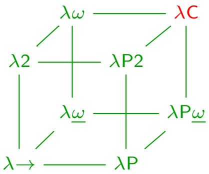
inductive types

Freek Wiedijk

Type Theory & Coq
2024–2025
Radboud University Nijmegen

October 4, 2024


$\lambda\omega$ λC
$\lambda 2$ $\lambda P2$
$\lambda\underline{\omega}$ $\lambda P\underline{\omega}$
$\lambda{\to}$ λP

introduction

today

minimal propositional logic	STT = simple type theory
minimal predicate logic	λP = dependent types
full Coq logic	CIC = Calculus of Inductive Constructions

CIC = λC + inductive types + coinductive types + universes + . . .

how are types introduced?

- free type variables

  STT = simple type theory

- in the context

  PTSs = pure type systems $\lambda\!\to$ λP $\lambda 2$ λC

  $$\mathsf{nat} : *,\ \mathsf{O} : \mathsf{nat},\ \mathsf{S} : \mathsf{nat} \to \mathsf{nat} \vdash \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{O})) : \mathsf{nat}$$

- definitions

  CIC = Calculus of Inductive Constructions

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

definitions in Coq

- axioms

  environment used like the context in λC
  disadvantage: reductions will get stuck

  `Axiom Parameter`

- definitions of constants

  `Definition`

  `Lemma`
  `Qed`

- inductive definitions

  `Inductive`

CIC

variants

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

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

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

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

context versus environment

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

- E is the environment of axioms and definitions
- Γ is the context of local variables

example of context versus environment

```
Parameter a : Prop.
Definition I : a -> a :=
  fun x : a => 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 : Prop.
Definition I : a -> a :=
  fun x : a => 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:

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

syntax

STT

$$\begin{aligned} A, B &::= a \mid A \to B \\ M, N &::= x \mid M\,N \mid \lambda x : A.\, M \end{aligned}$$

λC

$$\begin{aligned} M, N, A, B &::= x \mid M\,N \mid \lambda x : A.\, M \mid \Pi x : A.\, B \mid s \\ s &::= * \mid \Box \end{aligned}$$

CIC

$$\begin{aligned} M, N, A, B ::= {} & x \mid M\,N \mid \lambda x : A.\, M \mid \Pi x : A.\, B \mid s \mid \\ & \mathsf{let}\ x := N : A\ \mathsf{in}\ M \mid \\ & \mathsf{fix}\ \ldots \mid \mathsf{match}\ \ldots \mid \ldots \\ s ::= {} & \mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{SProp} \mid \mathsf{Type}(i) \end{aligned}$$

the universe levels i are explicit natural numbers

universes

λC

$$* : \Box$$

CIC

$$\{\mathsf{Set}, \mathsf{Prop}, \mathsf{SProp}\} : \mathsf{Type}(1) : \mathsf{Type}(2) : \mathsf{Type}(3) : \ldots$$

in λC the sort $\Box$ does not have a type
in CIC *every* term has a type

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

SProp is a proof irrelevant version of Prop

subtyping

$$\mathsf{Prop} \leq \mathsf{Set} \leq \mathsf{Type}(1) \leq \mathsf{Type}(2) \leq \mathsf{Type}(3) \leq \ \ldots$$

subtyping

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

```
Check True.
```

```
True
     : Prop
```

subtyping

$$\mathsf{Prop} \leq \mathsf{Set} \leq \mathsf{Type}(1) \leq \mathsf{Type}(2) \leq \mathsf{Type}(3) \leq \ \ldots$$

```
Check True.                   True : Set
Check (True : Set).                : Set
```

subtyping

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

```
Check True.                  True : Type
Check (True : Set).               : Type
Check (True : Type).
```

subtyping

$$\mathsf{Prop} \leq \mathsf{Set} \leq \mathsf{Type}(1) \leq \mathsf{Type}(2) \leq \mathsf{Type}(3) \leq \ldots$$

```
Check True.                   nat
Check (True : Set).                : Set
Check (True : Type).
Check nat.
```

subtyping

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

