CPSC-354 Report

Connor Jacobs
Chapman University

October 10, 2024

Abstract

This report will contain a summary of my learning progress throughout the course, so far including: lean proofs

Contents

1 Introduction

This report will document my learning through the course. The content will be structured week by week, with sections on mathematical notes, homework solutions, and personal reflections on the topics discussed.

2 Week by Week

2.1 Week 1

Mathematical Proof

Proving that $37x + q = 37x + q$ demonstrates the reflexivity property of equality. Reflexivity states that any mathematical expression is equal to itself. In this case, the expression $37x + q$ is compared to itself, and it is immediately clear by the reflexivity of equality that this statement is true.

Proof in Lean

In Lean, the theorem $37x + q = 37x + q$ is proven using the 'rfl' tactic. The 'rfl' tactic in Lean stands for "reflexivity" and handles proofs of the form $X = X$. When 'rfl' is executed, Lean verifies that both sides of the equation are equal and the proof is complete.

The command to execute in Lean is simply:

```
rfl
```

Connection Between Lean and Mathematics

In mathematics, we rely on the axiom of reflexivity to assert that $37x + q = 37x + q$. Likewise, in Lean, the 'rfl' tactic automates this process by invoking the same principle. The use of 'rfl' in Lean serves as a direct representation of reflexivity in mathematical logic, providing an automated and formalized way to conclude that an expression is equal to itself.

Thus, both in Lean and in traditional mathematics, the proof of $37x + q = 37x + q$ is an application of the same fundamental concept: the reflexivity of equality.

Answers to Levels 5-8

Level 5

```
rw [add_zero]
rw [add_zero]
rfl
```

Level 6

```
rw [add_zero c]
rw [add_zero b]
rfl
```

Level 7

```
rw [one_eq_succ_zero]
rw [add_succ]
rw [add_zero]
rfl
```

Level 8

```
rw [two_eq_succ_one]
rw [one_eq_succ_zero]
rw [four_eq_succ_three]
rw [three_eq_succ_two]
rw [two_eq_succ_one]
rw [one_eq_succ_zero]
rw [add_succ]
rw [add_succ]
rw [add_zero]
rfl
```

2.2 Week 2

Mathematical Proofs and Lean Implementation

Theorem (Associativity of Addition): For all natural numbers a, b, and c, the addition operation is associative: $(a + b) + c = a + (b + c)$.

Proof: By induction on c:

- **Base Case:** When $c = 0$, we have:

  $$(a + b) + 0 = a + (b + 0)$$

  Using the property addition with zero, this simplifies to:

  $$a + b = a + b$$

  This equality holds by the reflexivity of equality

- **Inductive Step:** Assume the hypothesis holds for some d

  $$(a + b) + d = a + (b + d)$$

  We must prove $c = \text{succ}(d)$:

  $$(a + b) + \text{succ}(d) = a + (b + \text{succ}(d))$$

  this can be rewritten as:

  $$\text{succ}((a + b) + d) = \text{succ}(a + (b + d))$$

  By the inductive hypothesis, we know $(a + b) + d = a + (b + d)$, so:

  $$\text{succ}((a + b) + d) = \text{succ}(a + (b + d))$$

  This completes the proof of associativity.

Connection to Lean Proof of Associativity of Addition:

The Lean proof is the same structure of the mathematical proof:

```
induction c with d hd
-- Base case
rw [add_zero]  -- Simplifies (a + b) + 0 to a + b using add_zero
rw [add_zero]  -- Simplifies a + (b + 0) to a + b using add_zero
rfl            -- Reflexivity: confirms a + b = a + b
-- Inductive step
rw [add_succ]  -- Rewrites (a + b) + succ d as succ ((a + b) + d)
rw [hd]        -- Applies the inductive hypothesis: (a + b) + d = a + (b + d)
rw [add_succ]  -- Rewrites a + (b + succ d) as succ (a + (b + d))
rw [add_succ]  -- Confirms that the expressions on both sides match
rfl            -- Reflexivity: confirms succ (a + (b + d)) = succ (a + (b + d))
```

Explanation:

- **Base Case in Lean:**
  - `rw [add_zero]` is used twice to simplify both sides of the equation to $a+b$, directly corresponding to the mathematical step where $a + b = a + b$ is shown by reflexivity.
  - `rfl` is then used to verify
- **Inductive Step in Lean:**

- `rw [add_succ]` rewrites the expression using the definition of addition with the successor, corresponding to the mathematical step where we transition from $(a+b)+\text{succ}(d)$ to $\text{succ}((a+b)+d)$.
- `rw [hd]` applies the inductive hypothesis, mirroring the assumption that $(a+b)+d = a+(b+d)$.

Week 2 questions

Question: Can Lean be used to verify complex operations in critical systems such as financial transactions or autonomous vehicles? What are the limitations of its use in such a high stakes environment?

