Leanduction from scratch?

Cheni Yuki Yang, Nicolas Arnold

26.07.2023

Table of Contents

Induction

In type theory, each inductive type can be assigned to a related induction principle.

- The natural numbers
- The binary tree
- A context-free grammar

Proof without Induction Tactic

Given an easy example: $\forall n : \mathbb{N}, n + 1 > n$.

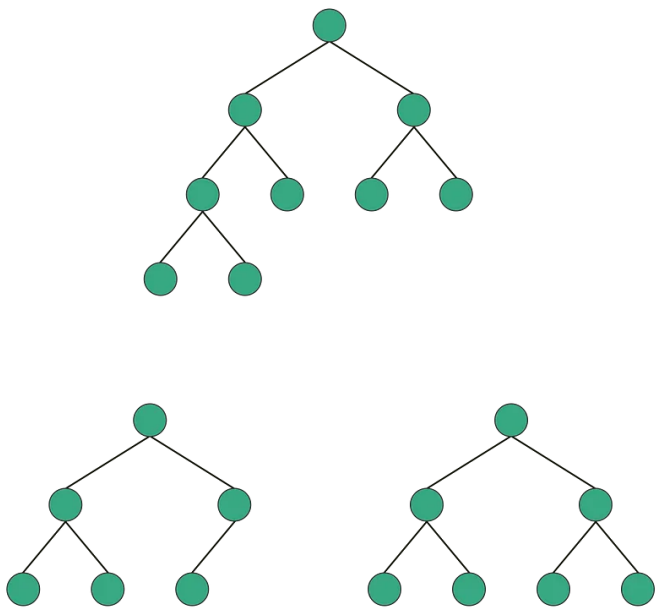
We implemented Peano numbers, some basic functions and properties.

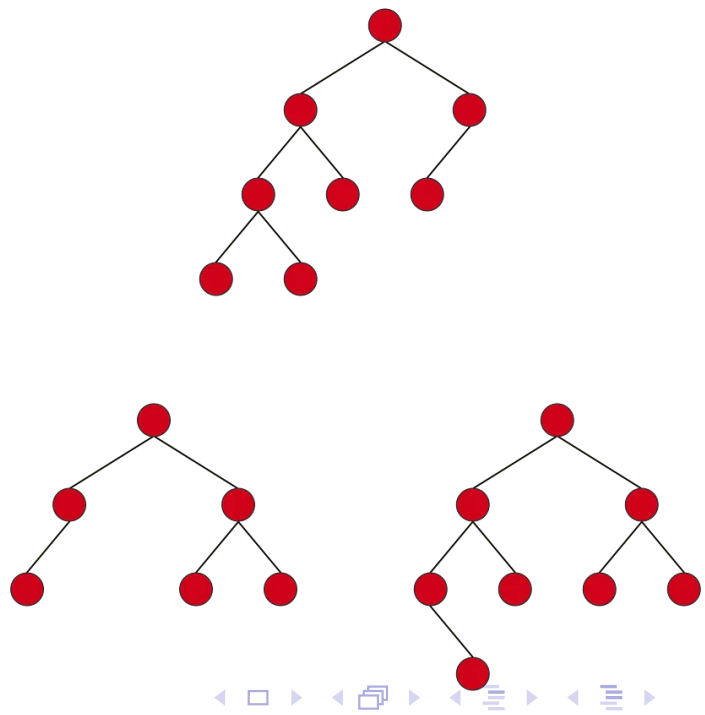
Induction Principle

The principle for N is easy to guess, but what is the principle for a Tree?
The generation of induction principles is computable! This leads us to the tactic "induction".

(Complete) Binary Trees

A Tree is a directed graph with no cycles. In a **binary** Tree each node has at most **two** children. Nodes with no child are called **Leaf**. In a binary heap (**complete binary tree**) all nodes appear *from left to right*, so the height is kept minimal. (list, array)





Binary Trees as a recursive data type

We use Haskell for prototyping because it has a very lean syntax.

tree.hs

```haskell
data BinaryTree a
  = Empty    -- Leaf
  | Node a (BinaryTree a) (BinaryTree a)
```

Figure: Binary Tree in Haskell, recursive

Height of a complete binary Tree

Proof upper bound for number of nodes n in a binary Tree with a given height h:

$$n \leq 2^{h+1} - 1$$

The height of any Tree is the number of edges of the longest path from root to leaf.

Thus the height of a binary Heap is just $\lfloor log_2 \rfloor n$.

