CPSC-354 Report

Jack Mazac
Chapman University

October 22, 2024

Abstract

This report chronicles the learning journey over the semester in CPSC-354. The course encompassed a diverse set of topics including formal systems, Lean theorem proving, and other key areas in computer science and logic. Throughout the semester, I engaged with a range of concepts, starting with an introduction to formal systems and basic Lean proofs, and gradually progressing to more complex topics. The report documents my notes, homework solutions, and critical reflections on the content covered each week. The goal is to provide a comprehensive overview of my understanding and development in these subjects.

Contents

1 Introduction

This report serves as a comprehensive documentation of my learning and development throughout the CPSC-354 course. The content spans from the basics of formal systems and Lean theorem proving to more advanced topics covered later in the semester. The report is structured week by week, with detailed notes, homework solutions, and reflections for each period. The aim is to capture both the technical and conceptual growth experienced over the course of the semester.

2 Week 1

Notes

In Week 1, I explored the foundational concepts of formal systems through the MU-puzzle and began working with Lean proof tactics. The MU-puzzle introduced me to the idea of rule-based transformations within a formal system, emphasizing the importance of adhering strictly to the rules—known as the "Requirement of Formality." This concept was mirrored in my Lean exercises, where I learned to apply specific proof tactics to simplify and verify logical statements.

Homework

The Lean tutorial levels 5 through 8 provided practical exercises that reinforced the theoretical concepts from the MU-puzzle. Below is a summary of the steps and lessons learned:

Level 5

In Level 5, I learned how to handle simple arithmetic involving the addition of zero. The steps were as follows:

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

This taught me the importance of simplifying expressions step by step and ensuring that both sides of an equation are identical before applying `rfl`.

Level 6

Level 6 focused on precision rewriting. I applied the following steps:

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

This exercise highlighted the need for targeted rewriting to simplify specific parts of an expression while maintaining overall accuracy.

Level 7

In Level 7, I worked with the successor function and addition:

```
rw [succ_eq_add_one]
rfl
```

This reinforced the relationship between successor functions and addition, showing how to simplify such expressions in Lean.

Level 8

Level 8 was the most complex, requiring multiple rewrites to simplify nested successor functions:

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

This level emphasized the importance of meticulous, step-by-step simplifications, especially when dealing with Peano arithmetic.

Comments and Questions

How do the abstract rules in formal systems like the MU-puzzle translate into practical applications in software engineering?

2.1 Week 2

...

2.2 Lessons from the Assignments

Throughout the semester, I encountered various challenges and learning opportunities that contributed to my understanding of formal systems, logic, and Lean theorem proving. Here are the key lessons:

2.3 Precision and Structure in Formal Systems

The early weeks of the course, particularly Week 1, emphasized the precision required when working within formal systems. The MU-puzzle illustrated how even simple rules can lead to complex problem-solving scenarios, mirroring the need for exactness in Lean proofs. As I progressed through the levels in Lean, this precision became even more critical, particularly when dealing with nested functions and arithmetic expressions.

2.4 The Art of Rewriting in Lean

Rewriting is a fundamental tactic in Lean, as demonstrated in Levels 5-8. The ability to identify which parts of an expression to rewrite—and in what order—can make the difference between a successful proof and a failed one. This process is not just mechanical; it involves a deep understanding of the underlying logical structure and the relationships between different components of an expression.

2.5 Conclusion

This course provided a rigorous exploration of formal systems and Lean theorem proving, both of which are foundational to understanding logic and formal reasoning in computer science.

3 Week 2

3.0.1 Lecture Content

In Week 2, we delved deeper into the Natural Number Game (NNG) and focused on the Addition World. The lectures covered the fundamental properties of addition for natural numbers and introduced us to more advanced proof techniques in Lean.

