

# inductive types

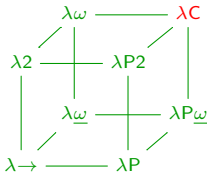
Freek Wiedijk

Type Theory & Coq

2023–2024

Radboud University Nijmegen

September 29, 2023



## introduction

today

---

minimal propositional logic   STT = simple type theory

minimal predicate logic    $\lambda P$  = dependent types

full Coq logic   CIC = Calculus of Inductive Constructions

$CIC = \lambda C + \text{inductive types} + \text{coinductive types} + \text{universes} + \dots$

## how are types introduced?

---

- ▶ free type variables

STT = simple type theory

- ▶ in the context

PTSs = pure type systems  $\lambda \rightarrow \lambda P \lambda 2 \lambda C$

`nat : *, O : nat, S : nat  $\rightarrow$  nat  $\vdash$  S (S (S O)) : nat`

- ▶ definitions

CIC = Calculus of Inductive Constructions

```
Inductive nat : Set :=  
  | 0 : nat  
  | S : nat -> nat.
```

## definitions in Coq

---

- ▶ axioms

environment used like the context in  $\lambda C$   
disadvantage: reductions will get stuck

Axiom Parameter

- ▶ definitions of constants

Definition

Lemma

Qed

- ▶ inductive definitions

Inductive

# CIC

## variants

---

$$\begin{aligned} \text{CIC} &= \text{Calculus of Inductive Constructions} \\ &= \\ &\lambda\text{C} = \text{Calculus of Constructions} \\ &+ \\ &\text{MLTT} = \text{Martin-Löf type theory} \end{aligned}$$

different systems have different variants of CIC:

- ▶ Coq
- ▶ Agda
- ▶ Lean
- ▶ ...



Thierry Coquand



Per Martin-Löf

## typing rules

---

### STT

3 rules

$$\Gamma \vdash M : A$$

### PTSs

7 rules

$$\Gamma \vdash M : A$$

$$M =_{\beta} N$$

### CIC

*many* rules

chapter 2.1 of the Coq manual

$$\mathcal{WF}(E)[\Gamma]$$

$$E[\Gamma] \vdash M : A$$

$$E[\Gamma] \vdash M =_{\beta\delta\iota\eta\zeta} N$$

$$E[\Gamma] \vdash M \leq_{\beta\delta\iota\eta\zeta} N$$

## examples of CIC typing rules from the Coq manual

---

$$\frac{\left\{ \begin{array}{l} \text{Ind } [p] (\Gamma_I := \Gamma_C) \in E \\ (E[] \vdash q_l : P'_l)_{l=1\dots r} \\ (E[] \vdash P'_l \leq_{\beta\delta\iota\zeta\eta} P_l \{p_u/q_u\}_{u=1\dots l-1})_{l=1\dots r} \\ 1 \leq j \leq k \end{array} \right.}{E[] \vdash I_j q_1 \dots q_r : \forall [p_{r+1} : P_{r+1}; \dots; p_p : P_p], (A_j)_{/s_j}}$$

$$\frac{\begin{array}{l} E[\Gamma] \vdash c : (I q_1 \dots q_r t_1 \dots t_s) \\ E[\Gamma] \vdash P : B \\ [(I q_1 \dots q_r) | B] \\ (E[\Gamma] \vdash f_i : \{(c_{p_i} q_1 \dots q_r)\}^P)_{i=1\dots l} \end{array}}{E[\Gamma] \vdash \mathbf{case}(c, P, f_1 | \dots | f_l) : (P t_1 \dots t_s c)}$$

## context versus environment

---

$$E[\Gamma] \vdash M : A$$

- ▶  $E$  is the **environment** of axioms and definitions
- ▶  $\Gamma$  is the **context** of local variables



## example of context versus environment

---

Parameter  $a : \text{Prop}$ .

Definition  $I : a \rightarrow a :=$

fun  $x : a \Rightarrow x$ .

the typing judgment for the subterm  $x$ :

$$(a : *) [x : a] \vdash x : a$$

$a$  is in the environment

$x$  is in the context

## example of context versus environment

---

Parameter  $a : \text{Prop}$ .

Definition  $I : a \rightarrow a :=$

fun  $x : a \Rightarrow x$ .

the typing judgment for the subterm  $x$ :

$$(a : *) [x : a] \vdash x : a$$

$a$  is in the environment

$x$  is in the context

after these three lines the environment is:

$$\underbrace{a : *}_{\text{axiom}}, \underbrace{I := (\lambda x : a. x) : a \rightarrow a}_{\text{definition}}$$

## STT

$$A, B ::= a \mid A \rightarrow B$$
$$M, N ::= x \mid MN \mid \lambda x : A. M$$

 $\lambda$ C

$$M, N, A, B ::= x \mid MN \mid \lambda x : A. M \mid \Pi x : A. B \mid s$$
$$s ::= * \mid \square$$

## CIC

$$M, N, A, B ::= x \mid MN \mid \lambda x : A. M \mid \Pi x : A. B \mid s \mid$$
$$\text{let } x := N : A \text{ in } M \mid$$
$$\text{fix } \dots \mid \text{match } \dots \mid \dots$$
$$s ::= \text{Set} \mid \text{Prop} \mid \text{SProp} \mid \text{Type}(i)$$

the universe levels  $i$  are explicit natural numbers

$\lambda C$

$*$  :  $\square$

CIC

$\{\text{Set}, \text{Prop}, \text{SProp}\} : \text{Type}(1) : \text{Type}(2) : \text{Type}(3) : \dots$

in  $\lambda C$  the sort  $\square$  does not have a type  
in CIC *every* term has a type

the universe  $\text{Type}(1)$  is often used like  $*$  too  
the universe levels  $i$  are generally inferred by the system

SProp is a proof irrelevant version of Prop

## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

True  
: Prop

## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

Check (True : Set).

True : Set

: Set

## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

Check (True : Set).

Check (True : Type).

True : Type

: Type



## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

Check (True : Set).

Check (True : Type).

Check nat.

nat

: Set

## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

Check (True : Set).

Check (True : Type).

Check nat.

Check (nat : Type).

nat : Type

: Type

## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

Check (True : Set).

Check (True : Type).

Check nat.

Check (nat : Type).

Check (nat : Prop).

Error:

The term "nat" has type "Set"  
while it is expected to have type  
"Prop"

(universe inconsistency: Cannot enforce  
Set = Prop).

## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

Check (True : Set).

Check (True : Type).

Check nat.

Check (nat : Type).

Check (nat : Prop).

Check (Type : Type).

Type : Type

: Type

## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

Check (True : Set).

Check (True : Type).

Check nat.

Check (nat : Type).

Check (nat : Prop).

Check (Type : Type).

conversion rule:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A' : s}{\Gamma \vdash M : A'} A =_{\beta} A'$$

## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

Check (True : Set).

Check (True : Type).

Check nat.

Check (nat : Type).

Check (nat : Prop).

Check (Type : Type).

conversion rule:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A' : s \quad A =_{\beta} A'}{\Gamma \vdash M : A'}$$

## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

Check (True : Set).

Check (True : Type).

Check nat.

Check (nat : Type).

Check (nat : Prop).

Check (Type : Type).

conversion rule:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A' : s \quad A =_{\beta\delta\iota\zeta\eta} A'}{\Gamma \vdash M : A'}$$

## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

Check (True : Set).

Check (True : Type).

Check nat.

Check (nat : Type).

Check (nat : Prop).

Check (Type : Type).

conversion rule:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A' : s \quad A \leq_{\beta\delta\iota\zeta\eta} A'}{\Gamma \vdash M : A'}$$



## subtyping

---

$\text{Prop} \leq \text{Set} \leq \text{Type}(1) \leq \text{Type}(2) \leq \text{Type}(3) \leq \dots$

Check True.

Check (True : Set).

Check (True : Type).

Check nat.

Check (nat : Type).

Check (nat : Prop).

