Induction principles

This document contains 32 examples of inductive types, together with their dependent induction principles and non-dependent recursion/induction principles. If the type is in `Set`, the non-dependent recursion principle (the 'recursor') is given. If it is in `Prop`, the non-dependent induction principle is given.

An inductive type in `Prop` sometimes has a recursion principle too, but only if it is non-recursive and has just a single constructor (this restriction is necessary for program extraction to be possible). Examples of such a type are the equality types at the end of Section 7.

The dependent induction principle of an inductive type has the form:

for all parameters,
for all predicates P on the type,
if *P is conserved under all constructors,* ***then***
P holds on the whole type

For example, the induction principle of the natural numbers (Example 3.1) is:

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

The type of polymorphic lists (Example 4.2) is an example with parameters. Its induction principle is:

```
forall (A : Set)
  (P : list A -> Prop),
    P (nil A) ->
    (forall (x : A) (l : list A), P l -> P (cons A x l)) ->
      forall l : list A, P l
```

The four parts of the induction principle are consistently indented in all examples.

To get the non-dependent principle from the corresponding dependent principle, one leaves out the red arguments.

Constructors have been underlined everywhere. For clarity, some constructor names (like `nil` and `cons`) reoccur in different examples.

The sorts are in blue, and can be independently changed from `Prop` to `Set` or `Type`.

Identifiers are generally the ones from the Coq library, but not always. What is `Set` here, often is `Type` in the Coq library.

Using type inference, implicit arguments, notation and other Coq features, these examples can be written in a much more compact and readable way. We have refrained from doing that here for conceptual clarity.

1 Finite types

1.1 Empty type and falsity

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

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

Dependent induction principle

```
  forall (P : Empty_set -> Prop)
      (x : Empty_set), P x
```

```
  forall (P : False -> Prop)
      (x : False), P x
```

Non-dependent recursion/induction principle

```
  forall P : Set,
      Empty_set -> P
```

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

$$\frac{\bot}{P}\,E\bot$$

1.2 Unit type and truth

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

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

$$\frac{}{\top}\,I\top$$

Dependent induction principle

```
forall P : unit -> Prop,
  P tt ->
    forall x : unit, P x
```

```
forall P : True -> Prop,
  P I ->
    forall x : True, P x
```

Non-dependent recursion/induction principle

```
forall P : Set,
  P ->
    unit -> P
```

```
forall P : Prop,
  P ->
    True -> P
```

$$\frac{\top \quad P}{P} E\top$$

1.3 Booleans

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

Dependent induction principle

```
forall P : bool -> Prop,
  P true ->
  P false ->
    forall x : bool, P x
```

Non-dependent recursion principle

```
forall P : Set,
  P ->
  P ->
    bool -> P
```

The recursor is the *if-then-else*, but with the condition *after* the two branches.

1.4 Dependent finite types

```
Inductive fin : nat -> Set :=
  | f0 : forall n : nat, fin (S n)
  | fS : forall n : nat, fin n -> fin (S n).
```

The type `fin n` has `n` elements. For example, the type `fin 3` has the elements:

```
  f0 2
  fS 2 (f0 1)
  fS 2 (fS 1 (f0 0))
```

Dependent induction principle

```
  forall P : forall n : nat, fin n -> Prop,
    (forall n : nat, P (S n) (f0 n)) ->
    (forall (n : nat) (x : fin n), P n x -> P (S n) (fS n x)) ->
      forall (n : nat) (x : fin n), P n x
```

Non-dependent recursion principle

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

2 Logical operators and corresponding datatype constructors

2.1 Cartesian product and conjunction

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

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

$$\frac{A \quad B}{A \wedge B} I\wedge$$

Dependent induction principle

```
forall (A B : Set)
  (P : prod A B -> Prop),
    (forall (x : A) (y : B), P (pair x y)) ->
      forall z : prod A B, P z
```

```
forall (A B : Prop)
  (P : and A B -> Prop),
    (forall (x : A) (y : B), P (conj x y)) ->
      forall z : and A B, P z
```

Non-dependent recursion/induction principle

```
forall (A B
  P : Set),
    (A -> B -> P) ->
      prod A B -> P
```

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

$$\frac{A \wedge B \qquad A \rightarrow B \rightarrow P}{P}\, E\wedge$$

2.2 Disjoint sum and disjunction

```
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.
```

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

Dependent induction principle

```
forall (A B : Set)
  (P : sum A B -> Prop),
    (forall x : A, P (inl x)) ->
    (forall y : B, P (inr y)) ->
      forall z : sum A B, P z
```

```
forall (A B : Prop)
  (P : or A B -> Prop),
    (forall x : A, P (or_introl x)) ->
    (forall y : B, P (or_intror y)) ->
      forall z : or A B, P z
```

