CPSC-354 Report

Liana Ikoyan
Chapman University

December 15, 2024

Abstract

This report is a compilation of lessons learned, technical and otherwise, from the CPSC 354 Programming Languages course. We started off by learning the discrete math and proofs behind programming languages. Then we covered Context-Free Grammars before moving into learning concepts in Lambda Calculus. Using each concept we slowly added more to our programming language project, before finalizing it with concepts of Abstract Reduction Systems, Fixed Point Conversions, and String Rewriting. With the programming language completed, I reflected on the technical details I learned from the project and what lessons I could take away from the course overall.

Contents

1 Introduction

Grading guidelines (see also below):

- Is typesetting and layout professional?
- Is the technical content, in particular the homework, correct?
- Did the student find interesting references [6] and cites them throughout the report?
- Do the notes reflect understanding and critical thinking?
- Does the report contain material related to but going beyond what we do in class?
- Are the questions interesting?

Do not change the template (fontsize, width of margin, spacing of lines, etc) without asking your first.

2 Week by Week

2.1 Week 1

2.1.1 Notes

Week 1 was dedicated to introducing the expectations and subject matter of the course, as well as covering the very basic foundations of discrete mathematics.

Computer programming languages are an intersection between mathematics and software engineering. For the first few weeks we will cover the discrete math portion. This week was a review of the basic concepts of descrete math:

- (Equivalence Class): It is possible to have different representations of the same data. For example, (n,m) = n-m can be represented by different pairs such as (1,2), (0,1), and (10,11). All of the will output the same value of -1. When you build up abstractions, the process of declaring lots of things to be equal, you have an equivalence class, or equivalence relation.
- (Notations): There are advantages and disadvantages of representing data as binary versus urnary (successor) notation. Binary is much more consice and efficient, but is more challengeing to add and subtract. On the other hand, urnary notation is better suited for proofs by induction and proving the correctness of algorithms. It uses successors of zero or other numbers to represent data.
- (Writing proofs): Lean is a programming lanugage where everything is a proof, and proofs can be programmed into the code. The program itself is not writing the proof but the language allows programmers to write a program as a proof. This is not for automated theorem proofing but serves as an interactive theorem prover or proof assistent.

I did some research on the way math and computer science intersects and found that there are many aspects of computer science, such as machine learning, algorithms, and numerical analysis that rely on mathematical concepts, allowing us to do things we do with technology [1]. Therefore, it seems important to study the underlying math.

We also covered the basics of the Lean language, learning to work toward the conclusion/goal from the assumption or vice versa using proofs, theorems, and tools such as:

- rfl (reflexive): the proof that anything equals anything, ie x = x
- rw (rewrite): substitutes the given proof, assumption, or other tool where the program finds the first instance to substitute
- one_eq_succ_zero: 1 = S0, the definition of 1
- two_eq_succ_one: 2 = S1, the defintion of 2
- opposite side arrow: using l + space to insert an arrow indicating a change to the right side instead of the left

2.1.2 Homework

Lessons 5-8 were completed in the Natural Number Game. The goal of lesson 5 was to proof the following:

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

The solution was to use the add-zero theorem which states that anything added to zero equals itself, ie a + 0 = a. With this theorem I could arrive at the solution:

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

```
rfl
```

This proof uses the algorithm of addition to show that for any natural number m, m + 0 = m.

Lesson 6 had the same goal but asked for more precise rewrites. Specifying which variables to rewrite, I arrived at the solution:

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

Lesson 7 asked for a proof of the goal:

$$\text{succ } n = n + 1$$

For this lesson, I had to use the theorem for the definition of 1, one-eq-succ-zero, to rewrite 1 as succ 0. Then I used the addition theorem add-succ to add the two factors together, followed by the add-zero theorem, arriving at the solution:

```
rw [one\_eq\_succ\_zero]
rw [add\_succ]
rw [add\_zero]
rfl
```

Lesson 8 was the final challenge, asking me to prove that 2 + 2 = 4.