2.3 Week 3

Discord Name: connorjacobs

Discord Posting: In my research, I took a look at how advancements in programming languages have balanced performance, safety, and ease of use, focusing on Rust and TypeScript. Rust's ownership model enforces memory safety by controlling how data is accessed and managed, preventing issues like memory leaks and race conditions without garbage collection. In contrast, TypeScript enhances JavaScript by introducing type checking, catching type errors early, and improving overall code reliability.

Link to ReadMe:
https://github.com/Conjac76/WeekThree354/blob/main/README.md

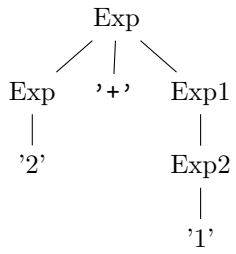
Links to two reviews I voted for:
https://github.com/tannerplatt/TannerHW3_354/blob/main/README.md
https://github.com/tyedwards37/Using-LLM-for-Literature-Review

2.4 Week 4

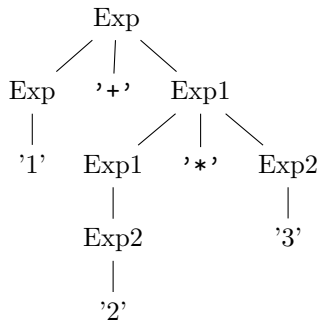
Derivation Trees

Below are the derivation trees for the given strings using the provided context-free grammar:

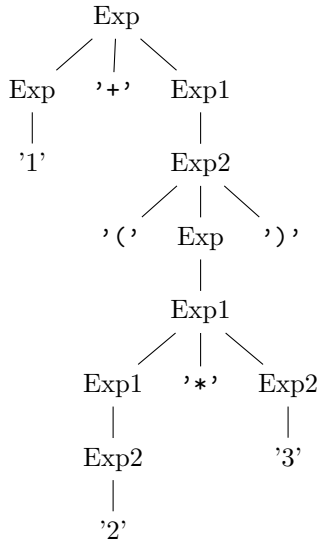
1. Derivation Tree for $2+1$



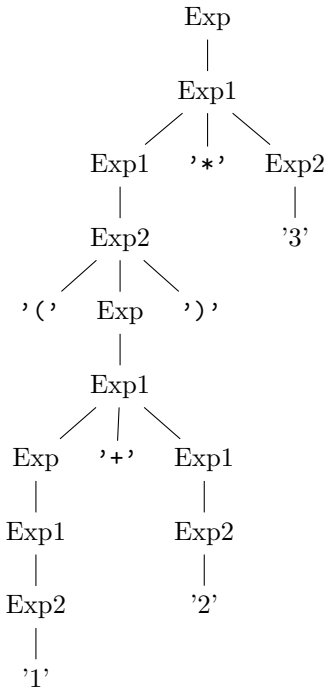
2. Derivation Tree for $1 + 2 * 3$



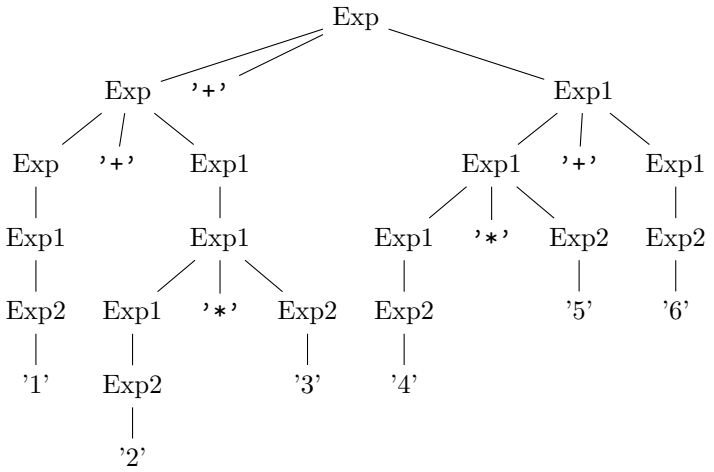
3. Derivation Tree for $1 + (2 * 3)$



4. Derivation Tree for $(1 + 2) * 3$



5. Derivation Tree for $1 + 2 * 3 + 4 * 5 + 6$



2.5 Week 5

Solutions to Levels 1-8 (Lean Syntax)

Level 1

```
exact todo_list
```

Level 2

```
exact and_intro p s
```

Level 3

```
exact and_intro (and_intro a i) (and_intro o u)
```

Level 4

```
have p := and_left vm
have p := vm.left
exact p
```

Level 5

```
exact h.right
```

Level 6

```
exact and_intro (and_left h1) (and_right h2)
```

Level 7

```
have h1 := h.left
have h2 := h1.right
have h3 := h2.left
have h4 := h3.left
have h5 := h4.right
exact h5
```

Level 8

```
have h1 := h.left
have h2 := h1.left
have h3 := h1.right
have h4 := h.right.right.left
have h5 := h4.left
exact and_intro h3 (and_intro h5 (and_intro h2.left h2.right))
```

Level 8: Proof in Mathematical Logic

If $P \wedge (Q \wedge R) \wedge ((S \wedge T) \wedge (U \wedge V))$, then $P \wedge (S \wedge U \wedge V)$.

