CENG 3549 – Functional Programming

Mathematical Induction & Induction over Lists & Structural Induction & Formal Verification of Functional Programs

Burak Ekici

December 8, 2022

Outline

When to use Mathematical Induction?

- prove some property P for all natural numbers

When to use Mathematical Induction?

- prove some property P for all natural numbers
- more formally, prove:

$$\forall n. P(n) \qquad (\text{where } n \in \mathbb{N})$$

When to use Mathematical Induction?

- prove some property P for all natural numbers
- more formally, prove:

$$\forall n.\, P(n) \qquad (\text{where } n \in \mathbb{N})$$

How is it Applied?

- mathematical induction consists of two steps:

When to use Mathematical Induction?

- prove some property P for all natural numbers
- more formally, prove:

$$\forall n.\, P(n) \qquad (\text{where } n \in \mathbb{N})$$

How is it Applied?

- mathematical induction consists of two steps:
  1. prove base case

$$P(0)$$

When to use Mathematical Induction?

- prove some property P for all natural numbers
- more formally, prove:

$$\forall n.\, P(n) \qquad (\text{where } n \in \mathbb{N})$$

How is it Applied?

- mathematical induction consists of two steps:
  1. prove base case

$$P(0)$$

show property for 0

When to use Mathematical Induction?

- prove some property P for all natural numbers
- more formally, prove:

$$\forall n.\, P(n) \qquad (\text{where } n \in \mathbb{N})$$

How is it Applied?

- mathematical induction consists of two steps:
  1. prove base case
     $$P(0)$$
  2. prove step case
     $$\forall k.\, (P(k) \implies P(k+1))$$

When to use Mathematical Induction?

- prove some property P for all natural numbers
- more formally, prove:

$$\forall n.\, P(n) \qquad (\text{where } n \in \mathbb{N})$$

How is it Applied?

- mathematical induction consists of two steps:
  1. prove base case
  $$P(0)$$
  2. prove step case
  $$\forall k.\, (P(k) \implies P(k+1))$$

assume $P(k)$ (induction hypothesis), show $P(k+1)$

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n.\, P(n)$)

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number $(\forall n.\, P(n))$

Example ($\boldsymbol{P(3)}$)

- have $P(0)$

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($\boldsymbol{P(3)}$)

- have $P(0)$
- and $P(0) \implies P(1)$

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k + 1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($P(3)$)

- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($P(3)$)

- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($\boldsymbol{P(3)}$)

- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($\boldsymbol{P(3)}$)

- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($P(3)$)

- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$
- have $P(3)$

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($\boldsymbol{P(3)}$)

- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$
- have $P(3)$

Idea

- reach arbitrary n s.t. $P(n)$
- hence, $\forall n. P(n)$

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n.\, P(n)$)

Example ($\boldsymbol{P(3)}$)

- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$
- have $P(3)$

Idea

- reach arbitrary n s.t. $P(n)$
- hence, $\forall n.\, P(n)$

Domino Effect

1. first domino falls
2. if domino falls, right neighbor falls

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($P(3)$)

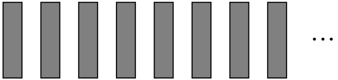
- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$
- have $P(3)$

Idea

- reach arbitrary n s.t. $P(n)$
- hence, $\forall n. P(n)$

Domino Effect

1. first domino falls
2. if domino falls, right neighbor falls



Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($\boldsymbol{P(3)}$)

- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$
- have $P(3)$

Idea

- reach arbitrary n s.t. $P(n)$
- hence, $\forall n. P(n)$

Domino Effect

1. first domino falls
2. if domino falls, right neighbor falls

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n.\, P(n)$)

Example ($P(3)$)

- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$
- have $P(3)$

Idea

- reach arbitrary n s.t. $P(n)$
- hence, $\forall n.\, P(n)$

Domino Effect

1. first domino falls
2. if domino falls, right neighbor falls

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n.\, P(n)$)

Example ($\boldsymbol{P(3)}$)

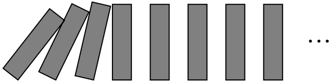
- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$
- have $P(3)$

Idea

- reach arbitrary n s.t. $P(n)$
- hence, $\forall n.\, P(n)$

Domino Effect

1. first domino falls
2. if domino falls, right neighbor falls



Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k + 1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($P(3)$)

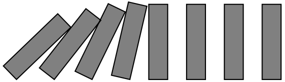
- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$
- have $P(3)$

Idea

- reach arbitrary n s.t. $P(n)$
- hence, $\forall n. P(n)$

Domino Effect

1. first domino falls
2. if domino falls, right neighbor falls



$\ldots$

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($\boldsymbol{P(3)}$)

- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$
- have $P(3)$

Idea

- reach arbitrary n s.t. $P(n)$
- hence, $\forall n. P(n)$

Domino Effect

1. first domino falls
2. if domino falls, right neighbor falls

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k+1$
- want to show P for every natural number ($\forall n.\, P(n)$)

Example ($P(3)$)

- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$
- have $P(3)$

Idea

- reach arbitrary n s.t. $P(n)$
- hence, $\forall n.\, P(n)$

Domino Effect

1. first domino falls
2. if domino falls, right neighbor falls

Why does this Work?

- have two facts:
  1. P true for 0
  2. for arbitrary k, if P true for k then P true for $k + 1$
- want to show P for every natural number ($\forall n. P(n)$)

Example ($\boldsymbol{P(3)}$)

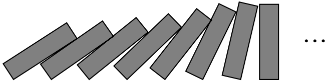
- have $P(0)$
- and $P(0) \implies P(1)$
- thus $P(1)$
- with $P(1) \implies P(2)$
- have $P(2)$
- with $P(2) \implies P(3)$
- have $P(3)$

Idea

- reach arbitrary n s.t. $P(n)$
- hence, $\forall n. P(n)$

Domino Effect

1. first domino falls
2. if domino falls, right neighbor falls



What is a “Property”?

