Principles of Programming Languages Lecture 3: Polymorphism. Higher-order functions. Logic in Coq.

Andrei Arusoaie[1]

[1]Department of Computer Science

October 12, 2021

Outline

Induction revisited

Polymorphism

Higher-order functions

Logic in Coq

Induction revisited

- Natural numbers

```
Inductive Nat := O : Nat | S : Nat -> Nat.
```

- Transitivity: strong vs. weak inductive hypothesis

```
Lemma le_Trans :
  forall m n p,
    le_Nat m n = true ->
    le_Nat n p = true ->
    le_Nat m p = true.
Proof.
(* Demo *)
```

Lists

- Lists of natural numbers:

```
Inductive ListNat :=
| Nil : ListNat
| Cons : Nat -> ListNat -> ListNat.
```

- Lists of booleans:

```
Inductive ListBool :=
| Nil : ListBool
| Cons : bool -> ListBool -> ListBool.
```

- They look very similar: code duplication!

Lists

- Lists of natural numbers:

```
Inductive ListNat :=
| Nil : ListNat
| Cons : Nat -> ListNat -> ListNat.
```

- Lists of booleans:

```
Inductive ListBool :=
| Nil : ListBool
| Cons : bool -> ListBool -> ListBool.
```

- They look very similar: code duplication!

Functions on lists

- Length of lists of natural numbers:

```
Fixpoint length(l : ListNat) :=
match l with
| Nil => O
| Cons _ l' => S (length l')
end.
```

- Length of lists of booleans:

```
Fixpoint length(l : ListBool) :=
match l with
| Nil => O
| Cons _ l' => S (length l')
end.
```

- Code duplication again (DRY!?!)

Functions on lists

- Length of lists of natural numbers:

```
Fixpoint length(l : ListNat) :=
match l with
| Nil => O
| Cons _ l' => S (length l')
end.
```

- Length of lists of booleans:

```
Fixpoint length(l : ListBool) :=
match l with
| Nil => O
| Cons _ l' => S (length l')
end.
```

- Code duplication again (DRY!?!)

Polymorphism

- Solution: polymorphism
- Polymorphic lists in Coq:

```
Inductive List (T : Type) : Type :=
| Nil : List T
| Cons : T -> List T -> List T.
```

- This is a similar definition of lists but this one is parametric in the type of its elements!
- `List` is a *function* from `Type`s to *inductive* definitions!

```
Check List.
List:
     Type -> Type
```

Polymorphism

- Solution: polymorphism
- Polymorphic lists in Coq:

```
Inductive List (T : Type) : Type :=
| Nil : List T
| Cons : T -> List T -> List T.
```

- This is a similar definition of lists but this one is parametric in the type of its elements!
- `List` is a *function* from `Type`s to *inductive* definitions!

```
Check List.
List:
    Type -> Type
```

Polymorphism

- Since `List` is a *function* from **`Type`**s to *inductive* definitions it means that we can use it to create new inductive definitions
- Here is the definition of lists of naturals:

```
Definition ListNat := List Nat.
```

- Here is the definition of lists of booleans:

```
Definition ListBool := List bool.
```

Polymorphism

- Since `List` is a *function* from **`Type`**s to *inductive* definitions it means that we can use it to create new inductive definitions
- Here is the definition of lists of naturals:

```
Definition ListNat := List Nat.
```

- Here is the definition of lists of booleans:

```
Definition ListBool := List bool.
```

Polymorphism

- Since `List` is a *function* from `Type`s to *inductive* definitions it means that we can use it to create new inductive definitions
- Here is the definition of lists of naturals:

  ```
  Definition ListNat := List Nat.
  ```
- Here is the definition of lists of booleans:

  ```
  Definition ListBool := List bool.
  ```

Polymorphism

- Automatically, the constructors are parametric too:

```
Check Nil.
Nil
     : forall T : Type, List T


Check Cons.
Cons
     : forall T : Type, T -> List T -> List T
```

Polymorphism

- Automatically, the constructors are parametric too:

```
Check Nil.
Nil
     : forall T : Type, List T
```

```
Check Cons.
Cons
     : forall T : Type, T -> List T -> List T
```