I first had to rewrite each 2 as succ (succ 0), then combined them together as a single chain of successors. I did the same to the other side, rewriting four as a chain of successors, and finally used the reflexive theorem to conclude both sides as the same:

```
rw [two_eq_succ_one]
rw [one_eq_succ_zero]
rw [add_succ]
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
```

2.1.3 Comments and Questions

It is interesting how even a simple elementary equation like 2 + 2 = 4 can have a much more complicated mathematical foundation. The majority of people, even those who work with math on a daily basis, will never have to consider numbers from the perspective of successors or confront the underlying theories that allow us to do so much with simple operations like addition.

The majority of the proofs and theorems we have covered so far use natural numbers as the basis for operations. Successor notation itself seems to start at zero and move only in the positive direction. How does discrete math take negative numbers, or more complicated concepts like imaginary numbers into account? Are those numbers even relevant in this sector of mathematics, and if no, why would they not be?

2.2 Week 2

2.2.1 Notes

First we went over translating math proofs into lean. We discussed how math is almost but not quite a programming language so you have to compromise with the syntax. There is a process of translating between the math and lean counterparts of proof definitions. For example:

def 1 = one_eq_succ_zero

You won't always find an easy match, but math can guide you through the proof process in lean.

Additionally, in lean you typically start from the conclusion and reason upwards, while in math you usually go the opposite way.

We also introduced induction as a tool in addition proofs. Natural numbers are defined by two rules: zero is a natural number, and the successor of natural number n is a natural number, the latter rule being a recursive defintion. Using these rules, we can get the definitions:

add(a,0) = a

add(a,Sb) = S add(a,b)

By defining addition in terms of itself, we get a recursive data type. Programming languages are essentially recursive data types like these, but scaled up by several factors.

Finally, we covered the Tower of Hanoi problem, a common example of this topic. Assuming there are poles 0, 1, and 2 with n+1 disks that have to move from 0 to 2 without a bigger disk stacked on a small one, we can come up with the following pseudo code solution:

```
hanoi 1 x y = move x y
hanoi (n+1) x y =
      hanoi n x (other x y )
      move x y
      hanoi n (other x y) y
```

2.2.2 Homework

For homework, we completed levels 1-5 of Addition World in the Natural Number Game.

Level 1 The goal was to prove the zero_add theorem, where for all natural numbers n, we have 0 + n = n. I arrived at the following solution:

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

Level 2 The goal was to prove succ_add where for all natural numbers a, b, we have succ(a) + b = succ(a + b). I got the solution:

```
induction b with d hd
rw[add_zero]
rw[add_zero]
rfl
```

```
rw[add_succ]
rw[hd]
rw[add_succ]
rfl
```

Level 3 I proved the theorem add_comm, with the goal that a + b = b + a, with the following:

```
induction a with d hd
rw[add_zero]
rw[zero_add]
rfl
rw[add_succ]
rw[succ_add]
rw[hd]
rfl
```

Level 4: To prove the theorem add_assoc, where (a + b) + c = a + (b + c), I got the following solution:

```
induction a with d hd
rw[zero_add]
rw[zero_add]
rfl
rw[succ_add]
rw[succ_add]
rw[succ_add]
rw[hd]
rfl
```

This is similar solution to the mathematical proof that uses induction. It starts by proving the base case that c = 0 getting

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

This can be solved with the defintion a + 0 = a, the mathematical equivalent of add_zero. Then, you prove

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

by starting with

$$(a + b) + S(c)$$

and using induction and a + S(b) = S(a + b), the math equivalent of add_succ, to get

$$a + (b + S(c))$$

Level 5: Without needing to use induction, I was able to prove the theorem add_right_comm, where (a + b) + c = (a + c) + b, with the following:

```
rw[add_assoc]
rw[add_assoc]
rw[add_comm c]
rfl
```

2.2.3 Comments and Questions

I found the Tower of Hanoi problem to be really interesting, but found it hard understanding the pseudo code that could be used to solve not just a three-disk stack, but be applied to taller and taller stacks. When I tried playing the game with more disks, I found it increasingly difficult, until I reached 6 disks and found that I could not win. I looked into this further, and while I could not find pseudo of the algorithm,

I found an article that used mathematic concepts to create as efficient of an algorithm as possible with a modified version of the magnetic Tower of Hanoi [2]. This demonstrates how, as mentioned in the previous week, math is an important part of the concepts we cover in computer science, so it is important that we do not overlook these mathematical algorithms.

