CPSC-354 Report

Connor Jacobs
Chapman University

December 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

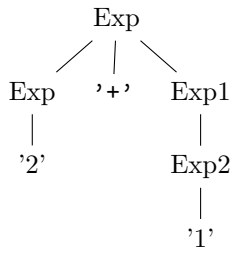
Links to two reviews I voted for:
https://github.com/tannerplatt/TannerHW3_354/blob/main/README.md
https://github.com/tyedwards37/Using-LLM-for-Literature-Review

2.4 Week 4

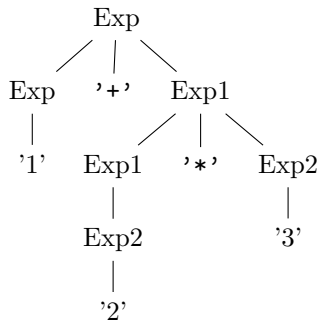
Derivation Trees

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

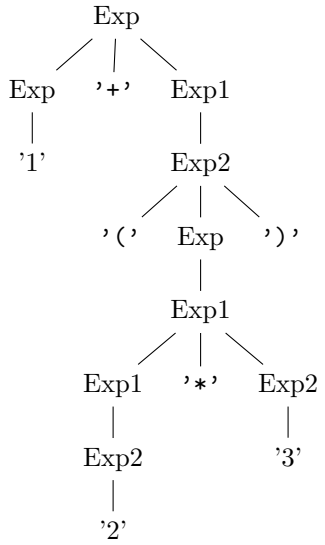
1. Derivation Tree for $2+1$



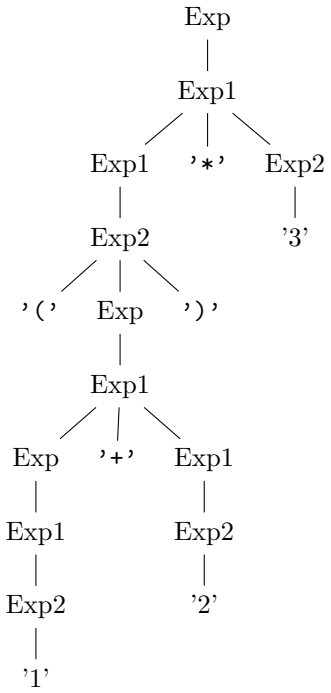
2. Derivation Tree for $1 + 2 * 3$



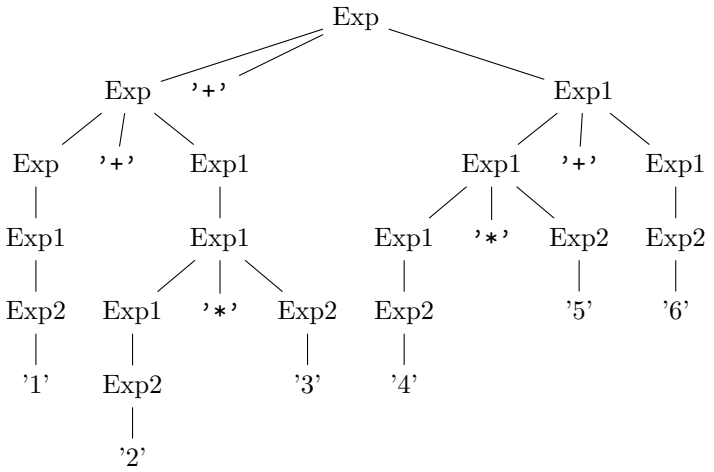
3. Derivation Tree for $1 + (2 * 3)$



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 and explain its semantic meaning:

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

Solution

Step-by-Step Beta Reduction: We start with the given expression:

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

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

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

2. Apply the result to $(\lambda f.\lambda x.f\,(f\,(f\,x)))$:

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

3. Substitute $(\lambda f.\lambda x.f\,(f\,(f\,x)))$ for f in $(\lambda x.f\,(f\,x))$:

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

4. Continue substitution and simplification. This corresponds to function application in Church numerals:

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

The result is the Church numeral 9:

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

Semantic Meaning: The term $(\lambda m.\lambda n.m\,n)$ implements a function on Church numerals that corresponds to **exponentiation with reversed arguments**. Given two Church numerals m and n representing natural numbers M and N, the term $m\,n$ corresponds to N^M.

For this specific case:

- $m = \lambda f.\lambda x.f\,(f\,x)$, which is the Church numeral 2 (representing 2). - $n = \lambda f.\lambda x.f\,(f\,(f\,x))$, which is the Church numeral 3 (representing 3).

Thus, $(m\,n) = (2\,3)$, which represents $3^2 = 9$.