```
Check True.                  nat : Type
Check (True : Set).              : Type
Check (True : Type).
Check nat.
Check (nat : Type).
```

subtyping

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

```
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

$$\mathsf{Prop} \leq \mathsf{Set} \leq \mathsf{Type}(1) \leq \mathsf{Type}(2) \leq \mathsf{Type}(3) \leq \ \ldots$$

```
Check True.                   Type : Type
Check (True : Set).                : Type
Check (True : Type).
Check nat.
Check (nat : Type).
Check (nat : Prop).
Check (Type : Type).
```

subtyping

$$\mathsf{Prop} \leq \mathsf{Set} \leq \mathsf{Type}(1) \leq \mathsf{Type}(2) \leq \mathsf{Type}(3) \leq \ \ldots$$

```
Check True.
Check (True : Set).
Check (True : Type).
Check nat.
Check (nat : Type).
Check (nat : Prop).
Check (Type : Type).
```

conversion rule:

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

subtyping

$$\mathsf{Prop} \leq \mathsf{Set} \leq \mathsf{Type}(1) \leq \mathsf{Type}(2) \leq \mathsf{Type}(3) \leq \ \ldots$$

```
Check True.
Check (True : Set).
Check (True : Type).
Check nat.
Check (nat : Type).
Check (nat : Prop).
Check (Type : Type).
```

conversion rule:

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

subtyping

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

```
Check True.
Check (True : Set).
Check (True : Type).
Check nat.
Check (nat : Type).
Check (nat : Prop).
Check (Type : Type).
```

conversion rule:

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

subtyping

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

```
Check True.
Check (True : Set).
Check (True : Type).
Check nat.
Check (nat : Type).
Check (nat : Prop).
Check (Type : Type).
```

conversion rule:

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

subtyping

$$\mathsf{Prop} \leq \mathsf{Set} \leq \mathsf{Type}(1) \leq \mathsf{Type}(2) \leq \mathsf{Type}(3) \leq \ \ldots$$

```
Check True.
Check (True : Set).
Check (True : Type).
Check nat.
Check (nat : Type).
Check (nat : Prop).
Check (Type : Type).
```

conversion rule:

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

reduction

`fun`	$\beta \ \eta$
`Definition`	δ
`fix match`	ι
`let`	ζ

reduction

`fun`	$\beta \;\; \eta$
`Definition`	δ
`fix match`	ι
`let`	ζ

$$\begin{aligned}
(\lambda x : A.\, M)N &\rightarrow_{\beta} M[x := N] \\
\lambda x : A.\, (Fx) &\rightarrow_{\eta} F \qquad\qquad\qquad \text{when } F : (\Pi x : A.\, B) \\
\text{let } x := N : A \text{ in } M &\rightarrow_{\zeta} M[x := N]
\end{aligned}$$

why let-in definitions when we have beta redexes?

$\text{let } A := \mathsf{nat} : \mathsf{Set} \text{ in } (\lambda x : A.\, x)\, \mathsf{O}$

is well-typed

$(\lambda A : \mathsf{Set}.\, ((\lambda x : A.\, x)\, \mathsf{O}))\, \mathsf{nat}$

is not well-typed

because the subterm

$\lambda A : \mathsf{Set}.\, ((\lambda x : A.\, x)\, \mathsf{O})$

is not well-typed

defining constants in Coq

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

defining constants in Coq

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

```
two = S (S O)
     : nat
```

defining constants in Coq

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

Lemma two’ : nat.
```

```
1 subgoal (ID 1)

  ============================
  nat
```

defining constants in Coq

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

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

```
1 subgoal (ID 2)

  ============================
  nat
```

defining constants in Coq

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

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

```
1 subgoal (ID 3)

  ============================
  nat
```

defining constants in Coq

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

Lemma two' : nat.
apply S.
apply S.
apply O.
```

```
No more subgoals.
```

defining constants in Coq

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

Lemma two’ : nat.
apply S.
apply S.
apply O.
Qed.
```

defining constants in Coq

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

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

```
two' = S (S O)
     : nat
```

defining constants in Coq

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

Lemma two’ : nat.
apply S.
apply S.
apply O.
Qed.
Print two’.

Lemma eq_two : two = two’.
```

```
1 subgoal (ID 3)

  ============================
  two = two’
```

defining constants in Coq

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

Lemma two' : nat.
apply S.
apply S.
apply O.
Qed.
Print two'.

