COMP2313
Formal Specification and Verification
Lecture 5: Rocq - Induction - Proof by Induction

Asieh Salehi Fathabadi

Prelude

MyEngagement code

963221

Recall: When Case Analysis Fails

Last lecture we saw that case analysis is not always sufficient:

```
Theorem plus_n_O : forall n:nat,
  n = n + 0.
Proof.
  intros n.
  destruct n as [| n'] eqn:E.
  - (* n = 0 *) reflexivity. (* Works! *)
  - (* n = S n' *)
    simpl.
    (* Goal: S n' = S (n' + 0) *)
    (* Stuck! Need to know n' = n' + 0 *)
```

To prove the successor case,

we need to know the result for `n'`.

Introduction to Induction

Why Induction?

Some properties require reasoning about all natural numbers:

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

- We can't check all natural numbers individually (infinitely many!)
- Case analysis only splits into finite cases
- **Induction** allows us to prove properties for infinitely many values
- Key idea: prove for n by assuming it holds for $n - 1$

University of Southampton

Principle of Mathematical Induction

To prove $P(n)$ for all natural numbers n:

1. **Base Case**: Prove $P(0)$
2. **Inductive Case**: Prove that $P(n') \Rightarrow P(S\,n')$
    - Assume $P(n')$ holds (the **inductive hypothesis**)
    - Prove that $P(S\,n')$ holds

Why does this work?

- $P(0)$ holds by base case
- $P(1) = P(S\,0)$ holds by inductive case (using $P(0)$)
- $P(2) = P(S\,1)$ holds by inductive case (using $P(1)$)
- And so on for all natural numbers...

Our First Induction Proof

Our First Induction Proof

```
Theorem plus_n_O : forall n:nat,
  n = n + 0.
Proof.
  intros n.
  induction n as [| n' IHn'].
  - (* n = 0 *)
    reflexivity.
  - (* n = S n' *)
    simpl.
    rewrite <- IHn'.
    reflexivity.
Qed.
```

- `induction n` applies mathematical induction
- `IHn’` is the inductive hypothesis: `n’ = n’ + 0`
- In the inductive case, we use `IHn’` to complete the proof

Understanding the Base Case

After `induction n`, the base case goal is:

```
============================
0 = 0 + 0
```

After `simpl`:

```
0 = 0
```

Then `reflexivity` completes this case.

This proves $P(0)$: the property holds for zero.

Understanding the Inductive Case

In the inductive step, we need to prove:

```
n' : nat
IHn' : n' = n' + 0
==============================
S n' = S n' + 0
```

After `simpl`:

```
S n' = S (n' + 0)
```

`rewrite <- IHn’` transforms `n’ + 0` to `n’`:

```
S n' = S n'
```

Then `reflexivity` completes the proof.

The Power of the Inductive Hypothesis

The inductive hypothesis (IH) is the key:

- We **assume** the property holds for `n'`
- We **prove** it holds for `S n'`
- This creates a "chain" of proofs:
  - Base case proves it for 0
  - Inductive case proves: if true for k, then true for $k + 1$
  - Therefore true for 1, 2, 3, ... all natural numbers!

Critical: We don't prove it separately for each number. We prove the general pattern!

More Example: Addition is Associative

```
Theorem plus_assoc : forall n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p.
  induction n as [| n' IHn'].
  - (* n = 0 *)
    reflexivity.
  - (* n = S n' *)
    simpl.
    rewrite -> IHn'.
    reflexivity.
Qed.
```

- Induction on the first argument `n`
- Inductive hypothesis: `n’ + (m + p) = (n’ + m) + p`
- Goal becomes: `S (n’ + (m + p)) = S ((n’ + m) + p)`

Let's reflect:Understanding Induction Proofs

```
Theorem minus_n_n : forall n,
  minus n n = 0.
Proof.
  intros n.
  induction n as [| n' IHn'].
  - (* n = 0 *)
    simpl. reflexivity.
  - (* n = S n' *)
    simpl. rewrite -> IHn'. reflexivity.
Qed.
```

Join at:
vevox.app

ID:
157-735-404

Understanding Induction Proofs

```
Theorem minus_n_n : forall n,
  minus n n = 0.
Proof.
  intros n.
  induction n as [| n' IHn'].
  - (* n = 0 *)
    simpl. reflexivity.
  - (* n = S n' *)
    simpl. rewrite -> IHn'. reflexivity.
Qed.
```

- Base case: `0 - 0 = 0` holds by definition
- Inductive case: assuming `n' - n' = 0`, prove `S n' - S n' = 0`
- By definition: `S n' - S n' = n' - n'`
  which equals 0 by IH

Proving Commutativity

Commutativity Requires a Lemma

To prove `n + m = m + n`, we first need:

```
Lemma plus_n_Sm : forall n m : nat,
  S (n + m) = n + (S m).
Proof.
  intros n m.
  induction n as [| n' IHn'].
  - (* n = 0 *)
    reflexivity.
  - (* n = S n' *)
    simpl.
    rewrite -> IHn'.
    reflexivity.
Qed.
```

This lemma states that successor can be moved from one side to the other in addition.

