Inductive and recursive proofs

CMPUT 664 · Formal Verification · October 19, 2026

Checking the result of automation

A tactic constructs a **proof term**. Lean's kernel checks that the term has the theorem's type.

For fixed propositions P and Q:

$$f : P \to Q, \qquad p : P, \qquad f\ p : Q.$$

Now consider a claim about a program's input and output:

"The list returned by `zeros n` has length `n`."

What type would a proof of this claim have?

A length contract

We will verify the length of a function that builds lists of zeros:

```
1 zeros 2 = [0, 0]
2 zeros 5 = [0, 0, 0, 0, 0]
```

Its ordinary interface is:

```
1 zeros : Nat → List Nat
```

Would a function that always returns `[]` satisfy this type?

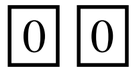
Yes. To express the length requirement, we need to relate the result to `n`:

$$\text{length}(\text{zeros}(n)) = n.$$

Let's first see how a length can appear inside a type.

Lengths in types

`Vector Nat n` describes a collection of exactly `n` natural numbers.

Result type	Example contents
`Vector Nat 2`	
`Vector Nat 5`	

Lean's library already provides this construction:

```
1 def zerosVector (n : Nat) : Vector Nat n :=
2   Vector.replicate n 0
```

Giving it `2` selects the result type `Vector Nat 2`; giving it `5` selects `Vector Nat 5`.

Dependent function types

The vector example has a **dependent function type**: its result type mentions the input value.

```
1 (n : Nat) → Vector Nat n
```

The name `n` before the arrow lets the result type refer to that same input.

In general, $(x : A) \to B(x)$ takes $a : A$ and returns a term of type $B(a)$.

An ordinary $A \to B$ keeps the result type B fixed, even when the returned value changes.

Which result type depends on `n`?

Interface	Result type
`(n : Nat) → Nat`	`Nat`
`(n : Nat) → Vector Nat n`	`Vector Nat n`

The type of the length proof

For a function `zeros : Nat → List Nat`, we can express the length guarantee as a separate theorem.

Using propositions as types, the proof we must construct has type:

```
1 (n : Nat) → (zeros n).length = n
```

Read this as: **given any `n`, supply a proof that `zeros n` has length `n`.**

The input determines which proposition the returned proof must establish:

Input	Required proof type
`2`	`(zeros 2).length = 2`
`5`	`(zeros 5).length = 5`

Predicates and their proofs

To give this family of length claims a short name, write:

$$P(n) \equiv \text{length}(\text{zeros}(n)) = n.$$

A **predicate** such as `P : Nat → Prop` assigns a proposition to each input.

Expression	Meaning
`P 3 : Prop`	The claim about input `3`
`h : (n : Nat) → P n`	Given `n`, supply evidence of `P n`
`h 3 : P 3`	Evidence of the claim about input `3`

What is the type of `h 4`? What length claim does it prove?

Universal proofs as functions

A proof that works for every input has two equivalent readings:

$$\forall n : \mathbb{N}.\ P(n) \qquad \text{and} \qquad (n : \mathbb{N}) \to P(n).$$

To construct it, take an **arbitrary** `n` and produce evidence of `P n`.

To use it, supply an input: `h 4` proves `P 4`.

For our length contract, the proof we want therefore has type:

```
1 ∀ n : Nat, (zeros n).length = n
```

Dependent type theory

Dependent type theory is the framework behind these examples: types can mention values, and terms must meet the requirements of their types.

Type	What a term of that type supplies
`Vector Nat n`	A collection of length `n`
`P n`	Evidence of the proposition at `n`
`(n : Nat) → P n`	Evidence of `P n` for any input `n`

Lean's kernel checks proofs using these type-theoretic rules.

With `Vector.replicate n 0`, we use a library function whose result type already guarantees the length.

Now we will write a recursive `zeros` returning a `List Nat` and prove its length contract ourselves.

How does the recursive definition guide that proof?

Defining zeros by cases

For our recursive version, let the input determine how we build the list:

- If `n = 0`, return the empty list.
- If `n = k + 1`, add one zero to the list built for `k`.

