CPSC-354 Report

Brandon Hughes
Chapman University

November 17, 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:

$$
\begin{array}{rr}
S(a+d+b) = S(a+d+b) & \texttt{rfl} \\
S(a+b+d) = S(a+d+b) & \texttt{rw[hd]} \\
S(a+d+b) = S(a+d)+b & \texttt{rw[succ\_add]} \\
S(a+d+b) = a+Sd+b & \texttt{rw[add\_succ]} \\
a+b+Sd = a+Sd+b & \texttt{rw[add\_succ]} \\
a+b = a+b & \texttt{rfl} \\
a+b = a+0+b & \texttt{rw[add\_zero]} \\
a+b+0 = a+0+b & \texttt{rw[add\_zero]} \\
a+b+c = a+c+b & \texttt{induction c with d hd}
\end{array}
$$

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

$$
\begin{array}{rr}
(a+b)+0 = a+(b+0) & \\
a+b = a+(b+0) & \text{def of } + \\
a+b = a+b & \text{def of } +
\end{array}
$$

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

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

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

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.

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.

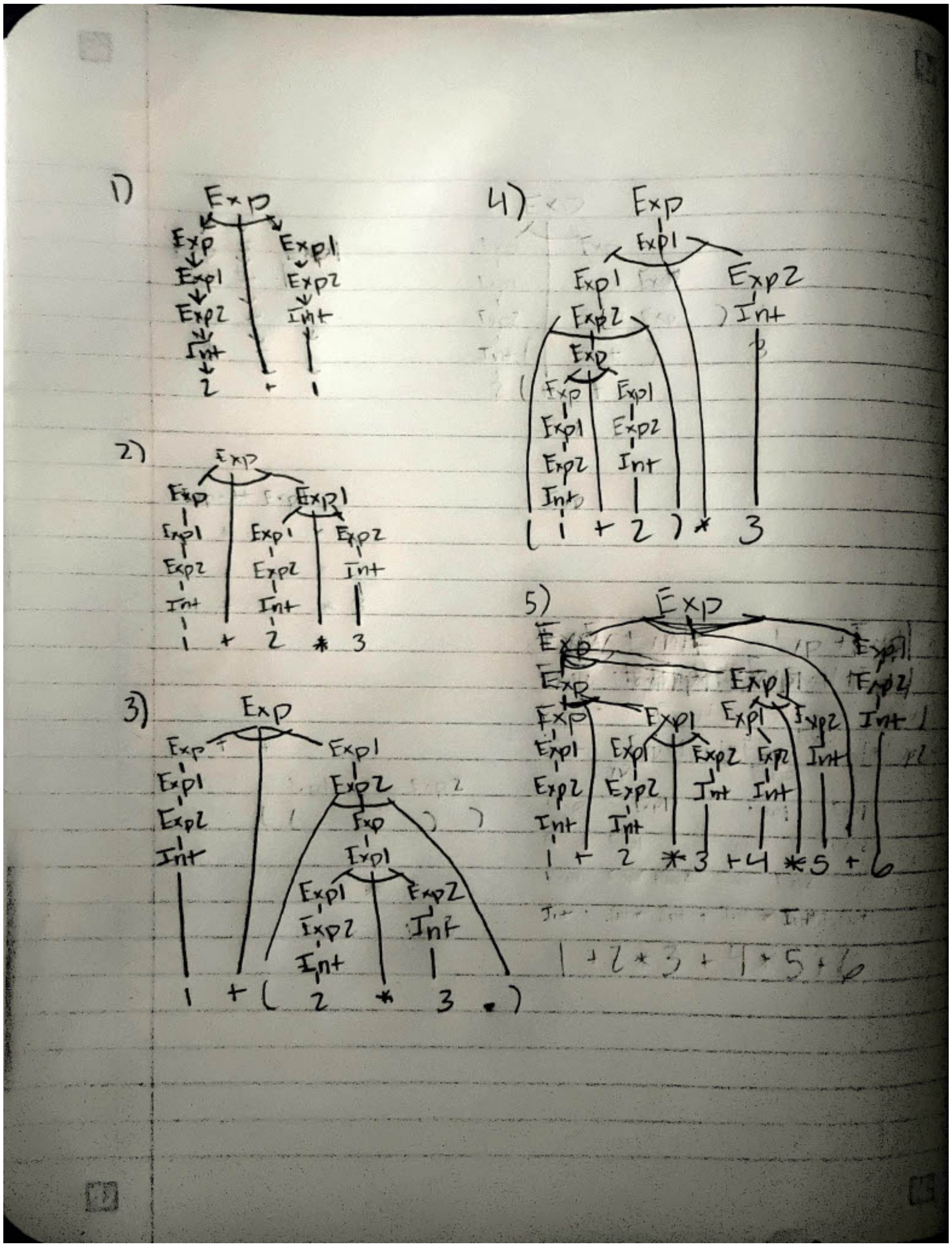
2.4.2 Homework

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+23
3. 1+23
4. 1 + 23
5. 1+23+45+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 Week 5

2.5.1 Notes

By practicing the application of context-free grammars (CFG) in a programming language, a calculator was created that took an input and broke it down into an Abstract Syntax Tree (AST) using a CFG. By breaking the expression into a AST, the problem was solved by reading through the AST to produce a correct result for the expression. This application served to practice parsing expressions into smaller pieces that are more easily read by programming languages, as well as provide an introduction to context-free grammars.

The second portion of this class week focused on an introduction and tutorial of constructive logic. Through the Lean logic game, students were introduced to how to write math expressions as logic-driven trees, similar to how a CFG might break an expression into components. While the process is similar, it differs by working with logical