- anything that depends on some input and is either true or false
- that is, some function `p :: a -> Bool`

What is a "Property"?

- anything that depends on some input and is either true or false
- that is, some function `p :: a -> Bool`

Remark

- base case may be changed
- e.g., if base case $P(1)$, property holds for all $n \geq 1$

What is a "Property"?

- anything that depends on some input and is either true or false
- that is, some function `p :: a -> Bool`

Remark

- base case may be changed
- e.g., if base case $P(1)$, property holds for all $n \geq 1$

Induction Principle

$$(P(m) \land \forall k \geq m. (P(k) \implies P(k+1))) \implies \forall n \geq m. P(n)$$

Lemma (Gauß's Formula)

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \cdots + x = \frac{x(x+1)}{2})$

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \cdots + x = \frac{x(x+1)}{2})$
- base case: $P(0)$

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \cdots + x = \frac{x(x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 \qquad\qquad\qquad)$

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \cdots + x = \frac{x(x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 \qquad\qquad)$

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \cdots + x = \frac{x(x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0(0+1)}{2})$

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \cdots + x = \frac{x(x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0(0+1)}{2})$
- step case: $P(k) \implies P(k+1)$

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \cdots + x = \frac{x(x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0(0+1)}{2})$
- step case: $P(k) \implies P(k+1)$
  IH: $P(k) = (1 + 2 + \cdots + k = \frac{k(k+1)}{2})$

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \cdots + x = \frac{x(x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0(0+1)}{2})$
- step case: $P(k) \implies P(k+1)$
  IH: $P(k) = (1 + 2 + \cdots + k = \frac{k(k+1)}{2})$
  show: $P(k+1)$

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \dots + x = \frac{x(x+1)}{2})$
- base case: $P(0) = (1 + 2 + \dots + 0 = 0 = \frac{0(0+1)}{2})$
- step case: $P(k) \implies P(k+1)$
  IH: $P(k) = (1 + 2 + \dots + k = \frac{k(k+1)}{2})$
  show: $P(k+1)$

$$1 + 2 + \dots + (k+1)$$

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \cdots + x = \frac{x(x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0(0+1)}{2})$
- step case: $P(k) \implies P(k+1)$
  IH: $P(k) = (1 + 2 + \cdots + k = \frac{k(k+1)}{2})$
  show: $P(k+1)$

$$1 + 2 + \cdots + (k+1) = (1 + 2 + \cdots + k) + (k+1)$$

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \cdots + x = \frac{x(x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0(0+1)}{2})$
- step case: $P(k) \implies P(k+1)$
  IH: $P(k) = (1 + 2 + \cdots + k = \frac{k(k+1)}{2})$
  show: $P(k+1)$

$$\begin{aligned} 1 + 2 + \cdots + (k+1) &= (1 + 2 + \cdots + k) + (k+1) \\ &\overset{IH}{=} \frac{k(k+1)}{2} + (k+1) \end{aligned}$$

Lemma (Gauß's Formula)

- $P(x) = (1 + 2 + \cdots + x = \frac{x(x+1)}{2})$
- base case: $P(0) = (1 + 2 + \cdots + 0 = 0 = \frac{0(0+1)}{2})$
- step case: $P(k) \implies P(k+1)$
  IH: $P(k) = (1 + 2 + \cdots + k = \frac{k(k+1)}{2})$
  show: $P(k+1)$

$$\begin{aligned} 1 + 2 + \cdots + (k+1) &= (1 + 2 + \cdots + k) + (k+1) \\ &\overset{IH}{=} \frac{k(k+1)}{2} + (k+1) \\ &= \frac{(k+1)(k+2)}{2} \end{aligned}$$

Outline

1. Mathematical Induction
2. Structural Induction
3. The Coq Proof Assistant

Structural Induction

- proof principle similar to mathematical induction
- works in the domain of recursively defined structures (not only in $\mathbb{N}$)

Structural Induction

- proof principle similar to mathematical induction
- works in the domain of recursively defined structures (not only in $\mathbb{N}$)

Example (Lists)

```
data List a where
       Nil  :: [a]
       Cons :: a -> [a] -> [a]
```

Structural Induction

- proof principle similar to mathematical induction
- works in the domain of recursively defined structures (not only in $\mathbb{N}$)

Example (Lists)

```
data List a where
       Nil  :: [a]
       Cons :: a -> [a] -> [a]
```

Notation

Denote

- `Nil` by []
- `Cons x xs` by $x : xs$

Structural Induction

- proof principle similar to mathematical induction
- works in the domain of recursively defined structures (not only in $\mathbb{N}$)

Example (Lists)

```
data List a where
       Nil  :: [a]
       Cons :: a -> [a] -> [a]
```

Notation

Denote

- `Nil` by []
- `Cons x xs` by $x : xs$

Remark (notes)

- lists are recursive structures
- non-recursive constructor (base case): []
- recursive constructor (step case): x:xs

Induction Principle for Lists – Informally

- to show $P(l)$ for all lists l
- show base case: $P([])$
- show step case: $P(xs) \implies P(x:xs)$ for arbitrary x and xs

Induction Principle for Lists – Informally

- to show $P(l)$ for all lists l
- show base case: $P([\,])$
- show step case: $P(xs) \implies P(x\!:\!xs)$ for arbitrary x and xs

Induction Principle for Lists – Formally

$$(P([\,]) \wedge \forall x.\, \forall xs.\, (P(xs) \implies P(x\!:\!xs))) \implies \forall l.\, P(l)$$