Check (Type : Type).

conversion rule:

$$\frac{E[\Gamma] \vdash M : A \quad E[\Gamma] \vdash A' : s \quad E[\Gamma] \vdash A \leq_{\beta\delta\iota\zeta\eta} A'}{E[\Gamma] \vdash M : A'}$$

## reduction

---

fun	$\beta \eta$
Definition	$\delta$
fix match	$\iota$
let	$\zeta$

## reduction

---

<b>fun</b>	$\beta$	$\eta$
Definition	$\delta$	
fix match	$\iota$	
<b>let</b>	$\zeta$	

$$(\lambda x : A. M)N \rightarrow_{\beta} M[x := N]$$

$$\lambda x : A. (Fx) \rightarrow_{\eta} F$$

when  $F : (\Pi x : A. B)$

$$\text{let } x := N : A \text{ in } M \rightarrow_{\zeta} M[x := N]$$

## why let-in definitions when we have beta redexes?

---

let  $A := \text{nat} : \text{Set}$  in  $(\lambda x : A. x) \text{O}$   
is well-typed

$(\lambda A : \text{Set}. ((\lambda x : A. x) \text{O})) \text{nat}$   
is not well-typed

because the subterm

$\lambda A : \text{Set}. ((\lambda x : A. x) \text{O})$   
is not well-typed

## defining constants in Coq

---

```
Definition two : nat :=      two is defined
  S (S 0).
```

## defining constants in Coq

---

```
Definition two : nat :=      two = S (S 0)
  S (S 0).                  : nat
Print two.
```

## defining constants in Coq

---

```
Definition two : nat :=  
  S (S 0).
```

```
Print two.
```

```
Lemma two' : nat.
```

```
1 subgoal (ID 1)
```

```
=====
```

```
nat
```

## defining constants in Coq

---

```
Definition two : nat :=  
  S (S 0).
```

```
Print two.
```

```
Lemma two' : nat.
```

```
apply S.
```

```
1 subgoal (ID 2)
```

```
=====
```

```
nat
```



## defining constants in Coq

---

```
Definition two : nat :=  
  S (S 0).
```

```
Print two.
```

```
Lemma two' : nat.
```

```
apply S.
```

```
apply S.
```

```
1 subgoal (ID 3)
```

```
=====
```

```
nat
```

## defining constants in Coq

---

```
Definition two : nat :=      No more subgoals.
```

```
  S (S 0).
```

```
Print two.
```

```
Lemma two' : nat.
```

```
  apply S.
```

```
  apply S.
```

```
  apply 0.
```

## defining constants in Coq

---

```
Definition two : nat :=  
  S (S 0).  
Print two.
```

```
Lemma two' : nat.  
apply S.  
apply S.  
apply 0.  
Qed.
```

## defining constants in Coq

---

```
Definition two : nat :=      two' = S (S 0)
  S (S 0).                  : nat
Print two.
```

```
Lemma two' : nat.
apply S.
apply S.
apply 0.
Qed.
Print two'.
```

## defining constants in Coq

---

```
Definition two : nat :=  
  S (S 0).
```

```
Print two.
```

```
Lemma two' : nat.
```

```
  apply S.
```

```
  apply S.
```

```
  apply 0.
```

```
Qed.
```

```
Print two'.
```

```
1 subgoal (ID 3)
```

```
=====
```

```
two = two'
```

```
Lemma eq_two : two = two'.
```

## defining constants in Coq

---

```
Definition two : nat :=      Error: two' is opaque.
```

```
  S (S 0).
```

```
Print two.
```

```
Lemma two' : nat.
```

```
  apply S.
```

```
  apply S.
```

```
  apply 0.
```

```
  Qed.
```

```
Print two'.
```

```
Lemma eq_two : two = two'.
```

```
  unfold two, two'.
```

## defining constants in Coq

---

```
Definition two : nat :=  
  S (S 0).  
Print two.
```

```
Lemma two' : nat.  
apply S.  
apply S.  
apply 0.  
Defined.
```

## defining constants in Coq

---

```
Definition two : nat :=      two' = S (S 0)
  S (S 0).                  : nat
Print two.
```

```
Lemma two' : nat.
apply S.
apply S.
apply 0.
Defined.
Print two'.
```



## defining constants in Coq

---

```
Definition two : nat :=      1 subgoal (ID 3)
  S (S 0).
Print two.                  =====
                             two = two'

Lemma two' : nat.
apply S.
apply S.
apply 0.
Defined.
Print two'.
```

Lemma eq\_two : two = two'.

## defining constants in Coq

---

```
Definition two : nat :=  
  S (S 0).
```

```
Print two.
```

```
1 subgoal (ID 5)
```

```
=====
```

```
S (S 0) = S (S 0)
```

```
Lemma two' : nat.
```

```
  apply S.
```

```
  apply S.
```

```
  apply 0.
```

```
Defined.
```

```
Print two'.
```

```
Lemma eq_two : two = two'.
```

```
  unfold two, two'.
```

## defining constants in Coq

---

```
Definition two : nat :=      No more subgoals.
  S (S 0).
Print two.
```

```
Lemma two' : nat.
  apply S.
  apply S.
  apply 0.
Defined.
Print two'.
```

```
Lemma eq_two : two = two'.
  unfold two, two'.
  reflexivity.
```

## defining constants in Coq

---

```
Definition two : nat :=  
  S (S 0).  
Print two.
```

```
Lemma two' : nat.  
apply S.  
apply S.  
apply 0.  
Defined.  
Print two'.
```

```
Lemma eq_two : two = two'.  
unfold two, two'.  
reflexivity.  
Qed.
```

## defining constants in Coq

---

```
Definition two : nat :=      No more subgoals.  
  S (S 0).  
Print two.
```

```
Lemma two' : nat.  
  apply S.  
  apply S.  
  apply 0.  
  Defined.  
Print two'.
```

```
Lemma eq_two : two = two'.  
  reflexivity.
```

## defining constants in Coq

---

```
Definition two : nat :=  
  S (S 0).  
Print two.
```

```
Lemma two' : nat.  
apply S.  
apply S.  
apply 0.  
Defined.  
Print two'.
```

```
Lemma eq_two : two = two'.  
reflexivity.  
Qed.
```

## defining constants in Coq

---

```
Definition two : nat :=  
  S (S 0).  
Print two.
```

```
Definition two' : nat.  
  apply S.  
  apply S.  
  apply 0.  
Defined.  
Print two'.
```

```
Lemma eq_two : two = two'.  
  reflexivity.  
Qed.
```

## defining constants in Coq

---

```
Definition two : nat :=  
  S (S 0).  
Print two.
```

```
Definition two' : nat.  
  apply S.  
  apply S.  
  apply 0.  
Defined.  
Print two'.
```

```
Lemma eq_two : two = two'.  
  reflexivity.  
Qed.
```

delta reduction:

$$\begin{aligned} \text{two} &\rightarrow_{\delta} S (S 0) \\ \text{two}' &\rightarrow_{\delta} S (S 0) \end{aligned}$$



## the natural numbers

### defining an inductive type

---

```
Inductive nat : Set :=  
| 0 : nat  
| S : nat -> nat.
```

$$\text{nat} = \{0, S\ 0, S\ (S\ 0), S\ (S\ (S\ 0)), \dots\}$$

## what is a type?

---

- ▶ syntax
  - ▶ string over some alphabet
- ▶ semantics: 'something like a set'
  - ▶ function types
  - ▶ inductive types

## what is a type?

---

- ▶ syntax
  - ▶ string over some alphabet
- ▶ semantics: 'something like a set'
  - ▶ function types
  - ▶ inductive types

an inductive type 'consists of'  
the terms you can make with the constructors

## what is a type?

---

- ▶ syntax
  - ▶ string over some alphabet
- ▶ semantics: 'something like a set'
  - ▶ function types
  - ▶ inductive types

an inductive type 'consists of'  
the terms you can make with the constructors

more precisely: the closed terms in normal form

closed = no free variables

normal form = does not reduce any further

normal forms are unique (CR = Church-Rosser)

every well-typed term has a normal form (SN = Strong Normalization)

Bishop-style **constructive** mathematics ( $\approx$  Coq)

classical mathematics

$\forall x \in \mathbb{R}. (x > 0) \vee \neg(x > 0)$

discontinuous functions

**intuitionistic** mathematics

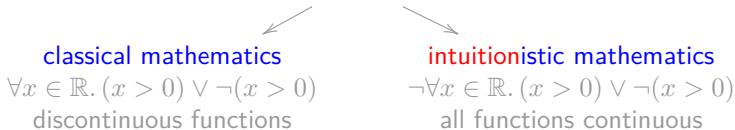
$\neg \forall x \in \mathbb{R}. (x > 0) \vee \neg(x > 0)$

all functions continuous



L.E.J. Brouwer

### Bishop-style constructive mathematics ( $\approx$ Coq)



the **ur-intuition** of time (synthetic a priori):

Deze intuïtie der **twee-eenigheid**, deze oerintuïtie der wiskunde scheidt niet alleen de getallen één en twee, doch tevens alle eindige ordinaalgetallen, daar één der elementen der twee-eenigheid als een nieuwe twee-eenigheid kan worden gedacht, en dit proces een **willekeurig aantal malen kan worden herhaald**.