inferences instead of numbers. The keyword "exact" is used to give the game's answer, which serves as the concluding statement. The symbol $\wedge$ denotes a logical "and," meaning the logic must follow both A and B. The logical "and" can be broken apart using the keywords ".left" and ".right" to access either the left or right item of the $\wedge$, respectively.

2.5.2 Homework

1. Finish the Lean Logic Game Tutorial World.
   a) Show the completed work for levels 1 through 8.
   b) For level 8, write down the proof in mathematical logic.

1a.
Level 1: If todo_list is P, then P.

exact todo_list

Level 2: If p is P and s is S, then P $\wedge$ S.

exact $\langle$p,s$\rangle$

Level 3: If a is A, i is I, o is O, and u is U, then A $\wedge$ I $\wedge$ O $\wedge$ U.

exact $\langle$a,$\langle$i,$\langle$o,u$\rangle\rangle\rangle$

Level 4: If vm is P $\wedge$ S, then P.

exact vm.left

Level 5: If h is P $\wedge$ Q, then Q.

exact vm.right

Level 6: If h1 is A $\wedge$ I and h2 is O $\wedge$ U, then A $\wedge$ U.

exact $\langle$h1.left,h2.right$\rangle$

Level 7: If h is (L $\wedge$ ((L $\wedge$ C) $\wedge$ L) $\wedge$ L $\wedge$ L $\wedge$ L), then C.

exact h.left.right.left.left.right

1b.

Level 8: If ((P ∧ S) ∧ A) ∧ ¬I ∧ (C ∧ ¬O) ∧ ¬U then A ∧ C ∧ P ∧ S

$$\begin{aligned} \text{assumption1} : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U & & (1)\\ A & \quad \text{and\_left and\_right on (1)} & (2)\\ C & \quad \text{and\_right and\_right and\_left and\_left on (1)} & (3)\\ P & \quad \text{and\_left and\_left and\_left on (1)} & (4)\\ S & \quad \text{and\_left and\_left and\_right on (1)} & (5)\\ A \wedge C \wedge P \wedge S & \quad \text{and\_intro on (2) (3) (4) (5)} & (6) \end{aligned}$$

2.5.3 Comments and Questions

In the creation of a context-free grammar, we can see that by establishing a hierarchy through constructs like exp, exp1, exp2, etc., we can control the order of operations. How can programmers determine how deep the hierarchy should go to ensure the grammar is well-structured and the language doesn't break?