Induction Principle for Lists – Informally

- to show $P(l)$ for all lists l
- show base case: $P([\,])$
- show step case: $P(xs) \implies P(x\!:\!xs)$ for arbitrary x and xs

Induction Principle for Lists – Formally

$$(P([\,]) \land \forall x.\, \forall xs.\, (P(xs) \implies P(x\!:\!xs))) \implies \forall l.\, P(l)$$

Remark

- for lists, P can be seen as function `p :: [a] -> Bool`

Definition (Append)

$$
\begin{aligned}
[\,] \quad &+\!\!+\ l_2 = l_2 \\
(x{:}xs)\ &+\!\!+\ l_2 = x : (xs +\!\!+ l_2)
\end{aligned}
$$

Definition (Append)

$$[\,] \mathbin{+\!\!+} l_2 = l_2$$
$$(x{:}xs) \mathbin{+\!\!+} l_2 = x : (xs \mathbin{+\!\!+} l_2)$$

Lemma (Nil is right identity of append)

$[\,]$ is right identity of $\mathbin{+\!\!+}$, that is,

$$l \mathbin{+\!\!+} [\,] = l$$

Definition (Append)

$$
\begin{aligned}
[\,] \mathbin{+\!\!+} l_2 &= l_2 \\
(x{:}xs) \mathbin{+\!\!+} l_2 &= x : (xs \mathbin{+\!\!+} l_2)
\end{aligned}
$$

Lemma (Nil is right identity of append)

$[\,]$ is right identity of $\mathbin{+\!\!+}$, that is,

$$l \mathbin{+\!\!+} [\,] = l$$

Lemma (Append is associative)

$\mathbin{+\!\!+}$ is associative, that is,

$$(l_1 \mathbin{+\!\!+} l_2) \mathbin{+\!\!+} l_3 \;=\; l_1 \mathbin{+\!\!+} (l_2 \mathbin{+\!\!+} l_3)$$

Definition (Length)

```
length []      =  0
length (_:xs)  =  1 + length xs
```

Definition (Length)

```
length []      =  0
length (_:xs)  =  1 + length xs
```

Lemma (Length and append)

length of combined list is sum of lengths, that is,

$$\texttt{length}\,(l_1 +\!\!+\; l_2) \;=\; \texttt{length}\; l_1 + \texttt{length}\; l_2$$

Example (Binary Trees)

```
data BTree a where
       Empty :: BTree a
       Node  :: a -> BTree a -> BTree a -> BTree a
```

Example (Binary Trees)

```
data BTree a where
        Empty :: BTree a
        Node  :: a -> BTree a -> BTree a -> BTree a
```

Induction Principle for Binary Trees

$$(P(\mathsf{Empty}) \land \forall x.\, \forall l.\, \forall r.\, ((P(l) \land P(r)) \implies P(\mathsf{Node}\ x\ l\ r))) \implies \forall t.\, P(t)$$

Example (Perfect Binary Trees)

- a binary tree is perfect if all leaf nodes have same depth

```
perfect Empty        = True
perfect (Node _ l r) =
  height l == height r && perfect l && perfect r

height Empty        = 0
height (Node _ l r) =
  max (height l) (height r) + 1

size Empty        = 0
size (Node _ l r) = size l + size r + 1
```

Lemma

a perfect binary tree t of height n has exactly $2^n - 1$ nodes, that is,

$$P(t) = \left(\texttt{perfect}\ t \implies \texttt{size}\ t = 2^{\texttt{height}\ t} - 1\right)$$

Example

```
data Term where
        Var    :: String -> Term
        Lambda :: String -> Term -> Term
        App    :: Term   -> Term -> Term
```

Example

```
data Term where
       Var    :: String -> Term
       Lambda :: String -> Term -> Term
       App    :: Term   -> Term -> Term
```

General Structures – Induction Principle

- for every non-recursive constructor, show base case
  - base case: $P(\texttt{Var x})$
- for every recursive constructor, show step case
  - step case 1: $(P(\texttt{s}) \wedge P(\texttt{t})) \implies P(\texttt{App s t})$
  - step case 2: $P(\texttt{t}) \implies P(\texttt{Lambda x t})$

Outline

Curry-Howard Isomorphism

Logic	∼	Type Theory
Proposition		Type
Proof		Program
⋮		⋮

The Coq Proof Assistant

- by Thierry Coquand (1985)
  - implements Curry-Howard Isomorphism
  - provides recursors + inductors

The Coq Proof Assistant

- Coq is a general purpose proof management system based on a rich type system called the Calculus of Inductive Constructions (CIC)

The Coq Proof Assistant

- Coq is a general purpose proof management system based on a rich type system called the Calculus of Inductive Constructions (CIC)
- CIC is a constructive logic: a proof is a process of constructing a witness for a given formula (statement) without employing classical axioms as LEM $\forall P, P \vee \neg P$ or any equivalent e.g., DNE $\neg\neg P \implies P$

The Coq Proof Assistant

- Coq is a general purpose proof management system based on a rich type system called the Calculus of Inductive Constructions (CIC)
- CIC is a constructive logic: a proof is a process of constructing a witness for a given formula (statement) without employing classical axioms as LEM $\forall P, P \vee \neg P$ or any equivalent e.g., DNE $\neg\neg P \implies P$
- In CIC, proofs are programs and formulae are types

The Coq Proof Assistant

- Coq is a general purpose proof management system based on a rich type system called the Calculus of Inductive Constructions (CIC)
- CIC is a constructive logic: a proof is a process of constructing a witness for a given formula (statement) without employing classical axioms as LEM $\forall P, P \vee \neg P$ or any equivalent e.g., DNE $\neg\neg P \implies P$
- In CIC, proofs are programs and formulae are types
- CIC is higher-order: quantifiers over predicates (propositional valued function) allowed