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

```
Error: two' is opaque.
```

defining constants in Coq

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

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

defining constants in Coq

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

Lemma two’ : nat.
apply S.
apply S.
apply O.
Defined.
Print two’.
```

```
two’ = S (S O)
     : nat
```

defining constants in Coq

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

Lemma two' : nat.
apply S.
apply S.
apply O.
Defined.
Print two'.

Lemma eq_two : two = two'.
```

```
1 subgoal (ID 3)

  ============================
  two = two'
```

defining constants in Coq

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

Lemma two' : nat.
apply S.
apply S.
apply O.
Defined.
Print two'.

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

```
1 subgoal (ID 5)

  ============================
  S (S O) = S (S O)
```

defining constants in Coq

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

Lemma two' : nat.
apply S.
apply S.
apply O.
Defined.
Print two'.

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

```
No more subgoals.
```

defining constants in Coq

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

Lemma two' : nat.
apply S.
apply S.
apply O.
Defined.
Print two'.

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

defining constants in Coq

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

Lemma two' : nat.
apply S.
apply S.
apply O.
Defined.
Print two'.

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

```
No more subgoals.
```

defining constants in Coq

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

Lemma two’ : nat.
apply S.
apply S.
apply O.
Defined.
Print two’.

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

defining constants in Coq

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

Definition two' : nat.
apply S.
apply S.
apply O.
Defined.
Print two'.

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

defining constants in Coq

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

Definition two’ : nat.
apply S.
apply S.
apply O.
Defined.
Print two’.

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

delta reduction:

$$\begin{array}{lcl} \mathsf{two} & \rightarrow_\delta & \mathsf{S}\ (\mathsf{S}\ \mathsf{O}) \\ \mathsf{two}' & \rightarrow_\delta & \mathsf{S}\ (\mathsf{S}\ \mathsf{O}) \end{array}$$

the natural numbers

defining an inductive type

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

$$\mathsf{nat} = \{\mathsf{O}, \mathsf{S\ O}, \mathsf{S\ (S\ O)}, \mathsf{S\ (S\ (S\ O))}, \ldots\}$$

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)

intuitionism

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

classical mathematics

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

discontinuous functions

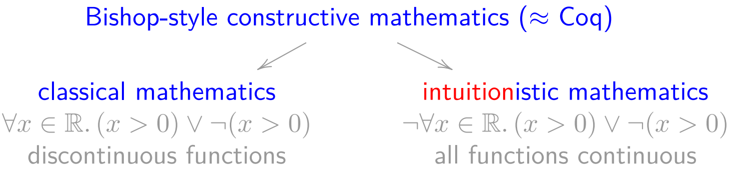
intuitionistic mathematics

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

all functions continuous

L.E.J. Brouwer

intuitionism



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

classical mathematics

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

discontinuous functions

intuitionistic mathematics

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

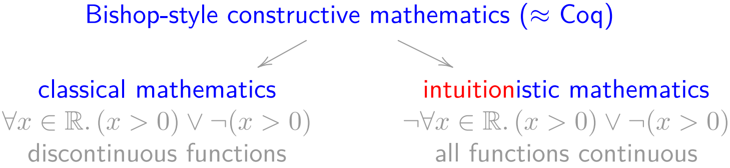
all functions continuous

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

Deze intuïtie der twee-eenigheid, deze oerintuïtie der wiskunde schept 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

intuitionism



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 :=
| O : 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
| O : nat                            : Set
| S : nat -> nat.