Lean expresses these two cases using the **constructors** of `Nat`:

Constructor	Meaning
`Nat.zero`, written `0`	Zero
`Nat.succ k`	The successor of `k`, or `k + 1`

For example, the input `3` is three successors of zero:

```
1 3 = Nat.succ (Nat.succ (Nat.succ 0))
```

A recursive implementation

`[]` is the empty list. `x :: xs` adds the element `x` to the front of `xs`.

```
1 def zeros : Nat → List Nat
2   | 0 => []
3   | Nat.succ k => 0 :: zeros k
```

For example:

```
1 zeros 3 = 0 :: zeros 2
2         = 0 :: 0 :: zeros 1
3         = 0 :: 0 :: 0 :: zeros 0
4         = [0, 0, 0]
```

Each call uses the smaller input `k`. Lean accepts this **structural recursion** because the input decreases toward the base case.

From one computation to every input

The trace of `zeros 3` already gives us a proof of its length:

```
1 example : (zeros 3).length = 3 := by
2   rfl
```

`rfl` succeeds because computing both sides gives `3 = 3`.

We want the same guarantee for **every** input:

```
1 ∀ n : Nat, (zeros n).length = n
```

For an arbitrary `n`, the computation has to wait: is `n` zero or a successor?

Let’s follow those two cases in the definition.

Following the recursive definition

The clauses of `zeros` tell us how to begin the length proof.

At `0`: the result is `[]`, so its length is 0.

At `Nat.succ k`: unfold one step using the defining equation

```
1 zeros (Nat.succ k) = 0 :: zeros k
```

The length goal becomes:

$$\text{length}(\text{zeros}(k)) + 1 = k + 1.$$

We can finish this step if we know that `zeros k` has length `k`.

That is the same length claim, at the smaller input.

Induction on the input

Induction supplies the smaller-input fact needed by the recursive step.

Let $P(n)$ mean $\text{length}(\text{zeros}(n)) = n$.

Base case: prove $P(0)$.

`zeros 0` is `[]`, whose length is 0.

Successor case: assume $P(k)$ and prove $P(k + 1)$.

The **induction hypothesis** says that `zeros k` has length `k`. Adding one zero gives the required length `k + 1`.

Starting with the base proof, the successor step builds a proof for 1, then 2, then 3, and so on. Every natural number is reached this way.

The successor argument

Here is the step that builds the proof at `k + 1` from the proof at `k`.

Induction hypothesis: $\text{length}(\text{zeros}(k)) = k.$

Goal: $\text{length}(\text{zeros}(k + 1)) = k + 1.$

$$\begin{aligned}\text{length}(\text{zeros}(k + 1)) &= \text{length}(0 :: \text{zeros}(k)) \\ &= \text{length}(\text{zeros}(k)) + 1 \\ &= k + 1.\end{aligned}$$

Unfold `zeros`, compute the length of a nonempty list, then use the induction hypothesis.

Induction in Lean

The proof follows the same base and successor cases:

```
1 theorem zerosLength :
2     ∀ n : Nat, (zeros n).length = n := by
3   intro n
4   induction n with
5   | zero =>
6       rfl
7   | succ k ih =>
8       change (zeros k).length + 1 = k + 1
9       rw [ih]
```

`induction` supplies an induction hypothesis in the successor case.

Using the induction hypothesis

In the successor branch, the context and goal are:

```
1 k  : Nat
2 ih : (zeros k).length = k
3 ⊢ (zeros (Nat.succ k)).length = Nat.succ k
```

`change` exposes a definitionally equal goal:

```
1 ⊢ (zeros k).length + 1 = k + 1
```

`rw [ih]` replaces `(zeros k).length` by `k`. The resulting equality closes by reflexivity.

The hypothesis proves the contract at `k`. This branch constructs its proof at `k + 1`.

Proofs defined recursively

We can also write the proof directly as the dependent function we wanted:

```
1 theorem zerosLengthRec : (n : Nat) → (zeros n).length = n
2   | 0 => rfl
3   | Nat.succ k => by
4       change (zeros k).length + 1 = k + 1
5       rw [zerosLengthRec k]
```

`zerosLengthRec k` provides the smaller proof that we previously called `ih`. Its input decreases from `Nat.succ k` to `k`.

We now have the evidence-producing function introduced at the start:

```
1 zerosLengthRec 5 : (zeros 5).length = 5
```

A recursive function with an accumulator

What changes in the proof if we build the list in an extra argument?