We mentioned in class how recursive data types can be scaled up to serve as the foundation for programming languages. How much did this concept play a role in the creation of programming languages we use today? How big of a role do these discrete mathematical concepts play in the field of computer science as a whole?

2.3 Week 3

2.3.1 Homework

This week was spent working on a literature review using LLMs. I used ChatGPT -4 to explore the question of how hardware advancements have effected the development of programming langauges. The summary of the review is as follows:

Using ChatGPT, I explored the topic of how hardware and computation advancements affect the development of programming languages. I looked into the effects of recent advancements of modern hardware over the last few decades, as well as effects of future advancements of quantum computing. For the former, I found that we have been able to greatly expand our programming capabilities through our improved hardware. Beyond basic speed and optimization increases, improved hardware has allowed for more complex abstractions and programming constructions without suffering performance issues, given us more sophisticated methods of memory management that make languages like Java and C# viable, and many more benefits for efficiency and ease of use. This got me wondering how programming languages would be affected by the advancement of quantum computing. While quantum capabilities would likely improve performance and calculation abilities, it would require many new models, languages, standardizations, methods of error handling, and educational tools to transition from classical to quantum programming. Thanks to extensive research contributions over the years from David Deutsch, Richard Feynman, and many other researchers, we are already working to create the tools and standards necessary for quantum programming.
(Link to the README.md file)

I posted the summary and link on Discord under the username LianaI and voted for two other submissions on Monday. The two submissions I voted for were Gabriel Davidson's review and Zack Dell's review.

2.4 Week 4

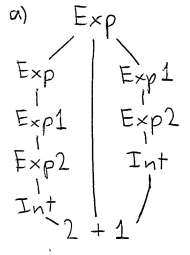
2.4.1 Homework

Week 4 was spent exploring Context-Free Grammar and how it could be used to represent parsing trees. These trees are built on grammar rules such as EXP : EXP '+' EXP1 that dictate the order in which numbers and operations can be addressed. We created trees for five different expressions following the CFG:

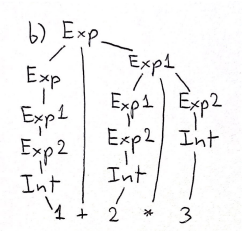
```
EXP : EXP '+' EXP1
EXP1 : EXP1 '*' EXP2
EXP2 : Integer
EXP2 : '(' EXP ')'
EXP : EXP1
EXP1 : EXP2
Integer : '1','2','3','4','5','6'
```

Using this grammar, I was able to represent the following expressions with trees:

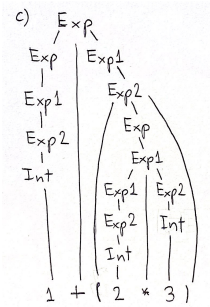
a: 2 + 1



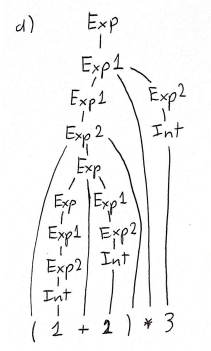
b: 1 + 2 * 3



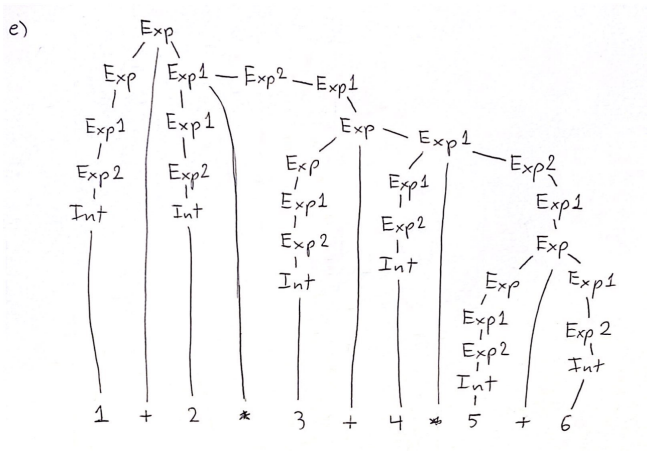
c: 1 + (2 * 3)



d: (1 + 2) * 3



e: 1 + 2 * 3 + 4 * 5 + 6



2.4.2 Comments and Questions

I found the translation process between mathematical expressions and the context-free grammar to be really fun. It is interesting just how much you can break down simple math into complex structures and algorithms.