The Coq Proof Assistant (cont'd)

1. Coq allows for

The Coq Proof Assistant (cont'd)

1. Coq allows for
   - defining functions and predicates;

The Coq Proof Assistant (cont'd)

1. Coq allows for
   - defining functions and predicates;
   - stating mathematical theorems and software specifications;

The Coq Proof Assistant (cont'd)

1. Coq allows for
   - defining functions and predicates;
   - stating mathematical theorems and software specifications;
   - interactively developing formal proofs of these theorems;

The Coq Proof Assistant (cont'd)

1. Coq allows for
   - defining functions and predicates;
   - stating mathematical theorems and software specifications;
   - interactively developing formal proofs of these theorems;
   - machine-checking proofs by a relatively small "(trusted) certification kernel"

The Coq Proof Assistant (cont'd)

1. Coq allows for
   - defining functions and predicates;
   - stating mathematical theorems and software specifications;
   - interactively developing formal proofs of these theorems;
   - machine-checking proofs by a relatively small "(trusted) certification kernel"
   - extracting certified programs from the constructive proof of its formal specification

The Coq Proof Assistant (cont'd)

1. Coq allows for
   - defining functions and predicates;
   - stating mathematical theorems and software specifications;
   - interactively developing formal proofs of these theorems;
   - machine-checking proofs by a relatively small "(trusted) certification kernel"
   - extracting certified programs from the constructive proof of its formal specification
2. Coq embodies

The Coq Proof Assistant (cont'd)

1. Coq allows for
   - defining functions and predicates;
   - stating mathematical theorems and software specifications;
   - interactively developing formal proofs of these theorems;
   - machine-checking proofs by a relatively small "(trusted) certification kernel"
   - extracting certified programs from the constructive proof of its formal specification
2. Coq embodies
   - functional programming language: to implement programs/mathematical objects

The Coq Proof Assistant (cont'd)

1. Coq allows for
   - defining functions and predicates;
   - stating mathematical theorems and software specifications;
   - interactively developing formal proofs of these theorems;
   - machine-checking proofs by a relatively small "(trusted) certification kernel"
   - extracting certified programs from the constructive proof of its formal specification
2. Coq embodies
   - functional programming language: to implement programs/mathematical objects
   - specification language: to develop property proofs of programs/mathematical objects

The Coq Proof Assistant (cont'd)

1. Coq allows for
   - defining functions and predicates;
   - stating mathematical theorems and software specifications;
   - interactively developing formal proofs of these theorems;
   - machine-checking proofs by a relatively small "(trusted) certification kernel"
   - extracting certified programs from the constructive proof of its formal specification
2. Coq embodies
   - functional programming language: to implement programs/mathematical objects
   - specification language: to develop property proofs of programs/mathematical objects
   - tactic language Ltac: to develop small code pieces (tactics) in order to manipulate proof states

The Coq Proof Assistant (cont'd)

1. Coq allows for
   - defining functions and predicates;
   - stating mathematical theorems and software specifications;
   - interactively developing formal proofs of these theorems;
   - machine-checking proofs by a relatively small "(trusted) certification kernel"
   - extracting certified programs from the constructive proof of its formal specification
2. Coq embodies
   - functional programming language: to implement programs/mathematical objects
   - specification language: to develop property proofs of programs/mathematical objects
   - tactic language Ltac: to develop small code pieces (tactics) in order to manipulate proof states
   - vernacular command language: to query and interact with the Coq type system

The Coq Proof Assistant (cont'd)

1. In CIC, every single construction has a type; there are

The Coq Proof Assistant (cont'd)

1. In CIC, every single construction has a type; there are
   - types for basic data/terms (atomic data-types)

The Coq Proof Assistant (cont'd)

1. In CIC, every single construction has a type; there are
   - types for basic data/terms (atomic data-types)
   - types for functions/programs

The Coq Proof Assistant (cont'd)

1. In CIC, every single construction has a type; there are
    - types for basic data/terms (atomic data-types)
    - types for functions/programs
    - types for proofs

The Coq Proof Assistant (cont'd)

1. In CIC, every single construction has a type; there are
   - types for basic data/terms (atomic data-types)
   - types for functions/programs
   - types for proofs
   - types for types themselves (sorts)

The Coq Proof Assistant (cont'd)

1. In CIC, every single construction has a type; there are
    - types for basic data/terms (atomic data-types)
    - types for functions/programs
    - types for proofs
    - types for types themselves (sorts)
2. Sorts are categorized in the three kinds:
    - `Prop`: sort (universe) of logical propositions

The Coq Proof Assistant (cont'd)

1. In CIC, every single construction has a type; there are
   - types for basic data/terms (atomic data-types)
   - types for functions/programs
   - types for proofs
   - types for types themselves (sorts)
2. Sorts are categorized in the three kinds:
   - `Prop`: sort (universe) of logical propositions
   - `Set`: sort (universe) of computations

The Coq Proof Assistant (cont'd)

1. In CIC, every single construction has a type; there are
   - types for basic data/terms (atomic data-types)
   - types for functions/programs
   - types for proofs
   - types for types themselves (sorts)
2. Sorts are categorized in the three kinds:
   - `Prop`: sort (universe) of logical propositions
   - `Set`: sort (universe) of computations
   - `Type`: super-sort of both

