CPSC-354 Report

Connor Jacobs
Chapman University

December 1, 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

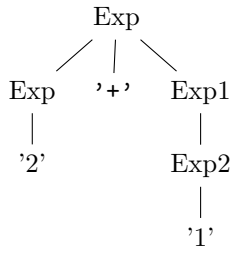
Links to two reviews I voted for:
https://github.com/tannerplatt/TannerHW3_354/blob/main/README.md
https://github.com/tyedwards37/Using-LLM-for-Literature-Review

2.4 Week 4

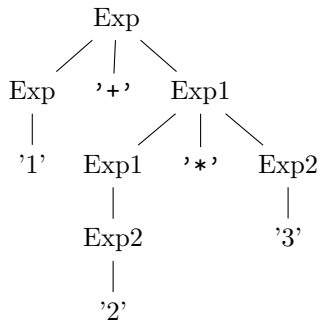
Derivation Trees

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

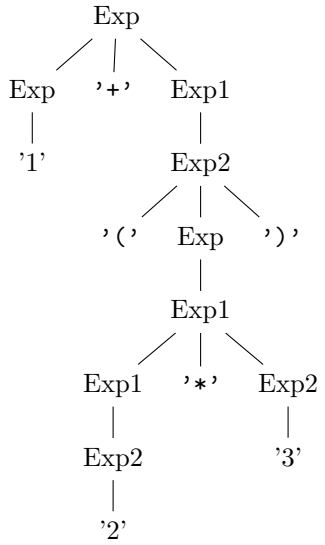
1. Derivation Tree for $2 + 1$



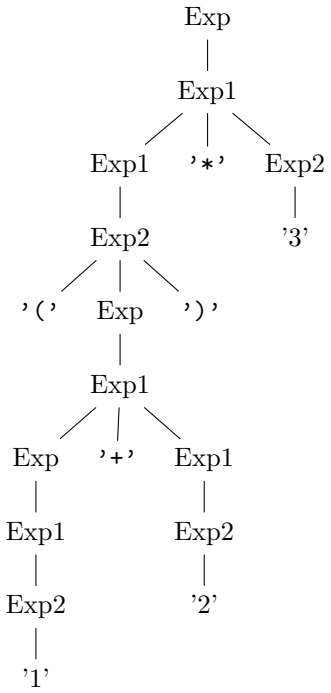
2. Derivation Tree for $1 + 2 * 3$



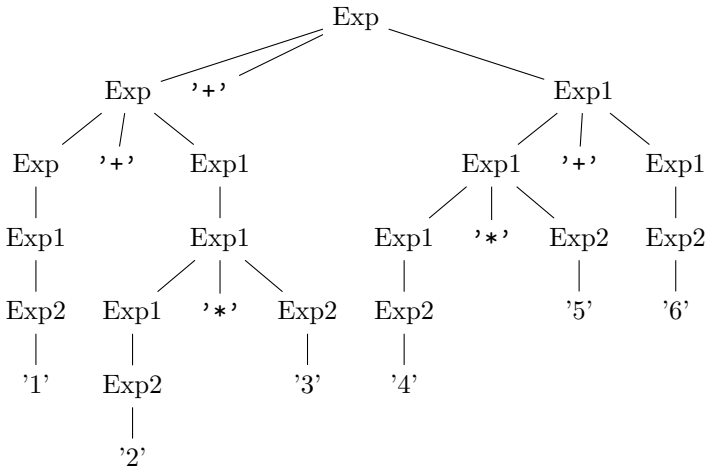
3. Derivation Tree for $1 + (2 * 3)$



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?

2.8 Week 8/9

Question 2.

The expression `a b c d` reduces to `(((a b) c) d)` becuase of the way lambda calculus handles application. Application is left associative. The function `a` is applied to `b`, then the result is applied to `c`, and finally, the result is applied to `d`. This can be written as:

$$(((a\,b)\,c)\,d)$$

Question 3.

In lambda calculus, capture-avoiding substitution prevents bound variable names from unintentionally clashing with free variables. This is implemented in the `substitute` function, where:

- **Variables**: If a variable matches the name being substituted, it is replaced with the new expression; otherwise, it remains unchanged.

- **Lambda Abstractions**: If the bound variable matches the name being substituted, substitution is skipped for that scope. If not, a fresh name is generated using the `NameGenerator` to avoid conflicts, and substitution proceeds recursively.

- **Applications**: Substitution occurs recursively in both the function and argument.

Example 1 (No Name Conflict):

```
(\x. \y. x) z
```

Reduces to:

$$\lambda y.z$$

Example 2 (Name Conflict):

```
(\x. \y. x) y
```

Reduces to:

$$\lambda \text{Var1}.y$$

where `Var1` is a fresh variable generated to avoid conflict with `y`.