Key topics included:

- Commutativity and associativity of addition
- The role of zero in addition
- Inductive proofs for properties of addition
- Tactics for rewriting and simplifying expressions in Lean

3.0.2 NNG Addition World Solutions

Levels 1-3: Lean Proofs

```
-- Level 1
rw [add_zero]

-- Level 2
rw [add_succ]
rw [succ_add]

-- Level 3
rw [succ_add]
rw [add_succ]
```

Level 4: Mathematical Proof **Theorem:** For all natural numbers a and b, a + succ(b) = succ(a + b)

Proof:

$$a + \text{succ}(b) = \text{succ}(a + b) \quad \text{[by definition of +]}$$

Level 5: Mathematical Proof **Theorem:** For all natural numbers n, 0 + n = n

Proof: By induction on n.

Base case: n = 0

$$0 + 0 = 0 \quad \text{[by definition of +]}$$

Inductive step: Assume 0 + k = k for some k. We need to prove 0 + succ(k) = succ(k).

$$0 + \text{succ}(k) = \text{succ}(0 + k) \quad \text{[by definition of +]}$$
$$= \text{succ}(k) \quad \text{[by inductive hypothesis]}$$

Therefore, by mathematical induction, 0 + n = n for all natural numbers n.

3.0.3 Relationship between Lean Proof and Mathematical Proof

Let's examine the relationship between the Lean proof and the mathematical proof for Level 5:

Lean proof:

```
induction n with d hd
  rw [add_zero]
 rfl
  rw [add_succ]
 rw [hd]
 rfl
```

The Lean proof closely mirrors the structure of the mathematical induction proof:

1. `induction n with d hd` corresponds to setting up the induction on n, with d representing the inductive variable and `hd` the inductive hypothesis.

2. The base case `rw [add_zero]` followed by `rfl` reflects the mathematical step $0 + 0 = 0$.

3. In the inductive step, `rw [add_succ]` corresponds to expanding $0 + \text{succ}(d)$ to $\text{succ}(0 + d)$.

4. `rw [hd]` applies the inductive hypothesis, similar to using $0 + k = k$ in the mathematical proof.

5. Finally, `rfl` completes the proof by asserting that the equality is now trivial, just as we conclude in the mathematical proof.

This comparison demonstrates how the rigorous structure of mathematical induction is encoded in Lean's tactics, providing a formal, machine-checkable version of the traditional proof.

3.0.4   Interesting Question

How does the use of induction in proving properties of natural numbers relate to recursive definitions in programming languages? Can we draw parallels between mathematical induction and recursive algorithms?

4   Week 3

4.1   Parsing and Context-Free Grammars

4.1.1   Lecture Content

In Week 3, we explored the concepts of parsing and context-free grammars (CFGs). The lectures covered the following key topics:

- Translating concrete syntax into abstract syntax

- The role of parsing in programming languages

- Defining the concrete syntax of a calculator using a CFG

- Writing a CFG using the Lark parser generator

- Generating a parser in Python to translate concrete arithmetic expressions to abstract ones

4.1.2   Homework Solutions

Derivation Trees   Using the provided context-free grammar, here are the derivation trees for the given strings: 2+1:

```
Exp
|
Exp '+' Exp1
|        |
Exp1     Exp2
|        |
Exp2   Integer
|        |
Integer    '1'
|
'2'
```

1+23:

```
Exp
|
Exp '+' Exp1
|        |
Exp1    Exp1 '' Exp2
|        |        |
```

```
Exp2    Exp2   Integer
|       |        |
Integer  Integer   '3'
|       |
'1'     '2'
```

1+(23):

```
Exp
|
Exp '+' Exp1
|        |
Exp1    Exp2
|        |
Exp2    '(' Exp ')'
|        |   |
Integer    Exp1 '' Exp2
|        |        |
'1'    Exp2    Integer
|        |
Integer    '3'
|
'2'
```

(1+2)3:

```
Exp
|
Exp1 '' Exp2
|        |
Exp2    Integer
|        |
'(' Exp ')'  '3'
|
Exp '+' Exp1
|        |
Exp1    Exp2
|        |
Exp2    Integer
|        |
Integer    '2'
|
'1'
```