Definition (Impredicativity)

a universe U is called impredicative if $\forall (P : U), P \to P : U$ holds

Definition (Impredicativity)

a universe U is called impredicative if $\forall (P : U), P \to P : U$ holds

Fact

`impredicativity` + `LEM` + `large elimination (computation)` $\to$ `False`

Definition (Impredicativity)

a universe U is called impredicative if $\forall (P : U), P \to P : U$ holds

Fact

impredicativity + LEM + large elimination (computation) → False

Coq Features (Predicativity and Impredicativity)

impredicativity + LEM ~~+ computation~~ Prop: Type

Definition (Impredicativity)

a universe U is called impredicative if $\forall (P : U), P \to P : U$ holds

Fact

$$\texttt{impredicativity} + \texttt{LEM} + \texttt{large elimination (computation)} \to \texttt{False}$$

Coq Features (Predicativity and Impredicativity)

$\texttt{impredicativity} + \text{LEM} \cancel{+ \text{computation}}$ $\texttt{Prop: Type}$

$\texttt{predicativity} + \text{LEM} + \text{computation}$ $\texttt{Set: Type}$

Definition (Impredicativity)

a universe U is called impredicative if $\forall (P : U), P \rightarrow P : U$ holds

Fact

impredicativity + LEM + large elimination (computation) → False

Coq Features (Predicativity and Impredicativity)

impredicativity + LEM ~~+ computation~~	Prop : Type
predicativity + LEM + computation	Set : Type

Coq Features (Stratification of Type)

$$Type_0 : Type_1 : Type_2 : Type_3 \cdots$$

Coq Features (Inductive Definitions)

1. the declaration

```
Variable T: Type
```

gives no a priori information on the number, or the properties of its inhabitants.

Coq Features (Inductive Definitions)

1. the declaration

```
Variable T: Type
```

gives no a priori information on the number, or the properties of its inhabitants.

2. In CIC, one can define a new type I inductively by giving its constructors together with their types which must be of the form:

$$\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \mathtt{I} \text{ with } n \geqslant 0$$

- any instance of I can be obtained by finite number of constructor applications

Coq Features (Inductive Definitions)

1. the declaration

```
Variable T: Type
```

gives no a priori information on the number, or the properties of its inhabitants.

2. In CIC, one can define a new type I inductively by giving its constructors together with their types which must be of the form:

$$\tau_1 \to \tau_2 \to \ldots \to \tau_n \to \mathtt{I} \text{ with } n \geqslant 0$$

- any instance of I can be obtained by finite number of constructor applications
- inductive types must be well-founded assured by the strict positivity

Example (Natural Numbers as an Inductive Type)

```
Inductive nat : Set ≜
  | O : nat
  | S : nat → nat.
```

Example (Natural Numbers as an Inductive Type)

```
Inductive nat : Set ≜
  | O : nat
  | S : nat → nat.

nat_ind: ∀(P : nat → Prop), P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Example (Natural Numbers as an Inductive Type)

```
Inductive nat : Set ≜
  | O : nat
  | S : nat → nat.

nat_ind: ∀(P : nat → Prop), P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
nat_rec: ∀(P : nat → Set), P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Example (Natural Numbers as an Inductive Type)

```
Inductive nat : Set ≜
  | O : nat
  | S : nat → nat.

nat_ind: ∀(P : nat → Prop), P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
nat_rec: ∀(P : nat → Set), P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
nat_rect: ∀(P : nat → Type), P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Examples (Logical Connectives as Inductive Types)

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

Examples (Logical Connectives as Inductive Types)

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

True_ind: ∀P : Prop, P → True → P
```

Examples (Logical Connectives as Inductive Types)

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

True_ind: ∀P : Prop, P → True → P

Inductive False: Prop ≜.
```

Examples (Logical Connectives as Inductive Types)

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

True_ind: ∀P : Prop, P → True → P

Inductive False: Prop ≜.

False_ind : ∀ P: Prop, False → P
```

Examples (Logical Connectives as Inductive Types)

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

True_ind: ∀P : Prop, P → True → P

Inductive False: Prop ≜.

False_ind : ∀ P: Prop, False → P

Inductive and (A B : Prop) : Prop ≜
  | conj : A → B → A ∧ B.
```

Examples (Logical Connectives as Inductive Types)

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

True_ind: ∀P : Prop, P → True → P

Inductive False: Prop ≜.

False_ind : ∀ P: Prop, False → P

Inductive and (A B : Prop) : Prop ≜
  | conj : A → B → A ∧ B.

and_ind: ∀A B P : Prop, (A → B → P) → A ∧ B → P
```

Examples (Logical Connectives as Inductive Types)

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

True_ind: ∀P : Prop, P → True → P

Inductive False: Prop ≜.

False_ind : ∀ P: Prop, False → P

Inductive and (A B : Prop) : Prop ≜
  | conj : A → B → A ∧ B.

and_ind: ∀A B P : Prop, (A → B → P) → A ∧ B → P

Inductive or (A B : Prop) : Prop ≜
  | or_introl : A → A ∨ B
  | or_intror : B → A ∨ B.
```

Examples (Logical Connectives as Inductive Types)

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

True_ind: ∀P : Prop, P → True → P

Inductive False: Prop ≜.

False_ind : ∀ P: Prop, False → P

Inductive and (A B : Prop) : Prop ≜
  | conj : A → B → A ∧ B.

and_ind: ∀A B P : Prop, (A → B → P) → A ∧ B → P

Inductive or (A B : Prop) : Prop ≜
  | or_introl : A → A ∨ B
  | or_intror : B → A ∨ B.

or_ind: ∀A B P : Prop, (A → P) → (B → P) → A ∨ B → P
```

Remark

- inductive type constructors are introduction rules

Remark

- inductive type constructors are introduction rules
- induction principles (_ind) are elimination rules

Remark

- inductive type constructors are introduction rules
- induction principles (_ind) are elimination rules
- how to prove

```
∀ (A B: Prop), A ∨ B → B ∨ A.
```

then?