2.6 Week 6

2.6.1 Notes

Lambda calculus is a programming language comprised of only variables and functions. Two main portions build lambda calculus, the syntax and the semantics. The syntax is made of three constructions abstraction, application, and variables. Abstraction is taking the program and using a placeholder variable so that it can change what you use. It's the creation of general-purpose functions that can take in many different inputs but will produce a structured output. Application is how we apply functions to one another. Lastly, variables are just variables. An important part of Lambda calculus is dropping the parenthesis for easier readability. You apply the left and move to the right to solve for lambda in the same parenthesis. The next portion of Lambda calculus is the semantics or how to execute it. By executing, we are either inputting numbers into variables or simplifying them into their lowest form. When having multiple inputs and variables within the same expression, apply it in the same parentheses or the farthest lambda to the left if you only have the function in one parenthesis but multiple inputs and variables. When solving problems, you may arrive at to capture avoiding substitution, when multiple inputs are being placed into a variable, many solutions act differently here it will place it into all of them while some of them place it only into the first variable possible. To solve this, they created bound and free variables where bound variables are set within the function at the beginning, while free variables are those that are inputted.

2.6.2 Homework

Finish the Lean Logic Game Tutorial World. Show the completed work for levels 1 through 9.

Level 1: If bakery_service is P → C, then C.

exact bakery_service p

Level 2: Show that C → C.

$$\text{exact } \lambda h \Rightarrow h$$

Level 3: Show that I ∧ S → S ∧ I.

$$\text{exact } \lambda h \Rightarrow \langle h.right, h.left \rangle$$

Level 4: If h1 is C → A and h2 is A → S, show that C → S.

$$\text{exact } \lambda c \Rightarrow h2(h1\, c)$$

Level 5: If h1 is P → Q, h2 is Q → R, h3 is Q → T, h4 is S → T, and h5 is T → U, then U.

$$\text{exact } h5(h3(h1\, p))$$

Level 6: If h is C ∧ D → S, then C → D → S.

$$\text{exact } \lambda c\, d \Rightarrow h\langle c, d\rangle$$

Level 7: If h is C → D → S, then C ∧ D → S.

$$\text{exact } \lambda\langle c, d\rangle \Rightarrow h\, c\, d$$

Level 8: If h is (S → C) ∧ (S → D), then S → C ∧ D.

$$\text{exact } \lambda s \Rightarrow \langle h.left\, s, h.right\, s\rangle$$

Level 9: Show that R → (S → R) ∧ (¬ S → R).

$$\text{exact } \lambda r \Rightarrow \langle \lambda s \Rightarrow r, \lambda s \Rightarrow r\rangle$$

2.6.3 Comments and Questions

What are some alternative solutions to capture-avoiding substitution beyond using bound and free variables?

2.7 Week 7

2.7.1 Notes

Through the usage of syntax and sematnics of lambda calculus we are able to create examples of programs that are made. Pure lambda calculus only have functions and we can use those functions to encode data types such as booleans, numbers, and lists. To define a boolean, we define that whenever a function is true it will return the first item but if it's false it will return the second item. For example, λ m λ n m is true because it returns the first item while λ m λn n returns false due to it returning the second item. Through this we are able to get an example of how lambda calculus creates functions to get basic outputs. Another example is the creation of numbers through Church numerals. We can define numbers as the amount of times a function is applied onto a variable. For example, 0 being defined as λf λ x x, while 1 is λf λ x f x.

Lambda calculus allows us to use variable binding to name functions using the keyword let. Through this we are able to define keywords that can be used to reference a function previously defined in the code.

Another idea that is explored is the implementation of recursive functions through lambda calculus. With the definition of conditionals, numbers and keywords, we are able to investigate terms that reduce (compute) to themselves. This is done through defining the function in a keyword and then using the reducing that definition of the function into itself, with the goal of having the keyword's function appear in the function again.