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 = {}

Terminating: Yes

Confluent: Yes

Unique Normal Forms: Yes

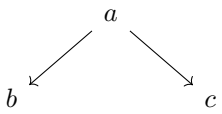
Question 3: A = {a} and R = {a, a}

Terminating: No

Confluent: Yes

Unique Normal Forms: No

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

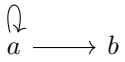


Terminating: Yes

Confluent: No

Unique Normal Forms: No

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

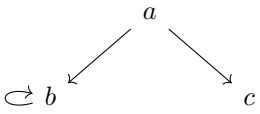


Terminating: No

Confluent: Yes

Unique Normal Forms: Yes

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

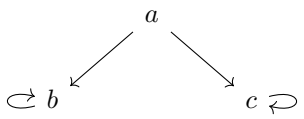


Terminating: No

Confluent: No

Unique Normal Forms: No

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

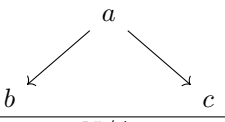
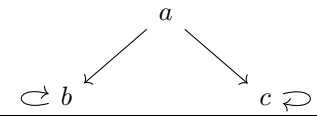


Terminating: No

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	
False	False	True	N/A
False	False	False	

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

Goal: We want to determine if we can reach a configuration where we have only one type of letter from the initial configuration:

$$a = 15, \quad b = 14, \quad c = 13.$$

Step 1: Total Letter Count Invariant

First, observe that all rules preserve the total number of letters:

$$\begin{aligned} ab \to cc &\implies a+b+c \to (a-1)+(b-1)+(c+2) = a+b+c, \\ ac \to bb &\implies a+c+b \to (a-1)+(c-1)+(b+2) = a+b+c, \\ bc \to aa &\implies b+c+a \to (b-1)+(c-1)+(a+2) = a+b+c. \end{aligned}$$

Since we start with $15 + 14 + 13 = 42$ letters, any reachable configuration must also have 42 letters in total.

Step 2: Differences as Invariants Modulo 3

Consider the differences:

$$a - b, \quad b - c, \quad c - a.$$

Let's check how each rule affects these differences:

- For $ab \to cc$:

$$(a, b, c) \to (a-1, b-1, c+2).$$

Thus:

$$\begin{aligned} a - b &\to (a-1) - (b-1) = a - b, \\ b - c &\to (b-1) - (c+2) = (b-c) - 3, \\ c - a &\to (c+2) - (a-1) = (c-a) + 3. \end{aligned}$$

- For $ac \to bb$:

$$(a, b, c) \to (a-1, b+2, c-1).$$

Thus:

$$\begin{aligned} a - b &\to (a-1) - (b+2) = (a-b) - 3, \\ b - c &\to (b+2) - (c-1) = (b-c) + 3, \\ c - a &\to (c-1) - (a-1) = c - a. \end{aligned}$$

- For $bc \to aa$:

$$(a, b, c) \to (a + 2, b - 1, c - 1).$$

Thus:

$$a - b \to (a + 2) - (b - 1) = (a - b) + 3,$$
$$b - c \to (b - 1) - (c - 1) = b - c,$$
$$c - a \to (c - 1) - (a + 2) = (c - a) - 3.$$

Notice that each rule only changes these differences by multiples of 3. Therefore, the values of $(a-b) \mod 3$, $(b - c) \mod 3$, and $(c - a) \mod 3$ are *invariants*—they do not change under any of the rewrite steps.

Checking the Initial Configuration

Initially:

$$a = 15, \quad b = 14, \quad c = 13.$$

Compute the differences modulo 3:

$$a - b = 15 - 14 = 1 \equiv 1 \pmod 3,$$
$$b - c = 14 - 13 = 1 \equiv 1 \pmod 3,$$
$$c - a = 13 - 15 = -2 \equiv 1 \pmod 3 \quad (\text{since } -2 \equiv 1 \mod 3).$$

Thus, initially:

$$(a - b) \mod 3 = 1, \quad (b - c) \mod 3 = 1, \quad (c - a) \mod 3 = 1.$$

Contradiction with Final Desired Configurations

Suppose we could reach a configuration of only a's. Then:

$$a = 42, \quad b = 0, \quad c = 0.$$

In this case:

$$a - b = 42 - 0 = 42 \equiv 0 \pmod 3,$$
$$b - c = 0 - 0 = 0 \equiv 0 \pmod 3,$$
$$c - a = 0 - 42 = -42 \equiv 0 \pmod 3.$$

This would give us $(a - b, b - c, c - a) \equiv (0, 0, 0) \mod 3$, which does not match our invariant $(1, 1, 1)$. Therefore, it is impossible to end up with only a's.

By symmetry, the same contradiction arises if we try to end up with only b's or only c's. Any single-type configuration would yield differences that are all $0 \mod 3$, which is inconsistent with our invariant.

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

The assignments helped me see how ideas from computer science theory connect to real world uses. We used Lean to prove basic math rules that we often take for granted. Including: Reflexivity (showing that something equals itself), Commutativity (showing that the order doesn't matter, like 2+3 = 3+2), and Associativity (showing that grouping doesn't matter, like (1+2)+3 = 1+(2+3)) Lean has special commands that help prove these rules step by step. For instance, when we need to prove that addition is associative, we used a method called induction - which is like proving that if something works for the first number, then showing that if it works for one number, it must work for the next one too. Lean helped by automatically checking each logical step. This work isn't just theoretical - it has practical value. When we build important computer systems that need to be mathematically perfect (like security systems or programs that check other mathematical proofs), tools like Lean help make sure everything works exactly as it should.

