Interactive Theorem Proving: An Intro to the Coq Proof Assistant

Presented by Lukas Convent and Prof. Dr. Martin Leucker as part of the *dependable software* course as taught at ISP in Lübeck in 2019.

Learning Goals

- *Programming*: Inductive Data Types and Recursive Functions
- *Specifying*: Encode Logical Formulas as Types
- *Proving*: Prove Logical Formulas about Programs

Outline

Introduction

Functional and Imperative Programs

Definition (Imperative Program)

An **imperative** program p describes a partial function on memory states: Given some initial state σ, the **execution** of p on σ either terminates with a final state σ' or it diverges. For example, the program $\texttt{x} := 2 * \texttt{x}$ maps state $\sigma = \{x \mapsto 1, \_ \mapsto 0\}$ to state $\sigma' = \{x \mapsto 2, \_ \mapsto 0\}$

Definition (Functional Program)

A **functional** program f describes a partial function on values: Given some input value v, the **reduction** of the expression $f(v)$ either terminates in a value v' or it diverges. For example, the program $f(x) := 2 * x$ with 1 given as a value, resulting in the expression $f(1)$, reduces to value 2

Program Verification

- We focus on verifying **functional programs**
- We do not limit ourselves though:
  - Imperative programs can be expressed as functional programs
  - The typical framework to prove properties about imperative programs is the Hoare calculus, which can be easily expressed in out framework
- **Our framework** is a functional language that allows to:
  - Write useful programs
  - Write specifications for these programs
  - Prove these specifications
- **Next:** Recap on what it means to prove a logical statement (such as a specification)

Start simple: Propositional Logic

- Syntax
  - Formulas $\varphi, \psi := p \in AP \mid \bot \mid \varphi \to \psi$
  - Atomic propositions AP
  - (further connectives $\neg, \wedge, \vee, \ldots$ can be used as notation)

Start simple: Propositional Logic

- Syntax
  - Formulas $\varphi, \psi := p \in AP \mid \bot \mid \varphi \to \psi$
  - Atomic propositions AP
  - (further connectives $\neg, \wedge, \vee, \ldots$ can be used as notation)
- Semantics
  - Truth domain $T := \{0, 1\}$
  - Interpretations $v \in AP \to T$
  - Evaluation function

$$\llbracket p \rrbracket_v := v(p)$$

$$\llbracket \bot \rrbracket_v := 0$$

$$\llbracket \varphi \to \psi \rrbracket_v := \begin{cases} 1 & \text{if } \llbracket \varphi \rrbracket_v = 0 \text{ or } \llbracket \psi \rrbracket_v = 1 \\ 0 & \text{otherwise} \end{cases}$$

  - Satisfaction $v \vDash \varphi \quad :\Leftrightarrow \quad \llbracket \varphi \rrbracket_v = 1$
    Validity $\vDash \varphi \quad :\Leftrightarrow \quad v \vDash \varphi$ for all v
  - φ is called a tautology if $\vDash \varphi$

Why *proving*?

- Goal: Given φ, does $\vDash \varphi$ hold?
- First approach: Evaluate and check $[\![\varphi]\!]_v = 1$ for all v

Why *proving*?

- Goal: Given φ, does $\models \varphi$ hold?
- First approach: Evaluate and check $[\![\varphi]\!]_v = 1$ for all v
- Problem:
  - for Propositional Logic: Possible, but there are 2^n interpretations (where n is the number of vars in φ)
  - for First-Order Logic: Impossible, there may be **infinitely** many interpretations

Why *proving*?

- Goal: Given φ, does $\models \varphi$ hold?
- First approach: Evaluate and check $[\![\varphi]\!]_v = 1$ for all v
- Problem:
  - for Propositional Logic: Possible, but there are 2^n interpretations (where n is the number of vars in φ)
  - for First-Order Logic: Impossible, there may be **infinitely** many interpretations
- Help:
  - Use a proof system
  - Idea: Construct a **finite** proof that φ holds
  - Proof system must be *sound*:
    If φ can be proven ($\vdash \varphi$), then φ is valid ($\models \varphi$)
  - Proof system *may* be *complete*:
    If φ is valid ($\models \varphi$), then φ can be proven ($\vdash \varphi$)

Proof System for Propositional Logic

- Natural deduction via entailment relation $\Gamma \vdash \varphi$
  - Γ is a finite set of formulas $\psi_1, \psi_2, ..., \psi_n$
  - "from Γ, one can **deduce** φ"

Proof System for Propositional Logic

- Natural deduction via entailment relation $\Gamma \vdash \varphi$
  - Γ is a finite set of formulas $\psi_1, \psi_2, ..., \psi_n$
  - "from Γ, one can **deduce** φ"
- Defined by inference rules:

$$\varphi \in \Gamma \ \frac{}{\Gamma \vdash \varphi}\ \textsc{Assump} \qquad \frac{\Gamma, (\varphi \to \bot) \vdash \bot}{\Gamma \vdash \varphi}\ \textsc{DoubleNeg}$$

$$\frac{\Gamma, \psi \vdash \varphi}{\Gamma \vdash \psi \to \varphi}\ \textsc{ImpIntro} \qquad \frac{\Gamma \vdash \psi \to \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi}\ \textsc{ImpElim}$$

Proof System for Propositional Logic

- Natural deduction via entailment relation $\Gamma \vdash \varphi$
  - Γ is a finite set of formulas $\psi_1, \psi_2, ..., \psi_n$
  - "from Γ, one can **deduce** φ"
- Defined by inference rules:

$$\varphi \in \Gamma \ \frac{}{\Gamma \vdash \varphi} \ \textsc{Assump} \qquad \frac{\Gamma, (\varphi \to \bot) \vdash \bot}{\Gamma \vdash \varphi} \ \textsc{DoubleNeg}$$

$$\frac{\Gamma, \psi \vdash \varphi}{\Gamma \vdash \psi \to \varphi} \ \textsc{ImpIntro} \qquad \frac{\Gamma \vdash \psi \to \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi} \ \textsc{ImpElim}$$

- This system is *sound*:
  If φ can be proven ($\vdash \varphi$), then φ is valid ($\vDash \varphi$)
- This system is *complete*:
  If φ is valid ($\vDash \varphi$), then φ can be proven ($\vdash \varphi$)

Proof Trees

- In order to check validity of $\varphi := p \to (q \to p)$...

Proof Trees

- In order to check validity of $\varphi := p \to (q \to p)$...
- ...prove $\vdash \varphi$, as witnessed by the following proof tree

$$
\dfrac{\dfrac{p \in \{p, q\} \quad \overline{p, q \vdash p}\ \textsc{Assump}}{p \vdash q \to p}\ \textsc{ImpIntro}}{\vdash p \to (q \to p)}\ \textsc{ImpIntro}
$$

Proof Trees

- In order to check validity of $\varphi := p \to (q \to p)$...
- ...prove $\vdash \varphi$, as witnessed by the following proof tree

$$\dfrac{\dfrac{p \in \{p, q\} \quad \dfrac{}{p, q \vdash p}\,\textsc{Assump}}{p \vdash q \to p}\,\textsc{ImpIntro}}{\vdash p \to (q \to p)}\,\textsc{ImpIntro}$$

- By soundness of $\vdash$, φ is valid

Proof Trees in Type Systems, 1

- In a typed programming language, we want to check that term t is of type T

Proof Trees in Type Systems, 1

- In a typed programming language, we want to check that term t is of type T
- Terms
  1. $3 + 2$
  2. $\textbf{if } true \textbf{ then } true \textbf{ else } false$
  3. $\lambda n.n$
  4. $\lambda n.\ (\lambda b.\ \textbf{if } b \textbf{ then } n \textbf{ else } n + n)$
  5. $\lambda n.\ \textbf{let } sq = n * n \textbf{ in } sq * sq$

Proof Trees in Type Systems, 1

- In a typed programming language, we want to check that term t is of type T
- Terms
  1. $3 + 2$
  2. $\textbf{if } \mathit{true} \textbf{ then } \mathit{true} \textbf{ else } \mathit{false}$
  3. $\lambda n.n$
  4. $\lambda n.\ (\lambda b.\ \textbf{if } b \textbf{ then } n \textbf{ else } n + n)$
  5. $\lambda n.\ \textbf{let } sq = n * n \textbf{ in } sq * sq$
- Types
  1. Int
  2. Bool
  3. $\mathit{Int} \to \mathit{Int}$
  4. $\mathit{Int} \to (\mathit{Bool} \to \mathit{Int})$
  5. $\mathit{Int} \to \mathit{Int}$

Proof Trees in Type Systems, 1

- In a typed programming language, we want to check that term t is of type T
- Terms
  1. $3 + 2$
  2. $\textbf{if } \mathit{true} \textbf{ then } \mathit{true} \textbf{ else } \mathit{false}$
  3. $\lambda n.n$
  4. $\lambda n.\ (\lambda b.\ \textbf{if } b \textbf{ then } n \textbf{ else } n + n)$
  5. $\lambda n.\ \textbf{let } sq = n * n \textbf{ in } sq * sq$
- Types
  1. Int
  2. Bool
  3. $\mathit{Int} \to \mathit{Int}$
  4. $\mathit{Int} \to (\mathit{Bool} \to \mathit{Int})$
  5. $\mathit{Int} \to \mathit{Int}$
- We write $\vdash t : T$ if term t has type T