2.7.2 Homework

1. Reduce the following lambda term: $((\lambda m.\lambda n.mn)(\lambda f.\lambda x.f(fx)))(\lambda f.\lambda x.f(f(fx)))$

$$
\begin{aligned}
&((\lambda m.\lambda n.mn)\ (\lambda f.\lambda x.f(fx)))\ (\lambda f.\lambda x.f(f(fx)))\\
&(\lambda n.\ (\lambda f.\lambda x.f(fx))n)\ (\lambda f.\lambda x.f(f(fx)))\\
&\lambda x.\ (\lambda f.\lambda x.f(f(fx)))\ x\ ((\lambda f.\lambda x.f(f(fx)))x\ x)\\
&\lambda x.\ \lambda x1.\ (\lambda f.\lambda x.f(f(fx)))\ x\ ((\lambda f.\lambda x.f(f(fx)))\ x\ x\ ((\lambda f.\lambda x.f(f(fx)))\ x\ x\ x1))\\
&\lambda x.\ \lambda x1.\ (\lambda x1.x(x(x\ x1)))\ ((\lambda f.\lambda x.f(f(fx)))\ x\ ((\lambda f.\lambda x.f(f(fx)))\ x\ x1))\\
&\lambda x.\ \lambda x1.\ x(x(x((\lambda f.\lambda x.f(f(fx)))\ x\ ((\lambda f.\lambda x.f(f(fx)))\ x\ x1))))\\
&\lambda x.\ \lambda x1.\ x(x(x(x(x(x(\lambda f.\lambda x.f(f(fx)))\ x\ x1)))))\\
&\lambda x.\ \lambda x1.\ x(x(x(x(x(x(\lambda x1.x(x(x\ x1)))\ x1)))))\\
&\lambda x.\ \lambda x1.\ x(x(x(x(x(x(x(x(x\ x1))))))))
\end{aligned}
$$

2. Explain what function on natural numbers $(\lambda m.\lambda n.mn)$ implements.

The lambda expression is about applying one numeral to another numeral. Church numerals are a way to represent natural numbers using functions. For example, we can represent 3 through f(f(f(x))) or "applying a function twice". In this expression, m is a numeral and n is another numeral and what it does is apply m to n. It takes two numerals or numebrs and applys one to the other like functions. This gives a way to combine the numerals and numbers using function application and not a direct operations such as addition or multiplication.

2.7.3 Comments and Questions

With the definition of lambda calculus in the basis of many programming languages, are there any issues that come when modeling complex data structures and alogrithms? If there are, how do programmers overcome them?

2.8 Week 8/9

2.8.1 Notes

This week focused on understanding an interpreter to solve lambda-calculus expressions through substituting and evaluating expressions. We were given a program and test expressions and had to create our test expressions to test each function and understand how they are changing the expression to solve them. Parse would break down the functions into the language that we are currently using to be able to run it through a lark file that would be able to interpret the expression. Without this step, it would be more complicated to solve the expression. The Substitute would take 3 inputs in which it would take the current tree, the name of the variable you are replacing, and the replacement variable. Through this, we can show an example substitution step where all the "names" inside of the "tree" are turned into "replacements." Through testing, I learned that there were some times when you would run into capture-avoiding substitution issues but through reading the function it solves this issue by generating a fresh name. This is done by checking if the variable character before the lambda is equal to the name you are replacing and if not then it changes all of the current variables' characters into a fresh name. Lastly, evaluate is our function that takes in the tree and returns the tree with the next step being done and simplifies all parenthesis into the correct form. Through this, we can see that each program can be simplified step by step and broken down into individual portions that can be stepped through to ultimately solve it. By taking this example, we can see that programs consist of a "lambda" or function and a "replacement" or input and return a simplified version of what is being done to it.

2.8.2 Homework

1. Complete exercises 2-8.

Exercise 2: Explain why a b c d reduces to (((a b) c) d) and why (a) reduces to a.
The point of this exercise is to understand how simplifying parentheses works inside of the program. It adds parenthesis onto those that go over the one character. If you would input "a b" it would add parentehsis around them to show that the current expression is "a and b". Then when you add c then d to the end, you are applying those onto the a and b, once at a time. If you would put "c and d" into parenthesis then you would apply "c and d" onto "a and b". The reason why (a) reduces down into just a, is that you aren't creating an expression. You are just having this single a that reduces down into itself leaving only an a.