Polymorphism

- Automatically, the constructors are parametric too:

```
Check Nil.
Nil
     : forall T : Type, List T
```

```
Check Cons.
Cons
     : forall T : Type, T -> List T -> List T
```

Polymorphism: Functions

- Functions have polymorphic versions as well:

```
Fixpoint length (T : Type) (l : List T) : nat :=
  match l with
  | Nil _ => 0
  | Cons _ _ l' => S (length T l')
  end.
```

Implicit arguments

- Calling the function as below could be cumbersome:

```
Compute length Nat (Cons Nat O (Nil Nat)).
= 1
: nat
```

- The type is passed to the function and all the constructors
- Solution: Implicit arguments

```
Arguments Nil {T}.
Arguments Cons {T}.
Arguments length {T}.
Compute length (Cons 0 Nil).
= 1
: nat
```

Implicit arguments

- Calling the function as below could be cumbersome:

```
Compute length Nat (Cons Nat O (Nil Nat)).
= 1
: nat
```

- The type is passed to the function and all the constructors
- Solution: Implicit arguments

```
Arguments Nil {T}.
Arguments Cons {T}.
Arguments length {T}.
Compute length (Cons 0 Nil).
= 1
: nat
```

Higher-order functions

- Higher-order functions: functions that manipulate other functions
  - Functions can take other functions as input parameters
  - Functions can return other functions
- Example: filter a list using `f : T -> bool`:

```
Fixpoint filter {T : Type}
                (f : T -> bool)
                (l : List T) : List T :=
  match l with
  | Nil => Nil
  | Cons x xs => if (f x)
                 then Cons x (filter f xs)
                 else filter f xs
  end.
```

Usage of the `filter` function

- First, we need a function `f : T -> bool` that is passed as an argument to `filter`:

```
Definition has_one_digit (n : nat) := leb n 9.
Check has_one_digit.
has_one_digit
     : nat -> bool
```

- The function `has_one_digit` returns `true` if the argument `n` is a single digit number, and `false` otherwise

Usage of the `filter` function

- Second, pick an example of a list

```
Example num_list :=
  Cons 2 (Cons 15 (Cons 7 (Cons 18 Nil))).
```

- `filter` call with `has_one_digit` as argument:

```
Compute filter has_one_digit num_list.
= Cons 2 (Cons 7 Nil)
    : List nat.
```

Usage of the `filter` function

- Second, pick an example of a list

```
Example num_list :=
  Cons 2 (Cons 15 (Cons 7 (Cons 18 Nil))).
```

- `filter` call with `has_one_digit` as argument:

```
Compute filter has_one_digit num_list.
= Cons 2 (Cons 7 Nil)
    : List nat.
```

Anonymous functions

- We can define functions "on the fly" without declaring them explicitly and use them by their name
- Example: an anonymous function having the same functionality as `has_one_digit`

```
Check (fun n => leb n 9).
fun n : nat => n <=? 9
     : nat -> bool
```

- The `<=?` is just a notation for Coq's builtin function `leb`

Back to our `filter` function

- Our previous example of a list was:

```
Example num_list :=
  Cons 2 (Cons 15 (Cons 7 (Cons 18 Nil))).
```

- `filter` call with an anonymous function as argument:

```
Compute filter (fun n => leb n 9) num_list.
= Cons 2 (Cons 7 Nil)
    : List nat.
```

Back to our `filter` function

- Our previous example of a list was:

```
Example num_list :=
  Cons 2 (Cons 15 (Cons 7 (Cons 18 Nil))).
```

- `filter` call with an anonymous function as argument:

```
Compute filter (fun n => leb n 9) num_list.
= Cons 2 (Cons 7 Nil)
    : List nat.
```

Functions that return other functions

- Classical example: function composition

```
Definition compose {A : Type}
                   {B : Type}
                   {C : Type}
           (f : B -> C)
           (g : A -> B) :=
  fun x => f (g x).
```

Type variables

- The type of `compose`

```
Check compose.
compose
     : (?B -> ?C) -> (?A -> ?B) -> ?A -> ?C
where
?A : [ |- Type]
?B : [ |- Type]
?C : [ |- Type]
```