Proof Trees in Type Systems, 1

- In a typed programming language, we want to check that term t is of type T
- Terms
  1. $3 + 2$
  2. $\textbf{if } \mathit{true} \textbf{ then } \mathit{true} \textbf{ else } \mathit{false}$
  3. $\lambda n.n$
  4. $\lambda n.\ (\lambda b.\ \textbf{if } b \textbf{ then } n \textbf{ else } n + n)$
  5. $\lambda n.\ \textbf{let } sq = n * n \textbf{ in } sq * sq$
- Types
  1. Int
  2. Bool
  3. $\mathit{Int} \to \mathit{Int}$
  4. $\mathit{Int} \to (\mathit{Bool} \to \mathit{Int})$
  5. $\mathit{Int} \to \mathit{Int}$
- We write $\vdash t : T$ if term t has type T
- *We can also talk of "soundness" here*: A type system is sound if $\vdash t : T$ implies that t won't "crash" on execution. E.g., `true` $+$ 4 crashes

Proof Trees in Type Systems, 2

- To check whether $t := \lambda x.(\lambda y.x)$ is of type $T := Int \to (Bool \to Int)$, we check whether there is a proof tree for $\vdash t : T$

Proof Trees in Type Systems, 2

- To check whether $t := \lambda x.(\lambda y.x)$ is of type $T := Int \to (Bool \to Int)$, we check whether there is a proof tree for $\vdash t : T$
- For our example, there is:

$$\dfrac{\dfrac{(x : Int) \in \{x : Int, y : Bool\} \quad \dfrac{}{x : Int, y : Bool \vdash x : Int}\ \textsc{Env}}{x : Int \vdash \lambda y.x : Bool \to Int}\ \textsc{Abs}}{\vdash \lambda x.(\lambda y.x) : Int \to (Bool \to Int)}\ \textsc{Abs}$$

Proof Trees in Type Systems, 2

- To check whether $t := \lambda x.(\lambda y.x)$ is of type $T := Int \to (Bool \to Int)$, we check whether there is a proof tree for $\vdash t : T$
- For our example, there is:

$$\dfrac{(x : Int) \in \{x : Int, y : Bool\} \quad \dfrac{}{x : Int, y : Bool \vdash x : Int}\,\textsc{Env}}{\dfrac{x : Int \vdash \lambda y.x : Bool \to Int}{\vdash \lambda x.(\lambda y.x) : Int \to (Bool \to Int)}\,\textsc{Abs}}\,\textsc{Abs}$$

- In a type system, the inference rules are designed s.t. for every pair $\vdash t : T$, there exists **at most one** proof tree

Proof Trees in Type Systems, 2

- To check whether $t := \lambda x.(\lambda y.x)$ is of type $T := \mathit{Int} \to (\mathit{Bool} \to \mathit{Int})$, we check whether there is a proof tree for $\vdash t : T$
- For our example, there is:

$$\dfrac{\dfrac{(x : \mathit{Int}) \in \{x : \mathit{Int}, y : \mathit{Bool}\} \quad \overline{x : \mathit{Int}, y : \mathit{Bool} \vdash x : \mathit{Int}}\ \textsc{Env}}{x : \mathit{Int} \vdash \lambda y.x : \mathit{Bool} \to \mathit{Int}}\ \textsc{Abs}}{\vdash \lambda x.(\lambda y.x) : \mathit{Int} \to (\mathit{Bool} \to \mathit{Int})}\ \textsc{Abs}$$

- In a type system, the inference rules are designed s.t. for every pair $\vdash t : T$, there exists **at most one** proof tree
- t itself witnesses its own proof tree of $\vdash t : T$

Proof Trees in Type Systems, 2

- To check whether $t := \lambda x.(\lambda y.x)$ is of type $T := Int \to (Bool \to Int)$, we check whether there is a proof tree for $\vdash t : T$
- For our example, there is:

$$\dfrac{\dfrac{(x : Int) \in \{x : Int, y : Bool\} \quad \dfrac{}{x : Int, y : Bool \vdash x : Int}\,\textsc{Env}}{x : Int \vdash \lambda y.x : Bool \to Int}\,\textsc{Abs}}{\vdash \lambda x.(\lambda y.x) : Int \to (Bool \to Int)}\,\textsc{Abs}$$

- In a type system, the inference rules are designed s.t. for every pair $\vdash t : T$, there exists **at most one** proof tree
- t itself witnesses its own proof tree of $\vdash t : T$
- *Intuition*: A term itself represents a syntax tree. Put this tree upside down. Traverse the tree, thereby annotating types according to the inference rules. If this works out, you have **the** proof tree. Otherwise, there is none.

Preview: Type System as a Proof System

- You noticed the similarity between the two proof trees?

$$\dfrac{\dfrac{p \in \{p, q\}}{p, q \vdash p}\ \textsc{Assump}}{\dfrac{p \vdash q \to p}{\vdash p \to (q \to p)}\ \textsc{ImpIntro}}\ \textsc{ImpIntro}$$

$$\dfrac{\dfrac{\dfrac{(x : \mathit{Int}) \in \{x : \mathit{Int}, y : \mathit{Bool}\}}{x : \mathit{Int}, y : \mathit{Bool} \vdash x : \mathit{Int}}\ \textsc{Env}}{x : \mathit{Int} \vdash \lambda y.x : \mathit{Bool} \to \mathit{Int}}\ \textsc{Abs}}{\vdash \lambda x.(\lambda y.x) : \mathit{Int} \to (\mathit{Bool} \to \mathit{Int})}\ \textsc{Abs}$$

- Is it be possible to encode a proof tree for a logic as a proof tree for a type system?

Preview: Type System as a Proof System

- You noticed the similarity between the two proof trees?

$$
\dfrac{\dfrac{p \in \{p, q\} \quad \overline{p, q \vdash p}\ \textsc{Assump}}{p \vdash q \to p}\ \textsc{ImpIntro}}{\vdash p \to (q \to p)}\ \textsc{ImpIntro}
$$

$$
\dfrac{\dfrac{(x : \mathit{Int}) \in \{x : \mathit{Int}, y : \mathit{Bool}\} \quad \overline{x : \mathit{Int}, y : \mathit{Bool} \vdash x : \mathit{Int}}\ \textsc{Env}}{x : \mathit{Int} \vdash \lambda y.x : \mathit{Bool} \to \mathit{Int}}\ \textsc{Abs}}{\vdash \lambda x.(\lambda y.x) : \mathit{Int} \to (\mathit{Bool} \to \mathit{Int})}\ \textsc{Abs}
$$

- Is it be possible to encode a proof tree for a logic as a proof tree for a type system?
- It is possible. It has been discovered in 1980 by Howard (Curry-Howard-Correspondence)

Preview: Type System as a Proof System

- You noticed the similarity between the two proof trees?

$$\dfrac{\dfrac{p \in \{p, q\} \quad \overline{p, q \vdash p}\ \textsc{Assump}}{p \vdash q \to p}\ \textsc{ImpIntro}}{\vdash p \to (q \to p)}\ \textsc{ImpIntro}$$

$$\dfrac{\dfrac{(x : Int) \in \{x : Int, y : Bool\} \quad \overline{x : Int, y : Bool \vdash x : Int}\ \textsc{Env}}{x : Int \vdash \lambda y.x : Bool \to Int}\ \textsc{Abs}}{\vdash \lambda x.(\lambda y.x) : Int \to (Bool \to Int)}\ \textsc{Abs}$$

- Is it be possible to encode a proof tree for a logic as a proof tree for a type system?
- It is possible. It has been discovered in 1980 by Howard (Curry-Howard-Correspondence)
- What do we need?
  1. Goal: Find a way of *proving* a specification φ
  2. We encode φ as a type T
  3. We find a term t that is well-typed, i.e. $\vdash t : T$
  4. But this means that t witnesses a proof tree for T
  5. Thus we interpret t as a proof of T and therefore of φ!