L.E.J. Brouwer

Bishop-style constructive mathematics ( $\approx$  Coq)

classical mathematics

$\forall x \in \mathbb{R}. (x > 0) \vee \neg(x > 0)$

discontinuous functions

intuitionistic mathematics

$\neg \forall x \in \mathbb{R}. (x > 0) \vee \neg(x > 0)$

all functions continuous

the **ur-intuition** of time (synthetic a priori):

This intuition of **two-oneness**, the basal intuition of mathematics, creates not only the numbers one and two, but also all finite ordinal numbers, inasmuch as one of the elements of the two-oneness may be thought of as a new two-oneness, which process **may be repeated indefinitely**.



L.E.J. Brouwer

## natural numbers in Coq

---

```
Inductive nat : Set :=  
| 0 : nat  
| S : nat -> nat.
```

```
nat is defined  
nat_rect is defined  
nat_ind is defined  
nat_rec is defined  
nat_sind is defined
```



## natural numbers in Coq

---

```
Inductive nat : Set :=      nat
| 0 : nat                  : Set
| S : nat -> nat.
```

Check nat.

## natural numbers in Coq

---

```
Inductive nat : Set := 0
| 0 : nat
| S : nat -> nat
```

Check nat.

Check 0.

## natural numbers in Coq

---

```
Inductive nat : Set := S
| 0 : nat
| S : nat -> nat
```

Check nat.

Check 0.

Check S.

## natural numbers in Coq

---

```
Inductive nat : Set :=          nat_ind
| 0 : nat                      : forall P : nat -> Prop,
| S : nat -> nat.              P 0 ->
                               (forall n : nat,
                                P n -> P (S n)) ->
                               forall n : nat, P n

Check nat.
Check 0.
Check S.
Check nat_ind.
```

## natural numbers in Coq

---

```
Inductive nat : Set :=
| 0 : nat
| S : nat -> nat.

Check nat.
Check 0.
Check S.
Check nat_ind.
Check nat_sind.
```

```
nat_sind
: forall P : nat -> SProp,
  P 0 ->
  (forall n : nat,
    P n -> P (S n)) ->
  forall n : nat, P n
```

## natural numbers in Coq

---

```
Inductive nat : Set :=          nat_rec
| 0 : nat                      : forall P : nat -> Set,
| S : nat -> nat.              P 0 ->
                               (forall n : nat,
                                P n -> P (S n)) ->
                               forall n : nat, P n

Check nat.
Check 0.
Check S.
Check nat_ind.
Check nat_sind.
Check nat_rec.
```

## natural numbers in Coq

---

```
Inductive nat : Set :=
| 0 : nat
| S : nat -> nat.

Check nat.
Check 0.
Check S.
Check nat_ind.
Check nat_sind.
Check nat_rec.
Check nat_rect.
```

```
nat_rect
: forall P : nat -> Type,
  P 0 ->
  (forall n : nat,
    P n -> P (S n)) ->
  forall n : nat, P n
```

## natural numbers in Coq

---

```
Inductive nat : Set :=  
| 0 : nat  
| S : nat -> nat.
```

```
Inductive nat : Set :=  
0 : nat | S : nat -> nat
```

```
Check nat.  
Check 0.  
Check S.  
Check nat_ind.  
Check nat_sind.  
Check nat_rec.  
Check nat_rect.
```

```
Print nat.
```



## natural numbers in Coq

---

```
Inductive nat : Set :=  
  | 0 : nat  
  | S : nat -> nat.
```

```
Inductive nat : Set :=  
  0 : nat | S : nat -> nat
```

```
Check nat.
```

```
Check 0.
```

```
Check S.
```

```
Check nat_ind.
```

```
Check nat_sind.
```

```
Check nat_rec.
```

```
Check nat_rect.
```

```
Print nat.
```

```
Print 0.
```

## natural numbers in Coq

---

```
Inductive nat : Set :=  
| 0 : nat  
| S : nat -> nat.
```

```
Inductive nat : Set :=  
0 : nat | S : nat -> nat
```

```
Check nat.
```

```
Check 0.
```

```
Check S.
```

```
Check nat_ind.
```

```
Check nat_sind.
```

```
Check nat_rec.
```

```
Check nat_rect.
```

```
Print nat.
```

```
Print 0.
```

```
Print S.
```

## natural numbers in Coq

---

```
Inductive nat : Set :=  
| 0 : nat  
| S : nat -> nat.
```

```
Check nat.
```

```
Check 0.
```

```
Check S.
```

```
Check nat_ind.
```

```
Check nat_sind.
```

```
Check nat_rec.
```

```
Check nat_rect.
```

```
Print nat.
```

```
Print 0.
```

```
Print S.
```

```
Print nat_ind.
```

```
nat_ind =  
fun (P : nat -> Prop) (f : P 0)  
  (f0 : forall n : nat,  
    P n -> P (S n)) =>  
fix F (n : nat) : P n :=  
  match n as n0 return (P n0) with  
  | 0 => f  
  | S n0 => f0 n0 (F n0)  
end  
: forall P : nat -> Prop,  
  P 0 ->  
  (forall n : nat,  
    P n -> P (S n)) ->  
  forall n : nat, P n
```

```
Arguments nat_ind _%function_scope  
_ _%function_scope
```

## natural numbers in Coq

---

```
Inductive nat : Set :=  
| 0 : nat  
| S : nat -> nat.
```

```
Check nat.
```

```
Check 0.
```

```
Check S.
```

```
Check nat_ind.
```

```
Check nat_sind.
```

```
Check nat_rec.
```

```
Check nat_rect.
```

```
Print nat.
```

```
Print 0.
```

```
Print S.
```

```
Print nat_ind.
```

```
Print nat_rect.
```

```
nat_rect =  
fun (P : nat -> Type) (f : P 0)  
  (f0 : forall n : nat,  
    P n -> P (S n)) =>  
fix F (n : nat) : P n :=  
  match n as n0 return (P n0) with  
  | 0 => f  
  | S n0 => f0 n0 (F n0)  
end  
: forall P : nat -> Type,  
  P 0 ->  
  (forall n : nat,  
    P n -> P (S n)) ->  
  forall n : nat, P n  
Arguments nat_ind _%function_scope  
  _ _%function_scope
```

## natural numbers in Coq

---

```
Inductive nat : Set :=  
| 0 : nat  
| S : nat -> nat.
```

```
Check nat.
```

```
Check 0.
```

```
Check S.
```

```
Check nat_ind.
```

```
Check nat_sind.
```

```
Check nat_rec.
```

```
Check nat_rect.
```

```
Print nat.
```

```
Print 0.
```

```
Print S.
```

```
Print nat_ind.
```

```
Print nat_rect.
```

```
nat_rect =  
fun (P : nat -> Type) (f : P 0)  
  (f0 : forall n : nat,  
    P n -> P (S n)) =>  
fix F (n : nat) : P n :=  
  match n as n0 return (P n0) with  
  | 0 => f  
  | S n0 => f0 n0 (F n0)  
end  
: forall P : nat -> Type,  
  P 0 ->  
  (forall n : nat,  
    P n -> P (S n)) ->  
  forall n : nat, P n
```

```
Arguments nat_ind _%function_scope  
_ _%function_scope
```

## the constants defined by an inductive type definition

---

```
Inductive nat : Set :=  
| 0 : nat  
| S : nat -> nat.
```

makes three kinds of constants available:

- ▶ the type  
primitive

nat : Set

- ▶ the constructors  
primitive

O : nat

S : nat → nat

- ▶ the destructors  
= eliminators = induction principles  
= recursors = recursion principles

defined using 'fix' and 'match'

## induction / recursion principles

---

nat\_ind : ...

nat\_sind : ...

nat\_rec : ...

nat\_rect : ...

correspond to predicates in {Prop, SProp, Set, Type}

## induction / recursion principles

---

`nat_ind` : ...  
`nat_sind` : ...  
`nat_rec` : ...  
`nat_rect` : ...

correspond to predicates in {Prop, SProp, Set, Type}

two variants:

- ▶ **dependent principle**  
(looks more complicated, easier to understand)
- ▶ **non-dependent principle**  
(can be derived from the dependent principle)

inductive types in **Prop** with more than two constructors:

program extraction  $\longrightarrow$  **only the first two, non-dependent**

inductive types in **Set** or **Type**: **all four, dependent**



## defining addition

---

```
Fixpoint add (n m : nat) {struct n} : nat :=  
  match n with  
  | 0 => m  
  | S n' => S (add n' m)  
  end.
```

## defining addition

---

```
Fixpoint add (n m : nat) {struct n} : nat :=  
  match n with  
  | 0 => m  
  | S n' => S (add n' m)  
  end.
```

structural recursion: recursive call has to be on a smaller term

## defining addition

---

```
Fixpoint add (n m : nat) : nat :=  
  match n with  
  | 0 => m  
  | S n' => S (add n' m)  
  end.
```

structural recursion: recursive call has to be on a smaller term

## defining addition

---

```
Fixpoint add (n m : nat) : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

structural recursion: recursive call has to be on a smaller term

```
Definition add' (n m : nat) : nat.
induction n as [|n' r].
- apply m.
- apply S. apply r.
Defined.
```

```
Definition add'' (n m : nat) : nat :=
  nat_rec (fun _ => nat) m (fun n' r => S r) n.
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

add is defined

add is recursively defined (guarded on

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

**Print add.**

```
add =
fix add (n m : nat) {struct n} :
  nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end
  : nat -> nat -> nat
```

## recursive definitions in Coq

---

```
Definition add := add is defined
  fix add (n m : nat)
    : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

## recursive definitions in Coq

---

```
Definition add :=  
  fix add (n m : nat)  
    : nat :=  
    match n with  
    | 0 => m  
    | S n' => S (add n' m)  
    end.
```

Print add.

