CPSC-354 Report

Brandon Hughes
Chapman University

September 22, 2024

Abstract

Contents

1 Introduction

2 Week by Week

2.1 Week 1

2.1.1 Notes

During this week, there was a review of Git and being introduced to Latex and Lean. Some helpful commands include git add, commit, status, and push. Through the website "https://sudorealm.com/blog/how-to-write-latex-documents-with-visual-studio-code-on-mac", we set up latex to be able to complete the weekly report.

2.1.2 Homework

1. Finish the Natural Number Game Tutorial World.
   a) Show the completed work for levels 5 through 8.
   b) For one level, explain in detail how the Lean proof is related to its corresponding proof in mathematics.

1a. Show the completed work for levels 5 through 8.
Level 5: Prove that a+(b+0)+(c+0)=a+b+c.

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

Level 6: Prove that a+(b+0)+(c+0)=a+b+c.

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

Level 7: Prove that for all natural numbers a, we have succ(a)=a+1.

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

Level 8: Prove that 2+2=4.

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

1b. For one level, explain in detail how the Lean proof is related to its corresponding proof in mathematics.
For level 7, we had to prove the therom of the succ(n) is also equal to n+1.

Lean Proof:

```
Start: succ(n) = n + 1
  1: rw[one_eq_succ_zero]
  Result: succ n = n + succ 0
  2: rw[add_succ]
  Result: succ n = succ (n + 0)
  3: rw[add_zero]
End: succ n = succ n

Thus, proving reflexitivity.
```

Proof by Mathematics: By using the induction we are able to prove the therom of the succ(a) is equal to a+1.

```
Base Case:
  Consider n = 0,

  S(0) = 0 + 1
```

```
  0 + 1 = 1

  Thus, 0 + 1 = S(0), which holds true.

Inductive hypothesis:

  Assume for some natural number k that k + 1 = S(k).

Inductive Step:

  We need to show that k + 1 + 1 = S(k + 1).

  (k + 1) + 1 = S(k + 1), by adding parenthesis

  S(k) + 1 = S(k + 1), by using the inductive hypothesis.

  S(k + 1) = S(k + 1), by using addition of successors.

Thus, proving reflexitivity.
```

Through these steps, we can see that the end goal of proving reflexitivity on both the Lean proof and its corresponding proof in mathematics. The similarites come from the lean proof and the inductive step however, as they are similar steps in being able to prove the theorem. The lean proof is more straight forward because instead of proving it through a basis, inductive hypothesis, and inductive step, you only have to prove it through rewriting the equation so that both sides are equal. Which is done through the indutive step of the mathematics proof.

2.1.3 Comments and Questions

When looking at Formal Systems from the textbook, we are given this example of an impossible puzzle to solve. The MU problem, where you are given a set of rules and have to obtain MU from MI, however, its impossible because you can never end up without having an odd number of I's inside the string. When it comes to Formal systems, and solving them a lot of times people will look towards actually doing compared to trying to assess the logic behind this however computers, mostly AI, generally start at the logic. How might combining human intuition and AI's logical reasoning lead to more effective problem-solving strategies? Would we be able to solve problems quicker, our would AI's logical reasoning overtake the human trial and error method?

2.2 Week 2

2.2.1 Notes

During this week, we learned in class that both Math and Lean can be seen as langauges. Math is a specification language while Lean is a programming language. A specification langauge is used to define the requirements and properties of a system. A Math proof can be written into Lean proof very easily since they use and follow the same rules when it comes to solving thoerems and problems. The difference between the two proofs however is that in Math we typically reason forwards from the problem to the answer, while in Lean we reason backwards from the answer to the problem. We could also do it the opposite way but it would become more challengening. Another idea that we learned in class is that a recursive data type, which could also be called an algebraic data type, and induction are of similar processes as you define what a number is inside of a number. An example of recursion can be seen in the Tower of Hanoi as you are solving the previous tree in the next tree. Tower of Hanoi are also similar to binary search trees since the amount of

nodes in a amount of 'n' level of a tree, if you have a balanced tree is the same amount of moves it takes to solve when you have ;n' amount of disks. Lastly, we also learned that when you write a recursive program it creates a stack behind the scenes to solve all the problems, it will always go to the one on top rather than starting back at the start of the problem. If you don't have a stack you could also write it on a rewriting machine.

