …
```

lakefile.lean