Interactive Theorem Provers

- Proofs are **manually written**, potentially with some automatic proof-search aid
- Proofs are **completely formal**
- Proofs can be **automatically checked**
- You have to trust in the soundness of the proof checker
  - Trust is usually established by providing a minimal base of the proof checker

Interactive Theorem Provers

- Proofs are **manually written**, potentially with some automatic proof-search aid
- Proofs are **completely formal**
- Proofs can be **automatically checked**
- You have to trust in the soundness of the proof checker
  - Trust is usually established by providing a minimal base of the proof checker
- Examples: Coq, Isabelle, Agda
- May be based on type theory, but not necessarily
- Applications
  1. Formalized Mathematics, e.g. Four-color theorem in 1976
  2. Correctness Properties
     - Certified C compiler CompCert, started in 2005
     - Soundness of type systems
     - Correctness of protocols
     - Further theorems about formalisms
  3. Generally: Verification where the system model or the property is "too complex" for automatic methods

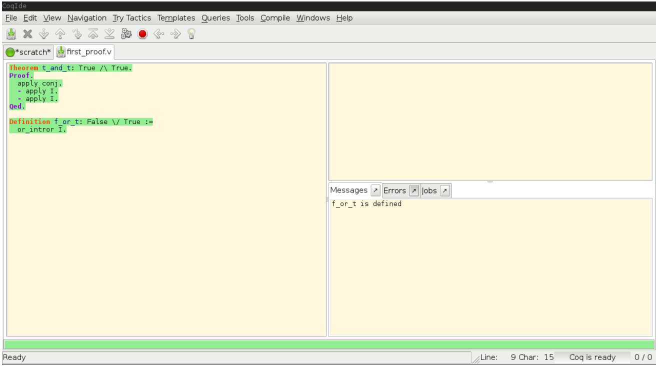
Coq Programming

Coq

- Coq is an interactive theorem prover
- Main idea: Propositions as Types, Proofs as Terms (Curry-Howard-Correspondence)
- One can define
  - **Types (Propositions)**
  - Well-typed **Terms (Proofs)**
- The underlying language Gallina
  - is a dependently-typed functional programming language
  - implements the Calculus of Inductive Constructions
  - is not Turing-complete (every function is total)

Getting started with Coq

1. Installation
   - Win/Mac: Download from `https://coq.inria.fr/`
   - Linux: We recommend installation via OPAM
     `https://coq.inria.fr/opam/www/using.html`
2. IDE
   - Recommendation: Coq IDE, shipped with Coq (see screenshot)
   - Popular plugin for Emacs: Proof General



Coq Programming (Inductive Data Types)

- An inductive data type definition introduces a **new type** and **new well-typed terms**

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

Inductive nat : Type :=
| O : nat
| S : nat → nat.
```

- An inductive data type definition introduces a **new type** and **new well-typed terms**

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

Inductive nat : Type :=
| O : nat
| S : nat → nat.
```

- `bool`, `nat` are types
- `true`, `false`, `O`, `S` are value constructors

Coq Programming (Definitions)

- A **Definition** gives a name to a term

```
Definition two:   nat ≔ S(S O).
Definition three: nat ≔ S(S(S O)).
```

Coq Programming (Definitions)

- A **`Definition`** gives a name to a term

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

- Definitions can be unfolded, which is a kind of *reduction*
- Two terms are *convertible* ($\equiv$) if they reduce to the same term
- E.g., `S two` and `three` are convertible

```
  S two
≡ S(S(S O))
≡ three
```

Coq Programming (Definitions)

- A **`Definition`** gives a name to a term

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

- Definitions can be unfolded, which is a kind of *reduction*
- Two terms are *convertible* ($\equiv$) if they reduce to the same term
- E.g., `S two` and `three` are convertible

```
  S two
≡ S(S(S O))
≡ three
```

- *Intuition:* Convertibility is “syntactic equality up-to certain manipulations”

Coq Programming (Functions, Pattern Matching)

- We can define functions that use pattern matching

```
Definition negb : bool → bool :=
  fun x ⇒ match x with
          | true  ⇒ false
          | false ⇒ true
          end.
```

Coq Programming (Functions, Pattern Matching)

- We can define functions that use pattern matching

```
Definition negb : bool → bool :=
  fun x ⇒ match x with
          | true  ⇒ false
          | false ⇒ true
          end.
```

- **fun** x ⇒ ... introduces a function (anonymous, “λ”)
- **match** ... **with** | ... **end** pattern-matches

Coq Programming (Functions, Pattern Matching)

- We can define functions that use pattern matching

```
Definition negb : bool → bool :=
  fun x ⇒ match x with
          | true  ⇒ false
          | false ⇒ true
          end.
```

- `fun x ⇒ ...` introduces a function (anonymous, "λ")
- `match ... with | ... end` pattern-matches
- Both constructs introduce a form of reduction and thus of convertibility

```
  negb true
≡ (fun x ⇒ ...) true
≡ match true with | true ⇒ false | ...
≡ false
```

Coq Programming (Short Notation for Functions)

- Recall our function

```
Definition negb : bool → bool :=
  fun x ⇒ match x with
          | true  ⇒ false
          | false ⇒ true
          end.
```

- We can use the following short notation

```
Definition negb (x: bool) : bool :=
  match x with
  | true  ⇒ false
  | false ⇒ true
  end.
```

Coq Programming (Type-Checking)

- In Coq, every term must be well-typed
- What does that mean?
  - We write $\Gamma \vdash t : T$ and call it a *(typing) judgement*
  - "Under context Γ, term t has type T"
  - Context Γ is a list of items $t : T$
  - E.g., we have $x : bool \vdash negb\ x : bool$

    ...but **not** $x : nat \vdash negb\ x : bool$

Coq Programming (Type-Checking)

- In Coq, every term must be well-typed
- What does that mean?
  - We write $\Gamma \vdash t : T$ and call it a *(typing) judgement*
  - "Under context Γ, term t has type T"
  - Context Γ is a list of items $t : T$
  - E.g., we have $x : bool \vdash negb\ x : bool$

    ...but **not** $x : nat \vdash negb\ x : bool$
- Coq can *try to find* a type T for Γ, t (a.k.a. **type inference**, generally undecidable)

Coq Programming (Type-Checking)

- In Coq, every term must be well-typed
- What does that mean?
  - We write $\Gamma \vdash t : T$ and call it a *(typing) judgement*
  - "Under context Γ, term t has type T"
  - Context Γ is a list of items $t : T$
  - E.g., we have $x : bool \vdash negb\ x : bool$

    ...but **not** $x : nat \vdash negb\ x : bool$
- Coq can *try to find* a type T for Γ, t (a.k.a. **type inference**, generally undecidable)
- Coq *decides* for a given judgement whether it holds (a.k.a. **type-checking**)

Coq Programming (Type-Checking, Reducing in Coq)

- Type-infer terms and compute (reduce) terms

```
Check (negb true).   ⇝ negb true: bool
Compute (negb true). ⇝ false: bool
```

  - Here, the context Γ is considered by Coq but not explicitly output

Coq Programming (Recursive Functions)

- We can define recursive functions

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

Coq Programming (Recursive Functions)

- We can define recursive functions

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

- Above was really a short notation for the following:

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

Coq Programming (Recursion must be structural)

```
Fixpoint plus (m n: nat) : nat :=
  match m with
  | O    ⇒ n
  | S m' ⇒ S (plus m' n)
  end.
```

- Recursive functions in Coq **always terminate** because only *structural recursion* is allowed

Coq Programming (Recursion must be structural)

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

- Recursive functions in Coq **always terminate** because only *structural recursion* is allowed
- Structural recursion means that recursion is only applied to sub-structures
- Here: `m'` is a sub-structure of `S m'`

Coq Programming (Recursion must be structural)

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

- Recursive functions in Coq **always terminate** because only *structural recursion* is allowed
- Structural recursion means that recursion is only applied to sub-structures
- Here: `m'` is a sub-structure of `S m'`
- Why this restriction? Remember: Proofs are programs, and non-terminating proofs must be avoided!
  (more later)

Coq Programming (Prelude and Notation)

- Standard data types, functions, notation are pre-defined via the Prelude[1]
- This allows us to write a term like `3 + 2` instead of `plus S(S(S O)) S(S O)`.
- We use the nice notation from now on wherever possible

[1] https://coq.inria.fr/library/Coq.Init.Prelude.html

Coq Programming (Polymorphic Data Types, 1)

- It is often useful to parameterize a data type to avoid multiple definitions such as `natList`, `boolList` etc.

```
Inductive list (X: Type) : Type ≔
| nil:  list X
| cons: X → list X → list X.
```

Coq Programming (Polymorphic Data Types, 1)

- It is often useful to parameterize a data type to avoid multiple definitions such as `natList`, `boolList` etc.

```
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

- We say that `list` is *polymorphic* in its *parameter* `X`

Coq Programming (Polymorphic Data Types, 1)

- It is often useful to parameterize a data type to avoid multiple definitions such as `natList`, `boolList` etc.

```
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

- We say that `list` is *polymorphic* in its *parameter* `X`
- We say that `list` is a *type constructor* (a function that constructs a type)

Coq Programming (Polymorphic Data Types, 1)

- It is often useful to parameterize a data type to avoid multiple definitions such as `natList`, `boolList` etc.

```
Inductive list (X: Type) : Type ≔
| nil:  list X
| cons: X → list X → list X.
```

- We say that `list` is *polymorphic* in its *parameter* `X`
- We say that `list` is a *type constructor* (a function that constructs a type)
- Applying this type constructor yields
  - `list nat:  Type`
  - `list bool: Type`
  - ...

Coq Programming (Polymorphic Data Types, 2)

In the parameterized definition

```
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

... the parameter X can be "multiplied-out" to ...