Question 4. No, not all computations in lambda calculus reduce to normal form, and there are situations where evaluation may not terminate or result in a normal form.

Infinite Loops (Divergence): An expression like the following doesn't have a normal form:

```
(\x. x x) (\x. x x)
```

Question 5. Smallest lambda expression that does not reduce to normal form is known as the Omega combinator

```
(\x. x x) (\x. x x)
```

Question 7.

```
((\m.\n. m n) (\f.\x. f (f x))) (\f.\x. f (f (f x)))
((\Var1. (\f.\x. f (f x)) Var1) ) (\f.\x. f (f (f x)))
(\Var2. (\f.\x. f (f (f x))) ((\f.\x. f (f (f x))) Var2))
```

Question 8.

```
12: eval (((\m.(\n.(m n))) (\f.(\x.(f (f x))))) (\f.(\x.(f x))))
  39: eval ((\m.(\n.(m n))) (\f.(\x.(f (f x)))))
    39: eval (\m.(\n.(m n)))
    44: substitute ((\n.(m n)), 'm', (\f.(\x.(f (f x)))))
    45: eval (\Var1.((\f.(\x.(f (f x)))) Var1))
  44: substitute (((\f.(\x.(f (f x)))) Var1), 'Var1', (\f.(\x.(f (f (f x))))))
  45: eval ((\Var2.(\Var4.(Var2 (Var2 Var4)))) (\f.(\x.(f (f (f x))))))
    39: eval (\Var2.(\Var4.(Var2 (Var2 Var4))))
    44: substitute ((\Var4.(Var2 (Var2 Var4))), 'Var2', (\f.(\x.(f (f (f x))))))
    45: eval (\Var5.((\f.(\x.(f (f (f x))))) ((\f.(\x.(f (f (f x))))) Var5)))
```

Question on Discord: -how does left-associative application impact the sequence and structure of reduction steps when evaluating complex expressions

2.9 Week 10

1. **Most challenging aspect:** The biggest challenge was managing substitution in a way that avoided unintended variable capture, especially within deeply nested expressions. In the old 'evaluate' function, substitution was only partially recursive, which sometimes led to incorrect evaluations when multiple applications or nested abstractions were involved. Ensuring that each function application and lambda abstraction was fully reduced required a more comprehensive approach, where both sides of applications were evaluated recursively and abstractions were reduced in sequence. This transformation was complex but essential for producing a reliable evaluation function.
2. **Key insight for Assignment 3:** The key insight emerged from recognizing that the old 'evaluate' function only partially evaluated applications by handling a single argument at a time. In the new version, 'evaluate' was enhanced to handle both sides of a function application ('app') recursively, which allowed for a more complete evaluation. Furthermore, an additional case was added to evaluate lambda abstractions ('lam') by reducing their bodies, creating a more robust and normalized output. This recursive depth and abstraction evaluation were critical in managing nested applications and ensuring proper reduction to normal form.
3. **Most interesting takeaway:** One of the most interesting takeaways from this assignment was understanding the full recursive structure required for accurate lambda calculus evaluation.

Question on Discord: In what ways do modern interpreters and compilers handle complex recursive evaluations to ensure expressions reduce accurately

2.10 Week 11

Question 1: A = {}

Since the set A is empty, there is nothing to draw for this ARS.

Terminating: Yes

Confluent: Yes

Unique Normal Forms: Yes

Question 2: A = {a} and R = {}

$$a$$

Terminating: Yes

Confluent: Yes

Unique Normal Forms: Yes

Question 3: A = {a} and R = {a, a}

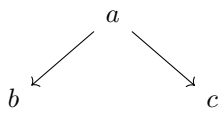
$$a \circlearrowleft$$

Terminating: No

Confluent: Yes

Unique Normal Forms: No

Question 4: A = {a, b, c} and R = {(a, b), (a, c)}

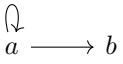


Terminating: Yes

Confluent: No

Unique Normal Forms: No

Question 5: A = {a, b} and R = {(a, a), (a, b)}

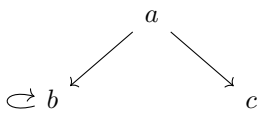


Terminating: No

Confluent: Yes

Unique Normal Forms: Yes

Question 6: A = {a, b, c} and R = {(a, b), (b, b), (a, c)}

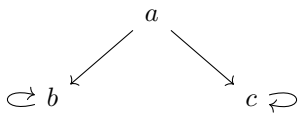


Terminating: No

Confluent: No

Unique Normal Forms: No

Question 7: A = {a, b, c} and R = {(a, b), (b, b), (a, c), (c, c)}



Terminating: No

Confluent: No

Unique Normal Forms: No

Homework: Try to find an example of an ARS for each of the possible 8 combinations. Draw pictures of these examples.