1+23+45+6:

```
Exp
|
Exp '+' Exp1
|        |
Exp '+' Exp1 Exp2
|        |     |
Exp1 Exp1 '' Exp2 Integer
|     |      |      |
```

```
Exp2  Exp2   Integer '6'
|     |          |
Integer Integer   '5'
|     |
'1'  '2' '' Exp2
|
Exp2
|
Integer
|
'3' '+' Exp1
|
Exp1 '*' Exp2
|        |
Exp2   Integer
|        |
Integer    '4'
|
'3'
```

Similarities and Differences between CFG and Algebraic Data Type The CFG and algebraic data type for expressions share some similarities:

- Both define the structure of valid expressions

- Both capture the precedence of operations (e.g., multiplication before addition)

- Both can be used to generate or recognize valid expressions

However, there are also some key differences:

- The CFG is used for parsing, i.e., translating concrete syntax (strings) into an abstract syntax tree (AST), while the algebraic data type directly defines the structure of the AST

- The CFG is a set of production rules, while the algebraic data type is a type definition with constructors

- The CFG can include redundant rules or ambiguity, while the algebraic data type is more precise and concise

4.1.3 Interesting Questions

- How can we extend the calculator CFG to handle more advanced features like variables, functions, or control structures?

- What are some real-world applications of parsing beyond programming languages (e.g., natural language processing, data serialization formats)?

- How do more advanced parsing techniques like parser combinators or parsing expression grammars compare to traditional CFG-based parsing?

5 Week 5

5.1 Level 1: Assumption

If P, then P.

Proof:

    1. P assumption

5.2   Level 2: Constructor

If P and Q, then $P \wedge Q$.

Proof:

    1. P assumption

    2. Q assumption

    3. $P \wedge Q$ and_intro (1) (2)

5.3   Level 3: Practice Makes Perfect

If P, Q, R, and S, then $(P \wedge Q) \wedge R \wedge S$.

Proof:

    1. P assumption

    2. Q assumption

    3. R assumption

    4. S assumption

    5. $P \wedge Q$ and_intro (1) (2)

    6. $(P \wedge Q) \wedge R$ and_intro (5) (3)

    7. $(P \wedge Q) \wedge R \wedge S$ and_intro (6) (4)

5.4   Level 4: Cases for a Conjunction

If $P \wedge Q$, then P.

Proof:

    1. $P \wedge Q$ assumption

    2. P and_left (1)

5.5   Level 5: Rinse and Repeat

If $P \wedge Q$, then Q.

Proof:

    1. $P \wedge Q$ assumption

    2. Q and_right (1)

5.6   Level 6: Nothing New

If $P \wedge Q$ and $R \wedge S$, then $P \wedge S$.

Proof:

    1. $P \wedge Q$ assumption

2. $R \wedge S$	assumption
3. P	and_left (1)
4. S	and_right (2)
5. $P \wedge S$	and_intro (3) (4)

5.7 Level 7: So Many Cases

If $(Q \wedge (((Q \wedge P) \wedge Q) \wedge Q \wedge Q \wedge Q)) \wedge (Q \wedge Q) \wedge Q$, then P.

Proof:

1. $(Q \wedge (((Q \wedge P) \wedge Q) \wedge Q \wedge Q \wedge Q)) \wedge (Q \wedge Q) \wedge Q$	assumption
2. $Q \wedge (((Q \wedge P) \wedge Q) \wedge Q \wedge Q \wedge Q)$	and_left (1)
3. $((Q \wedge P) \wedge Q) \wedge Q \wedge Q \wedge Q$	and_right (2)
4. $(Q \wedge P) \wedge Q$	and_left (3)
5. $Q \wedge P$	and_left (4)
6. P	and_right (5)

5.8 Level 8: BOSS LEVEL

If $((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U$, then $A \wedge C \wedge P \wedge S$.