Check nat.
```

natural numbers in Coq

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

Check nat.
Check O.
```

```
O
     : nat
```

natural numbers in Coq

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

Check nat.
Check O.
Check S.
```

natural numbers in Coq

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

Check nat.
Check O.
Check S.
Check nat_ind.
```

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

natural numbers in Coq

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

Check nat.
Check O.
Check S.
Check nat_ind.
Check nat_sind.
```

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

natural numbers in Coq

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

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

```
nat_rec
     : forall P : nat -> Set,
       P O ->
       (forall n : nat,
        P n -> P (S n)) ->
       forall n : nat, P n
```

natural numbers in Coq

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

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

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

natural numbers in Coq

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

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

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

Print nat.
```

natural numbers in Coq

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

Check nat.
Check O.
Check S.
Check nat_ind.
Check nat_sind.
Check nat_rec.
Check nat_rect.

Print nat.
Print O.
```

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

natural numbers in Coq

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

Check nat.
Check O.
Check S.
Check nat_ind.
Check nat_sind.
Check nat_rec.
Check nat_rect.

Print nat.
Print O.
Print S.
```

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

natural numbers in Coq

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

Check nat.
Check O.
Check S.
Check nat_ind.
Check nat_sind.
Check nat_rec.
Check nat_rect.

Print nat.
Print O.
Print S.
Print nat_ind.
```

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

Check nat.
Check O.
Check S.
Check nat_ind.
Check nat_sind.
Check nat_rec.
Check nat_rect.

Print nat.
Print O.
Print S.
Print nat_ind.
Print nat_rect.
```

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

Check nat.
Check O.
Check S.
Check nat_ind.
Check nat_sind.
Check nat_rec.
Check nat_rect.

Print nat.
Print O.
Print S.
Print nat_ind.
Print nat_rect.
```

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

makes three kinds of constants available:

- the type
  primitive
  $$\text{nat} : \text{Set}$$
- the constructors
  primitive
  $$\begin{aligned} \text{O} &: \text{nat} \\ \text{S} &: \text{nat} \to \text{nat} \end{aligned}$$
- the destructors
  = eliminators = induction principles
  = recursors = recursion principles

  defined using ‘fix’ and ‘match’

induction / recursion principles

$$\begin{aligned} \text{nat\_ind} &: \ldots \\ \text{nat\_sind} &: \ldots \\ \text{nat\_rec} &: \ldots \\ \text{nat\_rect} &: \ldots \end{aligned}$$

correspond to predicates in $\{\text{Prop}, \text{SProp}, \text{Set}, \text{Type}\}$

induction / recursion principles

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

correspond to predicates in $\{\mathsf{Prop}, \mathsf{SProp}, \mathsf{Set}, \mathsf{Type}\}$

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 ⟶ 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
  | O => m
  | S n’ => S (add n’ m)
  end.
```

defining addition

```
Fixpoint add (n m : nat) {struct n} : nat :=
  match n with
  | O => 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
  | O => 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
  | O => 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)      add is defined
    : nat :=                  add is recursively defined (guarded on
  match n with
  | O => m
  | S n' => S (add n' m)
  end.
```

recursive definitions in Coq

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

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

recursive definitions in Coq

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

```
add is defined
```

recursive definitions in Coq

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

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

recursive definitions in Coq

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

recursive definitions in Coq

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

Lemma add_1_1 :
  add (S O) (S O) = S (S O).
```

```
1 subgoal (ID 5)

  ============================
  add (S O) (S O) = S (S O)
```

recursive definitions in Coq

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

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

```
1 subgoal (ID 7)

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

recursive definitions in Coq

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

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

```
1 subgoal (ID 9)

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

recursive definitions in Coq

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

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

```
1 subgoal (ID 11)

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

recursive definitions in Coq

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

Lemma add_1_1 :
  add (S O) (S O) = S (S O).
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
         | O => m
         | S n’0 => S (add n’0 m)
         end) n’ (S O))) O =
  S (S O)
```

recursive definitions in Coq

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

Lemma add_1_1 :
  add (S O) (S O) = S (S O).
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
        | O => m
        | S n' => S (add n' m)
        end) O (S O)) =
  S (S O)
```

recursive definitions in Coq

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

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

```
1 subgoal (ID 17)

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

recursive definitions in Coq

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

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