This **accumulator** carries the partial result through the recursive calls:

```
1 def zerosAcc : Nat → List Nat → List Nat
2   | 0, acc => acc
3   | Nat.succ k, acc => zerosAcc k (0 :: acc)
```

Starting with two elements:

```
1 zerosAcc 2 [7, 8]
2   = zerosAcc 1 [0, 7, 8]
3   = zerosAcc 0 [0, 0, 7, 8]
4   = [0, 0, 7, 8]
```

We add n elements to the starting list, so the length contract is:

```
1 (zerosAcc n acc).length = n + acc.length
```

The changing accumulator

Try the same induction on `n`. After unfolding the successor branch, we need a fact about `zerosAcc k (0 :: acc)`.

If we keep `acc` fixed during induction, we get:

```
1 ih : (zerosAcc k acc).length = k + acc.length
```

This hypothesis talks about `acc`, but the call uses `0 :: acc`.

To cover that call, prove the property for **every accumulator** at each `n`:

```
1 ih : ∀ acc : List Nat,
2   (zerosAcc k acc).length = k + acc.length
```

Then `ih (0 :: acc)` gives the length of the recursive result.

Applying the hypothesis to the recursive call

In Lean, `induction n generalizing acc` gives a successor-case hypothesis that accepts **any accumulator**:

```
1 ih : ∀ a : List Nat, (zerosAcc k a).length = k + a.length
```

Supply the accumulator used by the recursive call:

```
1 ih (0 :: acc) :
2   (zerosAcc k (0 :: acc)).length = k + (0 :: acc).length
```

After unfolding `zerosAcc`, use `rw [ih (0 :: acc)]` and compute the length of `0 :: acc`. The goal reduces to:

$$k + (\text{length}(acc) + 1) = (k + 1) + \text{length}(acc).$$

Both sides count the original elements plus `k + 1` added zeros.

Using the stronger lemma

Call the resulting theorem `zerosAccLength`. It applies to every starting accumulator. To build a list from scratch, choose `[]`:

```
1 zerosAccLength n [] :
2   (zerosAcc n []).length = n + [].length
```

The empty list has length 0, giving the required length `n`:

```
1 theorem zerosAccEmptyLength (n : Nat) :
2     (zerosAcc n []).length = n := by
3   simpa using zerosAccLength n []
```

`simpa using` simplifies the supplied theorem's type to the required goal.

Proving properties of recursive functions

Both length proofs follow the same approach:

1. **State a property of the result** for arbitrary inputs.
2. **Inspect the recursive calls** to choose the induction argument and any parameters to generalize.
3. **Unfold the relevant defining equation** in each case.
4. **Use the induction hypothesis at the actual recursive arguments.**
5. **Prove the remaining local step**, using rewriting or suitable automation.

For `zeros`, the smaller-input fact concerns `zeros k`.

For `zerosAcc`, it must cover `zerosAcc k (0 :: acc)`.

Reviewing recursive proofs

If `h : ∀ n : Nat, P n`, what is the type of `h 4`?

`P 4`: evidence of the property at that input.

After unfolding `zeros (Nat.succ k)`, which fact lets us finish the length proof?

The induction hypothesis: `(zeros k).length = k`.

Why does the accumulator proof need `generalizing acc`?

The recursive call uses `0 :: acc`. The hypothesis must apply at that changed argument.

Optional example: recursion on lists

The same approach applies when the recursive argument is a list:

```
1 def append : List Nat → List Nat → List Nat
2   | [], ys => ys
3   | x :: xs, ys => x :: append xs ys
```

The recursive call uses the tail of the first list. The second list stays fixed.

To prove `append xs [] = xs`, induct on `xs`.

In the nonempty case, the hypothesis proves `append xs [] = xs`. Unfolding one step leaves the same head `x` and that smaller append call.

Optional example: proving the list property

The defining equations determine the two proof cases:

```
1 theorem appendNil (xs : List Nat) : append xs [] = xs := by
2   induction xs with
3   | nil =>
4       rfl
5   | cons x xs ih =>
6       change x :: append xs [] = x :: xs
7       rw [ih]
```

The induction hypothesis rewrites the recursive result. The same head `x` remains on both sides.