Non-dependent recursion/induction principle

```
forall (A B
  P : Set),
    (A -> P) ->
    (B -> P) ->
      sum A B -> P
```

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

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

2.3 Dependent product type (dependent pairs, Σ-type), subset type and existential quantifier

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

$$\frac{B[x := M]}{\exists x.\, B}\,I\exists$$

Coq has the following notations:

`{ x:A & B }`	is syntax for	`sigT A (fun x:A => B)`
`{ x:A | B }`	is syntax for	`sig A (fun x:A => B)`
`exists x:A, B`	is syntax for	`ex A (fun x:A => B)`

Dependent induction principle

```
forall (A : Set) (B : A -> Set)
  (P : sigT A B -> Prop),
    (forall (x : A) (y : B x), P (existT A B x y)) ->
      forall z : sigT A B, P z
```

```
forall (A : Set) (B : A -> Prop)
  (P : sig A B -> Prop),
    (forall (x : A) (y : B x), P (exist A B x y)) ->
      forall z : sig A B, P z
```

```
forall (A : Set) (B : A -> Prop)
  (P : ex A B -> Prop),
    (forall (x : A) (y : B x), P (ex_intro A B x y)) ->
      forall z : ex A B, P z
```

Non-dependent recursion/induction principle

```
forall (A : Set) (B : A -> Set)
  (P : Set),
    (forall x : A, B x -> P) ->
      sigT A B -> P
```

```
forall (A : Set) (B : A -> Prop)
  (P : Set),
    (forall x : A, B x -> P) ->
      sig A B -> P
```

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

$$\frac{\exists x.B \qquad \forall x.B \to P}{P} E\exists$$

3 Number types

3.1 Natural numbers

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

Dependent induction principle

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

Non-dependent recursion principle (= primitive recursion)

```
  forall P : Set,
    P ->
    (nat -> P -> P) ->
      nat -> P
```

3.2 Positive binary integers

```
Inductive positive : Set :=
  | xH : positive
  | xO : positive -> positive
  | xI : positive -> positive.
```

The constructors correspond to the following functions on the positive integers:

xH	1
xO	$\lambda n.\, 2n$
xI	$\lambda n.\, 2n+1$

For example the number $42 = 101010_2$ is represented by the term

```
xO (xI (xO (xI (xO xH))))
```

The constructors in this term match the bits of the number, but in the opposite order.

Dependent induction principle

```
forall P : positive -> Prop,
  P xH ->
  (forall n : positive, P n -> P (xO n)) ->
  (forall n : positive, P n -> P (xI n)) ->
    forall n : positive, P n
```

Non-dependent recursion principle

```
forall P : Set,
  P ->
  (forall n : positive, P -> P) ->
  (forall n : positive, P -> P) ->
    forall n : positive, P
```

3.3 Integers

```
Inductive Z : Set :=
  | Z0 : Z
  | Zpos : positive -> Z
  | Zneg : positive -> Z.
```

Dependent induction principle

```
forall P : Z -> Prop,
  P Z0 ->
  (forall n : positive, P (Zpos n)) ->
  (forall n : positive, P (Zneg n)) ->
    forall i : Z, P i
```

There are other, often more useful, induction principles for Z. For example, if one defines successor and predecessor functions ZS and ZP, one can prove:

```
forall P : Z -> Prop,
  P Z0
  (forall i : Z, P i -> P (ZS i)) ->
  (forall i : Z, P i -> P (ZP i)) ->
    forall i : Z, P i
```

Non-dependent recursion principle

```
forall P : Set,
  P ->
  (positive -> P) ->
  (positive -> P) ->
    Z -> P
```

The non-dependent recursion principle that corresponds to the alternate induction principle (compare this to the non-dependent recursion principle for the natural numbers) is:

```
forall P : Set,
  P
  (Z -> P -> P) ->
  (Z -> P -> P) ->
    Z -> P
```

4 List types

4.1 Lists of natural numbers

```
Inductive natlist : Set :=
  | nil : natlist
  | cons : nat -> natlist -> natlist.
```

Dependent induction principle

```
forall P : natlist -> Prop,
  P nil ->
  (forall (x : nat) (l : natlist), P l -> P (cons x l)) ->
    forall l : natlist, P l
```

Non-dependent recursion principle (= fold)

```
forall P : Set,
  P ->
  (nat -> natlist -> P -> P) ->
    natlist -> P
```