```
1 subgoal (ID 19)

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

recursive definitions in Coq

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

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

```
1 subgoal (ID 21)

  ============================
  S (S O) = S (S O)
```

recursive definitions in Coq

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

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

```
1 subgoal (ID 7)

  ============================
  S (S O) = S (S O)
```

recursive definitions in Coq

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

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

```
1 subgoal (ID 7)

  ============================
  S (S O) = S (S O)
```

recursive definitions in Coq

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

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

```
1 subgoal (ID 7)

  ============================
  S (S O) = S (S O)
```

recursive definitions in Coq

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

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

```
1 subgoal (ID 9)

  ============================
  S (S O) = S (S O)
```

recursive definitions in Coq

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

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

```
No more subgoals.
```

recursive definitions in Coq

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

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

```
1 subgoal (ID 7)

  ============================
  S (S O) = S (S O)
```

recursive definitions in Coq

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

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

```
No more subgoals.
```

recursive definitions in Coq

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

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

```
No more subgoals.
```

recursive definitions in Coq

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

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

```
No more subgoals.
```

recursive definitions in Coq

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

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

recursive definitions in Coq

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

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

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

```
= S (S O)
: nat
```

iota reduction

`fun`	$\beta \;\; \eta$
`Definition`	δ
`fix match`	ι
`let`	ζ

$$(\mathsf{fix}\, f \ldots := M) \ldots (\overset{\substack{\text{constructor}\\ \downarrow}}{c} \ldots) \ldots$$

$$\rotatebox{-90}{$\rightsquigarrow$}$$

$$M[f := (\mathsf{fix}\, f \ldots := M)] \; \ldots (c \ldots) \ldots$$

$$\mathsf{match}\ (c\, N_1 \ldots N_k)\ \mathsf{with}\ \ldots \mid (c\, x_1 \ldots x_k) \Rightarrow M \mid \ldots\ \mathsf{end}$$

$$\rotatebox{-90}{$\rightsquigarrow$}$$

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

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) :
  add 0 n = n.
reflexivity.
```

```
No more subgoals.
```

induction in Coq

```
Lemma add_O_n (n : nat) :
  add O n = n.
reflexivity.
Qed.
```

induction in Coq

```
Lemma add_n_O (n : nat) :
  add n O = n.
```

```
1 subgoal (ID 11)

  n : nat
  ============================
  add n O = n
```

induction in Coq

```
Lemma add_n_O (n : nat) :
  add n O = n.
reflexivity.
```

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

induction in Coq

```
Lemma add_n_O (n : nat) :
  add n O = n.
induction n as [|n’ IH].
```

```
2 subgoals (ID 15)

  ============================
  add O O = O

subgoal 2 (ID 18) is:
 add (S n’) O = S n’
```

induction in Coq

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

```
1 subgoal (ID 15)

  ============================
  add 0 0 = 0
```

induction in Coq

```
Lemma add_n_O (n : nat) :
  add n O = n.
induction n as [|n' IH].
- reflexivity.
```

```
1 subgoal (ID 18)

subgoal 1 (ID 18) is:
 add (S n') O = S n'
```

induction in Coq

```
Lemma add_n_O (n : nat) :
  add n O = n.
induction n as [|n' IH].
- reflexivity.
-
```

```
1 subgoal (ID 18)

  n' : nat
  IH : add n' O = n'
  ============================
  add (S n') O = S n'
```

induction in Coq

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

```
1 subgoal (ID 22)

  n' : nat
  IH : add n' O = n'
  ============================
  S (add n' O) = S n'
```

induction in Coq

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

```
1 subgoal (ID 23)

  n' : nat
  IH : add n' O = n'
  ============================
  S n' = S n'
```

induction in Coq

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

induction in Coq

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

induction in Coq

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

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

```
1 subgoal (ID 14)

  n, m : nat
  ============================
  nat
```

induction in Coq

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

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

```
2 subgoals (ID 18)

  m : nat
  ============================
  nat

subgoal 2 (ID 21) is:
 nat
```

induction in Coq

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

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