Remark

- inductive type constructors are introduction rules
- induction principles (_ind) are elimination rules
- how to prove

```
∀ (A B: Prop), A ∨ B → B ∨ A.
```

then? (coming soon!)

Example (Equality as an Inductive Type)

- no primitive notion named equality

Example (Equality as an Inductive Type)

- no primitive notion named equality
- inductively defined propositional (or Leibniz) equality:

```
Inductive eq (A : Type) (x : A) : A → Prop ≜
  | eq_refl : x = x

eq_ind: ∀(A : Type) (x : A) (P : A → Prop), P x → (∀ y : A, x = y → P y)
```

Example (Equality as an Inductive Type)

- no primitive notion named equality
- inductively defined propositional (or Leibniz) equality:

```
Inductive eq (A : Type) (x : A) : A → Prop ≜
  | eq_refl : x = x

eq_ind: ∀(A : Type) (x : A) (P : A → Prop), P x → (∀ y : A, x = y → P y)
```

- rewriting relies on the substitution principle eq_ind

Example (Equality as an Inductive Type)

- no primitive notion named equality
- inductively defined propositional (or Leibniz) equality:

```
Inductive eq (A : Type) (x : A) : A → Prop ≜
  | eq_refl : x = x

eq_ind: ∀(A : Type) (x : A) (P : A → Prop), P x → (∀ y : A, x = y → P y)
```

- rewriting relies on the substitution principle eq_ind
- no “extensionality property”

```
∀ (A B: Set) (f g: A → B) (x: A), f x = g x → f = g
```

Example (Equality as an Inductive Type)

- no primitive notion named equality
- inductively defined propositional (or Leibniz) equality:

```
Inductive eq (A : Type) (x : A) : A → Prop ≜
  | eq_refl : x = x

eq_ind: ∀(A : Type) (x : A) (P : A → Prop), P x → (∀ y : A, x = y → P y)
```

- rewriting relies on the substitution principle eq_ind
- no “extensionality property”

```
∀ (A B: Set) (f g: A → B) (x: A), f x = g x → f = g
```

- terms can also be definitionaly equal (next slide, ι-reduction)

Coq Features (Recursive Definitions/Functions)

- enables direct encoding of total recursive functions

Coq Features (Recursive Definitions/Functions)

- enables direct encoding of total recursive functions
- provides pattern matching

```
Fixpoint add (n m: nat): nat ≜
  match n with
    | O   ⇒ m
    | S k ⇒ S (add k m)
  end.
```

Coq Features (Recursive Definitions/Functions)

- enables direct encoding of total recursive functions
- provides pattern matching

```
Fixpoint add (n m: nat): nat ≜
  match n with
    | O   ⇒ m
    | S k ⇒ S (add k m)
  end.
```

- embodies "definitional equality" though computational behavior (ι-reduction) of (recursive) functions

$$\texttt{add}\ \ \texttt{O}\ \texttt{m} \xrightarrow{\iota} \texttt{m}$$

Coq Features (Recursive Definitions/Functions)

- enables direct encoding of total recursive functions
- provides pattern matching

```
Fixpoint add (n m: nat): nat ≜
  match n with
    | O   ⇒ m
    | S k ⇒ S (add k m)
  end.
```

- embodies “definitional equality” though computational behavior (ι-reduction) of (recursive) functions

$$\texttt{add}\ \ \texttt{O}\ \texttt{m} \xrightarrow{\iota} \texttt{m} \qquad \texttt{add}\ \ (\texttt{S}\ \ \texttt{p})\ \ \texttt{m} \xrightarrow{\iota} \texttt{S}\,(\texttt{add}\ \ \texttt{p}\ \ \texttt{m})$$

Coq Features (Recursive Definitions/Functions)

- enables direct encoding of total recursive functions
- provides pattern matching

```
Fixpoint add (n m: nat): nat ≜
  match n with
    | O   ⇒ m
    | S k ⇒ S (add k m)
  end.
```

- embodies "definitional equality" though computational behavior (ι-reduction) of (recursive) functions

$$\texttt{add}\ \ \texttt{O}\ \ \texttt{m} \xrightarrow{\iota} \texttt{m} \qquad\qquad \texttt{add}\ \ (\texttt{S}\ \ \texttt{p})\ \ \texttt{m} \xrightarrow{\iota} \texttt{S}\,(\texttt{add}\ \ \texttt{p}\ \ \texttt{m})$$

- runs termination checker over every single recursive definition

Remark (restricted form of general recursion)

- to convince Coq that your function terminates, you could embark on:
  - Fixpoint: termination measure “structurally decreasing arguments”

Remark (restricted form of general recursion)

- to convince Coq that your function terminates, you could embark on:
  - Fixpoint: termination measure “structurally decreasing arguments”
  - Fix: termination measure “well founded relations”

Remark (restricted form of general recursion)

- to convince Coq that your function terminates, you could embark on:
  - Fixpoint: termination measure “structurally decreasing arguments”
  - Fix: termination measure “well founded relations”
  - Program Fixpoint: takes a measure as an argument and generates a proof obligation that the measure decreases in each recursive call

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors
`destruct`	generates a sub-goal for every constructor of an inductive type skipping potential induction hypotheses

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors
`destruct`	generates a sub-goal for every constructor of an inductive type skipping potential induction hypotheses
`specialize`	instantiates universally quantified hypotheses by concrete terms

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors
`destruct`	generates a sub-goal for every constructor of an inductive type skipping potential induction hypotheses
`specialize`	instantiates universally quantified hypotheses by concrete terms
`simpl`	performs evaluation and simplifies the goal or hypotheses in the context, if applicable

