CS720

Logical Foundations of Computer Science

Lecture 3: induction

Tiago Cogumbreiro

Recap

- We are currently learning the Logical Foundations (volume 1 of the SF book)
- We are learning a **programming language** that allows us formalize programming languages

> What do we mean by formalizing programming languages?

Recap

- We are currently learning the Logical Foundations (volume 1 of the SF book)
- We are learning a **programming language** that allows us formalize programming languages

> What do we mean by formalizing programming languages?

1. A way to describe the abstract syntax (do we know how to do this?)
2. A way to describe how language executes (do we know how to do this?)
3. A way to describe properties of the language (do we know how to do this?)

Today we will learn...

- about proofs with recursive data structures
- how to use induction in Coq
- how to infer the induction principle
- about the difference between informal and mechanized proofs

Compile `Basic.v`

CoqIDE:

- Open Basics.v. In the "Compile" menu, click on "Compile Buffer".

Console:

- `make Basics.vo`

Example: prove this lemma (1/4)

```
Theorem plus_n_O : forall n:nat,
  n = n + 0.
Proof.
```

Example: prove this lemma (1/4)

```
Theorem plus_n_O : forall n:nat,
  n = n + 0.
Proof.
```

Tactic `simpl` does nothing.

Example: prove this lemma (1/4)

```
Theorem plus_n_O : forall n:nat,
  n = n + 0.
Proof.
```

Tactic `simpl` does nothing. Tactic `reflexivity` fails.

Example: prove this lemma (1/4)

```
Theorem plus_n_O : forall n:nat,
  n = n + 0.
Proof.
```

Tactic `simpl` does nothing. Tactic `reflexivity` fails. Apply `destruct n`.

```
2 subgoals
______________________________________(1/2)
0 = 0 + 0
______________________________________(2/2)
S n = S n + 0
```

Example: prove this lemma (2/4)

After proving the first, we get

```
1 subgoal
n : nat
______________________________________(1/1)
S n = S n + 0
```

Applying `simpl` yields:

```
1 subgoal
n : nat
______________________________________(1/1)
S n = S (n + 0)
```

Example: prove this lemma (2/4)

After proving the first, we get

```
1 subgoal
n : nat
______________________________________(1/1)
S n = S n + 0
```

Applying `simpl` yields:

```
1 subgoal
n : nat
______________________________________(1/1)
S n = S (n + 0)
```

Tactic `reflexivity` fails and there is nothing to rewrite.

We need an induction principle of `nat`

For some property `P` we want to prove.

- Show that $P(0)$ holds.
- Given the induction hypothesis $P(n)$, show that $P(n+1)$ holds.

Conclude that $P(n)$ holds for all n.

Example: prove this lemma (3/4)

Apply `induction n`.

```
2 subgoals
______________________________________(1/2)
0 = 0 + 0
______________________________________(2/2)
S n = S n + 0
```

> How do we prove the first goal?
> Compare `induction n` with `destruct n`.

Example: prove this lemma (4/4)

After proving the first goal we get

```
1 subgoal
n : nat
IHn : n = n + 0
______________________________________(1/1)
S n = S n + 0
```

applying `simpl` yields

```
1 subgoal
n : nat
IHn : n = n + 0
______________________________________(1/1)
S n = S (n + 0)
```

> How do we conclude this proof?

Intermediary results

```
Theorem mult_0_plus' : forall n m : nat,
  (0 + n) * m = n * m.
Proof.
  intros n m.
  assert (H: 0 + n = n). { reflexivity. }
  rewrite -> H.
  reflexivity. Qed.
```

- `H` is a variable name, you can pick whichever you like.
- Your intermediary result will capture all of the existing hypothesis.
- It may include `forall`.
- We use braces `{` and `}` to prove a sub-goal.

Formal versus informal proofs

- The objective of a mechanical (formal) proofs is to convince the proof checker.
- The objective of an informal proof is to convince (logically) the reader.
- `ltac` proofs are imperative, assume the reader can step through
- In informal proofs we want to help the reader reconstruct the proof state.