So far we have used context-free grammar to redefine concrete syntax in order to parse mathematical expressions into abstract syntax trees. How far could we push the concept of a CFG beyond simple expressions like EXP -> EXP1, and using different CFGs, how many different types of information could we redefine (ex concrete English grammar)?

2.5 Week 5

2.5.1 Notes

This week we covered Lean programming. We went over the concept of expressing ideas as propositions, and finding evidence to support them. The logic of the ideas in itself is not complicated, but the point was to learn how to express basic ideas through the more complex syntax of a programming language like Lean. We covered conjuctions using the $\wedge$ character to represent the concept of "and".

We also covered the LARL, a type of parser tree for Lark. They are an efficient class of parsers that work from left to right, symbol to symbol. You can traverse a tree by either reducing (moving up) or shifting (moving right), and moving forward until there is a conflict.

2.5.2 Homework

For homework, we completed lessons 1-8 of A Lean Intro to Logic. The problems, and subsequent solutions, are as follows:

A Lean Intro to Logic:

1. example (P : Prop)(todo_list : P) : P := by

   exact todo_list
2. example (P S : Prop)(p: P)(s : S) : P $\wedge$ S := by

   exact and_intro p s
3. example (A I O U : Prop)(a : A)(i : I)(o : O)(u : U) : (A $\wedge$ I) $\wedge$ O $\wedge$ U := by

   exact and_intro (and_intro a i) (and_intro o u)

4. example (P S : Prop)(vm: P ∧ S) : P := by

exact vm.left

5. example (P Q : Prop)(h: P ∧ Q) : Q := by

exact h.right

6. example (A I O U : Prop)(h1 : A ∧ I)(h2 : O ∧ U) : A ∧ U := by

have a := h1.left

have u := h2.right

exact ⟨ a , u ⟩

7. example (C L : Prop)(h: (L ∧ (((L ∧ C) ∧ L) ∧ L ∧ L ∧ L)) ∧ (L ∧ L) ∧ L) : C := b

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

8. example (A C I O P S U : Prop)(h: ((P ∧ S) ∧ A) ∧ ¬I ∧ (C ∧ ¬O) ∧ ¬U) : A ∧ C ∧ P ∧ S := by

have a := h.left.right

have c := h.right.right.left.left

have p := h.left.left.left

have s := h.left.left.right

exact ⟨ a, ⟨ c , ⟨ p , s ⟩ ⟩ ⟩

This can be rewritten in the form of a mathematical proof:

(1) (h: ((P ∧ S) ∧ A) ∧ ¬I ∧ (C ∧ ¬O) ∧ ¬U) assumption

(2) a and_left (1)

(3) a and_right (2)

(4) c and_right (1)

(5) c and_right (4)

(6) c and_right (5)

(7) c and_left (6)

(8) c and_left (7)

(9) p and_left (1)

(10) p and_left (9)

(11) p and_left (10)

(12) s and_left (1)

(13) s and_left (12)

(14) s and_right (13)

(15) A ∧ C ∧ P ∧ S and_intro ((3) and_intro ((8) and_intro ((11) (14))))

2.5.3 Comments and Quesitons

It's always interesting and a little confusing to me when the same expression can be represented in different formats. For example, the Lean problems can be presented as one long expression broken up by parentheses and colons, but can also be written as separate expressions specifiying the type, such as an object, assumption, or goal.

I noticed while doing the homework that there is different syntax to represent the same expression. For example, you could either write and_left h or h.left, with the latter being more efficient for representing deeply nested data. What is the purpose of having multiple syntax types for the same idea, especially if one is less efficient than the other? Is there a difference in how these syntax rules are represented in the logic of the programming language?

After asking this question I did some research and found an interesting article from Princeton which dives into the nuances of syntax and semantics [3]. It discusses the concepts of context-free and context-sensitive syntax that we have covered so far, and gives me a better understanding of where these different syntaxes come from [3].