Using compose

- The type of `compose` when called:

```
Check compose (fun x =>  x * 2)
              (fun x => x + 2).
compose (fun x : nat => x * 2)
        (fun x : nat => x + 2)
     : nat -> nat.
```

- Actual call:

```
Compute compose (fun x : nat => x * 2)
                (fun x : nat => x + 2)
        3.
= 10
: nat
```

Using compose

- The type of compose when called:

```
Check compose (fun x =>  x * 2)
              (fun x => x + 2).
compose (fun x : nat => x * 2)
        (fun x : nat => x + 2)
     : nat -> nat.
```

- Actual call:

```
Compute compose (fun x : nat => x * 2)
                (fun x : nat => x + 2)
        3.
= 10
: nat
```

Logic in Coq. Understanding `Props`

- In Coq we can *state* propositions
- Moreover, we can *prove* them
- Propositions have a type of their own called `Prop`

```
Check 10 = 10.      Check 10 = 11.
10 = 10             10 = 11
     : Prop              : Prop
```

- Not all propositions are *provable*!

```
Goal 10 = 10.       Goal 10 = 11.
Proof.              Proof.
  reflexivity.        (* Can't prove this *)
Qed.                Abort.
```

Logic in Coq. Understanding `Prop`s

- In Coq we can *state* propositions
- Moreover, we can *prove* them
- Propositions have a type of their own called `Prop`

```
Check 10 = 10.        Check 10 = 11.
10 = 10               10 = 11
     : Prop                : Prop
```

- Not all propositions are *provable*!

```
Goal 10 = 10.         Goal 10 = 11.
Proof.                Proof.
  reflexivity.          (* Can't prove this *)
Qed.                  Abort.
```

Logic in Coq. Understanding `Prop`s

- In Coq we can *state* propositions
- Moreover, we can *prove* them
- Propositions have a type of their own called `Prop`

```
Check 10 = 10.          Check 10 = 11.
10 = 10                 10 = 11
     : Prop                  : Prop
```

- Not all propositions are *provable*!

```
Goal 10 = 10.           Goal 10 = 11.
Proof.                  Proof.
  reflexivity.            (* Can't prove this *)
Qed.                    Abort.
```

Logic in Coq. Understanding `Props`

- In Coq we can *state* propositions
- Moreover, we can *prove* them
- Propositions have a type of their own called `Prop`

```
Check 10 = 10.
10 = 10
     : Prop
```

```
Check 10 = 11.
10 = 11
     : Prop
```

- Not all propositions are *provable*!

```
Goal 10 = 10.
Proof.
  reflexivity.
Qed.
```

```
Goal 10 = 11.
Proof.
  (* Can't prove this *)
Abort.
```

Logic in Coq. Understanding `Props`

- In Coq we can *state* propositions
- Moreover, we can *prove* them
- Propositions have a type of their own called `Prop`

```
Check 10 = 10.
10 = 10
     : Prop
```

```
Check 10 = 11.
10 = 11
     : Prop
```

- Not all propositions are *provable*!

```
Goal 10 = 10.
Proof.
  reflexivity.
Qed.
```

```
Goal 10 = 11.
Proof.
  (* Can't prove this *)
Abort.
```

Logic in Coq. Understanding `Props`

- In Coq we can *state* propositions
- Moreover, we can *prove* them
- Propositions have a type of their own called `Prop`

```
Check 10 = 10.
10 = 10
     : Prop
```

```
Check 10 = 11.
10 = 11
     : Prop
```

- Not all propositions are *provable*!

```
Goal 10 = 10.
Proof.
  reflexivity.
Qed.
```

```
Goal 10 = 11.
Proof.
  (* Can't prove this *)
Abort.
```

Logic in Coq. Understanding `Props`

- In Coq we can *state* propositions
- Moreover, we can *prove* them
- Propositions have a type of their own called `Prop`

```
Check 10 = 10.
10 = 10
     : Prop
```

```
Check 10 = 11.
10 = 11
     : Prop
```

- Not all propositions are *provable*!

```
Goal 10 = 10.
Proof.
  reflexivity.
Qed.
```

```
Goal 10 = 11.
Proof.
  (* Can't prove this *)
Abort.
```