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors
`destruct`	generates a sub-goal for every constructor of an inductive type skipping potential induction hypotheses
`specialize`	instantiates universally quantified hypotheses by concrete terms
`simpl`	performs evaluation and simplifies the goal or hypotheses in the context, if applicable
`rewrite`	rewrites a goal using an equality

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors
`destruct`	generates a sub-goal for every constructor of an inductive type skipping potential induction hypotheses
`specialize`	instantiates universally quantified hypotheses by concrete terms
`simpl`	performs evaluation and simplifies the goal or hypotheses in the context, if applicable
`rewrite`	rewrites a goal using an equality
`reflexivity`	applies reflexivity property for equality

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors
`destruct`	generates a sub-goal for every constructor of an inductive type skipping potential induction hypotheses
`specialize`	instantiates universally quantified hypotheses by concrete terms
`simpl`	performs evaluation and simplifies the goal or hypotheses in the context, if applicable
`rewrite`	rewrites a goal using an equality
`reflexivity`	applies reflexivity property for equality
`symmetry`	applies symmetry property for equality

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors
`destruct`	generates a sub-goal for every constructor of an inductive type skipping potential induction hypotheses
`specialize`	instantiates universally quantified hypotheses by concrete terms
`simpl`	performs evaluation and simplifies the goal or hypotheses in the context, if applicable
`rewrite`	rewrites a goal using an equality
`reflexivity`	applies reflexivity property for equality
`symmetry`	applies symmetry property for equality
`transitivity`	applies transitivity property for equality

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors
`destruct`	generates a sub-goal for every constructor of an inductive type skipping potential induction hypotheses
`specialize`	instantiates universally quantified hypotheses by concrete terms
`simpl`	performs evaluation and simplifies the goal or hypotheses in the context, if applicable
`rewrite`	rewrites a goal using an equality
`reflexivity`	applies reflexivity property for equality
`symmetry`	applies symmetry property for equality
`transitivity`	applies transitivity property for equality
`assumption`	match conclusion with an hypothesis

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors
`destruct`	generates a sub-goal for every constructor of an inductive type skipping potential induction hypotheses
`specialize`	instantiates universally quantified hypotheses by concrete terms
`simpl`	performs evaluation and simplifies the goal or hypotheses in the context, if applicable
`rewrite`	rewrites a goal using an equality
`reflexivity`	applies reflexivity property for equality
`symmetry`	applies symmetry property for equality
`transitivity`	applies transitivity property for equality
`assumption`	match conclusion with an hypothesis
`exact`	gives directly the exact proof term of the goal

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors
`destruct`	generates a sub-goal for every constructor of an inductive type skipping potential induction hypotheses
`specialize`	instantiates universally quantified hypotheses by concrete terms
`simpl`	performs evaluation and simplifies the goal or hypotheses in the context, if applicable
`rewrite`	rewrites a goal using an equality
`reflexivity`	applies reflexivity property for equality
`symmetry`	applies symmetry property for equality
`transitivity`	applies transitivity property for equality
`assumption`	match conclusion with an hypothesis
`exact`	gives directly the exact proof term of the goal
`left` / `right`	replaces a goal consisting of a disjunction $P \vee Q$ with just P or Q

Coq Features (Tactics)

`intro` / `intros`	introduces ∀ quantified variables and premises of implications into the context
`apply`	employs implications to transform goals and hypotheses
`induction`	performs induction on a given identifier, and generates a sub-goal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors
`destruct`	generates a sub-goal for every constructor of an inductive type skipping potential induction hypotheses
`specialize`	instantiates universally quantified hypotheses by concrete terms
`simpl`	performs evaluation and simplifies the goal or hypotheses in the context, if applicable
`rewrite`	rewrites a goal using an equality
`reflexivity`	applies reflexivity property for equality
`symmetry`	applies symmetry property for equality
`transitivity`	applies transitivity property for equality
`assumption`	match conclusion with an hypothesis
`exact`	gives directly the exact proof term of the goal
`left` / `right`	replaces a goal consisting of a disjunction $P \vee Q$ with just P or Q
`split`	replaces a goal consisting of a conjunction $P \wedge Q$ with two sub-goals P and Q

Coq Features (Proofs: Basic Logical Reasoning)

```
Lemma ex1_v1: ∀ (A B: Prop), A ∨ B → B ∨ A.
Proof. intros A B H.
       destruct H as [ H | H ].
       - apply or_intror.
         exact H.
       - apply or_introl.
         exact H.
Qed.
```

Coq Features (Proofs: Basic Logical Reasoning)

```
Lemma ex1_v1: ∀ (A B: Prop), A ∨ B → B ∨ A.
Proof. intros A B H.
       destruct H as [ H | H ].
       - apply or_intror.
         exact H.
       - apply or_introl.
         exact H.
Qed.
```

```
Lemma ex1_v2: ∀ (A B: Prop), A ∨ B → B ∨ A.
Proof. intros A B H.
       destruct H as [ H | H ].
       - right.
         exact H.
       - left.
         exact H.
Qed.
```

Coq Features (Proofs: Basic Logical Reasoning)

```
Lemma ex1_v1: ∀ (A B: Prop), A ∨ B → B ∨ A.
Proof. intros A B H.
       destruct H as [ H | H ].
       - apply or_intror.
         exact H.
       - apply or_introl.
         exact H.
Qed.
```

```
Lemma ex1_v2: ∀ (A B: Prop), A ∨ B → B ∨ A.
Proof. intros A B H.
       destruct H as [ H | H ].
       - right.
         exact H.
       - left.
         exact H.
Qed.
```

```
Lemma ex2_v1: ∀ (A B: Prop), A ∧ B → B ∨ A.
Proof. intros A B H.
       destruct H as (H1, H2).
       left.
       exact H2.
Qed.
```