2.2.2 Homework

1. Finish the Natural Number Game Addition World.

a) Show the completed work for levels 1 through 5.

b) For level 4 or 5, explain in some detail how the Lean proof is related to its corresponding proof in mathematics

1a.
Level 1:

$Sd = Sd$	`rfl`
$S(0 + d) = Sd$	`rw[hd]`
$0 + Sd = Sd$	`rw[add_succ]`
$0 = 0$	`rfl`
$0 + 0 = 0$	`rw[add_zero]`
$0 + n = n$	`induction n with d hd`

Level 2:

$SS(a + d) = SS(a + d)$	`rfl`
$S(Sa + d) = SS(a + d)$	`hd`
$S(Sa + d) = S(a + Sd)$	`rw[add_succ]`
$Sa + Sd = S(a + Sd)$	`rw[add_succ]`
$Sa = Sa$	`rfl`
$Sa = S(a + 0)$	`rw[add_zero]`
$Sa + 0 = S(a + 0)$	`rw[add_zero]`
$Sa + b = S(a + b)$	`induction b with d hd`

Level 3:

$S(d + a) = S(d + a)$	`rfl`
$S(a + d) = S(d + a)$	`rw[hd]`
$S(a + d) = Sd + a$	`rw[succ_add]`
$a + Sd = Sd + a$	`rw[add_succ]`
$a = a$	`rfl`
$a = 0 + a$	`rw[zero_add]`
$a + 0 = 0 + a$	`rw[add_zero]`
$a + b = b + a$	`induction b with d hd`

Level 4:

$S(a + (b + d)) = S(a + (b + d))$	`rfl`
$S(a + b + d) = S(a + (b + d))$	`rw[hd]`
$S(a + b + d) = a + S(b + d)$	`rw[add_succ]`
$S(a + b + d) = a + (b + Sd)$	`rw[add_succ]`
$a + b + Sd = a + (b + Sd)$	`rw[add_succ]`
$a + b = a + b$	`rfl`
$a + b = a + (b + 0)$	`rw[add_zero]`
$a + b + 0 = a + (b + 0)$	`rw[add_zero]`
$a + b + c = a + (b + c)$	`induction c with d hd`

Level 5:

$S(a + d + b) = S(a + d + b)$	`rfl`
$S(a + b + d) = S(a + d + b)$	`rw[hd]`
$S(a + d + b) = S(a + d) + b$	`rw[succ_add]`
$S(a + d + b) = a + Sd + b$	`rw[add_succ]`
$a + b + Sd = a + Sd + b$	`rw[add_succ]`
$a + b = a + b$	`rfl`
$a + b = a + 0 + b$	`rw[add_zero]`
$a + b + 0 = a + 0 + b$	`rw[add_zero]`
$a + b + c = a + c + b$	`induction c with d hd`

1b.

For Level 4, we are proving the associativity of addition. On the set of natural numbers, addition is associative. In other words, if a,b and c are arbitrary natural numbers, we have (a+b)+c=a+(b+c). In Math and Lean, we have to do a proof by induction on c.

Math Proof:

$$(a + b) + c = a + (b + c)$$

Base Case: $(a + b) + 0 = a + (b + 0)$

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

Induction Step: $(a + b) + Sd = a + (b + Sd)$

Induction Hypothesis: $(a + b) + d = a + (b + d)$

$$
\begin{aligned}
(a + b) + Sd &= a + (b + Sd) && \\
S((a + b) + d) &= a + (b + Sd) && \text{def of } + \\
S((a + b) + d) &= a + S(b + d) && \text{def of } + \\
S((a + b) + d) &= S(a + (b + d)) && \text{def of } + \\
S(a + (b + d)) &= S(a + (b + d)) && \text{Induction Hypothesis}
\end{aligned}
$$

The Lean proof written above is the exact same as the same steps in the Math proof just backwards. Instead of having add_zero and add_succ, we have the definition of addition as that can be proven to add both successors and zero to numbers.