```
add =  
fix add (n m : nat) {struct n} :  
  nat :=  
  match n with  
  | 0 => m  
  | S n' => S (add n' m)  
  end  
  : nat -> nat -> nat
```



## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
Print add.
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

Lemma add\_1\_1 :

add (S 0) (S 0) = S (S 0).

1 subgoal (ID 5)

```
=====
add (S 0) (S 0) = S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
cbv delta.
```

1 subgoal (ID 7)

```
=====
(fix add (n m : nat) {struct n} :
  nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end) (S 0) (S 0) =
S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
cbv delta. cbv iota.
```

```
1 subgoal (ID 9)
```

```
=====
(fun n m : nat =>
  match n with
  | 0 => m
  | S n' =>
      S
      ((fix add
        (n0 m0 : nat) {struct
          n0} : nat :=
        match n0 with
        | 0 => m0
        | S n'0 =>
            S (add n'0 m0)
        end) n' m)
      end) (S 0) (S 0) = S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
cbv delta. cbv iota.
cbv beta.
```

```
1 subgoal (ID 11)
```

```
=====
match S 0 with
| 0 => S 0
| S n' =>
  S
  ((fix add
    (n m : nat) {struct n} :
    nat :=
    match n with
    | 0 => m
    | S n'0 =>
      S (add n'0 m)
    end) n' (S 0))
end = S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
cbv delta. cbv iota.
cbv beta. cbv iota.
```

1 subgoal (ID 13)

```
=====
(fun n' : nat =>
  S
    ((fix add
      (n m : nat) {struct n} :
        nat :=
        match n with
        | 0 => m
        | S n'0 => S (add n'0 m)
        end) n' (S 0))) 0 =
  S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
cbv delta. cbv iota.
cbv beta. cbv iota.
cbv beta.
```

```
1 subgoal (ID 15)
```

```
=====
S
  ((fix add
    (n m : nat) {struct n} :
      nat :=
      match n with
      | 0 => m
      | S n' => S (add n' m)
      end) 0 (S 0)) =
S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
cbv delta. cbv iota.
cbv beta. cbv iota.
cbv beta. cbv iota.
```

1 subgoal (ID 17)

```
=====
S
((fun n m : nat =>
  match n with
  | 0 => m
  | S n' =>
    S
      ((fix add
        (n0 m0 : nat)
        {struct n0} :
          nat :=
          match n0 with
          | 0 => m0
          | S n'0 =>
            S (add n'0 m0)
          end) n' m)
      end) 0 (S 0)) = S (S 0)
```



## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
cbv delta. cbv iota.
cbv beta. cbv iota.
cbv beta. cbv iota.
cbv beta.
```

1 subgoal (ID 19)

```
=====
S
  match 0 with
  | 0 => S 0
  | S n' =>
    S
      ((fix add
        (n m : nat) {struct
          n} : nat :=
          match n with
          | 0 => m
          | S n'0 =>
            S (add n'0 m)
          end) n' (S 0))
      end = S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
cbv delta. cbv iota.
cbv beta. cbv iota.
cbv beta. cbv iota.
cbv beta. cbv iota.
```

1 subgoal (ID 21)

```
=====
S (S 0) = S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
compute.
```

1 subgoal (ID 7)

```
=====
S (S 0) = S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
vm_compute.
```

1 subgoal (ID 7)

```
=====
S (S 0) = S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
native_compute.
```

1 subgoal (ID 7)

```
=====
S (S 0) = S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
simpl.
```

1 subgoal (ID 9)

```
=====
S (S 0) = S (S 0)
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)      No more subgoals.
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
simpl.
reflexivity.
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
unfold add.
```

1 subgoal (ID 7)

```
=====
S (S 0) = S (S 0)
```



## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)      No more subgoals.
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
unfold add.
reflexivity.
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)      No more subgoals.
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
reflexivity.
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)      No more subgoals.
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
auto.
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
simpl.
reflexivity.
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)
  : nat :=
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
simpl.
reflexivity.
Qed.
```

## recursive definitions in Coq

---

```
Fixpoint add (n m : nat)           = S (S 0)
  : nat :=                          : nat
  match n with
  | 0 => m
  | S n' => S (add n' m)
  end.
```

Print add.

```
Lemma add_1_1 :
  add (S 0) (S 0) = S (S 0).
simpl.
reflexivity.
Qed.
```

```
Eval compute in
  add (S 0) (S 0).
```

## iota reduction

---

fun	$\beta$	$\eta$
Definition	$\delta$	
fix match	$\iota$	
let	$\zeta$	

constructor  
↓  
(fix  $f \dots := M$ ) ... ( $c \dots$ ) ...  
↓  
 $M[f := (\text{fix } f \dots := M)] \dots (c \dots) \dots$

match ( $c N_1 \dots N_k$ ) with ... | ( $c x_1 \dots x_k$ )  $\Rightarrow M$  | ... end

↓  
 $M[x_1 := N_1, \dots, x_k := N_k]$

## induction in Coq

---

```
Lemma add_0_n (n : nat) :  
  add 0 n = n.
```

```
1 subgoal (ID 8)
```

```
n : nat
```

```
=====
```

```
add 0 n = n
```



## induction in Coq

---

```
Lemma add_0_n (n : nat) :    No more subgoals.  
  add 0 n = n.  
reflexivity.
```

## induction in Coq

---

```
Lemma add_0_n (n : nat) :  
  add 0 n = n.  
reflexivity.  
Qed.
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :  
  add n 0 = n.
```

```
1 subgoal (ID 11)
```

```
n : nat
```

```
=====
```

```
add n 0 = n
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :  
  add n 0 = n.  
reflexivity.
```

```
Error:  
In environment  
n : nat  
Unable to unify "n" with  
"add n 0".
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :  
  add n 0 = n.  
induction n as [|n' IH].
```

```
2 subgoals (ID 15)
```

```
=====
```

```
add 0 0 = 0
```

```
subgoal 2 (ID 18) is:
```

```
add (S n') 0 = S n'
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 15)
  add n 0 = n.
induction n as [|n' IH].      =====
-                               add 0 0 = 0
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 18)
  add n 0 = n.
induction n as [|n' IH].      subgoal 1 (ID 18) is:
- reflexivity.                add (S n') 0 = S n'
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 18)
  add n 0 = n.
induction n as [|n' IH].      n' : nat
- reflexivity.                IH : add n' 0 = n'
-                               =====
                               add (S n') 0 = S n'
```



## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 22)
  add n 0 = n.
induction n as [|n' IH].      n' : nat
- reflexivity.                IH : add n' 0 = n'
- simpl.                      =====
                              S (add n' 0) = S n'
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 23)
  add n 0 = n.
induction n as [|n' IH].      n' : nat
- reflexivity.                IH : add n' 0 = n'
- simpl. rewrite IH.          =====
                               S n' = S n'
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      No more subgoals.
  add n 0 = n.
induction n as [|n' IH].
- reflexivity.
- simpl. rewrite IH.
  reflexivity.
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :  
  add n 0 = n.  
induction n as [|n' IH].  
- reflexivity.  
- simpl. rewrite IH.  
  reflexivity.
```

Qed.