(1)	$P \wedge (Q \wedge R) \wedge ((S \wedge T) \wedge (U \wedge V))$	assumption
(2)	$P \wedge (Q \wedge R)$	and_left (1)
(3)	P	and_left (2)
(4)	$(S \wedge T) \wedge (U \wedge V)$	and_right (1)
(5)	$S \wedge T$	and_left (4)
(6)	$U \wedge V$	and_right (4)
(7)	S	and_left (5)
(8)	U	and_left (6)
(9)	V	and_right (6)
(10)	$P \wedge (S \wedge U \wedge V)$	and_intro (3) (7) (8) (9)

2.6 Week 6

Level 1

```
have c : C := bakery_service p
exact c
```

Level 2

```
exact /lambda h : C -> h
```

Level 3

```
exact /lamba h : I ^ S -> ⟨h.right, h.left⟩
```

Level 4

```
exact /lamba c : C -> h2 (h1 c)
```

Level 5

```
have q : Q := h1 p
have t : T := h3 q
have u : U := h5 t
exact u
```

Level 6

```
exact fun c : C => fun d : D => h ⟨c, d⟩
```

Level 7

```
exact fun cd : C ^ D => h cd.left cd.right
```

Level 8

```
exact fun s : S => ⟨h.left s, h.right s⟩
```

Level 9

```
exact fun r : R => ⟨fun s : S => r, fun ns : ¬S => r⟩
```

Question:

Lambda calculus serves as a foundational framework for computation and formal logic. How can lambda calculus be used to reason about mnore advanced concepts, such as polymorphism?

2.7 Week 7

Question 1:

Reduce the following lambda term:

$$((\lambda m.\lambda n.m\,n)\,(\lambda f.\lambda x.f\,(f\,x)))\,(\lambda f.\lambda x.f\,(f\,(f\,x)))$$

Step 1:

$$(\lambda m.\lambda n.m\,n)\,(\lambda f.\lambda x.f\,(f\,x)) \to \lambda n.(\lambda f.\lambda x.f\,(f\,x))\,n$$

Step 2:

$$\lambda n.(\lambda f.\lambda x.f\,(f\,x))\,n\,(\lambda f.\lambda x.f\,(f\,(f\,x))) \to (\lambda f.\lambda x.f\,(f\,x))\,(\lambda f.\lambda x.f\,(f\,(f\,x)))$$

Step 3:

$$\lambda x.((\lambda f.\lambda x.f\,(f\,(f\,x))))\,((\lambda f.\lambda x.f\,(f\,(f\,x)))\,x)$$

$$(\lambda f.\lambda x.f\,(f\,x))\,(\lambda f.\lambda x.f\,(f\,(f\,x))) \to \lambda x.[(\lambda f.\lambda x.f\,(f\,(f\,x)))]\,((\lambda f.\lambda x.f\,(f\,(f\,x)))\,x)$$

Step 4:

$$(\lambda f.\lambda x.f\,(f\,(f\,x)))\,x \to \lambda x.x\,(x\,(x\,x))$$

$$(\lambda f.\lambda x.f\,(f\,(f\,x)))\,x \to x\,(x\,(x\,x))$$

Step 5:

$$\lambda x.((\lambda f.\lambda x.f\,(f\,(f\,x))))\,(x\,(x\,(x\,x)))$$

Step 6:

$$\lambda x.(x\,(x\,(x\,x)))\,(x\,(x\,(x\,x))\,x)$$

Step 7:

$$\lambda x.(x\,(x\,(x\,x)))\,(x\,(x\,(x\,x\,x)))$$

the function is applying x six times, which corresponds to the Church numeral 6.

Question 2:

Explain what function on natural numbers $\lambda m.\lambda n.m\,n$ implements.

Answer:

The lambda term $\lambda m.\lambda n.m\,n$ represents the multiplication operation in Church numerals.

Explanation: In Church encoding, natural numbers are represented as repeated applications of a function. The numeral n is encoded as $\lambda f.\lambda x.f^n\,x$. The term $m\,n$ applies the function m to n. Since m represents applying a function f m times, and n represents applying f n times, their composition results in $m \times n$ applications of f. Therefore, $\lambda m.\lambda n.m\,n$ takes two Church numerals and returns their product.

Question for Discord:

Is it possible to represent data structures like pairs or lists using Church numerals and lambda calculus? If so, how do these representations work?

3 Lessons from the Assignments

3.1 Key Lessons

The assignments taught the foundational properties in mathematics, such as reflexivity, commutativity, and associativity, and their implementation in Lean. Lean's tactics like 'rfl' and 'rw' simplify proofs by automating logical steps, making it easier to verify mathematical statements.

4 Conclusion

I have learned the connection between mathematical proofs and Lean verification.

References

[BLA] Author, Title, Publisher, Year.