Proof:

1. $((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U$	assumption
2. $(P \wedge S) \wedge A$	and_left (1)
3. $P \wedge S$	and_left (2)
4. A	and_right (2)
5. P	and_left (3)
6. S	and_right (3)
7. $\neg I \wedge (C \wedge \neg O) \wedge \neg U$	and_right (1)
8. $C \wedge \neg O$	and_left (and_right (7))
9. C	and_left (8)
10. $A \wedge C$	and_intro (4) (9)
11. $(A \wedge C) \wedge P$	and_intro (10) (5)
12. $A \wedge C \wedge P \wedge S$	and_intro (11) (6)

6 Week 6

6.1 Level 1: Let there be cake!

Theorem: If P $\rightarrow$ C and P, then C.

Proof:

1. P	assumption
2. P → C	assumption
3. C	modus ponens (1, 2)

6.2   Level 2: Is it Cake!?

Theorem: C → C

Proof:

1. Assume C	assumption
2. C	(1)
3. C → C	(1-2, →-introduction)

6.3   Level 3: Trouble with the cake

Theorem: I ∧ S → S ∧ I

Proof:

1. Assume I ∧ S	assumption
2. I	and-elimination left (1)
3. S	and-elimination right (1)
4. S ∧ I	and-introduction (3, 2)
5. (I ∧ S) → (S ∧ I)	(1-4, →-introduction)

6.4   Level 4: Transitivity Aside

Theorem: If C → A and A → S, then C → S.

Proof:

1. C → A	assumption
2. A → S	assumption
3. Assume C	assumption
4. A	modus ponens (3, 1)
5. S	modus ponens (4, 2)
6. C → S	(3-5, →-introduction)

6.5   Level 5: then the quest has begun

Theorem: If P → Q, Q → R, Q → T, S → T, T → U, and P, then U.

Proof:

1. P	assumption
2. P → Q	assumption
3. Q → R	assumption

4. Q → T   assumption

4. Q → T   assumption
5. S → T   assumption
6. T → U   assumption
7. Q   modus ponens (1, 2)
8. T   modus ponens (7, 4)
9. U   modus ponens (8, 6)

6.6   Level 6: Curry

Theorem: If C ∧ D → S, then C → (D → S).

Proof:

1. C ∧ D → S   assumption
2. Assume C   assumption
3. Assume D   assumption
4. C ∧ D   and-introduction (2, 3)
5. S   modus ponens (4, 1)
6. D → S   (3-5, →-introduction)
7. C → (D → S)   (2-6, →-introduction)

6.7   Level 7: Un-Curry

Theorem: If C → (D → S), then C ∧ D → S.

Proof:

1. C → (D → S)   assumption
2. Assume C ∧ D   assumption
3. C   and-elimination left (2)
4. D   and-elimination right (2)
5. D → S   modus ponens (3, 1)
6. S   modus ponens (4, 5)
7. C ∧ D → S   (2-6, →-introduction)

6.8   Level 8: Go buy chips and dip!

Theorem: If (S → C) ∧ (S → D), then S → (C ∧ D).

Proof:

1. (S → C) ∧ (S → D)   assumption
2. S → C   and-elimination left (1)
3. S → D   and-elimination right (1)
4. Assume S   assumption

5. C   modus ponens (4, 2)

6. D   modus ponens (4, 3)

7. C D   and-introduction (5, 6)

8. S → (C D)   (4-7, →-introduction)

6.9   Level 9: Riffin and Sybeth

Theorem: R → ((S → R) (¬S → R))

Proof:

1. Assume R   assumption

2. Assume S   assumption

3. R   (1)

4. S → R   (2-3, →-introduction)

5. Assume ¬S   assumption

6. R   (1)

7. ¬S → R   (5-6, →-introduction)

8. (S → R) (¬S → R)   and-introduction (4, 7)

9. R → ((S → R) (¬S → R))   (1-8, →-introduction)

7   Week 7

Part 1: Reduction of the Lambda Term

We will reduce the given lambda term step by step using beta reductions while avoiding variable capture.