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 14)
  add n 0 = n.
induction n as [|n' IH].      n, m : nat
- reflexivity.                =====
- simpl. rewrite IH.         nat
  reflexivity.
Qed.
```

```
Definition add' (n m : nat)
  : nat.
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      2 subgoals (ID 18)
  add n 0 = n.
induction n as [|n' IH].      m : nat
- reflexivity.                =====
- simpl. rewrite IH.         nat
  reflexivity.
Qed.                            subgoal 2 (ID 21) is:
                                nat

Definition add' (n m : nat)
  : nat.
destruct n as [|n']
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 18)
  add n 0 = n.
induction n as [|n' IH].      m : nat
- reflexivity.                =====
- simpl. rewrite IH.         nat
  reflexivity.
Qed.
```

```
Definition add' (n m : nat)
  : nat.
destruct n as [|n']
-
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 21)
  add n 0 = n.
induction n as [|n' IH].      subgoal 1 (ID 21) is:
- reflexivity.                nat
- simpl. rewrite IH.
  reflexivity.
Qed.
```

```
Definition add' (n m : nat)
  : nat.
destruct n as [|n']
- apply m.
```



## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 21)
  add n 0 = n.
induction n as [|n' IH].      n', m : nat
- reflexivity.                =====
- simpl. rewrite IH.          nat
  reflexivity.
Qed.
```

```
Definition add' (n m : nat)
  : nat.
destruct n as [|n']
- apply m.
-
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      2 subgoals (ID 18)
  add n 0 = n.
induction n as [|n' IH].      m : nat
- reflexivity.                =====
- simpl. rewrite IH.         nat
  reflexivity.
Qed.                            subgoal 2 (ID 22) is:
                                nat

Definition add' (n m : nat)
  : nat.
induction n as [|n' r].
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 18)
  add n 0 = n.
induction n as [|n' IH].      m : nat
- reflexivity.                =====
- simpl. rewrite IH.         nat
  reflexivity.
Qed.
```

```
Definition add' (n m : nat)
  : nat.
induction n as [|n' r].
-
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 22)
  add n 0 = n.
induction n as [|n' IH].      subgoal 1 (ID 22) is:
- reflexivity.                nat
- simpl. rewrite IH.
  reflexivity.
Qed.
```

```
Definition add' (n m : nat)
  : nat.
induction n as [|n' r].
- apply m.
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 22)
  add n 0 = n.
induction n as [|n' IH].      n', m, r : nat
- reflexivity.                =====
- simpl. rewrite IH.         nat
  reflexivity.
Qed.
```

```
Definition add' (n m : nat)
  : nat.
induction n as [|n' r].
- apply m.
-
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      1 subgoal (ID 23)
  add n 0 = n.
induction n as [|n' IH].      n', m, r : nat
- reflexivity.                =====
- simpl. rewrite IH.          nat
  reflexivity.
Qed.
```

```
Definition add' (n m : nat)
  : nat.
induction n as [|n' r].
- apply m.
- apply S.
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      No more subgoals.
  add n 0 = n.
induction n as [|n' IH].
- reflexivity.
- simpl. rewrite IH.
  reflexivity.
Qed.
```

```
Definition add' (n m : nat)
  : nat.
induction n as [|n' r].
- apply m.
- apply S. apply r.
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :  
  add n 0 = n.  
induction n as [|n' IH].  
- reflexivity.  
- simpl. rewrite IH.  
  reflexivity.  
Qed.
```

```
Definition add' (n m : nat)  
  : nat.  
induction n as [|n' r].  
- apply m.  
- apply S. apply r.  
Defined.
```



## induction in Coq

---

```
Lemma add_n_0 (n : nat) :  
  add n 0 = n.  
induction n as [|n' IH].  
- reflexivity.  
- simpl. rewrite IH.  
  reflexivity.  
Qed.
```

```
Definition add' (n m : nat)  
  : nat.  
induction n as [|n' r].  
- apply m.  
- apply S. apply r.  
Defined.  
Print add'.
```

```
add' =  
fun n m : nat =>  
nat_rec (fun _ : nat => nat) m  
  (fun _ r : nat => S r) n  
  : nat -> nat -> nat
```

## induction in Coq

---

```
Lemma add_n_0 (n : nat) :      add'' is defined
  add n 0 = n.
induction n as [|n' IH].
- reflexivity.
- simpl. rewrite IH.
  reflexivity.
Qed.
```

```
Definition add' (n m : nat)
  : nat.
induction n as [|n' r].
- apply m.
- apply S. apply r.
Defined.
Print add'.
```

```
Definition add'' (n m : nat)
  : nat :=
  nat_rec (fun _ => nat) m (fun n' r => S r) n.
```

## elimination tactics

---

`elim`

`destruct`

`intros + patterns`

`induction`

`inversion` ← details next week

## induction principle

---

```
nat_ind
  : forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n
```

## induction principle

---

```
nat_ind
  : forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n
```

### structure of an induction principle:

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

## induction principle

---

```
nat_ind
  : forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n
```

structure of an induction principle:

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

## induction principle

---

```
nat_ind
  : forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n
```

structure of an induction principle:

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

## induction principle

---

```
nat_ind
  : forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n
```

structure of an induction principle:

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type



## induction principle

---

```
nat_ind
  : forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n
```

structure of an induction principle:

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

## induction principle

---

```
nat_ind
  : forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n
```

### structure of an induction principle:

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

the induction tactic applies this

## recursion principle

---

$$\begin{aligned}f(0) &= g \\f(n + 1) &= h(n, f(n))\end{aligned}$$

## recursion principle

---

$$\begin{aligned}f(0) &= g \\f(n + 1) &= h n (fn)\end{aligned}$$

## recursion principle

---

$$f(0) = g$$
$$f(n + 1) = h n (fn)$$

$$f = \text{nat\_rec } A g h$$

## recursion principle

---

$$f(0) = g$$
$$f(n + 1) = h n (fn)$$

$$f = \text{nat\_rec } A g h$$

$$g : A$$

$$h : \text{nat} \rightarrow A \rightarrow A$$

$$f : \text{nat} \rightarrow A$$

## recursion principle

---

```
nat_rec
  : forall A : Set,
    A ->
    (nat -> A -> A) ->
    nat -> A
```

$$f(0) = g$$
$$f(n + 1) = h n (fn)$$

$$f = \text{nat\_rec } A g h$$

$$g : A$$

$$h : \text{nat} \rightarrow A \rightarrow A$$

$$f : \text{nat} \rightarrow A$$

## recursion principle

---

```
nat_rec
  : forall A : Set,
    A ->
    (nat -> A -> A) ->
    nat -> A
```

$$f(0) = g$$
$$f(n + 1) = h n (fn)$$

$$f = \text{nat\_rec } A g h$$

$$g : A(0)$$

$$h : \prod n : \text{nat}, A(n) \rightarrow A(n + 1)$$

$$f : \prod n : \text{nat}, A(n)$$



## recursion principle

---

```
nat_rec
  : forall A : Set,
    A ->
    (nat -> A -> A) ->
    nat -> A
```

$$f(0) = g$$
$$f(n + 1) = h n (f n)$$

$$f = \text{nat\_rec } A g h$$

$$g : A \text{ O}$$

$$h : \prod n : \text{nat}. A n \rightarrow A (\text{S } n)$$

$$f : \prod n : \text{nat}. A n$$

## recursion principle

---

```
nat_rec
  : forall A : nat -> Set,
    A 0 ->
    (forall n : nat, A n -> A (S n)) ->
    forall n : nat, A n
```

$$f(0) = g$$
$$f(n + 1) = h n (f n)$$

$$f = \text{nat\_rec } A g h$$

$$g : A 0$$

$$h : \prod n : \text{nat}. A n \rightarrow A (S n)$$

$$f : \prod n : \text{nat}. A n$$

## induction = recursion

---

`nat_rec`

```
: forall A : nat -> Set,  
  A 0 ->  
  (forall n : nat, A n -> A (S n)) ->  
  forall n : nat, A n
```

`nat_ind`

```
: forall P : nat -> Prop,  
  P 0 ->  
  (forall n : nat, P n -> P (S n)) ->  
  forall n : nat, P n
```

## non-dependent principle from dependent principle

---

```
nat_rec_dep
  : forall A : nat -> Set,
    A 0 ->
    (forall n : nat, A n -> A (S n)) ->
    forall n : nat, A n
```

```
nat_rec_nondep
  : forall A : Set,
    A ->
    (forall n : nat, A -> A) ->
    forall n : nat, A
```

```
nat_rec_nondep
  : forall A : Set,
    A ->
    (nat -> A -> A) ->
    nat -> A
```

## non-dependent principle from dependent principle

---

```
nat_rec_dep
  : forall A : nat -> Set,
    A 0 ->
    (forall n : nat, A n -> A (S n)) ->
    forall n : nat, A n
```

```
nat_rec_nondep
  : forall A : Set,
    A ->
    (forall n : nat, A -> A) ->
    forall n : nat, A
```

```
nat_rec_nondep
  : forall A : Set,
    A ->
    (nat -> A -> A) ->
    nat -> A
```

```
Inductive nat : Set :=
| 0 : nat
| S : nat -> nat.
```

```
Check nat_rec.
```