Coq Features (Proofs: Basic Logical Reasoning)

```
Lemma ex1_v1: ∀ (A B: Prop), A ∨ B → B ∨ A.
Proof. intros A B H.
       destruct H as [ H | H ].
       - apply or_intror.
         exact H.
       - apply or_introl.
         exact H.
Qed.
```

```
Lemma ex1_v2: ∀ (A B: Prop), A ∨ B → B ∨ A.
Proof. intros A B H.
       destruct H as [ H | H ].
       - right.
         exact H.
       - left.
         exact H.
Qed.
```

```
Lemma ex2_v1: ∀ (A B: Prop), A ∧ B → B ∨ A.
Proof. intros A B H.
       destruct H as (H1, H2).
       left.
       exact H2.
Qed.
```

```
Lemma ex2_v2: ∀ (A B: Prop), A ∧ B → B ∨ A.
Proof. intros A B H.
       destruct H as (H1, H2).
       right.
       exact H1.
Qed.
```

Coq Features (Proofs: Basic Logical Reasoning (cont'd))

```
Theorem ex3: ∀ (X: Type) (P: X → Prop),
  ~ (∃ (x: X), P x) → (∀ (x: X), ~ P x).
Proof. intros X P H x.
       unfold not.
       unfold not in H.
       intro px.
       apply H.
       ∃ x.
       exact px.
Qed.
```

Coq Features (Proofs: Basic Logical Reasoning (cont'd))

```
Theorem ex3: ∀ (X: Type) (P: X → Prop),
  ~ (∃ (x: X), P x) → (∀ (x: X), ~ P x).
Proof. intros X P H x.
       unfold not.
       unfold not in H.
       intro px.
       apply H.
       ∃ x.
       exact px.
Qed.
```

```
Theorem ex4: ∀ (X: Type) (P: X → Prop),
  (∀ (x: X), ~ P x) → ~ (∃ (x: X), P x).
Proof. intros X P H.
       unfold not.
       intro He.
       unfold not in H.
       destruct He as (x, He).
       specialize (H x He).
       exact H.
Qed.
```

Coq Features (Proofs: Basic Classical Logical Reasoning)

```
Axiom LEM: ∀ P: Prop, P ∨ ~P.
```

Coq Features (Proofs: Basic Classical Logical Reasoning)

```
Axiom LEM: ∀ P: Prop, P ∨ ~P.
```

```
Theorem dne: ∀ P, ~~P → P.
Proof. intros P H.
       unfold not in H.
       specialize (LEM P); intro HL.
       destruct HL as [ HL | HL ].
       - exact HL.
       - unfold not in HL.
         specialize (H HL).
         contradiction.
Qed.
```

Coq Features (Proofs: Basic Classical Logical Reasoning)

```
Axiom LEM: ∀ P: Prop, P ∨ ~P.
```

```
Theorem dne: ∀ P, ~~P → P.
Proof. intros P H.
       unfold not in H.
       specialize (LEM P); intro HL.
       destruct HL as [ HL | HL ].
       - exact HL.
       - unfold not in HL.
         specialize (H HL).
         contradiction.
Qed.
```

```
Theorem ex5: ∀ (X: Type) (P: X → Prop),
  ~ (∀ (x: X), ~ P x) → (∃ (x: X), P x).
Proof. intros X P H.
       specialize (LEM (∃ (x: X), P x)); intros HL.
       destruct HL as [ HL | HL ].
       - exact HL.
       - unfold not in *.
         destruct H.
         intros x px.
         apply HL.
         ∃ x.
         exact px.
Qed.
```

Coq Features (Proofs: Basic Equational Reasoning)

```
Lemma eq_trans_v1:
 ∀ (A: Type) (a b c: A), a = b → b = c → a = c.
Proof. intros A a b c Ha Hb.
       specialize (@eq_ind A b (fun x ⇒ a = x) Ha c Hb); intro H.
       simpl in H.
       exact H.
Qed.
```

Coq Features (Proofs: Basic Equational Reasoning)

```
Lemma eq_trans_v1:
 ∀ (A: Type) (a b c: A), a = b → b = c → a = c.
Proof. intros A a b c Ha Hb.
       specialize (@eq_ind A b (fun x ⇒ a = x) Ha c Hb); intro H.
       simpl in H.
       exact H.
Qed.
```

```
Lemma eq_trans_v2:
 ∀ (A: Type) (a b c: A), a = b → b = c → a = c.
Proof. intros A a b c Ha Hb.
       induction Ha.
       exact Hb.
Qed.
```

Coq Features (Proofs: Basic Equational Reasoning)

```
Lemma eq_trans_v1:
 ∀ (A: Type) (a b c: A), a = b → b = c → a = c.
Proof. intros A a b c Ha Hb.
       specialize (@eq_ind A b (fun x ⇒ a = x) Ha c Hb); intro H.
       simpl in H.
       exact H.
Qed.
```

```
Lemma eq_trans_v2:
 ∀ (A: Type) (a b c: A), a = b → b = c → a = c.
Proof. intros A a b c Ha Hb.
       induction Ha.
       exact Hb.
Qed.
```

```
Lemma eq_trans_v3:
 ∀ (A: Type) (a b c: A), a = b → b = c → a = c.
Proof. intros A a b c Ha Hb.
       rewrite Ha.
       exact Hb.
Qed.
```

Coq Features (Proofs: Basic Equational Reasoning (cont'd))

```
Lemma add_comm: ∀ (a b: nat), a + b = b + a.
```

Thanks! & Questions?