Implications

- Most of the properties are formulated as implications
- Dealing with implications in proofs: the `intros H` tactic extracts the hypothesis `H : n = 0`:

```
Lemma simple_impl :
  forall n, n = 0 -> n + 3 = 3.
Proof.
  intros n.
  intros H. (* H is the lhs of -> *)
  rewrite H.
  simpl.
  reflexivity.
Qed.
```

Implications

- Multiple implications are preferred instead of conjunctions because **intros** can extract the hypotheses easier:

```
Lemma not_so_simple_impl :
  forall m n, m = 0 -> n = 0 -> n + m = 0.
Proof.
  intros m n Hm Hn.
  (* Here, Hm is m = 0 and Hn is n = 0 *)
  rewrite Hn.
  rewrite Hm.
  simpl.
  reflexivity.
Qed.
```

Implications

- Naturally, one would formulate this property as:

```
Lemma not_so_simple_impl :
  forall m n, m = 0 /\ n = 0 -> n + m = 0.
```

instead of

```
Lemma not_so_simple_impl :
  forall m n, m = 0 -> n = 0 -> n + m = 0.
```

- But using `intros` will result in the following goal:

```
1 subgoal (ID 22)

  m, n : nat
  H : m = 0 /\ n = 0 (* <- not convenient *)
  ============================
  n + m = 0
```

Implications

- Naturally, one would formulate this property as:

```
Lemma not_so_simple_impl :
  forall m n, m = 0 /\ n = 0 -> n + m = 0.
```

instead of

```
Lemma not_so_simple_impl :
  forall m n, m = 0 -> n = 0 -> n + m = 0.
```

- But using `intros` will result in the following goal:

```
1 subgoal (ID 22)

  m, n : nat
  H : m = 0 /\ n = 0 (* <- not convenient *)
  ============================
  n + m = 0
```

Implications and conjunctions

Using implications instead of conjunctions is not a problem:

- $\varphi_1 \to (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi)$
- $(\varphi_1 \wedge \varphi_2) \to \varphi \equiv \neg(\varphi_1 \wedge \varphi_2) \vee \varphi \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$
- Since $\neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi) \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$ (because $\vee$ is associative) we also have by transitivity

$$\varphi_1 \to (\varphi_2 \to \varphi) \equiv (\varphi_1 \wedge \varphi_2) \to \varphi$$

- Conclusion: it's safe to use implications!

Implications and conjunctions

Using implications instead of conjunctions is not a problem:

- $\varphi_1 \to (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi)$
- $(\varphi_1 \wedge \varphi_2) \to \varphi \equiv \neg(\varphi_1 \wedge \varphi_2) \vee \varphi \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$
- Since $\neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi) \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$ (because $\vee$ is associative) we also have by transitivity

$$\varphi_1 \to (\varphi_2 \to \varphi) \equiv (\varphi_1 \wedge \varphi_2) \to \varphi$$

- Conclusion: it's safe to use implications!

Implications and conjunctions

Using implications instead of conjunctions is not a problem:

- $\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi) \equiv \neg\varphi_1 \vee (\varphi_2 \rightarrow \varphi) \equiv \neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi)$
- $(\varphi_1 \wedge \varphi_2) \rightarrow \varphi \equiv \neg(\varphi_1 \wedge \varphi_2) \vee \varphi \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$
- Since $\neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi) \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$ (because $\vee$ is associative) we also have by transitivity

$$\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi) \equiv (\varphi_1 \wedge \varphi_2) \rightarrow \varphi$$

- Conclusion: it's safe to use implications!

Implications and conjunctions

Using implications instead of conjunctions is not a problem:

- $\varphi_1 \to (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi)$
- $(\varphi_1 \wedge \varphi_2) \to \varphi \equiv \neg(\varphi_1 \wedge \varphi_2) \vee \varphi \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$
- Since $\neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi) \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$ (because $\vee$ is associative) we also have by transitivity

$$\varphi_1 \to (\varphi_2 \to \varphi) \equiv (\varphi_1 \wedge \varphi_2) \to \varphi$$

- Conclusion: it's safe to use implications!