An example of an `ltac` proof

```
Theorem plus_assoc : forall n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n as [| n' IHn'].
  - reflexivity.
  - simpl. rewrite -> IHn'. reflexivity. Qed.
```

1. The proof follows by induction on n.

An example of an `ltac` proof

```
Theorem plus_assoc : forall n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n as [| n' IHn'].
  - reflexivity.
  - simpl. rewrite -> IHn'. reflexivity. Qed.
```

1. The proof follows by induction on n.
2. In the base case, we have that $n = 0$. We need to show $0 + (m + p) = 0 + m + p$, which follows by the definition of $+$.

An example of an `ltac` proof

```
Theorem plus_assoc : forall n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n as [| n' IHn'].
  - reflexivity.
  - simpl. rewrite → IHn'. reflexivity. Qed.
```

1. The proof follows by induction on n.
2. In the base case, we have that $n = 0$. We need to show $0 + (m + p) = 0 + m + p$, which follows by the definition of $+$.
3. In the inductive case, we have $n = \mathsf{S}\ n'$ and must show $Sn' + (m + p) = Sn' + m + p$.
   From the definition of $+$ it follows that $\mathsf{S}\ (n' + (m + p)) = \mathsf{S}\ (n' + m + p)$.
   The proof concludes by applying the induction hypothesis $n' + (m + p) = n' + m + p$
   .

How do we define a data structure that holds two nats?

A pair of nats

```
Inductive natprod : Type :=
| pair : nat → nat → natprod.

Notation "( x , y )" := (pair x y).
```

> Explicit vs implicit: be cautious when declaring notations, they make your code harder to understand.

How do we read the contents of a pair?

Accessors of a pair

Accessors of a pair

```
Definition fst (p : natprod) : nat :=
```

Accessors of a pair

```
Definition fst (p : natprod) : nat :=
  match p with
  | pair x y ⇒ x
  end.

Definition snd (p : natprod) : nat :=
  match p with
  | (x, y) ⇒ y (* using notations in a pattern to be matched *)
   end.
```

How do we prove the correctness of our accessors?

(What do we expect fst/snd to do?)

Proving the correctness of our accessors:

```
Theorem surjective_pairing : forall (p : natprod),
  p = (fst p, snd p).
Proof.
  intros p.

1 subgoal
p : natprod
______________________________________(1/1)
p = (fst p, snd p)
```

Does `simpl` work? Does `reflexivity` work? Does `destruct` work? What about `induction`?

How do we define a list of nats?

A list of nats

```
Inductive natlist : Type :=
  | nil : natlist
  | cons : nat → natlist → natlist.

(* You don't need to learn notations, just be aware of its existence:*)

Notation "x :: l" := (cons x l) (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).

Compute cons 1 (cons 2 (cons 3 nil)).
```

outputs:

```
= [1; 2; 3]
: list nat
```

How do we concatenate two lists?

Concatenating two lists

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil ⇒ l2
  | h :: t ⇒ h :: (app t l2)
  end.

Notation "x ++ y" := (app x y) (right associativity, at level 60).
```

Proving results on list concatenation

```
Theorem nil_app_l : forall l:natlist,
  [] ++ l = l.
Proof.
  intros l.
```

Can we prove this with `reflexivity`? Why?

Proving results on list concatenation

```
Theorem nil_app_l : forall l:natlist,
  [] ++ l = l.
Proof.
  intros l.
```

> Can we prove this with `reflexivity`? Why?

```
  reflexivity.
Qed.
```

Nil is a neutral element wrt app

```
Theorem nil_app_l : forall l:natlist,
  l ++ [] = l.
Proof.
  intros l.
```

> Can we prove this with `reflexivity`? Why?

Nil is a neutral element wrt app

```
Theorem nil_app_l : forall l:natlist,
  l ++ [] = l.
Proof.
  intros l.
```

Can we prove this with `reflexivity`? Why?