```
1 subgoal (ID 18)

  m : nat
  ============================
  nat
```

induction in Coq

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

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

```
1 subgoal (ID 21)

subgoal 1 (ID 21) is:
 nat
```

induction in Coq

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

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

```
1 subgoal (ID 21)

  n', m : nat
  ============================
  nat
```

induction in Coq

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

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

```
2 subgoals (ID 18)

  m : nat
  ============================
  nat

subgoal 2 (ID 22) is:
 nat
```

induction in Coq

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

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

```
1 subgoal (ID 18)

  m : nat
  ============================
  nat
```

induction in Coq

```
Lemma add_n_O (n : nat) :
  add n O = 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.
```

```
1 subgoal (ID 22)

subgoal 1 (ID 22) is:
 nat
```

induction in Coq

```
Lemma add_n_O (n : nat) :
  add n O = 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.
-
```

```
1 subgoal (ID 22)

  n', m, r : nat
  ============================
  nat
```

induction in Coq

```
Lemma add_n_O (n : nat) :
  add n O = 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.
```

```
1 subgoal (ID 23)

  n', m, r : nat
  ============================
  nat
```

induction in Coq

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

```
No more subgoals.
```

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

induction in Coq

```
Lemma add_n_O (n : nat) :
  add n O = 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_O (n : nat) :
  add n O = 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_O (n : nat) :     add'' is defined
  add n O = 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` + pattern

`induction`

`inversion` ⟵ details next week

induction principle

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

induction principle

```
nat_ind
     : forall P : nat -> Prop,
       P O ->
       (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 O ->
       (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 O ->
       (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 O ->
       (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 O ->
       (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 O ->
       (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

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

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

recursion principle

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

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

$$\begin{aligned} &g : A \\ &h : \mathsf{nat} \to A \to A \\ &f : \mathsf{nat} \to A \end{aligned}$$

recursion principle

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

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

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

$$g : A$$
$$h : \mathsf{nat} \to A \to A$$
$$f : \mathsf{nat} \to A$$

recursion principle

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

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

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

$$\begin{aligned} &g : A(0) \\ &h : \Pi n : \mathsf{nat}.\, A(n) \to A(n+1) \\ &f : \Pi n : \mathsf{nat}.\, A(n) \end{aligned}$$

recursion principle

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

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

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

$$\begin{aligned} &g : A\ \mathsf{O} \\ &h : \Pi n : \mathsf{nat}.\ A\ n \to A\,(\mathsf{S}\ n) \\ &f : \Pi n : \mathsf{nat}.\ A\ n \end{aligned}$$

recursion principle

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

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

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

$$g : A\ \mathsf{O}$$
$$h : \Pi n : \mathsf{nat}.\ A\ n \to A\,(\mathsf{S}\ n)$$
$$f : \Pi n : \mathsf{nat}.\ A\ n$$

induction = recursion

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

nat_ind
     : forall P : nat -> Prop,
       P O ->
       (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 O ->
       (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 O ->
       (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 :=
| O : nat
| S : nat -> nat.

Check nat_rec.
```

non-dependent principle from dependent principle

```
nat_rec_dep
     : forall A : nat -> Set,
       A O ->
       (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 :=
| O : nat
| S : nat -> nat.

Check nat_ind.
```

iota reduction revisited

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

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

iota reduction revisited

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

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

$$\begin{aligned} \texttt{nat\_rec}\ A\ g\ h\ \mathsf{O} &= g \\ \texttt{nat\_rec}\ A\ g\ h\ (\mathsf{S}\ n) &= h\, n\, (\texttt{nat\_rec}\ A\ g\ h\ n) \end{aligned}$$

iota reduction revisited

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

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

$$\begin{aligned} \texttt{nat\_rec}\ A\ g\ h\ \mathsf{O} &\rightarrow_\iota g \\ \texttt{nat\_rec}\ A\ g\ h\ (\mathsf{S}\ n) &\rightarrow_\iota h\,n\,(\texttt{nat\_rec}\ A\ g\ h\ n) \end{aligned}$$