Implications and conjunctions

Using implications instead of conjunctions is not a problem:

- $\varphi_1 \to (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi)$
- $(\varphi_1 \wedge \varphi_2) \to \varphi \equiv \neg(\varphi_1 \wedge \varphi_2) \vee \varphi \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$
- Since $\neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi) \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$ (because $\vee$ is associative) we also have by transitivity

$$\varphi_1 \to (\varphi_2 \to \varphi) \equiv (\varphi_1 \wedge \varphi_2) \to \varphi$$

- Conclusion: it's safe to use implications!

Implications and conjunctions

Using implications instead of conjunctions is not a problem:

- $\varphi_1 \to (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi)$
- $(\varphi_1 \wedge \varphi_2) \to \varphi \equiv \neg(\varphi_1 \wedge \varphi_2) \vee \varphi \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$
- Since $\neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi) \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$ (because $\vee$ is associative) we also have by transitivity

$$\varphi_1 \to (\varphi_2 \to \varphi) \equiv (\varphi_1 \wedge \varphi_2) \to \varphi$$

- Conclusion: it's safe to use implications!

Implications and conjunctions

Using implications instead of conjunctions is not a problem:

- $\varphi_1 \to (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi)$
- $(\varphi_1 \wedge \varphi_2) \to \varphi \equiv \neg(\varphi_1 \wedge \varphi_2) \vee \varphi \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$
- Since $\neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi) \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$ (because $\vee$ is associative) we also have by transitivity

$$\varphi_1 \to (\varphi_2 \to \varphi) \equiv (\varphi_1 \wedge \varphi_2) \to \varphi$$

- Conclusion: it's safe to use implications!

Implications and conjunctions

Using implications instead of conjunctions is not a problem:

- $\varphi_1 \to (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi)$
- $(\varphi_1 \wedge \varphi_2) \to \varphi \equiv \neg(\varphi_1 \wedge \varphi_2) \vee \varphi \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$
- Since $\neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi) \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$ (because $\vee$ is associative) we also have by transitivity

$$\varphi_1 \to (\varphi_2 \to \varphi) \equiv (\varphi_1 \wedge \varphi_2) \to \varphi$$

- Conclusion: it's safe to use implications!

Implications and conjunctions

Using implications instead of conjunctions is not a problem:

- $\varphi_1 \to (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\varphi_2 \to \varphi) \equiv \neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi)$
- $(\varphi_1 \wedge \varphi_2) \to \varphi \equiv \neg(\varphi_1 \wedge \varphi_2) \vee \varphi \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$
- Since $\neg\varphi_1 \vee (\neg\varphi_2 \vee \varphi) \equiv (\neg\varphi_1 \vee \neg\varphi_2) \vee \varphi$ (because $\vee$ is associative) we also have by transitivity

$$\varphi_1 \to (\varphi_2 \to \varphi) \equiv (\varphi_1 \wedge \varphi_2) \to \varphi$$

- Conclusion: it's safe to use implications!

Conjunction as goals

- Using the `split` tactic:

```
Lemma simple_conjunction:
  2 + 3 = 5 /\ 5 + 5 = 10.
Proof.
  split.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

- When the same tactics apply to both goals generated by `split` we can use semicolon `;` to apply the next tactics to both goals:

```
Lemma simple_conjunction :
  2 + 3 = 5 /\ 5 + 5 = 10.
Proof.
  split; simpl; reflexivity.
Qed.
```

Conjunction as goals

- Using the **split** tactic:

```
Lemma simple_conjunction:
  2 + 3 = 5 /\ 5 + 5 = 10.
Proof.
  split.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

- When the same tactics apply to both goals generated by **split** we can use semicolon ; to apply the next tactics to both goals:

```
Lemma simple_conjunction :
  2 + 3 = 5 /\ 5 + 5 = 10.
Proof.
  split; simpl; reflexivity.
Qed.
```

Conjunction as hypothesis

There are two ways of breaking conjunctions in separate hypotheses:

- Using **destruct**:

```
Lemma conjunction_as_hypothesis:
 forall m n, n = 0 /\ m = 0 -> n + 3 = 3.
Proof.
  intros m n Hnm.
  destruct Hnm as [Hn Hm].
  rewrite Hn. simpl. reflexivity.
Qed.
```

