

Lecture 6: Inductive Types, Recursion, and Induction

Stefano M. Nicoletti

1 Lecture 6: Inductive Types, Recursion, and Induction

1.1 Aim of the lecture

In this lecture we move from domains whose structure is unknown to types for which we specify every possible value. We learn how to construct these values, define functions that follow their structure, and prove properties that hold for every element of the type.

This transition introduces three related ideas. An *inductive type* specifies how its values are built. A *recursive definition* computes by following that construction. A *proof by induction* proves a property by following the same cases. We also use `rfl`, `unfold`, and `rw` to recognize expressions that are equal, expose a definition, and rewrite with an equality.

1.2 Inductive types

In earlier lectures, `Thing : Type` left the structure of the domain open. We could declare objects, predicates, and functions, but Lean did not know which objects belonged to `Thing` or how they were constructed.

An inductive type provides this information by listing the ways in which its values can be built. These ways are called *constructors*. The declaration below introduces a new type and simultaneously states that its values can be constructed in three ways.

```
inductive TrafficLight where
| red
| yellow
| green
```

`TrafficLight.red`, `TrafficLight.yellow`, and `TrafficLight.green` are values of type `TrafficLight`. In other words, they are the constructors that produce values of the new type. Because no other constructors are declared, these three cases describe every possible `TrafficLight`.

Knowing every case allows us to define operations that behave differently on each one. We define a predicate by pattern matching:

```
def MustStop : TrafficLight → Prop
| .red => True
| .yellow => True
```

```
| .green => False
```

Pattern matching gives a result for each constructor. `MustStop .red` and `MustStop .yellow` reduce to `True`, while `MustStop .green` reduces to `False`. The definition covers all constructors, so it gives a result for every possible traffic light.

The same knowledge of constructors can be used in proofs. Given an arbitrary traffic light `light`, we do not initially know its colour. We do know that it must be one of the three declared values. The tactic `cases light` replaces the original problem with one problem for each constructor.

```
theorem lecture06_traffic_light_has_a_colour (light : TrafficLight) :  
  light = .red v light = .yellow v light = .green := by  
  cases light with  
  | red =>  
    apply Or.inl  
    rfl  
  | yellow =>  
    apply Or.inr  
    apply Or.inl  
    rfl  
  | green =>  
    apply Or.inr  
    apply Or.inr  
    rfl
```

In the red case, the goal is a disjunction whose first alternative is true. `Or.inl` selects that alternative, and `rfl` proves `.red = .red`. In the yellow case, we first select the right side of the outer disjunction and then the left side of the inner one. In the green case, we select the right side twice.

Here, `cases` is enough because we only need to inspect a finite number of possibilities. We do not need to relate one case to a smaller case.

1.3 Our natural numbers

We now consider a type with a different structure. The traffic-light colours are three independent cases. Natural numbers are instead built progressively: we start from zero and obtain a new natural by applying the successor constructor to one that has already been built.

To make this structure visible, we define our own natural-number type without using `Nat`:

```
inductive Natural where  
  | zero  
  | succ (n : Natural)
```

```
open Natural
```

```
def one : Natural := succ zero
```

```
open Natural
```

```
def two : Natural := succ one
```

```
open Natural
```

```
def three : Natural := succ two
```

The constructor `zero` takes no arguments and directly produces a natural number. The constructor `succ` takes a value `n : Natural` and produces its successor. The definition is recursive because `Natural` occurs as an argument to one of its own constructors.

The definitions of `one`, `two`, and `three` are readable abbreviations. If we unfolded them, we would obtain `succ zero`, `succ (succ zero)`, and `succ (succ (succ zero))`. Every finite value of `Natural` is therefore a chain of applications of `succ` ending in `zero`.

The declaration also states that there are no other ways to construct a `Natural`. This information will matter when we prove a property of every natural number: it will be enough to follow the constructors `zero` and `succ`.