Addition is Commutative

```
Theorem plus_comm : forall n m : nat,
  n + m = m + n.
Proof.
  intros n m.
  induction n as [| n' IHn'].
  - (* n = 0 *)
    rewrite <- plus_n_O.
    reflexivity.
  - (* n = S n' *)
    simpl.
    rewrite -> IHn'.
    rewrite -> plus_n_Sm.
    reflexivity.
Qed.
```

- Uses both `plus_n_O` and `plus_n_Sm` lemmas
- Shows the importance of proving helper lemmas first

University of Southampton

Strategy: Build Helper Lemmas

Complex proofs often require helper lemmas:

1. Identify what facts you need
2. Prove these facts as separate lemmas
3. Use the lemmas in the main proof

Example: To prove `n + m = m + n`:

- Need: `n + 0 = n` (proved as `plus_n_O`)
- Need: `n + S m = S (n + m)` (proved as `plus_n_Sm`)
- Then combine in main proof

This is like proving smaller theorems to build up to larger ones.

Advanced Proof Techniques

Using Assert for Lemmas

```
Theorem plus_rearrange : forall n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  assert (H: n + m = m + n).
  {
    rewrite -> plus_comm.
    reflexivity.
  }
  rewrite -> H.
  reflexivity.
Qed.
```

- assert introduces a new subgoal
- Once proved, it becomes available as hypothesis H
- Useful for complex proofs
  requiring intermediate results

Combining Multiple Techniques

```
Theorem plus_swap : forall n m p : nat,
  n + (m + p) = m + (n + p).
Proof.
  intros n m p.
  rewrite -> plus_assoc.
  rewrite -> plus_assoc.
  assert (H: n + m = m + n).
  { rewrite -> plus_comm. reflexivity. }
  rewrite -> H.
  reflexivity.
Qed.
```

Combines associativity, commutativity, and assert

to rearrange terms.

Choosing the Right Variable

Using Induction on Other Variables

Sometimes we need to choose which variable to induct on:

```
Theorem plus_n_Sm_alt : forall n m : nat,
  S (n + m) = n + (S m).
Proof.
  intros n m.
  induction n as [| n' IHn'].
  - (* n = 0 *)
    reflexivity.
  - (* n = S n' *)
    simpl.
    rewrite -> IHn'.
    reflexivity.
Qed.
```

Induction on `n` rather than `m` because addition is defined recursively on the first argument.

How to Choose Induction Variable

General principle: Induct on the variable that drives the recursion

- Look at the function definitions involved
- Which argument is pattern-matched?
- Induct on that variable

Example: For `plus n m`:

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

⇒ Induct on `n`, not `m`

University of Southampton

When to Use Induction

When to Use Induction vs Case Analysis

Use case analysis (`destruct`) when:

- The property holds for a finite number of cases
- Each case can be proved independently
- Example: boolean properties, small finite types

Use induction when:

- The property involves recursive data types
- Proving for arbitrary natural numbers
- The proof of case n depends on case $n - 1$
- Example: properties about all natural numbers

University of Southampton

Key Principles of Induction

1. **Choose the right variable**: Induct on the variable that drives the recursion
2. **Prove helper lemmas first**: Complex proofs often need intermediate results
3. **Use the inductive hypothesis**: The key step is applying IH correctly
4. **Structure matters**: Keep proofs organized with bullets and clear comments
5. **Build incrementally**: Start with simple cases, add complexity gradually

Common Induction Patterns

Pattern 1: Direct induction

- Induct on the variable, apply IH directly
- Example: `n + 0 = n`

Pattern 2: Induction with lemmas

- Need helper lemmas before main proof
- Example: `n + m = m + n` needs `n + S m = S (n + m)`

Pattern 3: Induction with rewriting

- Combine IH with other theorems
- Example: Associativity and commutativity proofs

Summary

Proof Strategies Summary

Technique	When to Use
`reflexivity`	When both sides are identical
`simpl`	To evaluate/reduce expressions
`rewrite`	To use equations/hypotheses
`destruct`	For case analysis on finite types
`induction`	For recursive/infinite types
`assert`	To prove intermediate facts
`replace`	To substitute specific subterms

Choose tactics based on the structure of your goal and available hypotheses.

Summary

1. **Mathematical Induction**:
    - Base case and inductive case
    - Using the inductive hypothesis
    - Choosing the right variable to induct on
2. **Helper Lemmas**:
    - Break complex proofs into smaller pieces
    - Prove fundamental properties first
    - Build up to main theorems
3. **Advanced Techniques**:
    - `assert` for inline lemmas
    - `replace` for substitution
    - Combining multiple tactics

University of Southampton

Key Takeaways

- **Induction is powerful**: Proves properties for infinitely many values
- **Two parts required**: Base case and inductive case
- **IH is the key**: Use it to connect n to $n + 1$
- **Choose carefully**: Induct on the recursive variable
- **Build incrementally**: Prove helper lemmas first
- **Practice essential**: Induction proofs become easier with experience

Typical Induction Proof Structure

```
Theorem property_name : forall n : nat,
  (* property about n *).
Proof.
  intros n.
  induction n as [| n' IHn'].
  - (* n = 0 : Base case *)
    (* Prove property for 0 *)
    simpl. reflexivity.
  - (* n = S n' : Inductive case *)
    (* IHn' available here *)
    simpl.
    rewrite -> IHn'.  (* Use IH *)
    (* Complete proof *)
    reflexivity.
Qed.
```

Additional Resources

- Software Foundations full textbook: `softwarefoundations.cis.upenn.edu`
- Rocq: `https://rocq-prover.org/`

Next Lecture

Topics for next time:

- Lists and structured data
- Induction on lists
- More complex data structures
- Additional proof patterns

Questions?

University of Southampton