Exercise 3: How does capture avoiding substitution work? Investigate both by making relevant test cases and by looking at the source code. How is it implemented?
In the program, capture avoiding substitution works by setting the bound variables to another name, such as VAR1 and VAR2. This works by keeping the free variables as they are currently. When the program does lambda substitution, on line 76 it checks for if the variable is equal to the name that you are replacing. If it is then it returns the tree otherwise it will generate a "fresh name" Allowing for the program to change the name of the variable to something else.

Exercise 4: Do you always get the expected result? Do all computations reduce to normal form?
Not all computations reduce down into normal form as there more applications of certain functions that can be perofmed. An example of this is in a non-terminating computation such as the one we saw in cases. Where the function would reduce into itself over and over again. If you apply this into the interpreter it will keep running into you arrive to a recurssion error.

Exercise 5: What is the smallest λ-expression you can find (minimal working example, MWE) that does not reduce to normal form?

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

Exercise 7: How does the interpreter evaluate ((\m.\n. m n) (\f.x. f (f x))) (\f.x. f (f (f x)))? Do a calculation similarly to when you evaluated ((\m.\n. m n) (\f.x. f (f x))) (\f.x. f (f (f x))) for the homework, but now follow precisely the steps taken by interpreter.py. Make a new line for each substitution.

$$\begin{aligned}
&((\lambda m.\lambda n.\,m\,n)\,(\lambda f.\lambda x.\,f\,(f\,x)))\,(\lambda f.\lambda x.\,f\,(f\,(f\,x)))\\
&(\lambda Var1.((\lambda f.(\lambda x.\,f\,(f\,x)))\,Var1))\,(\lambda f.\lambda x.\,f\,(f\,(f\,x)))\\
&((\lambda Var2.(\lambda Var4.(Var2\,(Var2\,Var4))))\,(\lambda f.(\lambda x.\,f\,(f\,(f\,x)))))\\
&(\lambda Var5.((\lambda f.(\lambda x.\,f\,(f\,(f\,x))))\,((\lambda f.(\lambda x.\,f\,(f\,(f\,x))))\,Var5)))
\end{aligned}$$

Exercise 8: Use ((\m.\n. m n) (\f.x. f (f x))) (\f.x. f x) as your input. Write out the trace of the interpreter in the format we used to picture the recursive trace of hanoi. Only write lines that contain calls to evaluate() or calls to substitute(). Add the line numbers.

12: evaluate $((\lambda m.(\lambda n.(m\, n)))\,(\lambda f.(\lambda x.(f\,(f\,x))))\,(\lambda f.(\lambda x.(f\,x))))$

39 : $evaluate((\lambda m.(\lambda n.(m\, n)))\,(\lambda f.(\lambda x.(f\,(f\,x)))))$

39 : $evaluate(\lambda m.(\lambda n.(m\, n)))$

50 :$(\lambda m.(\lambda n.(m\, n)))$

44 : $substitute(\lambda n.(m\, n)),\ m,\ (\lambda f.(\lambda x.(f\,(f\,x))))$

45 : $evaluate(\lambda Var1.((\lambda f.(\lambda x.(f\,(f\,x))))\,Var1))$

50 :$(\lambda Var1.((\lambda f.(\lambda x.(f\,(f\,x))))\,Var1))$

44 : $substitute((\lambda f.(\lambda x.(f\,(f\,x))))\,Var1),\ Var1,\ (\lambda f.(\lambda x.(f\,x)))$

45 : $evaluate((\lambda Var2.(\lambda Var4.(Var2\,(Var2\,Var4))))\,(\lambda f.(\lambda x.(f\,x))))$