```
Inductive list : Type → Type :=
| nil:  ∀ (X: Type), list X
| cons: ∀ (X: Type), X → list X → list X.
```

Coq Programming (Polymorphic Data Types, 2)

In the parameterized definition

```
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

... the parameter X can be "multiplied-out" to ...

```
Inductive list : Type → Type :=
| nil:  ∀ (X: Type), list X
| cons: ∀ (X: Type), X → list X → list X.
```

- The following judgements are introduced
  - `list: Type → Type`
  - `nil: ∀(X: Type), list X`
  - `cons: ∀(X: Type), X → list X → list X`

Coq Programming (Polymorphic Data Types, 2)

In the parameterized definition

```
Inductive list (X: Type) : Type ≔
| nil:  list X
| cons: X → list X → list X.
```

... the parameter `X` can be "multiplied-out" to ...

```
Inductive list : Type → Type ≔
| nil:  ∀ (X: Type), list X
| cons: ∀ (X: Type), X → list X → list X.
```

- The following judgements are introduced
  - `list: Type → Type`
  - `nil: ∀(X: Type), list X`
  - `cons: ∀(X: Type), X → list X → list X`
- The definitions are isomorphic (modulo technicalities), but the parameterized definition emphasises that *the structure* of `list` terms is independ. of the choice of the *"type of content"* `X`

Coq Programming (Implicit Parameters, 1)

```
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

- Recall that this introduces the judgements

  ```
  nil:  ∀(X: Type), list X
  cons: ∀(X: Type), X → list X → list X
  ```

Coq Programming (Implicit Parameters, 1)

```
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

- Recall that this introduces the judgements
  `nil:  ∀(X: Type), list X`
  `cons: ∀(X: Type), X → list X → list X`
- ... so the value constructors must be instantiated, e.g.
  `cons nat 42 (nil nat): list nat`

Coq Programming (Implicit Parameters, 1)

```
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

- Recall that this introduces the judgements
  `nil:  ∀(X: Type), list X`
  `cons: ∀(X: Type), X → list X → list X`
- ... so the value constructors must be instantiated, e.g.
  `cons nat 42 (nil nat): list nat`
- 42 is a `nat`, so `X` *must* be instantiated by `nat`. Can we let Coq infer this and simply write
  `cons 42 nil: list nat`
  instead?

Coq Programming (Implicit Parameters, 1)

```
Inductive list (X: Type) : Type ≔
| nil:  list X
| cons: X → list X → list X.
```

- Recall that this introduces the judgements
  `nil:  ∀(X: Type), list X`
  `cons: ∀(X: Type), X → list X → list X`
- ... so the value constructors must be instantiated, e.g.
  `cons nat 42 (nil nat): list nat`
- 42 is a `nat`, so `X` *must* be instantiated by `nat`. Can we let Coq infer this and simply write
  `cons 42 nil: list nat`
  instead?
- Yes! (See next slide)

Coq Programming (Implicit Parameters, 2)

- Recall our example term:
  `cons nat 42 (nil nat) : list nat`

[2] use `Require Import Coq.Lists.List.`

Coq Programming (Implicit Parameters, 2)

- Recall our example term:
  `cons nat 42 (nil nat) : list nat`
- We can manually mark arguments as implicit or enable this by default via

```
Set Implicit Arguments.
Set Contextual Implicit.
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

[2] use `Require Import Coq.Lists.List.`

Coq Programming (Implicit Parameters, 2)

- Recall our example term:
  `cons nat 42 (nil nat) : list nat`
- We can manually mark arguments as implicit or enable this by default via

```
Set Implicit Arguments.
Set Contextual Implicit.
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

- `X` is *strictly* implicit for `cons` (alway inferrable)

[2] use `Require Import Coq.Lists.List.`

Coq Programming (Implicit Parameters, 2)

- Recall our example term:
  `cons nat 42 (nil nat) : list nat`
- We can manually mark arguments as implicit or enable this by default via

```
Set Implicit Arguments.
Set Contextual Implicit.
Inductive list (X: Type) : Type ≔
| nil:  list X
| cons: X → list X → list X.
```

- `X` is *strictly* implicit for `cons` (alway inferrable)
- `X` is *contextually* implicit for `nil` (sometimes inferrable)

[2] use `Require Import Coq.Lists.List.`

Coq Programming (Implicit Parameters, 2)

- Recall our example term:
  `cons nat 42 (nil nat) : list nat`
- We can manually mark arguments as implicit or enable this by default via

```
Set Implicit Arguments.
Set Contextual Implicit.
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

- `X` is *strictly* implicit for `cons` (alway inferrable)
- `X` is *contextually* implicit for `nil` (sometimes inferrable)
- This allows to write the example term as:
  `cons 42 nil : list nat`

[2] use `Require Import Coq.Lists.List.`

Coq Programming (Implicit Parameters, 2)

- Recall our example term:
  `cons nat 42 (nil nat) : list nat`
- We can manually mark arguments as implicit or enable this by default via

```
Set Implicit Arguments.
Set Contextual Implicit.
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

- `X` is *strictly* implicit for `cons` (alway inferrable)
- `X` is *contextually* implicit for `nil` (sometimes inferrable)
- This allows to write the example term as:
  `cons 42 nil : list nat`
- But we lack the context to infer `nil : list nat`

[2] use **Require Import** `Coq.Lists.List.`

Coq Programming (Implicit Parameters, 2)

- Recall our example term:
  `cons nat 42 (nil nat) : list nat`
- We can manually mark arguments as implicit or enable this by default via

```
Set Implicit Arguments.
Set Contextual Implicit.
Inductive list (X: Type) : Type :=
| nil:  list X
| cons: X → list X → list X.
```

- `X` is *strictly* implicit for `cons` (alway inferrable)
- `X` is *contextually* implicit for `nil` (sometimes inferrable)
- This allows to write the example term as:
  `cons 42 nil : list nat`
- But we lack the context to infer `nil : list nat`
- There is further notation[2]: `42::nil : list nat`

[2] use `Require Import Coq.Lists.List.`

Coq Programming (Implicit Parameters, 3)

- Why do we have to bother about **explicit** parameters in expressions?
- In standard programming languages, parameters in terms can be simply deduced by the type checker and are therefore **implicit**, i.e. they are omitted in terms
- Here however, parameters are more subtle and cannot always be deduced
- Take-away: If we are using the usual programming constructs, we just use the options

```
Set Implicit Arguments.
Set Contextual Implicit.
```

and don't have to care about the majority of parameters!

Propositions as Types

The Idea, 1

- We have seen a fairly standard functional programming language (with restricted recursion)

The Idea, 1

- We have seen a fairly standard functional programming language (with restricted recursion)
- But wasn't Coq about giving proofs of propositions about programs?

The Idea, 1

- We have seen a fairly standard functional programming language (with restricted recursion)
- But wasn't Coq about giving proofs of propositions about programs?
- **Roadmap**: We add very few features to the language and show how we can prove propositions **within** the language, as opposed to using some meta framework

The Idea, 1

- We have seen a fairly standard functional programming language (with restricted recursion)
- But wasn't Coq about giving proofs of propositions about programs?
- **Roadmap**: We add very few features to the language and show how we can prove propositions **within** the language, as opposed to using some meta framework
- We need to establish the following mechanisms:
  - A **proposition** (logical statement) is encoded as a **type**
  - A **proof** of a proposition is encoded as a **term** of that type

The Idea, 1

- We have seen a fairly standard functional programming language (with restricted recursion)
- But wasn't Coq about giving proofs of propositions about programs?
- **Roadmap**: We add very few features to the language and show how we can prove propositions **within** the language, as opposed to using some meta framework
- We need to establish the following mechanisms:
  - A **proposition** (logical statement) is encoded as a **type**
  - A **proof** of a proposition is encoded as a **term** of that type
- The idea of using propositions as types is also called *Curry-Howard-Correspondence*

The Idea, 2

Spoilers:

Implication $P \to Q$	$\hat{\simeq}$	Funct. type `P -> Q`
Conjunction $P \land Q$	$\hat{\simeq}$	Product type `P × Q`
Disjunction $P \lor Q$	$\hat{\simeq}$	Sum type `P + Q`
Top $\top$	$\hat{\simeq}$	Unit type `1`
Bottom $\bot$	$\hat{\simeq}$	Empty type `0`
Univ. Qu. $\forall(x : A), P(x)$	$\hat{\simeq}$	Π-types
Exis. Qu. $\exists(x : A), P(x)$	$\hat{\simeq}$	Σ-types
Modus Ponens: From $P \to Q$ and P one deduces Q	$\hat{\simeq}$	Function application: `f: P -> Q` and `p: P` gives `f p: Q`
There is a proof tree for P	$\hat{\simeq}$	There is a term `t` such that `t: P`

Top and Bottom

- Let's define the proposition $\top$ ("truth")
- There should be a proof for $\top$

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

Top and Bottom

- Let's define the proposition $\top$ ("truth")
- There should be a proof for $\top$

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

- Let's define the proposition $\bot$ ("falsity")
- There should be **no** proof for $\bot$

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

Top and Bottom

- Let's define the proposition $\top$ ("truth")
- There should be a proof for $\top$

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

- Let's define the proposition $\bot$ ("falsity")
- There should be **no** proof for $\bot$

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

- In Coq, there is a special type universe for propositions, called `Prop` (more about universes later)
- From now on, we write $\top$ for `True` and $\bot$ for `False`

Conjunction

- We now come to our first connective, conjunction

```
Inductive and (A B: Prop) : Prop :=
  conj : A → B → and A B.
```

Conjunction

- We now come to our first connective, conjunction

```
Inductive and (A B: Prop) : Prop :=
  conj : A → B → and A B.
```

- `and` is a type constructor (better: **`Prop`** constructor):
  - Given two **`Prop`**'s, it establishes a new **`Prop`**
  - The two **`Prop`**'s are parameters (implicit for `conj`)
  - There is notation `A ∧ B` for `and A B`

Conjunction

- We now come to our first connective, conjunction