- Using **intros** with sugar syntax for conjunctions:

```
Lemma conjunction_as_hypothesis':
  forall m n, n = 0 /\ m = 0 -> n + 3 = 3.
Proof.
  intros m n [Hn Hm].
  rewrite Hn. simpl. reflexivity.
Qed.
```

Conjunction as hypothesis

There are two ways of breaking conjunctions in separate hypotheses:

- Using `destruct`:

```
Lemma conjunction_as_hypothesis:
 forall m n, n = 0 /\ m = 0 -> n + 3 = 3.
Proof.
  intros m n Hnm.
  destruct Hnm as [Hn Hm].
  rewrite Hn. simpl. reflexivity.
Qed.
```

- Using `intros` with sugar syntax for conjunctions:

```
Lemma conjunction_as_hypothesis':
  forall m n, n = 0 /\ m = 0 -> n + 3 = 3.
Proof.
  intros m n [Hn Hm].
  rewrite Hn. simpl. reflexivity.
Qed.
```

Disjunction as goal

There are two ways of proving a disjunction:

- Either prove the left prop using **left**:

```
Lemma simple_disjunction_left:
  2 + 3 = 5 \/ 5 + 6 = 10.
Proof.
  left.
  simpl.
  reflexivity.
Qed.
```

- Or prove the right right prop using **right**:

```
Lemma simple_disjunction_right:
  2 + 8 = 5 \/ 5 + 5 = 10.
Proof.
  right.
  simpl.
  reflexivity.
Qed.
```

Disjunction as goal

There are two ways of proving a disjunction:

- Either prove the left prop using **left**:

```
Lemma simple_disjunction_left:
  2 + 3 = 5 \/ 5 + 6 = 10.
Proof.
  left.
  simpl.
  reflexivity.
Qed.
```

- Or prove the right right prop using **right**:

```
Lemma simple_disjunction_right:
  2 + 8 = 5 \/ 5 + 5 = 10.
Proof.
  right.
  simpl.
  reflexivity.
Qed.
```

Disjunction as hypothesis

When a disjunction is a hypothesis we need to prove that both parts of the disjunction actually imply the property to be proved.

- We can use **intros** to break the goal in two goals:

```
Lemma disjunction_as_hypothesis:
  forall n, n = 0 \/ 5 + 5 = 11 -> n + 3 = 3.
Proof.
  intros n [Hn | Hn].
  - rewrite Hn. simpl. reflexivity.
  - inversion Hn.
Qed.
```

- Or we can use **destruct** to generate the two goals:

```
Lemma disjunction_as_hypothesis':
  forall n, n = 0 \/ 5 + 5 = 11 -> n + 3 = 3.
Proof.
  intros n H.
  destruct H as [Hn | Hn].
  - rewrite Hn. simpl. reflexivity.
  - inversion Hn.
Qed.
```

Disjunction as hypothesis

When a disjunction is a hypothesis we need to prove that both parts of the disjunction actually imply the property to be proved.

- We can use **intros** to break the goal in two goals:

```
Lemma disjunction_as_hypothesis:
  forall n, n = 0 \/ 5 + 5 = 11 -> n + 3 = 3.
Proof.
  intros n [Hn | Hn].
  - rewrite Hn. simpl. reflexivity.
  - inversion Hn.
Qed.
```

- Or we can use **destruct** to generate the two goals:

```
Lemma disjunction_as_hypothesis':
  forall n, n = 0 \/ 5 + 5 = 11 -> n + 3 = 3.
Proof.
  intros n H.
  destruct H as [Hn | Hn].
  - rewrite Hn. simpl. reflexivity.
  - inversion Hn.
Qed.
```

Disjunction as hypothesis

When a disjunction is a hypothesis we need to prove that both parts of the disjunction actually imply the property to be proved.