2.6 Week 6

2.6.1 Notes

This week we covered the dependently typeed programming language Lean, and used it to program Lambda calculus, the smallest programming language in the world. Using just functions and variables we can represent concepts on a single line with the keyword "exact".

We covered specific syntax for Lambda calculus, starting with the arrow →. Writing bakery_service P → C can be read as "If P then C". bakery_service is a function of P that gives C, so to get C we can write

exact bakery_service p

We need syntax to make functions. In math, that syntax typically looks like f(x) = x+2. But in Lambda calculus functions do not have names, you simply indicate what maps to what. For example, you can write

$x \mapsto x+2$ or $\lambda\ x : x+2$

Parentheses are also important, as with the case of "currying" for equations like

$(C \wedge D) \to S = C \to (D \to S)$

We also covered the theory of substitution, where different expressions can be substituted into a function in order to simplify and reduce expressions.

2.6.2 Homework

For homework we completed problems 1-9 of the Lean Logic implication world. I arrived at the following solutions: A Lean Intro to Logic:

1. example (P C: Prop)(p: P)(bakery_service : P → C) : C := by

   exact bakery_service p

2. example (C: Prop) : C → C := by

   exact λc:C ↦ c:C

3. example (I S: Prop) : I ∧ S → S ∧ I := by

   exact λ (h : I ∧ S) ↦ ⟨ h.right , h.left ⟩

4. example (C A S: Prop) (h1 : C → A) (h2 : A → S) : C → S := by

exact λ c ↦ h2 (h1 c)

5. example (P Q R S T U: Prop) (p : P) (h1 : P → Q) (h2 : Q → R) (h3 : Q → T) (h4 : S → T) (h5 : T → U) : U := by

have q := h1 p

have t := h3 q

exact h5 t

2.6.3 Comments and Questions

I find it interesting that there are entire subsections of math dedicated to representing functions as small as possible. I did a little more research on the concept of lambda calculus and found an article discussing the issue of "integrating non-functional aspects into functional programming langauges", explaining how higher-level programming langauges run into these problems, and how a language like lambda calculus can remedy these issues [4]. Addmittedly, it confused me why such a small and seemingly limited language would even be relevant today. But I realize now how that simplicity and elegance can be beneficial for many computer science problems.

With so many ways to represent functions, how does the method we choose affect how we perceive them? For example, would a problem seem easier to solve if represented in Lambda vs standard math notation?

2.7 Week 7

2.7.1 Notes

For this week, we continued to cover lambda calculus, focusing on using substitutions to reduce a lambda term. We covered the concept of beta reductions, and how functions are defined in lambda calculus using the scope of parentheses. Anything defined within the scope of a function is part of the function, and anything outside of its closed parentheses is the argument. Using this, you can substitute the argument for every instance of the variable within the function definition in order to reduce the function.

2.7.2 Homework

For homework I reduced the given lambda term and got the following solution after 7 reductions:

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

Solution: *Link to the embedded file `substitution.hs`*

Additionally, we discussed what function on natural numbers (\m. \n. m n) implements. Essentially, this term takes a function m and an element n and applies m to n.

2.7.3 Comments and Questions

When doing the homework I noticed how long and messy a lambda expression can get after just a few beta reductions. How do we strike a balance between the effectiveness of a program and the neatness/readability of its syntax?

2.8 Weeks 8-9

2.8.1 Notes

For week 8 we discussed normal forms, where an expression cannot be reduced any further, and then applied this to our programming assignment.

2.8.2 Homework

For weeks 8 and 9 we worked on exercises 2-8 to practice lambda calculus in python, specifically normalizing expressions.

For exercise 2 I ran some of the previous programs in test.lc and ran pyton interpreter.py test.lc. When the program ran a b c d it produced (((a b) c) d) but produced a when it ran (a). The latter case is because a has no function or argument, therefore can be reduced with no parentheses. Meanwhile, a b c d function like arguments and therefore have to be separated into different sections with parentheses.

For exercise 3 I ran test cases to test the implementation of capture avoiding substitution, which is when the original meaning of an expression is changed. I tested one expression (\f.f f) g, which produced (g g), and then (\f.f t) which produced (g t). This shows how you have to use unique names for different variables to avoid them being captured by a lambda abstraction.

For exercise 4, I found that not all expressions reduce to the expected result. Some seem to evaluate to an expression that includes Var1 or something similar.

