

Lecture 7: Induction, Simplification, and Software Verification

Stefano M. Nicoletti

1 Lecture 7: Induction, Simplification, and Software Verification

1.1 Aim of the lecture

We continue from the first induction proof in Lecture 6. We prove further properties of addition, introduce `simp`, and combine induction, rewriting, and simplification. We then apply the same tools to the verification of a recursive algorithm on lists.

1.2 Definitions and available results

We continue to use the natural numbers, addition, and doubling defined in Lecture 6:

`Natural` has the constructors `zero` and `succ`. The function `add (m n : Natural)` recurses on its second argument, while `double (n : Natural)` recurses on its only argument.

The argument examined by a definition determines when Lean can compute. The definition of `add` can select a branch when it knows the form of the second argument. Thus, `add m zero` and `add m (succ n)` reduce directly. By contrast, `add zero n` requires induction while `n` remains an unknown variable.

The computational equations and the final result from Lecture 6 are available:

```
open Natural
```

```
theorem lecture07_add_zero_right (m : Natural) : add m zero = m := by  
  rfl
```

```
open Natural
```

```
theorem lecture07_add_succ_right (m n : Natural) :  
  add m (succ n) = succ (add m n) := by  
  rfl
```

```
open Natural
```

```
theorem lecture07_double_succ (n : Natural) :
```

```
double (succ n) = succ (succ (double n)) := by
rfl
```

```
open Natural
```

```
theorem lecture07_zero_add (n : Natural) : add zero n = n := by
  induction n with
  | zero =>
    rfl
  | succ n ih =>
    unfold add
    rewrite [ih]
    rfl
```

The first three results state the computational behaviour of the definitions. The last is a general theorem proved by induction. This distinction remains important throughout the lecture: some transformations are computations, whereas others use previously proved results.

1.3 A second induction proof

We now prove that a successor in the first argument can be moved outside the addition:

$\text{add} (\text{succ } m) n = \text{succ} (\text{add } m n).$

The visible successor may suggest induction on m . The definition tells us otherwise. add examines its second argument, and that position contains n in both additions. The unknown form of n prevents the definition from choosing between its zero and succ branches, so we induct on n .

```
open Natural
```

```
theorem lecture07_succ_add (m n : Natural) :
  add (succ m) n = succ (add m n) := by
  induction n with
  | zero =>
    rfl
  | succ n ih =>
    unfold add
    rewrite [ih]
    rfl
```

In the base case, replacing n with zero lets both sides reduce to the same expression. In the induction step, unfold add opens both additions for the case $\text{succ } n$. The remaining goal contains the same equality for n , which is exactly the induction hypothesis ih . Rewriting with ih makes both sides identical.

The induction variable therefore follows the computation. We look for the variable in the argument examined by the recursive definition and check that the induction hypothesis will rewrite the recursive call produced in the successor case.

1.4 Associativity, commutativity, and ‘simp’

Associativity states that grouping three addends in different ways does not change the result. On the left, c is the second argument of the outer addition. On the right, it is the second argument of $\text{add } b \ c$, which in turn controls the outer computation. We therefore induct on c .

```
open Natural
```

```
theorem lecture07_add_assoc (a b c : Natural) :  
  add (add a b) c = add a (add b c) := by  
  induction c with  
  | zero =>  
    rfl  
  | succ c ih =>  
    rw [lecture07_add_succ_right]  
    rw [lecture07_add_succ_right]  
    rw [lecture07_add_succ_right]  
    rw [ih]
```

In the base case, additions with zero as their second argument reduce, and `rfl` closes the goal. In the successor case, three rewrites with `lecture07_add_succ_right` expose the recursive additions on both sides. The remaining equality is the induction hypothesis under `succ`, so `rw [ih]` completes the proof.

Commutativity states that exchanging the two addends does not change the result. It does not follow from computation alone because `add` treats its two arguments asymmetrically: it examines the second argument and leaves the first unchanged. We induct on b . In the base case, the left side computes while the right side uses `lecture07_zero_add`. In the successor case, `lecture07_add_succ_right` handles the successor on the right and `lecture07_succ_add` handles the successor that appears in the first argument after the exchange.

```
open Natural
```

```
theorem lecture07_add_comm (a b : Natural) :  
  add a b = add b a := by  
  induction b with  
  | zero =>  
    rw [lecture07_zero_add]  
    rfl  
  | succ b ih =>  
    rw [lecture07_add_succ_right]  
    rw [lecture07_succ_add]  
    rw [ih]
```

This proof explains why small supporting lemmas matter. The definition gives the equations for the second argument directly. The theorems `zero_add` and `succ_add` provide the corresponding forms for the first argument. Commutativity combines these two views.

When a proof requires many elementary rewrites, we can use the simplifier. `simp` applies equations oriented towards simpler forms and performs basic computations that Lean recognizes directly. It does not search arbitrarily for a proof. It uses a collection of rules called the *simp-set*.

