overview

inductive type

- a new type
- constructor functions
- pattern matching and \( \iota \)-reduction
- induction principle

inductive types in each universe

- inductive datatypes:
  \[ \text{Inductive} \ldots : \text{Set} := \]
- inductive predicates:
  \[ \text{Inductive} \ldots : \text{Prop} := \]
- Inductive \ldots : Type :=
universes of Coq: example

term : type : kind

S 0 : nat : Set : Type

fun x:A => x : A -> A : Prop : Type

singleton type

Inductive unit : Set :=
  tt : unit.

empty type

Inductive Empty_set : Set :=
  .

disjoint union of two types

Inductive sum (A B : Set) : Set :=
  | inl : A -> sum A B
  | inr : B -> sum A B.
cartesian product of two types

\[ \text{Inductive prod \ (A \ B : \text{Set}) : \text{Set} := } \]
\[ \text{pair : A \to B \to \text{prod A B}.} \]

truth: inductive definition

\[ \text{Inductive True : Prop := } \]
\[ \text{I : True.} \]

falsity: inductive definition

\[ \text{Inductive False : Prop := } \]
\[ \text{} \]

falsity: induction principle

\[ \text{False_ind : } \]
\[ \text{forall P : Prop, False \to P} \]

\[ \text{gives the elimination rule via the tactics} \]
\[ \bullet \text{elim h} \]
\[ \bullet \text{elimtype False} \]
\[ \bullet \text{apply False_ind} \]
**conjunction: inductive definition**

\[
\text{Inductive and (A : Prop) (B : Prop) : Prop := } \\
\text{conj : A -> B -> A \land B.}
\]

gives the introduction rule via the tactics

- apply conj
- split

**conjunction: induction principle**

\[
\text{and_ind: forall A B P : Prop, } \\
(A -> B -> P) -> A \land B -> P
\]

gives the elimination rule via the tactics

- elim h
- apply and_ind

**disjunction: inductive definition**

\[
\text{Inductive or (A : Prop) (B : Prop) : Prop := } \\
\text{or_introl : A -> A \lor B} \\
| \text{or_intror : B -> A \lor B}
\]

gives the introduction rule via the tactics

- left
- right

**disjunction: induction principle**

\[
\text{or_ind: forall A B P : Prop, } \\
(A -> P) -> (B -> P) -> A \lor B -> P
\]

gives the elimination rule via the tactic

- elim h
tactic elim

elim H can be used for every hypothesis H in some inductive type

datatypes in Set and logic in Prop

unit and True
Empty_set and False
prod and and
sum and or

Prop versus bool

I : True : Prop
true : bool : Set

True : inductive type
bool : inductive type
true : not a type

inductive predicate: even

Inductive even : nat -> Prop :=
| even0 : even 0
| evenSS : forall n:nat, even n -> even (S (S n)).
even: examples

even0 : even 0 : Prop
even 1 : Prop
evenSS 0 even0 : even 2 : Prop
even 3 : Prop
evenSS 2 (evenSS 0 even0) : even 4 : Prop

alternative definition: even and odd

Inductive ev : nat -> Prop :=
| ev0 : ev 0
| evS : forall n:nat, odd n -> ev (S n)

with odd : nat -> Prop :=
| oddS : forall n:nat, ev n -> odd (S n).

how to define inductive predicates

• constructors are axioms and should be intuitively true
• constructors define mutually exclusive cases
• make sure to test positive and negative cases

inductive predicates: induction principle

every inductive predicate comes with an induction principle
this is used in the tactic inversion
**even: induction principle**

```plaintext
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
```

**even: tactic inversion**

the tactic inversion (very) roughly does this: apply even_ind

**what does inversion do?**

elimination is used to prove properties

\[
\forall x_1, \ldots, x_k, I(x_1, \ldots, x_k) \rightarrow P(x_1, \ldots, x_k)
\]

sometimes our goal is \( I(t_1, \ldots, t_k) \rightarrow P(t_1, \ldots, t_k) \)

and the generalization \( \forall x_1, \ldots, x_k, I(x_1, \ldots, x_k) \rightarrow P(x_1, \ldots, x_k) \)
does not hold

then use

\[
\forall x_1, \ldots, x_k, I(x_1, \ldots, x_k) \rightarrow (x_1, \ldots, x_k) = (t_1, \ldots, t_k) \rightarrow
P(t_1, \ldots, t_k)
\]

**tactics inversion**

inversion H.
simple inversion H.
inversion_clear H.