CPSC-354 Report

Jack Mazac
Chapman University

September 24, 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).

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

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 $0 + \mathrm{succ}(d)$ to $\mathrm{succ}(0 + d)$.

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?

References

[BLA] Author, Title, Publisher, Year.