Steps:

1.
$$(\lambda m.\ \lambda n.\ m\ n)\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))$$

2.
$$(\lambda n.\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ n)\ (\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))$$

3.
$$(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))$$

4.
$$\lambda x.\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))\ x$$

5.
$$\lambda x.\ ((\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ (f\ (f\ (f\ y)))))\ x$$

6.
$$\lambda x.\ (\lambda z.\ (\lambda y.\ f\ (f\ y))\ (f\ (f\ (f\ z))))\ x$$

7.
$$\lambda x.\ (\lambda z.\ f\ (f\ (f\ (f\ (f\ (f\ (f\ (f\ (f\ z)))))))))\ x$$

13

Part 2: Function Implemented by $\lambda m.\ \lambda n.\ m\ n$

In **Part 1**, we start by applying the first lambda function to its argument:

1. The term $(\lambda m.\ \lambda n.\ m\ n)$ is applied to $(\lambda f.\ \lambda x.\ f\ (f\ x))$, resulting in $\lambda n.\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ n$.

2. Next, we apply this result to $(\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))$, giving $(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))$.

3. We then substitute $(\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))$ for f in the function, resulting in $\lambda x.\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda f.\ \lambda x.\ f\ (f\ (f\ x)))\ x.$

4. Continuing the substitutions and reducing, we eventually reach a term where the function f is applied nine times to x: $\lambda x.\ f^9\ x.$

This final term represents the Church numeral for 9, showing that the original expression computes $3^2 = 9$ when using Church numerals for 2 and 3.

In **Part 2**, the function $\lambda m.\ \lambda n.\ m\ n$ takes two Church numerals m and n and applies m to n. In the context of Church numerals, this operation corresponds to exponentiation. Specifically, it computes n raised to the power of m, since applying m to n results in n being applied m times.

8   Week 8 - 9

8.1   Running Tests and Adding New Test Cases

We begin by running the provided tests in `interpreter_test.py` to ensure that the interpreter is functioning correctly.

8.1.1   Running the Existing Tests

The existing tests can be run using the following command:

```
python interpreter_test.py
```

The output confirms that all tests pass:

```
Python version: 3.10.9 (main, Jan 11 2023, 09:18:20) [Clang 14.0.6 ]
Lark version: 1.1.9

TEST PARSING

AST x == ('var', 'x')
AST (((x)) ((y))) == ('app', ('var', 'x'), ('var', 'y'))
AST x y == ('app', ('var', 'x'), ('var', 'y'))
AST x y z == ('app', ('app', ('var', 'x'), ('var', 'y')), ('var', 'z'))
AST \x.y == ('lam', 'x', ('var', 'y'))
AST \x.x y == ('lam', 'x', ('app', ('var', 'x'), ('var', 'y')))
AST \x.x y z == ('lam', 'x', ('app', ('app', ('var', 'x'), ('var', 'y')), ('var', 'z')))
AST \x. \y. \z. x y z == ('lam', 'x', ('lam', 'y', ('lam', 'z', ('app', ('app', ('var', 'x'), ('var', ';
AST \x. x a == ('lam', 'x', ('app', ('var', 'x'), ('var', 'a')))
AST \x. x (\y. y) == ('lam', 'x', ('app', ('var', 'x'), ('lam', 'y', ('var', 'y'))))
AST \x. x (\y. y (\z. z z2)) == ('lam', 'x', ('app', ('var', 'x'), ('lam', 'y', ('app', ('var', 'y'), (
AST \x. y z (\a. b (\c. d e f)) == ('lam', 'x', ('app', ('app', ('var', 'y'), ('var', 'z')), ('lam', 'a

Parser: All tests passed!
```

```
TEST SUBSTITUTION

SUBST x [y/x] == ('var', 'y')
SUBST \x.x [y/x] == ('lam', 'x', ('var', 'x'))
SUBST (x x) [y/x] == ('app', ('var', 'y'), ('var', 'y'))
SUBST \y. x [y/x] == ('lam', 'Var1', ('var', 'y'))

substitute(): All tests passed!


TEST EVALUATION

EVAL x == x
EVAL x y == (x y)
EVAL x y z == ((x y) z)
EVAL x (y z) == (x (y z))
EVAL \x.y == \x.y
EVAL (\x.x) y == y

evaluate(): All tests passed!


TEST INTERPRETATION

Testing x --> x
Testing x y --> (x y)
Testing \x.x --> (\x.x)
Testing (\x.x) y --> y
Testing (\x.\y.x y) y --> (\Var2.(y Var2))

interpret(): All tests passed!
```

8.1.2 Adding New Test Cases

We have added a new test case to each function in `interpreter_test.py` to further verify the interpreter's correctness.