1.4 Recursion

A recursive definition solves a problem by using the result of the same problem on simpler data. For our natural numbers, the simpler value contained in `succ n` is `n`: one application of `succ` has been removed, bringing us closer to `zero`.

Defining a recursive function on `Natural` therefore means explaining what the function returns for `zero` and how it uses the result already computed for `n` in the `succ n` case. We begin with addition. The function takes two natural numbers, but its definition analyses the second argument.

```
open Natural
```

```
def add (m n : Natural) : Natural :=  
  match n with  
  | zero => m  
  | succ n => succ (add m n)
```

The first branch says that adding `zero` to `m` returns `m`. The second says that adding `succ n` to `m` applies `succ` to the result of `add m n`. For example, `add one two` computes as follows:

```
add one (succ one)  
succ (add one one)  
succ (succ (add one zero))  
succ (succ one)
```

Every step whose second argument has the form `succ n` shortens the chain and continues with `n`. When the second argument becomes zero, computation reaches the base case.

```
open Natural

def double (n : Natural) : Natural :=
  match n with
  | zero => zero
  | succ n => succ (succ (double n))
```

The definition of `double` follows the same structure. The double of zero is zero. In the `succ n` case, it uses the double of `n` and adds two applications of `succ`. The recursive call again removes one `succ`, so repeated computation eventually reaches the zero case.

`match n with` makes explicit which argument controls computation. In add, when the form of the second argument is known, Lean can select a branch of the definition. When that position contains a variable whose form is unknown, computation must wait.

1.5 Computation, 'rfl', and 'unfold'

Two expressions are *definitionally equal* when Lean can compute them until they become the same expression. No previously proved theorem is needed: Lean only unfolds definitions and follows their computation rules.

The tactic `rfl` proves an equality when both sides coincide after this computation. In the first example, Lean unfolds one, two, and add, and then recognizes that both sides describe the same value.

```
open Natural

example : add one one = two := by
  rfl
```

```
open Natural

theorem lecture06_add_zero_right (m : Natural) :
  add m zero = m := by
  rfl
```

```
open Natural

theorem lecture06_add_succ_right (m n : Natural) :
  add m (succ n) = succ (add m n) := by
  rfl
```

```
open Natural
```

```
theorem lecture06_double_succ (n : Natural) :  
  double (succ n) = succ (succ (double n)) := by  
  rfl
```

Consider the next three theorems. In `add m zero` and `add m (succ n)`, the second argument already shows which branch of the match must be selected. Similarly, `double (succ n)` immediately selects the successor branch.

These equalities should be distinguished from statements such as `add zero n = n`. In the latter, the second argument is the variable `n`. Lean cannot select a branch until it knows whether `n` is `zero` or `succ k`.

In the first two names, “right” says that `zero` or `succ n` is the second argument of `add`. That is the argument examined by `match n with`, so these equalities follow directly from the definition. The analogous statements with `zero` or `succ` on the left need induction.

`unfold` makes a definition explicitly visible in the goal. It is useful when we want to inspect steps that `rfl` can normally perform without displaying them:

```
open Natural
```

```
example : double one = two := by  
  unfold double  
  unfold one two  
  rfl
```

After `unfold double`, the left side exposes the pattern matching used by the function. Unfolding `one` and `two` then allows Lean to select the `succ` branch and reduce both sides to the same chain of constructors. `unfold` therefore makes an existing definition visible by expanding it.

1.6 Rewriting with ‘rw’

Not every useful equality comes directly from computation. Sometimes an equality is an assumption or a theorem proved earlier, and we want to use it to transform the goal.