## non-dependent principle from dependent principle

---

```
nat_rec_dep
  : forall A : nat -> Set,
    A 0 ->
    (forall n : nat, A n -> A (S n)) ->
    forall n : nat, A n
```

```
nat_rec_nondep
  : forall A : Set,
    A ->
    (forall n : nat, A -> A) ->
    forall n : nat, A
```

```
nat_rec_nondep
  : forall A : Set,
    A ->
    (nat -> A -> A) ->
    nat -> A
```

```
Inductive nat : Prop :=
| 0 : nat
| S : nat -> nat.
```

```
Check nat_ind.
```

$$\begin{aligned}f(0) &= g \\f(n + 1) &= h\ n\ (fn)\end{aligned}$$

$$f = \text{nat\_rec } A\ g\ h$$

$$\begin{aligned}f(0) &= g \\f(n + 1) &= h\ n\ (fn)\end{aligned}$$

$$f = \text{nat\_rec } A\ g\ h$$

$$\begin{aligned}\text{nat\_rec } A\ g\ h\ \mathbf{O} &= g \\ \text{nat\_rec } A\ g\ h\ (\mathbf{S}\ n) &= h\ n\ (fn)\end{aligned}$$



$$f(0) = g$$
$$f(n + 1) = h\ n\ (fn)$$

$$f = \text{nat\_rec } A\ g\ h$$

$$\text{nat\_rec } A\ g\ h\ \mathbf{O} \rightarrow_{\iota} g$$
$$\text{nat\_rec } A\ g\ h\ (\mathbf{S}\ n) \rightarrow_{\iota} h\ n\ (fn)$$

$$f(0) = g$$
$$f(n + 1) = h\ n\ (fn)$$

$$f = \text{nat\_rec } A\ g\ h$$

$$\text{nat\_rec } A\ g\ h\ \mathbf{O} \rightarrow_{\beta\delta\iota} g$$
$$\text{nat\_rec } A\ g\ h\ (\mathbf{S}\ n) \rightarrow_{\beta\delta\iota} h\ n\ (fn)$$

## examples of inductive types

### Curry-Howard

---

datatypes

Set

$\mathbb{1}$

$\mathbb{0}$

$A \rightarrow B$

$A \times B$

$A + B$

$\prod x : A. B$

$\sum x : A. B$

functions

pairs

functions

pairs

logic

Prop

$\top$

$\perp$

$A \rightarrow B$

$A \wedge B$

$A \vee B$

$\forall x : A. B$

$\exists x : A. B$

## unit and empty types

---

```
Inductive unit : Set :=  
| tt : unit.
```

```
Inductive True : Prop :=  
| I : True.
```

```
Inductive Empty_set : Set := .
```

```
Inductive False : Prop := .
```

## product and sum types

---

```
Inductive prod (A B : Set) : Set :=  
| pair : A -> B -> prod A B.
```

```
Inductive and (A B : Prop) : Prop :=  
| conj : A -> B -> and A B.
```

```
Inductive sum (A B : Set) : Set :=  
| inl : A -> sum A B  
| inr : B -> sum A B.
```

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

```
Inductive sumbool (A B : Prop) : Set :=  
| left : A -> or A B  
| right : B -> or A B.
```

## Sigma types and the existential quantifier

---

```
Inductive prod (A B : Set) : Set :=  
| pair : A -> B -> prod A B.
```

```
Inductive sigT (A : Set) (B : A -> Set) : Set :=  
| existsT : forall x : A, B x -> sigT A B.
```

## Sigma types and the existential quantifier

---

```
Inductive prod (A B : Set) : Set :=  
| pair : A -> B -> prod A B.
```

```
Inductive sigT (A : Set) (B : A -> Set) : Set :=  
| existsT : forall x : A, B x -> sigT A B.
```

```
Inductive sig (A : Set) (B : A -> Prop) : Set :=  
| exist : forall x : A, B x -> sig A B.
```

```
Inductive ex (A : Set) (B : A -> Prop) : Prop :=  
| ex_intro : forall x : A, B x -> ex A B.
```

## Sigma types and the existential quantifier

---

```
Inductive prod (A B : Set) : Set :=  
| pair : A -> B -> prod A B.
```

```
Inductive sigT (A : Set) (B : A -> Set) : Set :=  
| existsT : forall x : A, B x -> sigT A B.
```

```
Inductive sig (A : Set) (B : A -> Prop) : Set :=  
| exist : forall x : A, B x -> sig A B.
```

```
Inductive ex (A : Set) (B : A -> Prop) : Prop :=  
| ex_intro : forall x : A, B x -> ex A B.
```

notation:

$A \times B$	$A * B$	prod A B
$A + B$	$A + B$	sum A B
$\Sigma_{x:A} B$	$\{x : A \& B\}$	@sigT A (fun x : A => B)
$\{x : A \mid B\}$	$\{x : A \mid B\}$	@sig A (fun x : A => B)
$\exists x : A. B$	$\text{exists } x : A, B$	@ex A (fun x : A => B)



## proof rules

---

logical connectives as inductive types:

the proposition  $\longleftrightarrow$  the type

introduction rules  $\longleftrightarrow$  the constructors

elimination rule  $\longleftrightarrow$  the eliminator  
= the induction principle

## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

```
or_ind_dep  
:
```

## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

```
or_ind_dep  
  : forall (A B : Prop)
```

## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

```
or_ind_dep  
: forall (A B : Prop)  
  (P : or A B -> Prop),
```

## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

```
or_ind_dep  
: forall (A B : Prop)  
  (P : or A B -> Prop),  
  (forall H : A, P (or_introl H)) ->  
  (forall H : B, P (or_intror H)) ->
```

## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

```
or_ind_dep  
: forall (A B : Prop)  
  (P : or A B -> Prop),  
  (forall H : A, P (or_introl H)) ->  
  (forall H : B, P (or_intror H)) ->  
  forall H : or A B, P H
```

## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

```
or_ind_dep  
: forall (A B : Prop)  
  (P : or A B -> Prop),  
  (forall H : A, P (or_introl H)) ->  
  (forall H : B, P (or_intror H)) ->  
  forall H : or A B, P H
```



## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

```
or_ind  
: forall (A B : Prop)  
  (P : Prop),  
  (forall H : A, P) ->  
  (forall H : B, P) ->  
  forall H : or A B, P
```

## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

```
or_ind  
: forall (A B : Prop)  
  (P : Prop),  
  (forall H : A, P) ->  
  (forall H : B, P) ->  
  forall H : or A B, P
```

## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

```
or_ind  
: forall A B  
  P : Prop,  
  (A -> P) ->  
  (B -> P) ->  
  or A B -> P
```

## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

`or_ind`

```
: forall A B  
  P : Prop,  
  (A -> P) ->  
  (B -> P) ->  
  or A B -> P
```

$$\frac{A}{A \vee B} \text{I}\vee \qquad \frac{B}{A \vee B} \text{I}\text{r}\vee$$

$$\frac{A \vee B \quad A \rightarrow P \quad B \rightarrow P}{P} \text{E}\vee$$

## example: disjunction elimination

---

```
Inductive or (A B : Prop) : Prop :=  
| or_introl : A -> or A B  
| or_intror : B -> or A B.
```

for all parameters of the type,  
for all predicates over the type,  
if the predicate is preserved by the constructors,  
then the predicate holds on the full type

`or_ind`

```
: forall A B  
  P : Prop,  
  (A -> P) ->  
  (B -> P) ->  
  or A B -> P
```

$$\frac{A}{A \vee B} \text{I}\vee \qquad \frac{B}{A \vee B} \text{I}\text{r}\vee$$

$$\frac{A \vee B \quad A \rightarrow P \quad B \rightarrow P}{P} \text{E}\vee$$

## propositions versus Booleans

---

two very different types for truth values:

- ▶ **Prop**  
elements are types, does not support if-then-else  
predicates map to Prop
- ▶ **bool**  
elements are data, supports if-then-else  
decision procedures map to bool

## propositions versus Booleans

---

two very different types for truth values:

- ▶ **Prop**  
elements are types, does not support if-then-else  
predicates map to Prop
- ▶ **bool**  
elements are data, supports if-then-else  
decision procedures map to bool

Prop : Type

True : Prop

False : Prop

! : True

bool : Set

true : bool

false : bool

## datatypes: lists and vectors

---

```
Inductive blist : Set :=  
| bnil : blist  
| bcons : bool -> blist -> blist.
```



## datatypes: lists and vectors

---

```
Inductive blist : Set :=  
| bnil : blist  
| bcons : bool -> blist -> blist.
```

```
Inductive list (A : Set) : Set :=  
| nil : list A  
| cons : A -> list A -> list A.
```

```
Inductive vec (A : Set) : nat -> Set :=  
| vnil : vec A 0  
| vcons : forall n : nat, A -> vec A n -> vec A (S n).
```