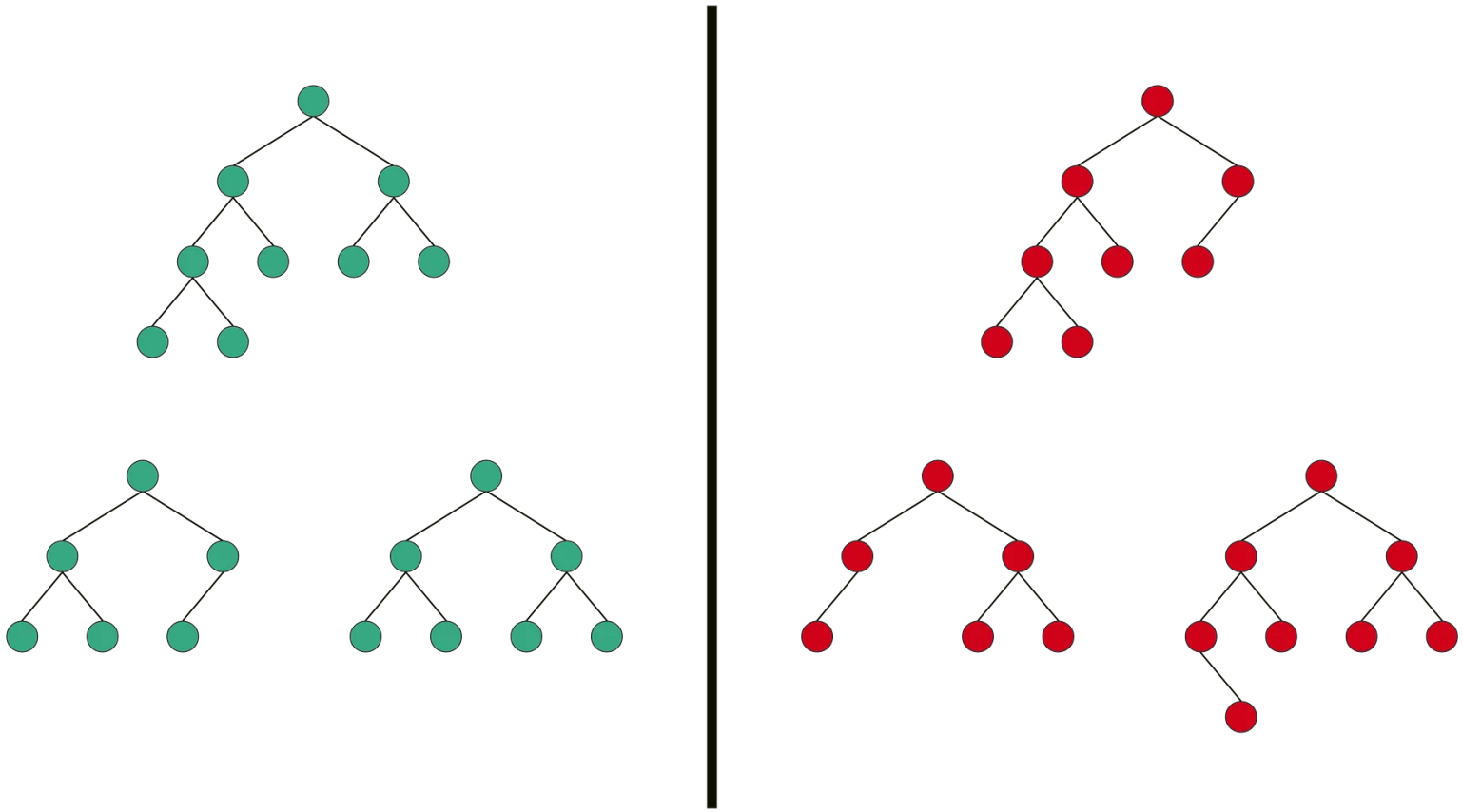
Integer Log base 2

Of course we can't use mathlib's log on our own Peano numbers and we don't want to implement real-based log.

What can we do to calculate the height of a complete binary Tree and implement our own Integer log-function?

Integer Log base 2 implementation 1/3

The leftmost path is always the longest path, in a complete binary Tree.



Integer Log base 2 implementation 2/3

So we can simply calculate the height recursively by pattern matching on our data structure:

```
heapHeight :: BinaryTree a -> Int
heapHeight Empty = -1
heapHeight (Node a left right) = 1 + heapHeight left
```

Figure: heapHeight, computing height recursively

We replace each node with 1 until we reach a Leaf (compensating for leaf edge) and sum it up.

Integer Log base 2 implementation 3/3

We can implement log_n by building a n-ary heap and compute it's height.

This takes time...

Now we've implemented an Integer log function based on the recursive data structure of our interest.

As you can see, DIY takes quite some effort...

So at some point we left this path and used internal Nat and it's properties in order to use induction tactic on our custom BinaryTree data structure.

Itchy and Scratchy

From taking the DIY path we learned it can be a fruitful journey to think closely about data structures and define their properties directly instead of just relying on the abstractions provided by powerful libraries.

Our thinking becomes deeply connected to the recursive nature of the data structure, which might inspire new proof ideas / shortcuts and improve our understanding of the nature of the problem.

And we precisely see, what definitions are necessary for the proof to work, if we provide our own data structures and functions.

Induction Tactic

The induction tactic only does two things:

- apply the principle on certain variables
- split the context into more cases

We can write our own induction tactic with some programming skills (OCaml).

A long Journey of Treeduction

There is an upper bound for the number of nodes in a binary tree (complete binary tree).
Given following proposition: $\forall t : \text{Tree}, |t| \leq 2^{H(t)} - 1$

Combinators

Combinators combine tactics and cases together to save the redundant codes.

- all_goals and any_goals
- repeat
- try

Higher Tactics

Some user-defined tactics(especially in Mathlib) help us solve some trivial goals easily.

- ring
- omega
- split_ifs

Table of Contents

Types of Equality

- In Martin-Löf type theory $\equiv$
- Leibniz equality $\doteq$
- In Homotopy type theory $\cong$

The Martin-Löf equality is equivalent to the Leibniz equality, which should be hold in every type system!
$\forall (A : \text{Type})(x\ y : A), x \doteq y \iff x \equiv y.$

The Reverse of Leibniz?

The function extensionality can be proven on the quotient of setoid of functions. It also works when we set it just as a new axiom of the system.
$\forall(A : \text{Type})(B : A \to \text{Type})(f_1\ f_2 : \Pi x : A, Bx), \forall(x : A), f_1(x) = f_2(x) \implies f_1 = f_2.$

Table of Contents

References

 Abel et al.
Leibniz equality is isomorphic to Martin-Löf identity, parametrically.

 LeanCommunity
Tutorial: tactic writing in Lean.

 J, Avigad.
A proof of function extensionality from quotients.