4.2 Polymorphic lists

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

Dependent induction principle

```
forall (A : Set)
  (P : list A -> Prop),
    P (nil A) ->
    (forall (x : A) (l : list A), P l -> P (cons A x l)) ->
      forall l : list A, P l
```

Non-dependent recursion principle

```
forall (A
  P : Set),
    P ->
    (A -> list A -> P -> P) ->
      list A -> P
```

4.3 Vectors of natural numbers

```
Inductive natvec : nat -> Set :=
  | nil : natvec O
  | cons : forall n : nat, nat -> natvec n -> natvec (S n).
```

Dependent induction principle

```
  forall P : forall n : nat, natvec n -> Prop,
    P O nil ->
    (forall (n x : nat) (v : natvec n), P n v -> P (S n) (cons n x v)) ->
      forall (n : nat) (v : natvec n), P n v
```

Non-dependent recurson principle

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

4.4 Polymorphic vectors

```
Inductive vec (A : Set) : nat -> Set :=
  | nil : vec A O
  | cons : forall n : nat, A -> vec A n -> vec A (S n).
```

Dependent induction principle

```
forall (A : Set)
  (P : forall n : nat, vec A n -> Prop),
    P O (nil A) ->
    (forall (n : nat) (x : A) (v : vec A n), P n v -> P (S n) (cons A n x v)) ->
      forall (n : nat) (v : vec A n), P n v
```

Non-dependent recursion principle

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

5 Tree types

5.1 Unlabeled binary trees

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

Dependent induction principle

```
  forall P : bintree -> Prop,
    P leaf ->
    (forall l : bintree, P l -> forall r : bintree, P r ->
        P (node l r)) ->
      forall t : bintree, P t
```

Non-dependent recursion principle

```
  forall P : Set,
    P ->
    (bintree -> P -> bintree -> P ->
        P) ->
      bintree -> P
```

5.2 Binary trees with natural numbers at the leaves

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

Dependent induction principle

```
forall P : bintree -> Prop,
  (forall n : nat, P (leaf n) ->
  (forall l : bintree, P l -> forall r : bintree, P r ->
      P (node l r)) ->
    forall t : bintree, P t
```

Non-dependent recursion principle

```
forall P : Set,
  (nat -> P) ->
  (bintree -> P -> bintree -> P ->
      P) ->
    bintree -> P
```

5.3 Alternative representation for unlabeled binary trees

```
Inductive bintree : Set :=
  | leaf : (fin 0 -> bintree) -> bintree
  | node : (fin 2 -> bintree) -> bintree.
```

Compare this to Example 5.1. The definition of the `fin` types is Example 1.4.

Dependent induction principle

```
forall P : bintree -> Prop,
  (forall u : fin 0 -> bintree, (forall i : fin 0, P (u i)) ->
      P (leaf u)) ->
  (forall lr : fin 2 -> bintree, (forall i : fin 2, P (lr i)) ->
      P (node lr)) ->
    forall t : bintree, P t
```

Non-dependent recursion principle

```
forall P : Set,
  ((fin 0 -> bintree) -> (fin 0 -> P) ->
      P) ->
  (fin 2 -> bintree) -> (fin 2 -> P) ->
      P) ->
    bintree -> P
```

5.4 Unlabeled finitely branching trees

```
Inductive tree : Set :=
  | node : forall n : nat, (fin n -> tree) -> tree.
```

Dependent induction principle

```
forall P : tree -> Prop,
  (forall (n : nat) (u : fin n -> tree),
      (forall i : fin n, P (u i)) -> P (node n u)) ->
    forall t : tree, P t
```

Non-dependent recursion principle

```
forall P : Set,
  (forall n : nat, (fin n -> tree) ->
      (fin n -> P) -> P) ->
    tree -> P
```

5.5 W-types

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

Traditionally, what is called `node` in the previous examples is called `sup` in the case of W-types.

Compared to the previous example, `A` takes the place of the natural numbers and `B` takes the place of the `fin` types. This means that if a node is labeled with `x` in `A`, then there is an edge going down from it for every element of `B x`. This means that the unlabeled binary trees can also be represented using W-types, as:

```
W bool (fun b : bool => if b then fin 0 else fin 2)
```

In this, the notation '`if b then ... else ...`' is syntax for:

```
match b with
| true => ...
| false => ...
end
```

Dependent induction principle

```
forall (A : Set) (B : A -> Set)
  (P : W A B -> Prop),
    (forall (x : A) (u : B x -> W A B),
        (forall i : B x, P (u i)) -> P (sup A B x u)) ->
      forall t : W A B, P t
```