## datatypes: lists and vectors

---

```
Inductive blist : Set :=
| bnil : blist
| bcons : bool -> blist -> blist.
```

```
Inductive list (A : Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.
```

```
Inductive vec (A : Set) : nat -> Set :=
| vnil : vec A 0
| vcons : forall n : nat, A -> vec A n -> vec A (S n).
```

```
Fixpoint vappend (A : Set) (n m : nat)
  (v : vec A n) (w : vec A m) : vec A (add n m) :=
  match v with
  | vnil _ => w
  | vcons _ n' h t => vcons A (add n' m) h (vappend A n' m t w)
  end.
```

## datatypes: lists and vectors

---

```
Inductive blist : Set :=  
| bnil : blist  
| bcons : bool -> blist -> blist.
```

```
Inductive list (A : Set) : Set :=  
| nil : list A  
| cons : A -> list A -> list A.
```

```
Inductive vec (A : Set) : nat -> Set :=  
| vnil : vec A 0  
| vcons : forall n : nat, A -> vec A n -> vec A (S n).
```

```
Fixpoint vappend (A : Set) (n m : nat)  
  (v : vec A n) (w : vec A m) : vec A (add n m) :=  
  match v in vec _ n return vec A (add n m) with  
  | vnil _ => w  
  | vcons _ n' h t => vcons A (add n' m) h (vappend A n' m t w)  
  end.
```

## datatypes: lists and vectors

---

```
Inductive blist : Set :=
| bnil : blist
| bcons : bool -> blist -> blist.
```

```
Inductive list (A : Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.
```

```
Inductive vec (A : Set) : nat -> Set :=
| vnil : vec A 0
| vcons : forall n : nat, A -> vec A n -> vec A (S n).
```

```
Fixpoint vappend (A : Set) (n m : nat)
  (v : vec A n) (w : vec A m) : vec A (add n m) :=
match v in vec _ n return vec A (add n m) with
| vnil _ => w
| vcons _ n' h t => vcons A (add n' m) h (vappend A n' m t w)
end.
```

## datatypes: lists and vectors

---

```
Inductive blist : Set :=
| bnil : blist
| bcons : bool -> blist -> blist.
```

```
Inductive list (A : Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.
```

```
Inductive vec (A : Set) : nat -> Set :=
| vnil : vec A 0
| vcons : forall n : nat, A -> vec A n -> vec A (S n).
```

Arguments vcons {A} {n}.

```
Fixpoint vappend {A : Set} {n m : nat}
  (v : vec A n) (w : vec A m) : vec A (add n m) :=
  match v in vec _ n return vec A (add n m) with
  | vnil _ => w
  | vcons h t => vcons h (vappend t w)
  end.
```

## extended match

---

```
match ... as y in Ix1 ... xn return A with
| ...
| (ci ...) ⇒ Mi
| ...
end
```

for all  $i$ :

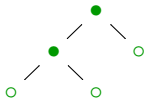
$$(c_i \dots) : IN_1 \dots N_1$$
$$\Downarrow$$
$$M_i : A[x_1 := N_1, \dots, x_n := N_n, y := (c_i \dots)]$$

## trees

---

```
Inductive bintree : Set :=  
| node : bintree -> bintree -> bintree  
| leaf : bintree.
```

```
node (node leaf leaf) leaf
```



## W-types

```
Inductive W (A : Set) (B : A -> Set) : Set :=  
| sup : forall x : A, (B x -> W A B) -> W A B.
```

nodes are labeled with elements of  $A$   
edges are labeled with elements of  $Bx$   
(with  $x$  the label of the node)

## inductive predicates

### rules

---

Coq formalization of any system of rules of the form:

$$\frac{\text{hyp}_1 \quad \dots \quad \text{hyp}_n}{\text{conclusion}}$$

- ▶ logics: proof rules
- ▶ type systems: typing rules
- ▶ programming language semantics
- ▶ ...



## examples

---

```
Inductive even : nat -> Prop :=  
| even_0 : even 0  
| even_SS : forall n : nat, even n -> even (S (S n)).
```

$$\frac{}{\text{even } 0} \qquad \frac{\text{even } n}{\text{even } (n + 2)}$$

```
Inductive le : nat -> nat -> Prop :=  
| le_n : forall n : nat, le n n  
| le_S : forall n m : nat, le n m -> le n (S m).
```

```
Inductive le (n : nat) : nat -> Prop :=  
| le_n : le n n  
| le_S : forall m : nat, le n m -> le n (S m).
```

$$\frac{}{n \leq n} \qquad \frac{n \leq m}{n \leq m + 1}$$

## proving that four is even

---

```
Inductive even
  : nat -> Prop :=
| even_0 : even 0
| even_SS n :
  even n ->
  even (S (S n)).
```

```
even is defined
even_ind is defined
even_sind is defined
```

## proving that four is even

---

```
Inductive even
  : nat -> Prop :=
| even_0 : even 0
| even_SS n :
  even n ->
  even (S (S n)).
```

```
1 subgoal (ID 38)
```

```
=====
even (S (S (S (S 0))))
```

```
Lemma even_4 :
  even (S (S (S (S 0)))).
```

## proving that four is even

---

```
Inductive even
  : nat -> Prop :=
| even_0 : even 0
| even_SS n :
  even n ->
  even (S (S n)).
```

```
Lemma even_4 :
  even (S (S (S (S 0)))).
apply even_SS.
```

```
1 subgoal (ID 39)
```

```
=====
even (S (S 0))
```

## proving that four is even

---

```
Inductive even
  : nat -> Prop :=
| even_0 : even 0
| even_SS n :
  even n ->
  even (S (S n)).
```

```
Lemma even_4 :
  even (S (S (S (S 0)))).
apply even_SS.
apply even_SS.
```

```
1 subgoal (ID 40)
```

```
=====
even 0
```

## proving that four is even

---

Inductive even

```
  : nat -> Prop :=  
| even_0 : even 0  
| even_SS n :  
  even n ->  
  even (S (S n)).
```

No more subgoals.

```
Lemma even_4 :  
  even (S (S (S (S 0)))).  
apply even_SS.  
apply even_SS.  
apply even_0.
```

## proving that four is even

---

Inductive even

```
  : nat -> Prop :=  
| even_0 : even 0  
| even_SS n :  
  even n ->  
  even (S (S n)).
```

Lemma even\_4 :

```
  even (S (S (S (S 0)))).  
apply even_SS.  
apply even_SS.  
apply even_0.
```

Qed.

## proving that four is even

---

```
Inductive even
  : nat -> Prop :=
| even_0 : even 0
| even_SS n :
  even n ->
  even (S (S n)).
```

```
Lemma even_4 :
  even (S (S (S (S 0)))).
apply even_SS.
apply even_SS.
apply even_0.
Qed.
```

**Print even\_4.**

```
even_4 = even_SS (S (S 0))
          (even_SS 0 even_0)
          : even (S (S (S (S 0))))
```

$$\frac{\text{even } 0}{\text{even } (0 + 2)} \\ \frac{\text{even } (0 + 2)}{\text{even } ((0 + 2) + 2)}$$



## proving that three is not even: inversion

---

```
Inductive even
  : nat -> Prop :=
| even_0 : even 0
| even_SS n :
  even n ->
  even (S (S n)).
```

```
even is defined
even_ind is defined
even_sind is defined
```

## proving that three is not even: inversion

---

```
Inductive even
  : nat -> Prop :=
| even_0 : even 0
| even_SS n :
  even n ->
  even (S (S n)).
```

```
1 subgoal (ID 39)
```

```
=====
~ even (S (S (S 0)))
```

```
Lemma odd_3 :
  ~ even (S (S (S 0))).
```

## proving that three is not even: inversion

---

```
Inductive even
  : nat -> Prop :=
| even_0 : even 0
| even_SS n :
  even n ->
  even (S (S n)).
```

```
Lemma odd_3 :
  ~ even (S (S (S 0))).
intro H.
```

```
1 subgoal (ID 41)
```

```
H : even (S (S (S 0)))
```

```
=====
```

```
False
```

## proving that three is not even: inversion

---

```
Inductive even                                2 subgoals (ID 53)
  : nat -> Prop :=
| even_0 : even 0                             =====
| even_SS n :
  even n ->
  even (S (S n)).                             False

Lemma odd_3 :
  ~ even (S (S (S 0))).
intro H.
induction H.
```

## proving that three is not even: inversion

---

```
Inductive even                               1 subgoal (ID 71)
  : nat -> Prop :=
| even_0 : even 0                          H : even (S (S (S 0)))
| even_SS n :                               n : nat
  even n ->                                H1 : even (S 0)
  even (S (S n)).                          H0 : n = S 0
                                           =====
                                           False

Lemma odd_3 :
  ~ even (S (S (S 0))).
intro H.
inversion H.
```