- We can use **intros** to break the goal in two goals:

```
Lemma disjunction_as_hypothesis:
  forall n, n = 0 \/ 5 + 5 = 11 -> n + 3 = 3.
Proof.
  intros n [Hn | Hn].
  - rewrite Hn. simpl. reflexivity.
  - inversion Hn.
Qed.
```

- Or we can use **destruct** to generate the two goals:

```
Lemma disjunction_as_hypothesis':
  forall n, n = 0 \/ 5 + 5 = 11 -> n + 3 = 3.
Proof.
  intros n H.
  destruct H as [Hn | Hn].
  - rewrite Hn. simpl. reflexivity.
  - inversion Hn.
Qed.
```

Disjunctions

- If one part of the disjunction is not sufficient to prove the goal, then the entire proof fails:

```
Lemma disjunction_as_hypothesis_unprovable:
  forall n, n = 0 \/ 5 + 5 = 10 -> n + 3 = 3.
Proof.
  intros n [Hn | Hn].
  - rewrite Hn. simpl. reflexivity.
  - (* stuck: can't prove this case *)
    (* the hypothesis 5 + 5 = 10 is useless *)
Abort.
```

Negations

- Negations are in fact implications `P -> False` in Coq.
- We can use **unfold** in proofs to reveal implications:

```
Lemma simple_negation:
  forall (x : nat), ~ x <> x.
Proof.
  intros x.
  unfold not.
  (* Here the goal is:
    x : nat
  ============================
  (x = x -> False) -> False *)
  intros Hx.
  apply Hx.
  reflexivity.
Qed.
```

Contradiction in proofs

- Sometimes we have to prove a goal by contradiction
- We can use **inversion**:

```
Theorem prove_false:
  forall P, False -> P.
Proof.
  intros P HF.
  inversion HF.
Qed.
```

- Or we can use `exfalso`:

```
Theorem ex_falso:
  forall P, False -> P.
Proof.
  intros P HF.
  exfalso.
  assumption.
Qed.
```

Existential quantifiers in goals

- In Coq, proving properties of the form $\exists x.P(x)$ requires a value v for x s.t. $P(v)$.
- In proofs, this is done via the **exists** tactic:

```
Lemma exists_zero:
  exists (n : nat), n = 0.
Proof.
 (* 0 is the only value that
    satisfies the equality *)
  exists 0.
  reflexivity.
Qed.
```

Existential quantifiers in hypotheses

- When in hypotheses, existentially quantified properties $\exists x.P(x)$ can be decomposed into pairs $(v, P(v))$, where v is a name chosen for the value that satisfies P.
- This decomposition can be done via **destruct** or **intros** (as shown below):

```
Lemma exists_as_hypothesis:
  forall m, (exists n, m = 2 + n) ->
            (exists n', m = 1 + n').
Proof.
  intros m [n Hmn].
  exists (1 + n).
  rewrite Hmn.
  simpl.
  reflexivity.
Qed.
```

Universal quantifiers in hypotheses

- So far we proved universally quantified properties
- Universally quantified hypothese can be applied to the other hypotheses:

```
Lemma forall_hyp:
  forall n,
    (forall m, m > 10 -> m > 0) -> n > 10 -> n > 0.
Proof.
  intros n H H'.
  (* here H is instantiated over n > 10 *)
  apply H in H'.
  (* H changed to n > 0 *)
  assumption.
Qed.
```

Universal quantifiers in hypotheses

- Universally quantified hypotheses can be applied directly to goals:

```
Lemma forall_hyp':
  forall n,
    (forall m, m > 10 -> m > 0) -> n > 10 -> n > 0.
Proof.
  intros n H H'.
  (* here the conclusion of H matches the goal *)
  apply H.
  (* the precondition of H needs to be proved *)
  assumption.
Qed.
```

Conclusions

- We've learned about
  1. polymorphism
  2. higher-order functions
  3. anonymous functions
  4. logic in Coq and new tactics
- Bibliography:

  1. Chaper Polymorphism and Higher-Order Functions and Chaper Logic in Coq in Software Foundations - Volume 1, Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, Brent Yorgey https://softwarefoundations.cis.upenn.edu/lf-current/toc.html

Conclusions

- We've learned about
  1. polymorphism
  2. higher-order functions
  3. anonymous functions
  4. logic in Coq and new tactics
- Bibliography:

  1. Chaper Polymorphism and Higher-Order Functions and Chaper Logic in Coq in Software Foundations - Volume 1, Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, Brent Yorgey `https://softwarefoundations.cis.upenn.edu/lf-current/toc.html`