39 : $evaluate(\lambda Var2.(\lambda Var4.(Var2\,(Var2\,Var4))))$

50 :$(\lambda Var2.(\lambda Var4.(Var2\,(Var2\,Var4))))$

44 : $substitute(\lambda Var4.(Var2\,(Var2\,Var4))),\ Var2,\ (\lambda f.(\lambda x.(f\,x)))$

45 : $evaluate(\lambda Var5.((\lambda f.(\lambda x.(f\,x)))\,((\lambda f.(\lambda x.(f\,x)))\,Var5)))$

50 : $return(\lambda Var5.((\lambda f.(\lambda x.(f\,x)))\,((\lambda f.(\lambda x.(f\,x)))\,Var5)))$

2.8.3 Comments and Questions

Week 8: How does knowing lambda calculus help prove the fundamental structure of programming languages and aid in designing new ones?
Week 9: Are we able to solve the expression with lamda calculus without changing it into the grammer and how much more complicated would it be to do the subtiutions?

2.9 Week 10

2.9.1 Notes

Through the usage of an abstraction reduction system ARS, we are able to show a relation between sets. There are special situations when an ARS is an algorithm, an example of this being an interpreters. An ARS is terminating if it does not allow an infinite computation. This means that you have the "size" of the problem or the number of steps you are making to solve for it and every application of a rule makes the "size" smaller. An ARS is confluent if for every "peak" there is a "valley" meaning that their are multiple paths that the problem can take but will end up at one ending. An element has normal form if it is not reducible. If an element has only one normal form, the normal form can be said to be a unique normal form. If every element in the ARS has a unique normal form, then the ARS has unique normal forms as well. Through some additional theorems we are able to define some additional theorems such as if an ARS is confluent and terinating then all elements reduce to a unique normal form. Some examples of rewriting through ARS can be seen in high school algebra equations and calculators.

2.9.2 Homework

1. What did you find most challenging when working through Homework 8/9 and Assignment 3?
2. How did you come up with the key insight for Assignment 3?
3. What is your most interesting takeaway from Homework 8/9 and Assignment 3?

The most challenging I thought was the creation of the minimal working example that wouldn't work in the program provided. Since, I understood how the program worked it was just thinking of ways that it wouldn't produce a simplified expression. With the help of my group members, I was farther able to understand how we could create a minimum working example and be able to solve the problem inside of the code for situations where it wouldn't simplfy. I wasn't able to come up with it through my own work but being able to be understand where the problem lies helped me to understand how to create the solution for Assignment 3. The issue was in the evalute not solving for when a "lam" is in front and not an "app". The most interesting takeawy from solving homework 8/9 and assignment 3 was the understanding of how programs break down their functions into something much smaller. I gained a deeper understanding of how programming langauges can actually read how certain inputs are needed and how they apply those inputs into the call being made. Another understanding is how programs are actually similar to lambda calculus in that they are only being simplified from a bigger picture into something that is concise for the reader.

2.9.3 Comments and Questions

When coming up with whether an abstraction reduction system is confluent or not, we are able to add intermediary steps to produce a confluent relation through starred relations. Are we able to use this to support the idea behind not having normal forms, if we infinitely return back to the same element with with a starred relation.

2.10 Week 11

2.10.1 Notes

In-Class Notes

(Exse 1 Bubble Sort)
implementation: ba → ab
$\overset{*}{\longleftrightarrow}$
w = v
spec if w and v have the same number of a's and b's

(Exse 2 XOR)
implementation:
aa → a
bb → a
ab → b
ba → b
- the ARS reduces its size with every step that it takes - 3 normal forms: """ (empty string), a, b
- $bbbb = b^4$
- $b^{2(n+1)}, where n > 0$
- $1 * x = x$
- $\epsilon * w = w$

- Rules:
- $b^{2(n+1)}, so n > 0$
- ba^n
- if non-empty and even b's = a - if non-empty and odd b's - b

(Exse 3)
implementation:
aa → a
bb → b
ba → ab

ab → ba
- does not terminate due to the last rules always being run
- 3 normal forms: ”””(empty string), a, b