The assignments taught me about lambda calculus. I think of it as a very simple programming language that's become the foundation for many modern programming languages. This work helped me understand how modern programming languages handle variables and functions, especially in languages like Haskell and Scala where functions can be passed around like any other data. We added math operations and if statements to our simple system. This showed how it can be built upon to create a full programming language. It's like seeing how you can start with just a few bricks and build something quite complex. Another valuable lesson came from evaluating Church numerals. This exercise not only showcased how numbers and arithmetic operations can be represented functionally but also highlighted the elegance and power of recursion. Also non-terminating evaluations—became apparent. This emphasized the importance of carefully managing recursion in programming and algorithm design.

The assignments reinforced the importance of invariants in reasoning about system behavior. One notable exercise involved examining rewrite rules for strings while preserving the total count of characters. This reasoning provided an approach to solving problems and demonstrated how to detect impossibilities, such as reducing a mixed system of letters to a single type. This principle is broadly applicable in areas like database management, where maintaining data integrity is critical, and in distributed systems, where consistent state synchronization is paramount. It's like having an anchor point that helps you check if things are going wrong and proves whether certain changes are even possible.

I learned about using Lean to helps check if mathematical proofs are correct. Instead of having humans check every step of a proof (which can be time consuming and error prone), Lean does this automatically. Its like having a super-thorough proofreader who never gets tired and never misses a mistake. This is especially important when we're building systems where mistakes could be dangerous or very expensive, like: Software that controls airplanes or Systems that handle large financial transactions. The cool thing about Lean is that it makes this kind of careful checking more: Doable (because computers can check things much faster than humans), Reliable (because computers don't get tired or distracted), and Scalable (because once you've proven something, you can use that proof as a building block for bigger proofs) It's like turning mathematical reasoning into a language that computers can understand and verify, which gives us a powerful way to tackle really complex problems while being sure our solutions are correct.

A particularly challenging but rewarding part of the assignments was managing nested expressions and achieving normal form in lambda calculus. The recursive evaluation process required careful handling of abstractions, applications, and substitutions to avoid unintended variable capture. Enhancing the evaluate function to handle deeply nested expressions underscored the recursive nature of computation and the importance of designing robust algorithms that handle edge cases effectively.

We learned about a special tool in programming called the fixed point combinator that lets us write functions that can call themselves without using normal loops. While this sounds complex, it's actually a clever way to make recursive functions work in a very basic programming system. This might seem like a roundabout way to do something simple, but it shows something really powerful: you can create complex calculations using nothing but functions that transform other functions. It's like building a complex machine using only

simple parts. This matters because: It helps us understand how programming languages like Haskell work under the hood, it shows how abstract math ideas can solve real programming problems, and it demonstrates how to build complex things from very simple building blocks

We learned about turning text that humans write into a structure that computers can more easily work with. This process is called parsing, and it's like taking apart a sentence to understand its grammar, but for computer code. When you write '1 + 2 * 3', a human knows that multiplication should happen before addition. But a computer needs this spelled out more clearly. So we turn it into a tree-like structure that shows exactly what should happen in what order. It's like adding parentheses to make everything crystal clear: '1 + (2 * 3)'. We used a tool called Lark in Python that helps build these parsers. Lark is like having a set of rules (called a grammar) that tells the computer: how to recognize numbers, which operations should happen first and how to handle more complex expressions. It's like being a translator between human language and computer language, where every little detail needs to be precisely defined. This precision is crucial because computers need absolutely clear instructions to work correctly.

We did an interesting assignment using LLMs to help us research the history and development of programming languages. We Started with a question we were curious about in programming languages, used the LLM to help us explore that topic deeper , found connections between different areas of programming language research and discovered who the important researchers were and what they contributed. I learned to carefully check and edit what the llm suggests and how llms can help you find connections and patterns. This homework taught me two main things: How different areas of programming languages connect and influence each other and how ow to use models effectively as a research tool. It's like having a smart brainstorming partner who can help you explore ideas and make connections, but occasionally likes to make stuff up.

4 Conclusion

This course has taught me how theoretical computer science ideas connect to real world programming. First, I got hands-on experience with different tools and concepts: Using Lean to check mathematical proofs automatically, which showed me how computers can help ensure critical systems work correctly, learning about lambda calculus, which is like a simple blueprint that modern programming languages are built on, understanding how computers read and understand code by breaking it down into a structure they can process and more.

The part of the course focused on turning human written into something computers can understand taught me why programming languages need to be designed so carefully - it's like creating a new language that needs to be clear to both humans and machines.

The biggest takeaway was seeing how abstract math and logic aren't just theoretical - they help solve real problems. The course gave me both knowledge and new ways to tackle complex challenges. It's like learning to build things with both the basic building blocks (the theory) and modern power tools (the practical applications), and understanding why both are important.

References

[BLA] Author, Title, Publisher, Year.