```
Inductive and (A B: Prop) : Prop :=
  conj : A → B → and A B.
```

- `and` is a type constructor (better: **`Prop`** constructor):
  - Given two **`Prop`**'s, it establishes a new **`Prop`**
  - The two **`Prop`**'s are parameters (implicit for `conj`)
  - There is notation `A ∧ B` for `and A B`
- How do you prove `A ∧ B`?

Conjunction

- We now come to our first connective, conjunction

```
Inductive and (A B: Prop) : Prop :=
  conj : A → B → and A B.
```

- `and` is a type constructor (better: `Prop` constructor):
  - Given two `Prop`'s, it establishes a new `Prop`
  - The two `Prop`'s are parameters (implicit for `conj`)
  - There is notation `A ∧ B` for `and A B`
- How do you prove `A ∧ B`?
- Give proofs `a: A` and `b: B` and apply `conj` to them

Conjunction

- We now come to our first connective, conjunction

```
Inductive and (A B: Prop) : Prop :=
  conj : A → B → and A B.
```

- `and` is a type constructor (better: **`Prop`** constructor):
  - Given two **`Prop`**'s, it establishes a new **`Prop`**
  - The two **`Prop`**'s are parameters (implicit for `conj`)
  - There is notation `A ∧ B` for `and A B`
- How do you prove `A ∧ B`?
- Give proofs `a: A` and `b: B` and apply `conj` to them
- Example: Prove $\top \land \top$

```
Definition t_and_t: ⊤ ∧ ⊤ :=
  conj I I.
```

Disjunction

- We now come to our second connective, disjunction

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

Disjunction

- We now come to our second connective, disjunction

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

- `or` is a type constructor (better: **`Prop`** constructor):
  - Given two **`Prop`**'s, it establishes a new **`Prop`**
  - The two **`Prop`**'s are parameters (implicit for `or_introl`, `or_intror`)
  - There is notation `A ∨ B` for `or A B`

Disjunction

- We now come to our second connective, disjunction

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

- `or` is a type constructor (better: **`Prop`** constructor):
  - Given two **`Prop`**'s, it establishes a new **`Prop`**
  - The two **`Prop`**'s are parameters (implicit for `or_introl`, `or_intror`)
  - There is notation `A ∨ B` for `or A B`
- How do you prove `A ∨ B`?

Disjunction

- We now come to our second connective, disjunction

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

- `or` is a type constructor (better: **`Prop`** constructor):
  - Given two **`Prop`**'s, it establishes a new **`Prop`**
  - The two **`Prop`**'s are parameters (implicit for `or_introl`, `or_intror`)
  - There is notation `A ∨ B` for `or A B`
- How do you prove `A ∨ B`?
- Prove `a: A` and apply `or_introl` *or*
  prove `b: B` and apply `or_intror`

Disjunction

- We now come to our second connective, disjunction

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

- `or` is a type constructor (better: **`Prop`** constructor):
  - Given two **`Prop`**'s, it establishes a new **`Prop`**
  - The two **`Prop`**'s are parameters (implicit for `or_introl`, `or_intror`)
  - There is notation `A ∨ B` for `or A B`
- How do you prove `A ∨ B`?
- Prove `a: A` and apply `or_introl` *or*
  prove `b: B` and apply `or_intror`
- Example: Prove $\bot \vee \top$

```
Definition f_or_t: ⊥ ∨ ⊤ :=
  or_intror I.
```

Types, so far

- Let's recall what we know about types in Coq so far
- First, how are types formed? We saw 3 possibilities:

1a) **`Inductive`** types (atomic)

- `bool` : **`Type`**
  - with value `true`, `false`
- $\bot$: **`Prop`**
  - no proofs

1b) **`Inductive`** types (applied type constructors)

- `list nat` : **`Type`**
  - with value `1::2::3::nil`, ...
- $\top \wedge \top$: **`Prop`**
  - with proof `conj I I`

Types, so far

2) Function Types

- `bool → bool` : **Type**
  - with values `negb`, `(fun x ⇒ x)`, ...
- `nat → nat → nat` : **Type**
  - with values `plus`, ...
  - ...
- $\bot \to \bot$: **Prop**
  - with... a proof? Yes: `fun x ⇒ x`
- $\top \to \bot$: **Prop**
  - with... a proof? No.

Types, so far

3) Polymorphic Types

- `∀ (X: Type), list X` : **Type**
  - with one value: `nil`
- `∀ (X: Type), X → list X` : **Type**
  - with value `fun (X: Type) (x: X) ⇒ x::x::x::nil`
  - ...
- `∀ (A B: Prop), A → B → A ∧ B` : **Prop**
  - with proof `conj`
- `∀ (P: Prop), ⊤∨ P` : **Prop**
  - with... a proof? Yes:
    `fun (_: Prop) ⇒ or_introl I`

- **Polymorphic types** are **function types** that take types as arguments
- **Polymorphic values** are **functions** that take types as arguments

Polymorphic Propositions

- As we have seen, propositions can be polymorphic, too
- Example: *For all propositions* P, *we have* $\top \vee P$

Polymorphic Propositions

- As we have seen, propositions can be polymorphic, too
- Example: *For all propositions P, we have* $\top \vee P$
- We formulate and prove this proposition as follows:

```
Definition t_or_p:
  ∀ (P: Prop), ⊤ ∨ P ≔
  fun (_: Prop) ⇒ or_introl I.
```

Polymorphic Propositions

- As we have seen, propositions can be polymorphic, too
- Example: *For all propositions* P, *we have* $\top \vee P$
- We formulate and prove this proposition as follows:

```
Definition t_or_p:
  ∀ (P: Prop), ⊤ ∨ P ≔
  fun (_: Prop) ⇒ or_introl I.
```

- Short notation:

```
Definition t_or_p:
  ∀ P, ⊤ ∨ P ≔
  fun _ ⇒ or_introl I.
```

Polymorphic Propositions

- As we have seen, propositions can be polymorphic, too
- Example: *For all propositions* P, *we have* $\top \vee P$
- We formulate and prove this proposition as follows:

```
Definition t_or_p:
  ∀ (P: Prop), ⊤ ∨ P ≔
  fun (_: Prop) ⇒ or_introl I.
```

- Short notation:

```
Definition t_or_p:
  ∀ P, ⊤ ∨ P ≔
  fun _ ⇒ or_introl I.
```

- The type is *polymorphic* in P

Polymorphic Propositions

- As we have seen, propositions can be polymorphic, too
- Example: *For all propositions P, we have* $\top \lor P$
- We formulate and prove this proposition as follows:

```
Definition t_or_p:
  ∀ (P: Prop), ⊤ ∨ P ≔
  fun (_: Prop) ⇒ or_introl I.
```

- Short notation:

```
Definition t_or_p:
  ∀ P, ⊤ ∨ P ≔
  fun _ ⇒ or_introl I.
```

- The type is *polymorphic* in `P`
- The proof is *polymorphic* in `P`

Implication

- We now come to our third logical connective, implication

Implication

- We now come to our third logical connective, implication
- You have seen it, as it is already built-in: An *implication* is a *function type*!

Implication

- We now come to our third logical connective, implication
- You have seen it, as it is already built-in: An *implication* is a *function type*!
- Example: Prove that for all propositions `P`, we have `P` $\rightarrow$ $\top \wedge$ `P`.

```
Definition true_p:
  ∀ (P: Prop), P → (⊤ ∧ P) :=
  fun (P: Prop) (p: P) ⇒ conj I p.
```

Implication

- We now come to our third logical connective, implication
- You have seen it, as it is already built-in: An *implication* is a *function type*!
- Example: Prove that for all propositions P, we have P $\rightarrow$ $\top \wedge$ P.

```
Definition true_p:
  ∀ (P: Prop), P → (⊤ ∧ P) :=
  fun (P: Prop) (p: P) ⇒ conj I p.
```

- Short notation:

```
Definition true_p:
  ∀ (P: Prop), P → (⊤ ∧ P) :=
  fun P p ⇒ conj I p.
```

Proving with Tactics, 1

- A conjunction can be proven as follows

```
Definition t_and_t: ⊤ ∧ ⊤ :=
  conj I I.
```

Proving with Tactics, 1

- A conjunction can be proven as follows

```
Definition t_and_t: ⊤ ∧ ⊤ :=
  conj I I.
```

- We call the value of a proposition a **proof term**

Proving with Tactics, 1

- A conjunction can be proven as follows

```
Definition t_and_t: ⊤ ∧ ⊤ ≔
  conj I I.
```

- We call the value of a proposition a **proof term**
- This proof term can equivalently be obtained via **tactics**

```
Lemma t_and_t: ⊤ ∧ ⊤.
Proof.
  apply conj.
  - apply I.
  - apply I.
Qed.
```

Proving with Tactics, 1

- A conjunction can be proven as follows

```
Definition t_and_t: ⊤ ∧ ⊤ ≔
  conj I I.
```

- We call the value of a proposition a **proof term**
- This proof term can equivalently be obtained via **tactics**

```
Lemma t_and_t: ⊤ ∧ ⊤.
Proof.
  apply conj.
  - apply I.
  - apply I.
Qed.
```

- Tactics **generate proof terms**

Proving with Tactics, 1

- A conjunction can be proven as follows