- 00 → 0
- 11 → 1
- 01 → 10
- 10 → 01

- bbaa → baba → abba → aba → aab → ab
- bbaa → ba

implementation:
aa → a
bb → b
ba → ab
ab → ba

- at least one a no b = a - at least one b, no a = b - at least one a, one b = ab - if the two are equal then reduce them to lowest form, otherwise swap the positions

Summary

This week focused on String rewriting, which is rewriting strings using a specific rules and finding a pattern or understanding of what the rule is attempting to produce. This week we were able to see examples of string rewriting through bubble sort and XOR notation to further develop our understanding. As you can see in the notes taken from in class above, in excersise 1 we explore the implementation of bubble sort and cases we would be able to deifne as equal. Through specifiying when sorts take place, we are able to say that two strings are equal if they contain the same number of a's and b's since they will always end up at the same end goal. Since the process terminates whenever there are all the a's on the left and all the b's on the right and depending on the steps that you take you will always end up at the same string, the problem has a unique normal form such that we are able to determine that. In excersize 2, we explore the implementation of an XOR string. The ARS terminates since with every step that it takes, it reduces it's size and no matter the steps that you take you will always end up at the same normal form it is also confluent. The normal forms for this problem consist of the empty set, a, or b. The specification of this implementation is that if the string is non-empty and have an even number of b's then it will end up in the a normal form, or if it's nonempty and have an odd number of b's then it will end up in the b normal form. Otherwise, it's an empty set.

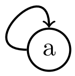
2.10.2 Homework

Draw a picture for each of the ARSs.

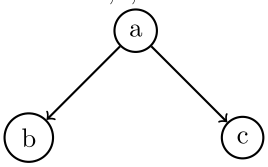
1. A =

2. A = a and R =
a

3. A = a and R = (a,a)

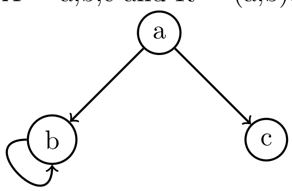


4. A = a,b,c and R = (a,b),(a,c)

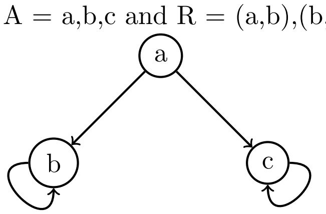


5. A = a,b and R = (a,a),(a,b)

6. A = a,b,c and R = (a,b),(b,b)(a,c)

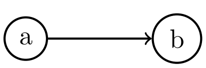


7. A = a,b,c and R = (a,b),(b,b)(a,c),(c,c)



Try to find an example of an ARS for each of the possible 8 combinations of confluent, terminating, and has unique normal forms. Draw pictures of these examples.

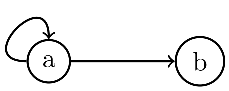
T / T / T example:



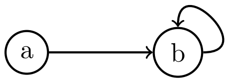
T / T / F example:

Cannot exist due the Theorem that If an ARS is confluent and terminating then all elements reduce to a unique normal form. Since this is both confluent and terminating, it must have a unique normal form.

T / F / T example:



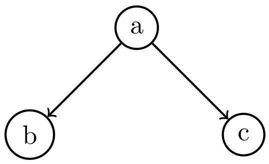
T / F / F example:



F / T / T example:

Confluent and having a unique normal form must be the same, one cannot be true while the other is false since an ARS has unique normal forms if and only if it is confluent and normalising.

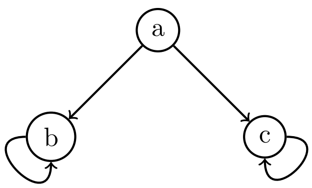
F / T / F example:



F / F / T example:

Same as above, confluent and having a unique normal form must be the same, one cannot be true while the other is false since an ARS has unique normal forms if and only if it is confluent and normalising.

F / F / F example:



2.10.3 Comments and Questions

Since we can represent basic programs like bubble sort and XOR as an ARS, how does this help us understand the logic and behavior of these programs better?