2.2.3 Comments and Questions

In the beginning of the reading, it takes about how recursion is different from paradox or infinite regress, since it never defines somethin in terms of iteself, but always in terms of simpler versions of itself. Is the only difference between a paradox and a recursively solveable problem be that it has an exit statement at its very simplest version or are there more differences? If some paradoxs were proposed recursively, would we be able to break down some harder problems into simpler version to prove if they are unsolveable logically?

2.3 Week 3

2.3.1 Homework

For this homework assignment, I used ChatGPT to explore language interoperability, which refers to how different programming languages interact with one another to create a single system or application. In modern software development and application creation, it is common for multiple programming languages to be used together to meet different kinds of functions and performance requirements. As a result, being able to integrate between these languages flawless is essential for building efficient and reliable systems. This report goes over some of the current methods used to ensure compatibility between programming languages and investigates ways to enhance these systems further. While many languages already possess some interoperability with one another, challenges arise when trying to bridge the gaps between the languages that were not designed to work together. These challenges can lead to issues such as performance inefficiencies, data type mismatches, and increase complexity in code maintenance. By examining these difficulties, potential improvements can be seen to create more solutions for the future. Additionally, advancements in tools, frameworks, and compiler technology could further enhance interoperability efforts. By improving in ways in which programming languages collaborate, developers will be able to build more versatile, efficient, and scalable systems. Addressing these challenges and making enhancements will lead to more seamless development processes and better applications.

https://github.com/Brandon-Hughes/LLM_Literature_Review

2.4 Week 4

2.4.1 Notes

Concrete syntax considers a program as a string in the form of a sequence of characters. This is the syntax that we are used to seeing and interacting with on a day to day basis. Abstract syntax considers a program as a tree-like structure that represents the program's structure and organization. An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. ASTs can write type checkers, interpreters, and compilers via recursion on these algebraic data types, but translating strings into trees is very difficult. The main idea is that you can translate a concrete syntax (string) into a concrete syntax tree, but translating a concrete syntax tree (string) into an abstract syntax tree (tree) is very difficult. Parsing is about putting the parentheses in the correct position. A context-free grammar is a set of rules that describe how to form strings in a language. The purpose of a context-free grammar is to define a set of strings or a language, in which a particular string in the lanaguage can be derived from the start symbol. The symbols that will appear in the strings (terminals), are those that are enclosed in single quotes. Other symbols may never appear in the parsed string, but only control which strings can be derived. These are called non-terminals.