```
Definition t_and_t: ⊤ ∧ ⊤ :=
  conj I I.
```

- We call the value of a proposition a **proof term**
- This proof term can equivalently be obtained via **tactics**

```
Lemma t_and_t: ⊤ ∧ ⊤.
Proof.
  apply conj.
  - apply I.
  - apply I.
Qed.
```

- Tactics **generate proof terms**
- Display proof term via `Print t_and_t.`

Proving with Tactics, 2

```
Lemma t_and_t: ⊤ ∧ ⊤.
Proof.
  apply conj.
  - apply I.
  - apply I.
Qed.
```

- Enables backwards-directed reasoning

Proving with Tactics, 2

```
Lemma t_and_t: ⊤ ∧ ⊤.
Proof.
  apply conj.
  - apply I.
  - apply I.
Qed.
```

- Enables backwards-directed reasoning
- The *goal* (proof obligation) is simplified/divided/reduced to smaller *subgoals*

Proving with Tactics, 2

```
Lemma t_and_t: ⊤ ∧ ⊤.
Proof.
  apply conj.
  - apply I.
  - apply I.
Qed.
```

- Enables backwards-directed reasoning
- The *goal* (proof obligation) is simplified/divided/reduced to smaller *subgoals*
- **`apply`** can be used to apply a value constructor `conj`

Proving with Tactics, 2

```
Lemma t_and_t: ⊤ ∧ ⊤.
Proof.
  apply conj.
  - apply I.
  - apply I.
Qed.
```

- Enables backwards-directed reasoning
- The *goal* (proof obligation) is simplified/divided/reduced to smaller *subgoals*
- **`apply`** can be used to apply a value constructor `conj`
- If the value constructor expects further arguments, further subgoals are generated

Proving with Tactics, 2

```
Lemma t_and_t: ⊤ ∧ ⊤.
Proof.
  apply conj.
  - apply I.
  - apply I.
Qed.
```

- Enables backwards-directed reasoning
- The *goal* (proof obligation) is simplified/divided/reduced to smaller *subgoals*
- **`apply`** can be used to apply a value constructor `conj`
- If the value constructor expects further arguments, further subgoals are generated
- This is the case in our example: We have two subgoals $\top$ and $\top$

Tactic: **exact**

```
Lemma t_and_t: ⊤ ∧ ⊤.
Proof.
  exact (conj I I).
Qed.
```

- By **exact**, one can give an explicit proof term

Tactic: **exact**

```
Lemma t_and_t: ⊤ ∧ ⊤.
Proof.
  exact (conj I I).
Qed.
```

- By **exact**, one can give an explicit proof term
- In this example, we give the whole proof term just by a single **exact**, which is equivalent to the two other definitions of t_and_t

Tactic: **intros**

```
Lemma p_q_p: ∀ (P Q: Prop), P → Q → P.
Proof.
  intros P Q p q.
  apply p.
Qed.
```

- By **intros**, arguments are assumed

Tactic: **intros**

```
Lemma p_q_p: ∀ (P Q: Prop), P → Q → P.
Proof.
  intros P Q p q.
  apply p.
Qed.
```

- By **intros**, arguments are assumed
- They are now available as **hypotheses** in the context Γ

Tactic: **intros**

```
Lemma p_q_p: ∀ (P Q: Prop), P → Q → P.
Proof.
  intros P Q p q.
  apply p.
Qed.
```

- By **`intros`**, arguments are assumed
- They are now available as **hypotheses** in the context Γ
- Correspondence to proof terms:
  `intros` `x y.` introduces **`fun`** `x y` $\Rightarrow$ ...

Tactic: **destruct** (1)

```
Lemma pq_p: ∀ (P Q: Prop), P ∧ Q → P.
Proof.
  intros P Q H.
  destruct H as [p q].
  apply p.
Qed.
```

- By **destruct**, a hypothesis is case-analyzed

Tactic: **destruct** (1)

```
Lemma pq_p: ∀ (P Q: Prop), P ∧ Q → P.
Proof.
  intros P Q H.
  destruct H as [p q].
  apply p.
Qed.
```

- By **destruct**, a hypothesis is case-analyzed
- In this example, there is only one case, `conj`

Tactic: **destruct** (1)

```
Lemma pq_p: ∀ (P Q: Prop), P ∧ Q → P.
Proof.
  intros P Q H.
  destruct H as [p q].
  apply p.
Qed.
```

- By **destruct**, a hypothesis is case-analyzed
- In this example, there is only one case, `conj`
- Correspondence to proof terms:
  Introduces **match** ... **with** `conj p q` ⇒ ...

Tactics: **destruct** (2), **left**, **right**

```
Lemma pq_or_qp:
  ∀ (P Q: Prop), P ∨ Q → Q ∨ P.
Proof.
  intros P Q H.
  destruct H as [p | q].
  - right. apply p.
  - left. apply q.
Qed.
```

- By **destruct**, a hypothesis is case-analyzed

Tactics: **destruct** (2), **left**, **right**

```
Lemma pq_or_qp:
  ∀ (P Q: Prop), P ∨ Q → Q ∨ P.
Proof.
  intros P Q H.
  destruct H as [p | q].
  - right. apply p.
  - left. apply q.
Qed.
```

- By **destruct**, a hypothesis is case-analyzed
- For each case, there is a subgoal

Tactics: **destruct** (2), **left**, **right**

```
Lemma pq_or_qp:
  ∀ (P Q: Prop), P ∨ Q → Q ∨ P.
Proof.
  intros P Q H.
  destruct H as [p | q].
  - right. apply p.
  - left. apply q.
Qed.
```

- By **destruct**, a hypothesis is case-analyzed
- For each case, there is a subgoal
- Correspondence to proof terms:
  Introduces **match** ... **with** | ... | ... ⇒ ...
- By **left** (**right**), the first (second) constructor is selected
- Correspondence to proof terms:
  Introduces `or_introl` resp. `or_intror`

Tactic: **split**

```
Lemma and_comm:
  ∀ (P Q: Prop), P ∧ Q → Q ∧ P.
Proof.
  intros P Q H.
  destruct H as [p q].
  split.
  - apply q.
  - apply p.
Qed.
```

- By **split**, a goal is split into subgoals

Tactic: **split**

```
Lemma and_comm:
  ∀ (P Q: Prop), P ∧ Q → Q ∧ P.
Proof.
  intros P Q H.
  destruct H as [p q].
  split.
  - apply q.
  - apply p.
Qed.
```

- By **split**, a goal is split into subgoals
- For each case, there is a subgoal

Tactic: **split**

```
Lemma and_comm:
  ∀ (P Q: Prop), P ∧ Q → Q ∧ P.
Proof.
  intros P Q H.
  destruct H as [p q].
  split.
  - apply q.
  - apply p.
Qed.
```

- By **split**, a goal is split into subgoals
- For each case, there is a subgoal
- Correspondence to proof terms:
  Introduces `conj ... ...`

Tactic: **exfalso**

```
Lemma false_proves_anything:
  ∀ (P: Prop), ⊥ → P.
Proof.
  intros P f.
  exfalso.
  exact f.
Qed.
```

- **exfalso** replaces the current goal by $\bot$

Tactic: **exfalso**

```
Lemma false_proves_anything:
  ∀ (P: Prop), ⊥ → P.
Proof.
  intros P f.
  exfalso.
  exact f.
Qed.
```

- **exfalso** replaces the current goal by $\bot$
- In other words, proving $\bot$ suffices to prove any P

Tactic: **exfalso**

```
Lemma false_proves_anything:
  ∀ (P: Prop), ⊥ → P.
Proof.
  intros P f.
  exfalso.
  exact f.
Qed.
```

- **`exfalso`** replaces the current goal by $\bot$
- In other words, proving $\bot$ suffices to prove any `P`
- The correspondence to proof terms is very interesting. Recall that $\bot$ is an empty type. What happens if we assume a proof of $\bot$ (as in the example)? As with every value, we can case-analyze it and prove `P` **for every case**. But there are **no cases**, so we are done!

Tactic: **exfalso**

```
Lemma false_proves_anything:
  ∀ (P: Prop), ⊥ → P.
Proof.
  intros P f.
  exfalso.
  exact f.
Qed.
```

- **`exfalso`** replaces the current goal by $\bot$
- In other words, proving $\bot$ suffices to prove any `P`
- The correspondence to proof terms is very interesting. Recall that $\bot$ is an empty type. What happens if we assume a proof of $\bot$ (as in the example)? As with every value, we can case-analyze it and prove `P` **for every case**. But there are **no cases**, so we are done!
- Crucial part of the corresponding proof term:
  `match` `f` **`with`** *(nothing here)* **`end`**

Tactics: **simpl**, **reflexivity**

```
Lemma negb_tf: negb true = false.
Proof.
  simpl.
  reflexivity.
Qed.
```

- By **simpl**, a goal is maximally reduced

Tactics: **simpl**, **reflexivity**

```
Lemma negb_tf: negb true = false.
Proof.
  simpl.
  reflexivity.
Qed.
```

- By **simpl**, a goal is maximally reduced
- This yields the subgoal `false = false`

Tactics: **simpl**, **reflexivity**

