schema

inductive definitions: predicates

inductive predicates: logical connectives

overview

inductive definitions: predicates

inductive predicates: logical connectives

inductive type

a new type

constructor functions

pattern matching and ι-reduction

induction principle

inductive types in each universe

- inductive data-types:
  `Inductive ... : Set :=`
- inductive predicates:
  `Inductive ... : Prop :=`
- `Inductive ... : Type :=`

universes of Coq: example

term	:	type	:	kind		
`(S O)`	:	`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

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

overview

inductive definitions: predicates

inductive predicates: logical connectives

truth: inductive definition

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

falsity: inductive definition

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

falsity: induction principle

```
False_ind :
  forall P : Prop, False -> P
```

gives the elimination rule via the tactics

- `elim h`
- `elimtype False`
- `apply False_ind`

conjunction: inductive definition

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

gives the introduction rule via the tactics

- `apply conj`
- `split`

conjunction: induction principle

```
and_ind: forall A B P : Prop,
  (A -> B -> P) -> A /\ B -> P
```

gives the elimination rule via the tactics

- `elim h`
- `apply and_ind`

disjunction: inductive definition

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

gives the introduction rule via the tactics

- `left.`
- `right.`

disjunction: induction principle

gives the elimination rule via the tactic

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

- `elim h.`

tactic elim

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

data-types in Set and logic in Prop

`unit` and `True`

`Empty_set` and `False`

`propd` 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
- 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

```
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 `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 $I(t_1, \ldots, t_k) \rightarrow P(t_1, \ldots, t_j k)$

and the generaliszation $\forall x_1, \ldots, x_k.\, I(x_1, \ldots, x_k) \rightarrow P(x_1, \ldots, x_k)$ 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.
```

existential quantification in Set

inductive type:

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

syntax:

```
{x:A | P x}
```

for program extraction

use existential quantification in Set

schema

program extraction

program extraction: examples

verified programs: alternative approach

examples lambda P

successor: existence proof and extracted program

specification:

```
Theorem successor:
  forall n:nat, {m:nat | m = S n} .
```

extracted program :

```
let successor n =
  S n
```

predecessor: existence proof and extracted program

specification:

```
Theorem predecessor :
  forall n:nat, ~(n=0) -> {m:nat | S m = n} .
```

extracted program :

```
let rec predecessor = function
  | 0 -> assert false (* absurd case *)
  | S n0 -> n0
```

insertion sort: existence proof

```
Theorem Sort :
  forall l : natlist,
  {l' : natlist | permutation l l' /\ sorted l'}.
```

insertion sort: predicate permutation

```
Inductive permutation : natlist->natlist->Prop :=
  | permutation_nil : permutation nil nil
  | permutation_cons :
      forall (n : nat) (l l' l'' : natlist),
        permutation l l' ->
        inserted n l' l'' ->
        permutation (cons n l) l''.
```

insertion sort: predicate inserted

```
Inductive inserted (n:nat):
natlist->natlist->Prop :=
  | inserted_front :
      forall l : natlist, inserted n l (cons n l)
  | inserted_cons :
      forall (m : nat) (l l' : natlist),
        inserted n l l' ->
        inserted n (cons m l) (cons m l').
```

le: family of inductive predicates

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

```
le_ind
     : forall (n : nat) (P : nat -> Prop),
       P n ->
       (forall m : nat, le n m -> P m -> P (S m)) ->
       forall n0 : nat, le n n0 -> P n0
```

le: examples

```
                        le_n 0  :  le 0 0  :  Prop
                        le_n 7  :  le 7 7  :  Prop
              le_S 0 0 (le_n 0)  :  le 0 1  :  Prop
le_S 0 1 (le_S 0 0 (le_n 0))  :  le 0 2  :  Prop
```

insertion sort: predicate sorted

```
Inductive sorted : natlist -> Prop :=
| sorted0 : sorted nil
| sorted1 : forall n:nat , sorted (cons n nil)
| sorted2 : forall n h:nat , forall t:natlist ,
              le n h ->
              sorted (cons h t) ->
              sorted (cons n (cons h t)).
```

Leibniz equality

two terms are equal if they have the same properties

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  refl_equal

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