## proving that three is not even: inversion

---

```
Inductive even                                2 subgoals (ID 46)
  : nat -> Prop :=
| even_0 : even 0                            H : even (S (S (S 0)))
| even_SS n :                                H0 : 0 = S (S (S 0))
  even n ->
  even (S (S n)).                          =====
                                           False

Lemma odd_3 :                                subgoal 2 (ID 51) is:
  ~ even (S (S (S 0))).                      even n -> False
intro H.
simple inversion H.
```

## proving that three is not even: inversion

---

```
Inductive even                                1 subgoal (ID 73)
  : nat -> Prop :=
| even_0 : even 0                            H0 : even (S 0)
| even_SS n :
  even n ->
  even (S (S n)).                          =====
                                           False

Lemma odd_3 :
  ~ even (S (S (S 0))).
intro H.
inversion_clear H.
```

## proving that three is not even: inversion

---

```
Inductive even                               1 subgoal (ID 72)
  : nat -> Prop :=
| even_0 : even 0                          H : even (S (S (S 0)))
| even_SS n :                               H1 : even (S 0)
  even n ->
  even (S (S n)).                          =====
                                           False

Lemma odd_3 :
  ~ even (S (S (S 0))).
intro H.
inversion H; subst.
```



## proving that three is not even: inversion

---

```
Inductive even                                1 subgoal (ID 73)
  : nat -> Prop :=
| even_0 : even 0                            H1 : even (S 0)
| even_SS n :
  even n ->
  even (S (S n)).                          =====
                                           False

Lemma odd_3 :
  ~ even (S (S (S 0))).
intro H.
inversion H; clear H; subst.
```

## proving that three is not even: inversion

---

```
Inductive even                                my_inversion is defined
  : nat -> Prop :=
| even_0 : even 0
| even_SS n :
  even n ->
  even (S (S n)).
```

```
Ltac my_inversion H :=
  inversion H; clear H; subst.
```

```
Lemma odd_3 :
  ~ even (S (S (S 0))).
intro H.
my_inversion H.
```

## proving that three is not even: inversion

---

```
Inductive even                                1 subgoal (ID 73)
  : nat -> Prop :=
| even_0 : even 0                               H1 : even (S 0)
| even_SS n :
  even n ->
  even (S (S n)).
=====
False

Ltac my_inversion H :=
  inversion H; clear H; subst.

Lemma odd_3 :
  ~ even (S (S (S 0))).
intro H.
my_inversion H.
```

## proving that three is not even: inversion

---

```
Inductive even                                No more subgoals.
  : nat -> Prop :=
| even_0 : even 0
| even_SS n :
  even n ->
  even (S (S n)).

Ltac my_inversion H :=
  inversion H; clear H; subst.

Lemma odd_3 :
  ~ even (S (S (S 0))).
intro H.
my_inversion H.
my_inversion H1.
```

## proving that three is not even: inversion

---

Inductive even

```
  : nat -> Prop :=  
| even_0 : even 0  
| even_SS n :  
  even n ->  
  even (S (S n)).
```

```
Ltac my_inversion H :=  
  inversion H; clear H; subst.
```

```
Lemma odd_3 :  
  ~ even (S (S (S 0))).  
intro H.  
my_inversion H.  
my_inversion H1.  
Qed.
```

## exercise: figure out the induction principle of even

---

### dependent induction principle of `nat`

```
nat_ind
  : forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n
```

### non-dependent induction principle of `even`

```
even_ind
  : forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S (S n))) ->
    forall n : nat, even n -> P n
```

## exercise: figure out the induction principle of even

---

### dependent induction principle of `nat`

```
nat_ind
  : forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n
```

### non-dependent induction principle of `even`

```
even_ind
  : forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, even n -> P n -> P (S (S n))) ->
    forall n : nat, even n -> P n
```

## equality

---

```
Inductive le (n : nat) : nat -> Prop :=  
| le_n : le n n  
| le_S : forall m : nat, le n m -> le n (S m).
```

```
Inductive eq_nat (n : nat) : nat -> Prop :=  
| eq_n : eq_nat n n.
```



## equality

---

```
Inductive le (n : nat) : nat -> Prop :=  
| le_n : le n n  
| le_S : forall m : nat, le n m -> le n (S m).
```

```
Inductive eq_nat (n : nat) : nat -> Prop :=  
| eq_n : eq_nat n n.
```

## equality

---

```
Inductive le (n : nat) : nat -> Prop :=  
| le_n : le n n  
| le_S : forall m : nat, le n m -> le n (S m).
```

```
Inductive eq_nat (n : nat) : nat -> Prop :=  
| eq_n : eq_nat n n.
```

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  
| eq_refl : eq A x x.
```

## equality

---

```
Inductive le (n : nat) : nat -> Prop :=  
| le_n : le n n  
| le_S : forall m : nat, le n m -> le n (S m).
```

```
Inductive eq_nat (n : nat) : nat -> Prop :=  
| eq_n : eq_nat n n.
```

```
Inductive eq (A : Type) : A -> A -> Prop :=  
| eq_refl : forall x : A, eq A x x.
```

## equality

---

```
Inductive le (n : nat) : nat -> Prop :=  
| le_n : le n n  
| le_S : forall m : nat, le n m -> le n (S m).
```

```
Inductive eq_nat (n : nat) : nat -> Prop :=  
| eq_n : eq_nat n n.
```

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  
| eq_refl : eq A x x.
```

## equality

---

```
Inductive le (n : nat) : nat -> Prop :=  
| le_n : le n n  
| le_S : forall m : nat, le n m -> le n (S m).
```

```
Inductive eq_nat (n : nat) : nat -> Prop :=  
| eq_n : eq_nat n n.
```

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  
| eq_refl : eq A x x.
```

### eq\_ind\_dep

```
: forall (A : Type) (x : A)  
  (P : forall y : A, eq A x y -> Prop),  
  P x (eq_refl A x) ->  
  forall (y : A) (H : eq A x y), P y H
```

## equality

---

```
Inductive le (n : nat) : nat -> Prop :=
| le_n : le n n
| le_S : forall m : nat, le n m -> le n (S m).
```

```
Inductive eq_nat (n : nat) : nat -> Prop :=
| eq_n : eq_nat n n.
```

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq A x x.
```

### eq\_ind\_dep

```
: forall (A : Type) (x : A)
  (P : forall y : A, eq A x y -> Prop),
  P x (eq_refl A x) ->
  forall (y : A) (H : eq A x y), P y H
```

## equality

---

```
Inductive le (n : nat) : nat -> Prop :=
| le_n : le n n
| le_S : forall m : nat, le n m -> le n (S m).
```

```
Inductive eq_nat (n : nat) : nat -> Prop :=
| eq_n : eq_nat n n.
```

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq A x x.
```

### eq\_ind

```
: forall (A : Type) (x : A)
  (P : forall y : A, Prop),
  P x ->
  forall (y : A) (H : eq A x y), P y
```

## equality

---

```
Inductive le (n : nat) : nat -> Prop :=  
| le_n : le n n  
| le_S : forall m : nat, le n m -> le n (S m).
```

```
Inductive eq_nat (n : nat) : nat -> Prop :=  
| eq_n : eq_nat n n.
```

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  
| eq_refl : eq A x x.
```

### eq\_ind

```
: forall (A : Type) (x : A)  
  (P : A -> Prop)  
  P x ->  
  forall (y : A), eq A x y -> P y
```



## equality

---

```
Inductive le (n : nat) : nat -> Prop :=  
| le_n : le n n  
| le_S : forall m : nat, le n m -> le n (S m).
```

```
Inductive eq_nat (n : nat) : nat -> Prop :=  
| eq_n : eq_nat n n.
```

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  
| eq_refl : eq A x x.
```

### eq\_ind

```
: forall (A : Type) (x : A)  
  (P : A -> Prop)  
  P x ->  
  forall (y : A), eq A x y -> P y
```

### Leibniz equality

$$\frac{P(x) \quad x = y}{P(y)}$$

## overview

---

- ▶ CIC (it's complicated)
  - ▶ universes: Prop, Set, Type
  - ▶ reduction:  $\rightarrow_{\beta\delta\iota\zeta\eta}$
- ▶ inductive types
  - ▶ constructors
  - ▶ induction/recursion principles
- ▶ Coq
  - ▶ Inductive
  - ▶ Fixpoint and match
  - ▶ induction
  - ▶ inversion (more next week)
- ▶ examples
  - ▶ logical operators
  - ▶ datatypes
  - ▶ inductive predicates
  - ▶ Leibniz equality

## table of contents

contents

---

introduction

CIC

the natural numbers

examples of inductive types

inductive predicates

conclusion

table of contents