```
Lemma negb_tf: negb true = false.
Proof.
  simpl.
  reflexivity.
Qed.
```

- By **`simpl`**, a goal is maximally reduced
- This yields the subgoal `false = false`
- By **`reflexivity`**, we can prove such a goal

Tactics: **simpl**, **reflexivity**

```
Lemma negb_tf: negb true = false.
Proof.
  simpl.
  reflexivity.
Qed.
```

- By **`simpl`**, a goal is maximally reduced
- This yields the subgoal `false = false`
- By **`reflexivity`**, we can prove such a goal
- How is $=$ encoded as a type and what proof term does **`reflexivity`** introduce?

Tactics: **`simpl`**, **`reflexivity`**

```
Lemma negb_tf: negb true = false.
Proof.
  simpl.
  reflexivity.
Qed.
```

- By **`simpl`**, a goal is maximally reduced
- This yields the subgoal `false = false`
- By **`reflexivity`**, we can prove such a goal
- How is = encoded as a type and what proof term does **`reflexivity`** introduce?
- The answer is "as an inductive type" but the details are not relevant at this point

Negation

- We now come to our fourth logical connective, negation

Negation

- We now come to our fourth logical connective, negation
- You have seen it, as it is already built-in: The negation of a proposition `P` is the implication `P` $\rightarrow$ $\perp$

Negation

- We now come to our fourth logical connective, negation
- You have seen it, as it is already built-in: The negation of a proposition `P` is the implication `P -> ⊥`
- We use the notation `~P` for `P -> ⊥`

Negation

- We now come to our fourth logical connective, negation
- You have seen it, as it is already built-in: The negation of a proposition `P` is the implication `P → ⊥`
- We use the notation `~P` for `P → ⊥`
- Example: Prove that for all propositions `P`, we have `P → ~(~P)`.

```
Lemma not_not:
  ∀ (P: Prop), P → ~(~P).
Proof.
  intros P p.
  intros H.
  apply H.
  exact p.
Qed.
```

Types that Depend on Terms

- Recall polymorphic types, i.e. types that depend **on types**

3) Polymorphic Types

- `∀ (X: Type), list X` : **Type**
  - with value `nil`
- $\forall$ `(P: Prop),` $\top \vee$ `P` : **Prop**
  - with proof `fun (_: Prop) ⇒ or_introl I`

Types that Depend on Terms

- Recall polymorphic types, i.e. types that depend **on types**

3) Polymorphic Types

- `∀ (X: Type), list X` : **Type**
  - with value `nil`
- `∀ (P: Prop), ⊤∨ P` : **Prop**
  - with proof `fun (_: Prop) ⇒ or_introl I`

- Types can also depend **on terms**

4) Dependent Types

- `∀ (b: bool), negb (negb b) = b` : **Prop**
- `∀ (n: nat), 0 + n = n` : **Prop**
- `∀ (n: nat), n + 0 = n` : **Prop**
- `∀ (m n: nat), m + n = n + m` : **Prop**

Types that Depend on Terms

- Recall polymorphic types, i.e. types that depend **on types**

3) Polymorphic Types

- `∀ (X: Type), list X` : **Type**
  - with value `nil`
- `∀ (P: Prop), ⊤∨ P` : **Prop**
  - with proof `fun (_: Prop) ⇒ or_introl I`

- Types can also depend **on terms**

4) Dependent Types

- `∀ (b: bool), negb (negb b) = b` : **Prop**
- `∀ (n: nat), 0 + n = n` : **Prop**
- `∀ (n: nat), n + 0 = n` : **Prop**
- `∀ (m n: nat), m + n = n + m` : **Prop**

- *Roadmap*: We prove all of the above properties!

Type Universes

- The type of a type is called a *type universe*: Either **`Type`** or **`Prop`**[3]

```
bool : Type    ⊤                    : Prop
nat  : Type    10 = 4               : Prop
               ∀ (P: Prop), ⊤ ∨ P   : Prop
```

[3] This is a simplified view that is sufficient for now.

Type Universes

- The type of a type is called a *type universe*: Either **Type** or **Prop**[3]

```
bool : Type    ⊤                  : Prop
nat  : Type    10 = 4             : Prop
               ∀ (P: Prop), ⊤ ∨ P : Prop
```

- A type of **Type** (e.g. `nat`) contains **data values**

```
O     : nat            true  : bool
S O   : nat            false : bool
plus  : nat → nat → nat
```

[3]This is a simplified view that is sufficient for now.

Type Universes

- The type of a type is called a *type universe*: Either **`Type`** or **`Prop`**[3]

```
bool : Type    ⊤                    : Prop
nat  : Type    10 = 4               : Prop
               ∀ (P: Prop), ⊤ ∨ P   : Prop
```

- A type of **`Type`** (e.g. `nat`) contains **data values**

```
O      : nat             true  : bool
S O    : nat             false : bool
plus   : nat → nat → nat
```

- A type of **`Prop`** (e.g. ⊤) contains **proofs**

```
I      : ⊤
(fun (_: Type) ⇒ or_introl I)
       : ∀ (P: Prop), ⊤ ∨ P
```

[3] This is a simplified view that is sufficient for now.

Back to Booleans, 1

```
Definition negb (x: bool) : bool :=
  match x with
  | true  => false
  | false => true
  end.
```

- Let's prove `negb (negb b) = b` for all `b`

Back to Booleans, 1

```
Definition negb (x: bool) : bool :=
  match x with
  | true  => false
  | false => true
  end.
```

- Let's prove `negb (negb b) = b` for all `b`
- The term `negb (neg b)` does not reduce

Back to Booleans, 1

```
Definition negb (x: bool) : bool :=
  match x with
  | true  => false
  | false => true
  end.
```

- Let's prove `negb (negb b) = b` for all `b`
- The term `negb (neg b)` does not reduce
- Why not? `negb` performs pattern matching, but since we don't know anything about `b` (it could be *any bool*), we don't know which case will match

Back to Booleans, 1

```
Definition negb (x: bool) : bool :=
  match x with
  | true  ⇒ false
  | false ⇒ true
  end.
```

- Let's prove `negb (negb b) = b` for all `b`
- The term `negb (neg b)` does not reduce
- Why not? `negb` performs pattern matching, but since we don't know anything about `b` (it could be *any bool*), we don't know which case will match
- But there are only two possible values for `b`. So let's do a case analysis and prove every case!
  1. `b` is `true`. Then `negb (negb true)` reduces to `true`.
  2. `b` is `false`. Then `negb (negb false)` reduces to `false`.

Back to Booleans, 2

```
Definition negb (x: bool) : bool :=
  match x with
  | true  => false
  | false => true
  end.
```

- We have all the tools to prove this in Coq

Back to Booleans, 2

```
Definition negb (x: bool) : bool ≔
  match x with
  | true  ⇒ false
  | false ⇒ true
  end.
```

- We have all the tools to prove this in Coq

```
Lemma negb_inverse:
  ∀ (b: bool), negb (negb b) = b.
Proof.
  intros b.
  destruct b.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

Back to Natural Numbers

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

- We can easily prove that $0 + n = n$ for all n

Back to Natural Numbers

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

- We can easily prove that `0 + n = n` for all `n`
- Because: `0 + n` reduces to `n` by definition of `plus`

Back to Natural Numbers

```
Fixpoint plus (m n: nat) : nat :=
  match m with
  | O    ⇒ n
  | S m' ⇒ S (plus m' n)
  end.
```

- We can easily prove that `0 + n = n` for all `n`
- Because: `0 + n` reduces to `n` by definition of `plus`

```
Lemma O_plus_n: ∀ (n: nat), 0 + n = n.
Proof.
  intros n.
  simpl.
  reflexivity.
Qed.
```

Natural induction

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

- What about the other way round, `m + 0 = m`?

Natural induction

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

- What about the other way round, `m + 0 = m`?
- This should hold, but we cannot reduce `m + 0`

Natural induction

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

- What about the other way round, `m + 0 = m`?
- This should hold, but we cannot reduce `m + 0`
- Why not? `plus` pattern-matches on the first argument, but since we don't know anything about `m` (it could be *any number*), we don't know which case will match

Natural induction

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

- What about the other way round, `m + 0 = m`?
- This should hold, but we cannot reduce `m + 0`
- Why not? `plus` pattern-matches on the first argument, but since we don't know anything about `m` (it could be *any number*), we don't know which case will match
- "*any number*" rings a bell: Proof by Natural Induction!

Natural induction

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

- What about the other way round, `m + 0 = m`?
- This should hold, but we cannot reduce `m + 0`
- Why not? `plus` pattern-matches on the first argument, but since we don't know anything about `m` (it could be *any number*), we don't know which case will match
- "*any number*" rings a bell: Proof by Natural Induction!
- **Base case.** To show: $0 + 0 = 0$. By definition.

Natural induction

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

- What about the other way round, `m + 0 = m`?
- This should hold, but we cannot reduce `m + 0`
- Why not? `plus` pattern-matches on the first argument, but since we don't know anything about `m` (it could be *any number*), we don't know which case will match
- "*any number*" rings a bell: Proof by Natural Induction!
- **Base case.** To show: $0 + 0 = 0$. By definition.
- **Inductive case.** Let IH be $m + 0 = m$. To show: $(S\ m) + 0 = S\ m$. But this is by def. of `plus` convertible to $S(m + 0) = S\ m$. Now use IH, so we have to show $S\ m = S\ m$ which holds by reflexivity of $=$.