We compare three uses of the same theorem. `rw` performs a specific rewrite chosen by us. `simp only [lemmas]` uses only the listed lemmas, together with the simplifier's essential reductions. `simp [lemma]` uses the standard rules and adds lemma for that command.

```
open Natural

theorem lecture07_zero_add_with_rw (n : Natural) : add zero n = n := by
  rw [lecture07_zero_add]
```

The first theorem names the exact rewrite. The second asks the simplifier to work with one declared rule. The third adds the theorem to the *simp-set* for a single invocation. All three prove the same statement, but they provide different degrees of control and automation.

```
open Natural

theorem lecture07_zero_add_with_simp_only (n : Natural) : add zero n = n := by
  simp only [lecture07_zero_add]
```

```
open Natural

theorem lecture07_zero_add_with_simp (n : Natural) : add zero n = n := by
  simp [lecture07_zero_add]
```

```
open Natural

section LocalSimplification

attribute [local simp] lecture07_zero_add lecture07_succ_add

theorem lecture07_succ_add_with_simp (m n : Natural) :
  add (succ m) n = succ (add m n) := by
  simp

end LocalSimplification
```

The `local simp` attribute applies only inside the section. It lets us extend the *simp-set* without changing every later proof in the module. Inside the section, `simp` finds `lecture07_succ_add` automatically, so the theorem need not be named in the tactic call.

```
open Natural

@[simp] theorem lecture07_double_zero :
```

```
double zero = zero := by
  rfl
```

`@[simp]` registers a rule for the rest of the module. We use it for a canonical equation such as `double zero = zero`. Associativity and commutativity are not appropriate simp rules because they only move parentheses or swap arguments.

Rule selection matters. A good simplification rule predictably moves an expression towards a shorter or canonical form. Associativity would require a global choice of grouping. Commutativity could keep exchanging arguments without producing an evidently simpler expression. We therefore apply these results explicitly where they are needed.

1.5 A final proof

We combine induction and rewriting to show that the double of a sum is the sum of the doubles. The subexpression that remains blocked is `add m n`, so we induct on `n`.

```
open Natural

theorem lecture07_double_add (m n : Natural) :
  double (add m n) = add (double m) (double n) := by
  induction n with
  | zero =>
    rfl
  | succ n ih =>
    rw [lecture07_add_succ_right]
    rw [lecture07_double_succ]
    rw [lecture07_double_succ]
    rw [lecture07_add_succ_right]
    rw [lecture07_add_succ_right]
    rw [ih]
```

The base case follows from the definitions of `add` and `double`. In the successor case, the first rewrite moves the successor out of `add m (succ n)`. The two rewrites with `lecture07_double_succ` describe doubling a successor on both sides. The next two addition rewrites expose the same successor structure. Finally, `ih` replaces the double of the smaller sum by the sum of the doubles.

After checking each step, we can express the same proof with a controlled collection of simplification rules:

```
open Natural

theorem lecture07_double_add_short (m n : Natural) :
  double (add m n) = add (double m) (double n) := by
  induction n with
  | zero =>
    rfl
  | succ n ih =>
    simp only [lecture07_add_succ_right, lecture07_double_succ, ih]
```

The shorter version does not use a different mathematical argument. It uses the same equations but lets `simp` only choose their occurrences. The explicit list remains useful because it records which results justify the proof.

1.6 From induction to software verification

A program takes an input and produces an output. Verification states an expected property precisely and proves it for every input. To apply this idea to a new data structure, we begin by defining the type of lists inductively.

```
inductive CustomList  $\alpha$  ( : Type) where
| empty
| cons (head :  $\alpha$ ) (tail : CustomList  $\alpha$ )
```

A list is therefore empty or built from a first element and a smaller list. In the next lecture, we will use this structure to define an algorithm and verify some of its properties.

1.7 Sources

- Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich, *Theorem Proving in Lean 4*, “Inductive Types”: https://lean-lang.org/theorem_proving_in_lean4/Inductive-Types/.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich, *Theorem Proving in Lean 4*, “Induction and Recursion”: https://lean-lang.org/theorem_proving_in_lean4/Induction-and-Recursion/.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich, *Theorem Proving in Lean 4*, “Tactics”: https://lean-lang.org/theorem_proving_in_lean4/Tactics/.
- David Thrane Christiansen, *Functional Programming in Lean*, “Datatypes and Patterns”: https://lean-lang.org/functional_programming_in_lean/Getting-to-Know-Learn/Datatypes-and-Patterns/.
- Lean project, *The Lean Language Reference*, “Recursive Definitions”: <https://lean-lang.org/doc/reference/latest/Definitions/Recursive-Definitions/>.
- Kevin Buzzard and Jon Eugster, *Natural Number Game*: <https://adam.math.hu.de/>.
- Vrije Universiteit Amsterdam, *Logical Verification*: <https://research.vu.nl/en/courses/logical-verification-5/>.
- Marieke Huisman and Anton J. Wijs, *Concise Guide to Software Verification: From Model Checking to Annotation Checking*: <https://research.utwente.nl/en/publications/concise-guide-to-software-verification-from-model-checking-to-ann/>.

- Hana Chockler, Joseph Y. Halpern, and Orna Kupferman, *What Causes a System to Satisfy a Specification?*: <https://arxiv.org/abs/cs/0312036>.