Test Parsing In the `test_parse()` function, we added:

```
assert ast(r"(\x.x) (\y.y)") == ('app', ('lam', 'x', ('var', 'x')), ('lam', 'y', ('var', 'y')))
print(f"AST {MAGENTA}(\\x.x) (\\y.y){RESET} == ('app', ('lam', 'x', ('var', 'x')), ('lam', 'y', ('var',
```

This tests parsing of an application of two lambda abstractions.

Test Substitution In the `test_substitute()` function, we added:

```
# SUBST (\x.x y) [y/z] == (\Var1.Var1 y)
assert substitute(('lam', 'x', ('app', ('var', 'x'), ('var', 'y'))), 'y', ('var', 'z')) == ('lam', 'Var
print(f"SUBST {MAGENTA}\\x.x y [z/y]{RESET} == ('lam', 'Var1', ('app', ('var', 'Var1'), ('var', 'z')))"
```

This tests substitution where the variable to be substituted is free in the body.

Test Evaluation In the `test_evaluate()` function, we added:

```
# EVAL (\x.\y.x y) a b == (a b)
assert linearize(evaluate(ast(r"(\x.\y.x y) a b"))) == "(a b)"
print(f"EVAL {MAGENTA}(\\x.\\y.x y) a b{RESET} == (a b)")
```

This tests evaluation of nested lambda abstractions with applications.

Test Interpretation In the `test_interpret()` function, we added:

```
input=r"(\x.\y.y x) a b"; output = interpret(input); print(f"Testing {input} --> {output}")
```

Which outputs:

```
Testing (\x.\y.y x) a b --> (b a)
```

8.2   Running New Test Cases

After adding the new test cases, we run the tests again:

```
python interpreter_test.py
```

The tests pass, confirming that our additions are correct.

8.3   Adding Programs to `test.lc` and Running the Interpreter

We added the following lambda calculus expressions to `test.lc`:

```
-- Identity function applied to itself
(\x.x) (\x.x)

-- Function that applies its argument to itself
(\x.x x) (\x.x x)

-- Combinator K (\x.\y.x)
(\x.\y.x) a b

-- Combinator S (\x.\y.\z.x z (y z))
(\x.\y.\z.((x z) (y z))) a b c
```

We run the interpreter with:

```
python interpreter.py test.lc
```

The interpreter outputs the results for each expression:

```
(\x.x)
((\x.x x) (\x.x x))
a
((a c) (b c))
```

8.4   Reduction of Expressions

Reduction of `a b c d` The expression `a b c d` reduces as follows:

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

This is due to the left-associative nature of function application in lambda calculus. Each application groups to the left.

Reduction of `(a)` The expression `(a)` reduces to `a` because the parentheses are just grouping symbols and do not affect the evaluation.

8.5 Capture-Avoiding Substitution

Capture-avoiding substitution ensures that when substituting an expression for a variable, we do not accidentally change the meaning of the expression by capturing free variables.

Implementation In `interpreter.py`, substitution is implemented in the `substitute()` function. When substituting into a lambda abstraction, if the bound variable is the same as the variable we are substituting for, we leave the abstraction unchanged. If there is a potential for variable capture, we generate a fresh variable name using the `NameGenerator` class.

```
elif tree[0] == 'lam':
    if tree[1] == name:
        return tree  # Variable bound; do not substitute
    else:
        fresh_name = name_generator.generate()
        new_body = substitute(tree[2], tree[1], ('var', fresh_name))
        return ('lam', fresh_name, substitute(new_body, name, replacement))
```

Test Cases We tested this with the following expression:

```
SUBST \y. x y [y/x] == (\Var1. (y Var1))
```

This shows that when substituting `y` for `x` in `.  x y`, we avoid capturing the variable `y` by renaming the bound variable to `Var1`.