Confluent	Terminating	Has UNF	Example
True	True	True	$a \longrightarrow b$
True	True	False	N/A (Impossible)
True	False	True	$a \longrightarrow b \circlearrowright$
True	False	False	$a \circlearrowright$
False	True	True	N/A
False	True	False	a, $a \to b$, $a \to c$
False	False	True	N/A
False	False	False	$a \longrightarrow b \longrightarrow c$, $c \to a$

Question on Discord: Why is it impossible to have an ARS that is both Confluent and Terminating but does not have Unique Normal Forms?

2.11 Week 12

Starting Point:

$$\text{let rec fact} = \lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1) \text{ in fact } 3$$

Step 1: Expanding let rec

$$\text{let fact} = (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))) \text{ in fact } 3$$

Step 2: Expanding let

$$(\lambda \text{fact.fact } 3)(\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)))$$

Step 3: Applying fix

$$(\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))) \ 3$$

$$= (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))(\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))) \ 3$$

Step 4: Applying the function

$$(\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)))(n-1)) \ 3$$

Step 5: Computing if n = 0

$$\text{if } 3 = 0 \text{ then } 1 \text{ else } 3 \times (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)))(3-1)$$

$$= 3 \times (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))) \ 2$$

Step 6: Applying fix again

$$3 \times (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))(\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))) \ 2$$

Step 7: Applying the function

$$3 \times (\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)))(n-1)) \ 2$$

Step 8: Computing if n = 0 for n = 2

$$3 \times (\text{if } 2 = 0 \text{ then } 1 \text{ else } 2 \times (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)))(2-1))$$

$$= 3 \times (2 \times (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))) \ 1)$$

Step 9: Applying fix one more time

$$3 \times 2 \times (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))(\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))) \ 1$$

Step 10: Applying the function

$$3 \times 2 \times (\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)))(n-1)) \ 1$$

Step 11: Computing if n = 0 for n = 1

$$3 \times 2 \times (\text{if } 1 = 0 \text{ then } 1 \text{ else } 1 \times (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)))(1-1))$$

$$= 3 \times 2 \times (1 \times (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))) \ 0)$$

Step 12: Applying fix again

$$3 \times 2 \times 1 \times (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))(\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))) \ 0$$

Step 13: Applying the function

$$3 \times 2 \times 1 \times (\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)))(n-1)) \ 0$$

Step 14: Computing if n = 0 for n = 0

$$3 \times 2 \times 1 \times (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 \times (\text{fix } (\lambda \text{fact}.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)))(0-1))$$

$$= 3 \times 2 \times 1 \times 1 \quad (\text{since } 0 = 0, \text{ take the then branch})$$

Final Computation:

$$3 \times 2 \times 1 \times 1 = 6$$

The result is 6.

Question for discord:

are there specific cases where using recursion in lambda calculus can lead to inefficiencies that an iterative approach could solve more cleanly?

2.12 Week 13

Excercise 8: Consider the rewrite rules:

$$ab \to cc, \quad ac \to bb, \quad bc \to aa$$

plus rules saying that the order of letters does not matter. Starting from $15a$, $14b$, and $13c$, is it possible to reach a configuration in which there are only a's, only b's, or only c's?

Solution

1. **Invariant:** A helpful approach is to use an invariant that remains unchanged under the rewrite rules. By analyzing the given rules:

- $ab \to cc$: reduces $a + b$, increases c by $+2$.
- $ac \to bb$: reduces $a + c$, increases b by $+2$.
- $bc \to aa$: reduces $b + c$, increases a by $+2$.

The total number of letters $f(a, b, c) = a + b + c$ remains constant under all rules.

Initial value:

$$f(15, 14, 13) = 15 + 14 + 13 = 42.$$

Therefore, 42 letters must be preserved at all times.

2. **Parity Analysis:** Under each rule:

- $ab \to cc$: The count of $a + b + c$ remains unchanged.
- $ac \to bb$: Similarly, parity does not change.
- $bc \to aa$: Again, parity does not change.

Since the initial counts of a, b, and c are $15, 14, 13$ respectively (all odd or even), parity is preserved.

To reach a state where only one type of letter (a, b, or c) remains:

- At least one letter type must dominate entirely while the other two reduce to zero.
- However, the rules $ab \to cc$, $ac \to bb$, and $bc \to aa$ always generate new letters of the other two types.

It is impossible to reach a state containing only a's, b's, or c's because the system keeps "mixing" types.

—

Conclusion

Starting from $15a$, $14b$, $13c$, it is not possible to reduce the configuration to only a's, b's, or c's. The invariant $f(a, b, c) = 42$ and the parity analysis confirm this result.

Question for discord:

How could introducing invariants based on patterns (e.g., tracking the balance of transformations across rules) lead to unexpected equivalence classes?

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.