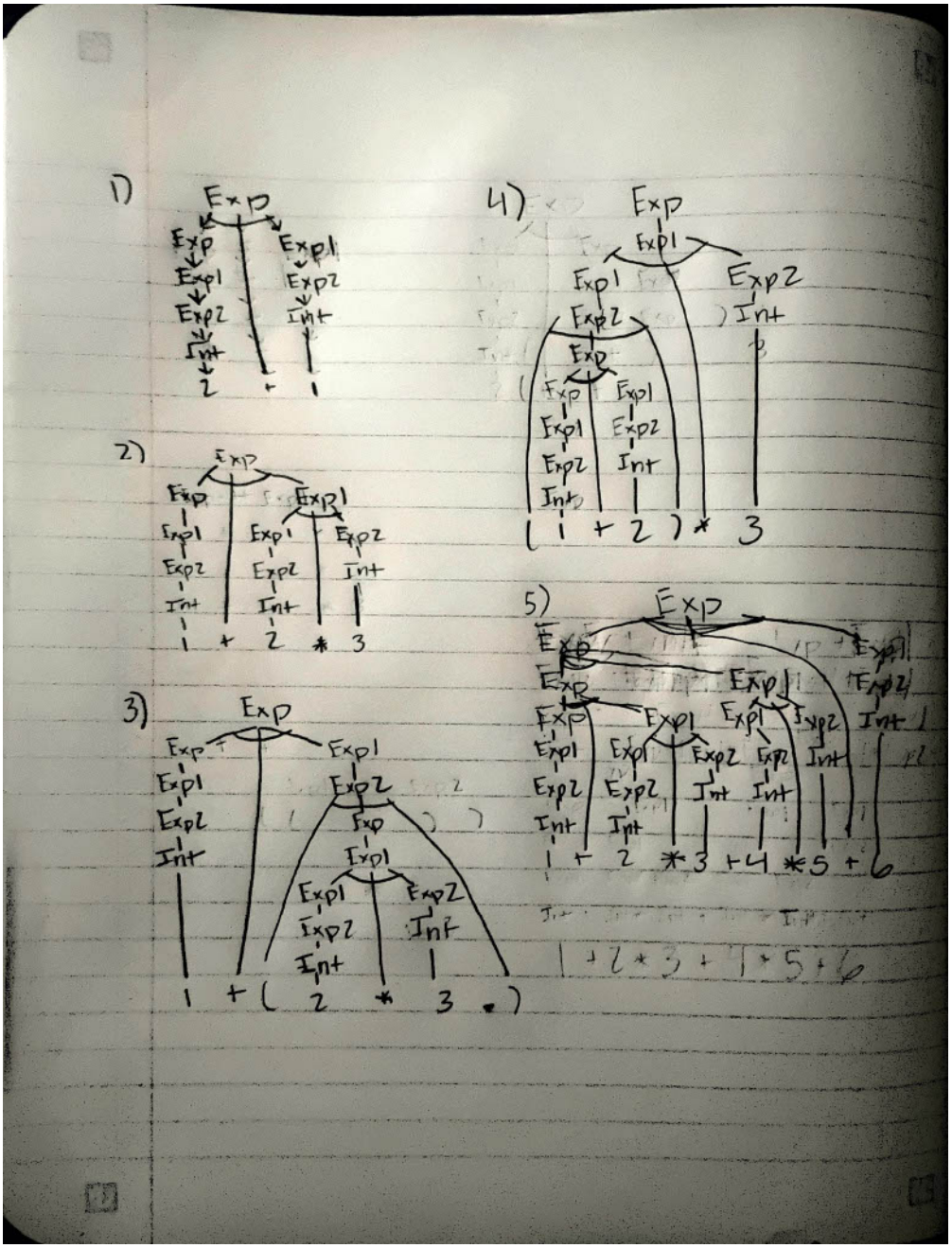
2.4.2 Homework

Using the context-free grammar:

$$
\begin{aligned}
Exp &\rightarrow Exp + Exp1 \\
Exp1 &\rightarrow Exp1 * Exp2 \\
Exp2 &\rightarrow Integer \\
Exp2 &\rightarrow (Exp) \\
Exp &\rightarrow Exp1 \\
Exp1 &\rightarrow Exp2
\end{aligned}
$$

Generate the abstract syntax tree for the expressions.
1. 2+1
2. 1+2*3
3. 1+(2*3)
4. (1+2)*3
5. 1+2*3+4*5+6

The Abstract Syntax Trees for Problems 1 through 5 are shown below:



As we can see from the image above, each of the problems can be solved by using the rules of the context-free grammar with the start symbol Exp.

2.4.3 Comments and Questions

During this week we've explored the idea of applying context-free grammar in the basics of Math and we are going to start applying that into programming languages. Besides the usage in math and programming lanaguages, are we able to see or use context-free grammar in other examples or in our lives?

2.5 ...

...

3 Lessons from the Assignments

4 Conclusion

References

[label] Andrew Moshier, Contemporary Discrete Mathematics, M&H Publishing, 2024