8.6 Normal Form and Non-Terminating Expressions

Do All Computations Reduce to Normal Form? Not all lambda calculus expressions reduce to a normal form due to the possibility of infinite reductions. For example, the following expression does not reduce to a normal form:

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

Minimal Working Example The smallest lambda expression that does not reduce to normal form is the self-application of the identity function to itself:

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

This expression causes infinite recursion during evaluation.

8.7 Using the Debugger to Trace Executions

Setting Up the Debugger To trace the execution of the interpreter, we set breakpoints in `interpreter.py` at the calls to `evaluate()` and `substitute()`.

Tracing the Evaluation We input the expression:

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

By stepping through the interpreter, we observe the following calls to `evaluate()` and `substitute()`:

```
evaluate(('app',
          ('app',
           ('lam', 'm', ('lam', 'n', ('app', ('var', 'm'), ('var', 'n')))),
           ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('app', ('var', 'f'), ('var', 'x')))))),
          ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('var', 'x'))))))
evaluate(('app',
          ('lam', 'm', ('lam', 'n', ('app', ('var', 'm'), ('var', 'n')))),
          ('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('app', ('var', 'f'), ('var', 'x')))))))
evaluate(('lam', 'm', ('lam', 'n', ('app', ('var', 'm'), ('var', 'n')))))
substitute(('lam', 'n', ('app', ('var', 'm'), ('var', 'n'))), 'm', ('lam', 'f', ('lam', 'x', ('app', ('
...
```

Understanding the Evaluation Strategy By following the recursive calls and substitutions, we see how the interpreter applies beta reduction and maintains variable bindings without capture.

8.8   Implementation Challenges and Solutions

Throughout the development of our lambda calculus interpreter, we encountered several challenges that required careful consideration and innovative solutions. This section outlines these issues, our approach to solving them, and the final implementation.

Issues Encountered

1. **Infinite Recursion:** Initially, the interpreter would enter infinite recursion when evaluating non-terminating expressions like $(\lambda x.xx)(\lambda x.xx)$ (the omega combinator). This caused stack overflow errors.

2. **Depth Limitation Ineffectiveness:** We attempted to solve the infinite recursion problem by introducing a maximum evaluation depth (MAX_EVAL_DEPTH). However, this approach was not effective as the recursion error persisted in the substitute function.

3. **Stack Overflow in Recursive Approach:** The recursive implementation of both evaluate and substitute functions led to stack overflow errors for deeply nested expressions or non-terminating computations.

4. **IndexError in Iterative Approach:** When we switched to an iterative approach for substitute, we encountered an IndexError due to attempting to pop from an empty list. This occurred because the stack management in the substitute function was not robust enough to handle all cases.

5. **Partial Evaluation of Non-terminating Expressions:** While not strictly an "issue," we had to decide how to handle non-terminating expressions. The initial solution returned partially evaluated expressions when the maximum number of iterations was reached.

Solutions Implemented To address these issues, we implemented the following solutions:

1. **Iterative Approach:** We replaced the recursive implementations of evaluate and substitute with iterative versions to avoid stack overflow errors.

2. **Maximum Iterations:** We introduced a MAX_ITERATIONS constant to limit the number of reduction steps and prevent infinite loops while allowing for deep evaluations.

3. **Robust Stack Management:** We improved the stack management in the substitute function to handle all cases without errors, using separate stacks for the traversal and the result building.

4. **Graceful Handling of Non-terminating Expressions:** The interpreter now returns partially evaluated expressions for non-terminating computations, allowing it to handle a wider range of expressions without crashing.

5. **Separate Processing of Expressions:** We modified the main loop to process each expression separately, allowing the interpreter to continue even if one expression doesn't terminate.