iota reduction revisited

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

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

$$\begin{aligned} \texttt{nat\_rec}\ A\ g\ h\ \mathsf{O} &\twoheadrightarrow_{\beta\delta\iota} g \\ \texttt{nat\_rec}\ A\ g\ h\ (\mathsf{S}\ n) &\twoheadrightarrow_{\beta\delta\iota} h\, n\, (\texttt{nat\_rec}\ A\ g\ h\ n) \end{aligned}$$

examples of inductive types

Curry-Howard

datatypes		logic
Set		Prop
$\mathbb{1}$		$\top$
$\mathbb{0}$		$\bot$
$A \to B$	functions	$A \to B$
$A \times B$	pairs	$A \wedge B$
$A + B$		$A \vee B$
$\Pi x : A.\, B$	functions	$\forall x : A.\, B$
$\Sigma x : A.\, B$	pairs	$\exists x : A.\, B$

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 -> sumbool A B
| right : B -> sumbool 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 | B}`	`@sig  A (fun x : A => B)`
$\exists x : A.\, B$	`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
```

$$\frac{A}{A \lor B}\, Il\lor \qquad \frac{B}{A \lor B}\, Ir\lor$$

$$\frac{A \lor B \quad A \to P \quad B \to P}{P}\, E\lor$$

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
```

$$\frac{A}{A \vee B}\,Il\vee \qquad \frac{B}{A \vee B}\,Ir\vee$$

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

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

$$\begin{aligned} \mathsf{Prop} &: \mathsf{Type} \\ \mathsf{True} &: \mathsf{Prop} \\ \mathsf{False} &: \mathsf{Prop} \\ \mathsf{I} &: \mathsf{True} \end{aligned}$$

$$\begin{aligned} \mathsf{bool} &: \mathsf{Set} \\ \mathsf{true} &: \mathsf{bool} \\ \mathsf{false} &: \mathsf{bool} \end{aligned}$$

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 O
| 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 O
| 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 O
| 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 O
| 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

$$\begin{array}{l}\textsf{match } \ldots \textsf{ in } I x_1 \ldots x_n \textsf{ as } y \textsf{ return } A \textsf{ with} \\ \mid \ldots \\ \mid (c_i \ldots) \Rightarrow M_i \\ \mid \ldots \\ \textsf{end}\end{array}$$

for all i:

$$(c_i \ldots) : I N_1 \ldots N_1$$

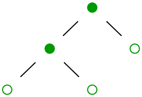
$$\Downarrow$$

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

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_O : even O
| 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)).

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

```
1 subgoal (ID 38)

  ============================
  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)).

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

```
No more subgoals.
```

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))))
```

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

proving that three is not even: inversion

```
Inductive even
    : nat -> Prop :=
| even_O : even O
| 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)).

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

```
1 subgoal (ID 39)

  ============================
  ~ 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
    : 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.
induction H.
```

```
2 subgoals (ID 53)

  ============================
  False

subgoal 2 (ID 57) is:
 False
```

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.
inversion H.
```

```
1 subgoal (ID 71)

  H : even (S (S (S 0)))
  n : nat
  H1 : even (S 0)
  H0 : n = S 0
  ============================
  False
```

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.
simple inversion H.
```

```
2 subgoals (ID 46)

  H : even (S (S (S 0)))
  H0 : 0 = S (S (S 0))
  ============================
  False

subgoal 2 (ID 51) is:
 even n -> False
```

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.
inversion_clear H.
```

```
1 subgoal (ID 73)

  H0 : even (S 0)
  ============================
  False
```

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.
inversion H; subst.
```

```
1 subgoal (ID 72)

  H : even (S (S (S 0)))
  H1 : even (S 0)
  ============================
  False
```

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.
inversion H; clear H; subst.
```

```
1 subgoal (ID 73)

  H1 : even (S 0)
  ============================
  False
```

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 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)).

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

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

```
1 subgoal (ID 73)

  H1 : even (S 0)
  ============================
  False
```

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.
```

```
No more subgoals.
```

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 O ->
       (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 O ->
       (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

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

conclusion

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