Non-dependent recursion principle

```
forall (A : Set) (B : A -> Set)
  (P : Set),
    (forall x : A, (B x -> W A B) ->
        (B x -> P) -> P) ->
      W A B -> P
```

6 Option type

6.1 Option type

```
Inductive option (A : Set) : Set :=
  | None : option A
  | Some : A -> option A.
```

Dependent induction principle

```
forall (A : Set)
  (P : option A -> Prop),
    P (None A) ->
    (forall x : A, P (Some A x)) ->
      forall x : option A, P x
```

Non-dependent recursion principle

```
forall (A : Set)
  (P : Set),
    P ->
    (A -> P) ->
      option A -> P
```

7 Inductive predicates

7.1 Even natural numbers

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

Dependent induction principle

```
  forall P : forall n : nat, even n -> Prop,
    P O even_O ->
    (forall (n : nat) (H : even n), P n H -> P (S (S n)) (even_SS n H)) ->
      forall (n : nat) (H : even n), P n H
```

Non-dependent induction principle

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

$$\frac{P(0) \qquad \forall n.\,\mathsf{even}(n) \wedge P(n) \rightarrow P(n+2)}{\forall n.\,\mathsf{even}(n) \rightarrow P(n)}$$

7.2 Sorted polymorphic lists

```
Inductive sorted (A : Set) (le : A -> A -> Prop) : list A -> Prop :=
  | sorted_0 : sorted A le (nil A)
  | sorted_1 : forall x : A, sorted A le (cons A x nil)
  | sorted_2 : forall (x y : A) (l : list A), le x y ->
      sorted A le (cons A y l) -> sorted A le (cons A x (cons A y l)).
```

Of course, in practice one would make `A` and `le` implicit, and the terms would look much less hairy.

Dependent induction principle

```
forall (A : Set) (le : A -> A -> Prop)
  (P : forall l : list A, sorted A le l -> Prop),
    P (nil A) (sorted_0 A le) ->
    (forall x : A, P (cons A x (nil A)) (sorted_1 A le x)) ->
    (forall (x y : A) (l : list A) (Hle : le x y)
         (H : sorted A le (cons A y l)),
         P (cons A y l) H ->
         P (cons A x (cons A y l)) (sorted_2 A le x y l Hle H)) ->
      forall (l : list A) (H : sorted A le l), P l H
```

Non-dependent induction principle

```
forall (A : Set) (le : A -> A -> Prop)
  (P : list A -> Prop),
    P (nil A) ->
    (forall x : A, P (cons A x (nil A))) ->
    (forall (x y : A) (l : list A), le x y ->
         sorted A le (cons A y l) ->
         P (cons A y l) ->
         P (cons A x (cons A y l))) ->
      forall (l : list A) (H : sorted A le l), P l
```

$$\frac{P([]) \qquad \forall x.\,P([x]) \qquad \forall x\,y\,l.\;x \leq y \wedge \mathsf{sorted}(y :: l) \wedge P(y :: l) \rightarrow P(x :: y :: l)}{\forall l.\,\mathsf{sorted}(l) \rightarrow P(l)}$$

7.3 Less-or-equal on natural numbers

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

Dependent induction principle

```
forall (n : nat)
  (P : forall m : nat, le n m -> Prop),
    P n (le_refl n) ->
    (forall (m : nat) (H : le n m), P m H -> P (S m) (le_S n m H)) ->
      forall (m : nat) (H : le n m), P m H
```

Non-dependent induction principle

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

7.4 Equality on natural numbers

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

Dependent induction principle

```
forall (n : nat)
  (P : forall m : nat, eq_nat n m -> Prop),
    P n (eq_refl n) ->
      forall (m : nat) (H : eq_nat n m), P m H
```

Non-dependent induction principle

```
forall (n : nat)
  (P : nat -> Prop),
    P n ->
      forall m : nat, eq_nat n m -> P m
```

7.5 Alternative version of equality on natural numbers

```
Inductive eq_nat : nat -> nat -> Prop :=
  | eq_refl : forall n : nat, eq_nat n n.
```

Dependent induction principle

```
  forall P : forall n m : nat, eq_nat n m -> Prop,
    (forall n : nat, P n n (eq_refl n)) ->
      forall (n m : nat) (H : eq_nat n m), P n m H
```

Non-dependent induction principle

```
  forall P : nat -> nat -> Prop,
    (forall n : nat, P n n) ->
      forall n m : nat, eq_nat n m -> P n m
```

7.6 Polymorphic equality

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

Dependent induction principle

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

Non-dependent induction principle

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

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