Natural induction in Coq, 1

- Let's do the same proof, but in Coq

Natural induction in Coq, 1

- Let's do the same proof, but in Coq

```
Lemma n_plus_O: ∀ (n: nat), n + 0 = n.
Proof.
  intros n.
  induction n.
  - simpl. reflexivity.
  - simpl. rewrite IHn. reflexivity.
Qed.
```

- **`induction`** `n` does a case-analysis on `n` (like **`destruct`**), but provides an additional inductive hypothesis

Natural induction in Coq, 1

- Let's do the same proof, but in Coq

```
Lemma n_plus_O: ∀ (n: nat), n + 0 = n.
Proof.
  intros n.
  induction n.
  - simpl. reflexivity.
  - simpl. rewrite IHn. reflexivity.
Qed.
```

- **`induction`** `n` does a case-analysis on `n` (like **`destruct`**), but provides an additional inductive hypothesis
- **`rewrite`** `IHn` uses the equation `IHn` to substitute a subterm in the current goal

Natural induction in Coq, 2

- Remember the very first property we wanted to show?
  `m + n = n + m` for all `m`, `n`

Natural induction in Coq, 2

- Remember the very first property we wanted to show?
  `m + n = n + m` for all `m`, `n`
- Proof by induction over `m`.
  - **Base case**. To show: $0 + n = n + 0$. By our last lemma, we know $n + 0 = n$; by definition of `plus` we know $0 + n = n$; thus we are done.
  - **Inductive case**. Let IH be $m + n = n + m$. To show: $(S\ m) + n = n + (S\ m)$. This goal reduces to $S(m + n) = n + (S\ m)$. By the IH, we can reduce the goal to $S(n + m) = n + (S\ m)$. Here we have the same problem as with proving $n + 0 = n$: It does not hold by definition, because `plus` pattern-matches on the *first* argument. Thus we prove this as an extra lemma, after which the proof is completed.

Natural induction in Coq, 3

- Let's prove the little lemma
  `m + (S n) = S (m + n)` for all `m`, `n`
  ... by induction on `m`

Natural induction in Coq, 3

- Let's prove the little lemma
  `m + (S n) = S (m + n)` for all `m`, `n`
  … by induction on `m`

```
Lemma m_plus_S:
  ∀ (m n: nat), m+(S n) = S (m+n).
Proof.
  intros m n.
  induction m.
  - simpl. reflexivity.
  - simpl. rewrite IHm. reflexivity.
Qed.
```

Natural induction in Coq, 4

- Now we can prove commutativity of addition in Coq, following the proof sketch given before

Natural induction in Coq, 4

- Now we can prove commutativity of addition in Coq, following the proof sketch given before

```
Lemma plus_comm:
  ∀ (m n: nat), m + n = n + m.
Proof.
  intros m.
  induction m.
  - intros. rewrite m_plus_O.
    simpl. reflexivity.
  - intros n. simpl. rewrite IHm.
    rewrite m_plus_S. reflexivity.
Qed.
```

Induction in Coq: Outlook, 1

- We know the principle of natural induction from maths

[4] For further reading: "x is-a-substructure-of y" is well-founded

Induction in Coq: Outlook, 1

- We know the principle of natural induction from maths
- Why have we considered this principle a sound proof method?

[4] For further reading: "x is-a-substructure-of y" is well-founded

Induction in Coq: Outlook, 1

- We know the principle of natural induction from maths
- Why have we considered this principle a sound proof method?
- Because our objects (here: natural numbers) are constructed out of **finitely** many steps[4]. We can view a proof of induction as a recipe on how to obtain a proof for **any concrete** number n.

[4] For further reading: "x is-a-substructure-of y" is well-founded

Induction in Coq: Outlook, 1

- We know the principle of natural induction from maths
- Why have we considered this principle a sound proof method?
- Because our objects (here: natural numbers) are constructed out of **finitely** many steps[4]. We can view a proof of induction as a recipe on how to obtain a proof for **any concrete** number n.
- Example: How do we prove $3 + 0 = 3$?
  - Use Inductive Case, but need to prove $2 + 0 = 2$. How?
  - Use Inductive Case, but need to prove $1 + 0 = 1$. How?
  - Use Inductive Case, but need to prove $0 + 0 = 0$. How?
  - Use Base Case.

[4] For further reading: "x is-a-substructure-of y" is well-founded

Induction in Coq: Outlook, 1

- We know the principle of natural induction from maths
- Why have we considered this principle a sound proof method?
- Because our objects (here: natural numbers) are constructed out of **finitely** many steps[4]. We can view a proof of induction as a recipe on how to obtain a proof for **any concrete** number n.
- Example: How do we prove $3 + 0 = 3$?
  - Use Inductive Case, but need to prove $2 + 0 = 2$. How?
  - Use Inductive Case, but need to prove $1 + 0 = 1$. How?
  - Use Inductive Case, but need to prove $0 + 0 = 0$. How?
  - Use Base Case.
- Doesn't this proof construction look a lot like... a recursive program?

---

[4] For further reading: "x is-a-substructure-of y" is well-founded

Induction in Coq: Outlook, 2

- Example: How do we prove $3 + 0 = 3$?
  - Use Inductive Case, but need to prove $2 + 0 = 2$. How?
  - Use Inductive Case, but need to prove $1 + 0 = 1$. How?
  - Use Inductive Case, but need to prove $0 + 0 = 0$. How?
  - Use Base Case.

[5] You can take a look e.g. via `Print nat_ind`.

Induction in Coq: Outlook, 2

- Example: How do we prove $3 + 0 = 3$?
  - Use Inductive Case, but need to prove $2 + 0 = 2$. How?
  - Use Inductive Case, but need to prove $1 + 0 = 1$. How?
  - Use Inductive Case, but need to prove $0 + 0 = 0$. How?
  - Use Base Case.
- In Coq, an **inductive proof** is a **recursive function**

[5] You can take a look e.g. via `Print nat_ind`.

Induction in Coq: Outlook, 2

- Example: How do we prove $3 + 0 = 3$?
  - Use Inductive Case, but need to prove $2 + 0 = 2$. How?
  - Use Inductive Case, but need to prove $1 + 0 = 1$. How?
  - Use Inductive Case, but need to prove $0 + 0 = 0$. How?
  - Use Base Case.
- In Coq, an **inductive proof** is a **recursive function**
- Every `Inductive` type has this essential property that each object is constructed out of finitely many steps

[5] You can take a look e.g. via `Print nat_ind`.

Induction in Coq: Outlook, 2

- Example: How do we prove $3 + 0 = 3$?
  - Use Inductive Case, but need to prove $2 + 0 = 2$. How?
  - Use Inductive Case, but need to prove $1 + 0 = 1$. How?
  - Use Inductive Case, but need to prove $0 + 0 = 0$. How?
  - Use Base Case.
- In Coq, an **inductive proof** is a **recursive function**
- Every `Inductive` type has this essential property that each object is constructed out of finitely many steps
- There is an induction principle for every type, and it is automatically generated[5]

[5] You can take a look e.g. via `Print nat_ind`.

Summary

- Coq is a **programming language** (with restricted recursion)

Summary

- Coq is a **programming language** (with restricted recursion)
- Data is defined inductively, i.e. all values are **finite objects**

Summary

- Coq is a **programming language** (with restricted recursion)
- Data is defined inductively, i.e. all values are **finite objects**
- Functions are defined recursively over this inductive structure

Summary

- Coq is a **programming language** (with restricted recursion)
- Data is defined inductively, i.e. all values are **finite objects**
- Functions are defined recursively over this inductive structure
- The Curry-Howard-Correspondence provides clever tricks to **encode propositions as types**

Summary

- Coq is a **programming language** (with restricted recursion)
- Data is defined inductively, i.e. all values are **finite objects**
- Functions are defined recursively over this inductive structure
- The Curry-Howard-Correspondence provides clever tricks to **encode propositions as types**
- A proposition is proven by a well-typed **proof term**

Summary

- Coq is a **programming language** (with restricted recursion)
- Data is defined inductively, i.e. all values are **finite objects**
- Functions are defined recursively over this inductive structure
- The Curry-Howard-Correspondence provides clever tricks to **encode propositions as types**
- A proposition is proven by a well-typed **proof term**
- A **proof by induction** is a recipe for constructing proofs for **any** element
  - This recipe is a recursive function!

Summary

- Coq is a **programming language** (with restricted recursion)
- Data is defined inductively, i.e. all values are **finite objects**
- Functions are defined recursively over this inductive structure
- The Curry-Howard-Correspondence provides clever tricks to **encode propositions as types**
- A proposition is proven by a well-typed **proof term**
- A **proof by induction** is a recipe for constructing proofs for **any** element
  - This recipe is a recursive function!
- Tactics assist the user in finding a proof term

Literature

[1] B. Pierce et al., Software Foundations
https://softwarefoundations.cis.upenn.edu/
(Free online book series)

[1] G. Smolka, Lecture Notes of *Introduction to Computational Logic*
https://www.ps.uni-saarland.de/courses.html