```
In environment
l : natlist
Unable to unify "l" with "l ++ [ ]".
```

How can we prove this result?

We need an induction principle of `natlist`

For some property `P` we want to prove.

- Show that $P([])$ holds.
- Given the induction hypothesis $P(l)$ and some number n, show that $P(n :: l)$ holds.

Conclude that $P(l)$ holds for all l.

> How do we know this principle? Hint: compare `natlist` with `nat`.

Comparing nats with natlists

```
Inductive natlist : Type :=
 | O : natlist                           | A: T
 | S : nat → nat.                        | B: T → T
```

1. $\vdash P(A)$
2. $t : T, P(t) \vdash P(B\ t)$

```
Inductive natlist : Type :=
 | nil : natlist                         | A: T
 | cons : nat → natlist → natlist.       | B: X → T → T
```

1. $\vdash P(A)$
2. $x : X, t : T, P(t) \vdash P(B\ t)$

How do we know the induction principle?

Use search

```
Search natlist.
```

which outputs

```
 nil: natlist
 cons: nat → natlist → natlist
(* trimmed output *)
natlist_ind:
   forall P : natlist → Prop,
   P [] →
   (forall (n : nat) (l : natlist), P l → P (n::l)) → forall n : natlist, P n
```

Nil is neutral on the right (1/2)

```
Theorem nil_app_r : forall l:natlist,
  l ++ [] = l.
Proof.
  intros l.
  induction l.
  - reflexivity.
  -
```

yields

```
1 subgoal
n : nat
l : natlist
IHl : l ++ [ ] = l
______________________________________(1/1)
(n :: l) ++ [ ] = n :: l
```

Nil is neutral on the right (2/2)

```
1 subgoal
n : nat
l : natlist
IHl : l ++ [ ] = l
______________________________________(1/1)
(n :: l) ++ [ ] = n :: l
```

Nil is neutral on the right (2/2)

```
1 subgoal
n : nat
l : natlist
IHl : l ++ [ ] = l
______________________________________(1/1)
(n :: l) ++ [ ] = n :: l

simpl.          (* app (n::l) [] = n :: (app l []) *)
rewrite → IHl.  (*   n :: (app l []) = n :: l *)
                (*        ^^^^^^^^^          ^ *)
reflexivity.    (* conclude *)
```

> Can we apply rewrite directly without simplifying?
> Hint: before and after stepping through a tactic show/hide notations.
> How do we state a theorem that leads to the same proof state (without ltac)?

How do we signal failure in a functional language?

Partial functions

How declare a function that is not defined for empty lists?

```
(* Pairs the head and the list *)
Fixpoint indexof n (l:natlist) :=
  match l with
  | [] => ???
  | h :: t =>
    match beq_nat h n with
    | true => 0
    | false => S (indexof t)
    end
  end.
```

Optional results

```
Inductive natoption : Type :=
  | Some : nat → natoption
  | None : natoption.
```

How do we declare indexof with optional types?

```
Fixpoint indexof n (l:natlist) : natoption :=
```

How do we declare indexof with optional types?

```
Fixpoint indexof n (l:natlist) : natoption :=
  match l with
  | [] => None
  | h :: t =>
    match beq_nat h n with
    | true => Some 0
    | false => S (indexof n t)
    end
  end.
```

How do we declare indexof with optional types?

```
Fixpoint indexof n (l:natlist) : natoption :=
  match l with
  | [] => None
  | h :: t =>
    match beq_nat h n with
    | true => Some 0
    | false => S (indexof n t)
    end
  end.
```

```
   | false => S (indexof n t)
                ^^^^^^^^^^^
```

```
The term "indexof n t" has type "natoption" while it is expected to have type "nat".
```

How do we declare indexof with optional types?

```
Fixpoint indexof (n:nat) (l:natlist) : natoption :=
  match l with
  | [] ⇒ None
  | h :: t ⇒
    match beq_nat h n with
    | true ⇒ Some 0            (* element found at the head *)
    | false ⇒
      match indexof n t with   (* check for error *)
      | Some i ⇒ Some (S i)    (* increment successful result *)
      | None ⇒ None            (* propagate error *)
      end
    end
  end.
```

Summary

Summary

- implemented containers: pair, list, option
- partial functions via option types
- reviewed case analysis, proof by induction
- used `Search` to browse definitions

Next class: read Poly.v

Ltac vocabulary

- `simpl`
- `reflexivity`
- `intros`
- `rewrite`
- `destruct`
- `induction`
- `assert`

(Nothing new from Lesson 2.)