2.11 Week 12

2.11.1 Notes

This week explored the concept of invariants and their significance through different systems, such as mathematics and programming. Invariants are properties that do not change when we apply rules to the systems. Formally, this can be defined as a function $P : A \to B$ for an Abstract Reduction System (ARS) $(A, \to)$ such that if $a \to b$, then $P(a) = P(b)$ for all $a, b \in A$. This means that even when you apply the function P to a and b, they will still be equal to each other.

An example of an invariant in an ARS is as follows: under the rule $ab \to ba$, invariants could include the total length of the string, the number of a's or b's, and the sum of a's and b's, since these are values that will never change no matter how the rule is applied. Strong invariants are symmetric and are the method of choice for studying equivalence classes, while weak invariants are appropriate if the direction of time is important.

Furthermore, if P is an invariant property of the ARS $(A, \to)$ and $P(a) = \text{true}$ and $P(b) = \text{false}$, then $\neg(a \xrightarrow{*} b)$, $\neg(a \xleftrightarrow{*} b)$, and a and b are in different equivalence classes. Invariants can be used to demonstrate that certain transformations or operations are impossible by showing that two elements belong to different equivalence classes.

2.11.2 Homework

Rule: (fix (\fact. \n. if n=0 then 1 else n * fact (n-1))) → (fix (\fact. \n. if n=0))

let rec fact = \n. if n=0 then 1 else n * fact (n-1) in fact 3	**def of let rec**
let fact = (fix (\fact. \n. if n=0)) in fact 3	**def of let**
(\fact. fact 3) (fix (\fact. \n. if n=0))	**beta rule : substitute fix F**
(fix (\fact. \n. if n=0)) 3	**def of fix**
(\fact. \n. if n=0) (fix (\fact. \n. if n=0)) 3	**beta rule : substitute fix F**
(\n. if n=0 then 1 else n * (fix (\fact. \n1. if n1=0)) (n-1)) 3	**beta rule : substitute 3**
if 3=0 then 1 else 3 * (fix (\fact. \n1. if n1=0)) (3-1)	**def of if**
3 * (fix (\fact. \n1. if n1=0)) 2	**def of fix**
3 * (\fact. \n1. if n1=0) (fix (\fact. \n1. if n1=0)) 2	**beta rule : substitute fix F**
3 * (\n1. if n1=0 then 1 else n1 * (fix (\fact. \n2. if n2=0)) (n1-1)) 2	**beta rule : substitute 2**
3 * (if 2=0 then 1 else 2 * (fix (\fact. \n2. if n2=0)) (2-1))	**def of if**
3 * (2 * (fix (\fact. \n2. if n2=0)) 1)	**def of fix**
3 * (2 * (\fact. \n2. if n2=0) (fix (\fact. \n2. if n2=0)) 1)	**beta rule : substitute fix F**
3 * (2 * (\n2. if n2=0 then 1 else n2 * (fix (\fact. \n3. if n3=0)) (n2-1)) 1)	**beta rule : substitute 1**
3 * (2 * (if 1=0 then 1 else 1 * (fix (\fact. \n3. if n3=0)) (1-1)))	**def of if**
3 * (2 * (1 * (fix (\fact. \n3. if n3=0)) 0))	**def of fix**
3 * (2 * (1 * (\fact. \n3. if n3=0) (fix (\fact. \n3. if n3=0)) 0))	**beta rule : substitute fix F**
3 * (2 * (1 * (\n3. if n3=0 then 1 else n3 * (fix (\fact. \n4. if n4=0)) (n3-1)) 0))	**beta rule : substitute 0**
3 * (2 * (1 * (if 0=0 then 1 else 0 * (fix (\fact. \n4. if n4=0)) (0-1))))	**def of if**
3 * (2 * (1 * 1))	**def of** $*$
3 * (2 * 1)	**def of** $*$
3 * 2	**def of** $*$
6	

2.11.3 Comments and Questions

Are there any cases where a problem may not follow the invariant but still becomes solvable? Would that make the invariant invalid as a whole or would it just not be a good one?...

3 Lessons from the Assignments

4 Conclusion

References

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