If we have `h : a = b`, `rw [h]` searches the goal for occurrences of `a` and replaces them with `b`. `rw [← h]` uses the same equality in the opposite direction, replacing `b` with `a`. The arrow only changes the direction in which we use it.

```
open Natural
```

```
theorem lecture06_rw_forward  
  (a b : Natural)  
  (h : a = b) :  
  add a zero = b := by
```

```
rw [h]
rfl
```

Before the first rewrite, the goal is `add a zero = b`. With `rw [h]`, the occurrence of `a` is replaced by `b`, and the goal becomes `add b zero = b`. The definition of `add` reduces the left side to `b`, so `rfl` concludes the proof.

In the opposite direction, the goal contains `b` on the left and `a` on the right. Using `rw [← h]`, Lean replaces `b` with `a` and obtains `add a zero = a`.

```
open Natural

theorem lecture06_rw_backward
  (a b : Natural)
  (h : a = b) :
  add b zero = a := by
  rw [← h]
  rfl
```

We choose the rewrite direction by inspecting which substitution moves the goal towards a simpler or already known form.

1.7 Induction

`add m zero = m` reduces immediately because the second argument is explicitly `zero`. Lean selects the first branch of `add`, and both sides become identical.

In `add zero n = n`, by contrast, the second argument is the variable `n`. The theorem does not assign a concrete value to `n`: it must hold for any natural number. Lean therefore cannot directly reduce `add zero n`, because it does not yet know whether to use the `zero` branch or the `succ` branch.

Checking `zero`, `one`, `two`, and further examples may suggest that the statement is correct, but it does not prove the result for every finite chain of successors. We need an argument that covers all values generated by the constructors of `Natural`.

A proof by induction follows precisely these constructors. In the *base case*, we prove the property for `zero`. In the *inductive step*, we consider a number of the form `succ n`: we assume that the property holds for `n` and prove that it also holds for `succ n`. The assumption received in this step is the *induction hypothesis*.

These two cases are sufficient because every natural number is obtained by starting from `zero` and applying `succ` finitely many times. Induction thus gives the initial case and a way to transfer the result from each natural number to its successor.

In the proof script, `induction n with` creates one branch for each constructor. The `zero` branch must prove `add zero zero = zero`, which follows directly from the definition. The `succ n ih` branch introduces both the preceding natural number `n` and the induction hypothesis `ih : add zero n = n`.

In the step below, `rewrite [ih]` makes the same substitution as `rw [ih]`. We use the longer form here because it leaves the resulting goal visible, and we close it explicitly

with `rfl`. `rw` can instead close a goal automatically when rewriting leaves a goal of the form `t = t`.

```
open Natural

theorem lecture06_zero_add (n : Natural) :
  add zero n = n := by
  induction n with
  | zero =>
    rfl
  | succ n ih =>
    unfold add
    rewrite [ih]
    rfl
```

Finally, let us evaluate how we choose the induction variable. The definition of `add` analyses its second argument. In the goal `add zero n = n`, that position contains `n`, and its unknown form is precisely what blocks computation. We therefore induct on `n`.

In the successor case, `unfold add` exposes the equation `add zero (succ n) = succ (add zero n)`. The goal becomes `succ (add zero n) = succ n`. The induction hypothesis concerns exactly the remaining subexpression `add zero n`, and rewriting replaces it with `n`. The new goal is `succ n = succ n`, which `rfl` proves.

1.8 Sources

- Lean developers, *Theorem Proving in Lean 4*, «Inductive Types»: https://lean-lang.org/theorem_proving_in_lean4/Inductive-Types/.
- Lean developers, *Theorem Proving in Lean 4*, «Induction and Recursion»: https://lean-lang.org/theorem_proving_in_lean4/Induction-and-Recursion/.
- Lean developers, *Functional Programming in Lean*, «Datatypes and Patterns»: https://lean-lang.org/functional_programming_in_lean/Getting-to-Know-Lean/Datatypes-and-Patterns/.
- Lean developers, *The Lean Language Reference*, «Recursive Definitions»: <https://lean-lang.org/doc/reference/latest/Definitions/Recursive-Definitions/>.
- Lean community, *Natural Number Game*: <https://adam.math.hhu.de/>.