Final Implementation The final implementation of our lambda calculus interpreter incorporates these solutions, resulting in a robust and flexible tool. Key features of the final implementation include:

- **Iterative Evaluation:** The `evaluate` function uses a while loop with a counter to prevent infinite loops:

```
def evaluate(tree):
    iterations = 0
    while iterations < MAX_ITERATIONS:
        if tree[0] == 'app':
            # ... (evaluation logic)
        else:
            return tree
        iterations += 1
    return tree  # Return partially evaluated tree if max iterations reached
```

- **Iterative Substitution:** The `substitute` function uses an explicit stack for traversal and a result stack for building the substituted expression:

```
def substitute(tree, name, replacement):
    stack = [(tree, False)]
    result_stack = []

    while stack:
        # ... (substitution logic)

    if result_stack:
        return result_stack[0]
    else:
        return tree  # Return original tree if no substitution occurred
```

- **Flexible Expression Handling:** The main function processes each expression in the input file separately:

```
def main():
    # ... (file reading logic)
    for expression in expressions:
        if expression.strip() and not expression.strip().startswith('--'):
            result = interpret(expression)
            print(f"Expression: {expression}")
            print(f"Result: \033[95m{result}\033[0m")
            print()
```

This implementation successfully handles both terminating and non-terminating expressions, providing a balance between functionality and preventing infinite computations. It demonstrates the practical application of theoretical concepts in lambda calculus while addressing real-world programming challenges.

9 Detailed Evaluation of Complex Expressions

To better understand how the interpreter handles complex expressions, we'll examine the evaluation of:

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

The evaluation proceeds as follows:

1. The outermost application is evaluated first.

2. The left part `((..  m n) (..  f (f x)))` is evaluated:
   - `m` is substituted with `(..  f (f x))`
   - This results in `(.  (..  f (f x)) n)`

3. The result is applied to `(..  f (f (f x)))`:
   - `n` is substituted with `(..  f (f (f x)))`
   - This yields `(..  f (f x)) (..  f (f (f x)))`

4. The final beta-reduction occurs:
   - `f` is substituted with `(.  f (f (f x)))`
   - The result is `(.  (.  f (f (f x))) ((.  f (f (f x))) x))`

This evaluation demonstrates how the interpreter handles nested lambda abstractions and applications.

10 Tracing Recursive Calls

To gain insight into the evaluation strategy, we trace the recursive calls to `evaluate()` for the expression:

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

The trace would show the sequence of `evaluate()` and `substitute()` calls, illustrating how the interpreter traverses the expression tree and applies beta-reductions. Due to the iterative nature of our final implementation, the trace would show a series of iterations within the `evaluate()` function, each handling a part of the expression until the final result is reached or the maximum number of iterations is hit.

Handling the Minimal Working Example (MWE) The Minimal Working Example (MWE) that does not reduce to normal form is:

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

Our modified interpreter handles this expression by:

1. The interpreter begins evaluating the expression.

2. It applies beta-reduction, substituting `(.  x x)` for `x` in the body of the first lambda.

3. This results in `(.  x x) (.  x x)`, which is identical to the starting expression.

4. The process repeats until MAX_ITERATIONS is reached.

5. The interpreter returns the partially evaluated expression, preventing an infinite loop.

10.1 Conclusion

Through this project, we've implemented a lambda calculus interpreter that balances theoretical correctness with practical considerations. Key achievements include:

- Implementing capture-avoiding substitution.
- Handling both terminating and non-terminating expressions.
- Using an iterative approach to prevent stack overflow.
- Providing a flexible tool for experimenting with lambda calculus.

The interpreter demonstrates the challenges of implementing theoretical concepts in practice and showcases solutions to common issues like infinite recursion and variable capture. It serves as a valuable educational tool for understanding lambda calculus and interpreter design.

10.2 Question

The Minimal Working Example (MWE) in our `test.lc` file, (.x x) (.x x), is a non-terminating expression that demonstrates a form of recursion. How does this relate to the Y combinator and fixed point combinators in lambda calculus? How might we extend our interpreter to support more complex recursive structures while still managing potential non-termination?

References

[1] Henk Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, 1984.