For exercise 5, I found (\f.\x.f) (\f.x) which reduces to (\Var1.(\f.x))

For exercise 6, I launched the python debugger and went through the steps of the interpreter.

For exercise 7, I tested the given expression using a debugger, adding breaks at each substitute() function call and using the linearize() function in the debugger to see the AST output. The results were the following:

```
\Var5.((\f.(\\x.(f (f (f x))))) ((\f.(\x.(f (f (f x))))) Var5))

(\f.(\x.(f(f x))))

 (\f.(\x.(f (f (f x)))))

 (\x.(f(f x)))

 (f (f x))

 (f x)

 (f Var3)

 (\f.(\x.(f ( f(f x)))))

 (Var2 (Var2 Var3))
```

```
(Var2 Var3)

(Var2 Var4)

(\f.(\x.(f ( f(f x)))))

(\Var4.(Var2 (Var2 Var4)))

(\f.(\x.(f ( f (f x)))))

(\Var5.( \f.(\x.(f ( f (f x))))))  ((\f. (\x. (f ( f (f x))) Var5)))
```

For exercise 8, I followed a similar process, this time putting breaks at each call of the evaluate() function, its return statements, or calls to the substitute() function and seeing what the debugger outputted for the variables. The results were as follows:

```
13. eval (\\x.x) a\n// (\\x.\\y.x) a b\n// (\\x.\\y.y) a b\n// ((\\m.\\n. m n)
(\\f.\\x. f (f x))) (\\f.\\x. f (f (f x))) \n\n(((\\m.(\\n.(m n))) (\\f.(\\x.(f (f x)))))
(\\f.(\\x.(f x))))


39. eval ('app', ('app', (...), (...)), ('app', (...), (...)))

39. ('app', ('lam', 'x', (...)), ('var', 'a'))

53. return ('lam', 'x', ('var', 'x'))

44. sub body = ('var', 'x'), name = 'x', arg = ('var', 'a')

45. eval ('var', 'a')

53. ('var', 'a')

53. ('var', 'a')

48. eval result = 'app', e1 = ('var', 'a'),
tree = ('app', ('app', (...), (...)), ('app', (...), (...)))

53. return ('app', ('var', 'a'), ('app', (...), (...)))

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

2.8.3 Comments and Questions

For week 8, would there be a cleaner way to automate lambda reduction? For week 9, is there any way to iterate on these lambda calculus algorithms to make them more efficient or accurate?

2.9 Week 10

2.9.1 Notes

This week we discussed algorithms and went over the concepts of confluence, termination, and (unique) normal forms. We used the bubble sort as an example of confluence, to explain how we need an algorithm that always produces the same output for a given input, making it predictable and reliable. Additionally we

discussed termination, where the algorithm will end eventually. In order to determine this, the "size" of the data needs to be defined, as well as proof that the "size" descreases with each loop. We also went over the syntax of the star arrow, what mathemeticians call the reflexive and transitive closure of ->.

2.9.2 Homework

We continued working on the prgramming assignment and concepts from weeks 8 and 9. We were asked to reflect on our experience with the previous homework and with programming assignment 3.

1. I found the most challenging part of the homework to be trying to understand the theory behind the code and how to interpret the functions. Once I figured that out I found working with it significantly easier.

2. In order to come up with how to solve Assignment 3 I had to understand what each function was doing and use the debugger to understand where the output was wrong.

3. I think it is interesting how we can take syntaxes that seem so imcompatible with programming languages and turn them into a working program, even down to the little details of how variables are processed. It really shows the power of coding and programming languages in general.

2.9.3 Comments and Questions

Are we only considering these qualities of an algorithm in a vacuum? Would there not be other factors that could effect the predicatbility and performance, or is the underlying logic immune to external, unpredicatble factors?

2.10 Week 11

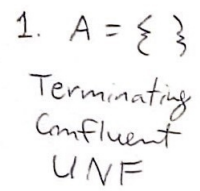
2.10.1 Notes

For week 11 we covered the concept of abstract reduction systems and explored how to interpret and solve them. These systems only require the reduction of strings rather than an entire tree, allowing us to study its properties such as its normal form and whether or not it terminates.

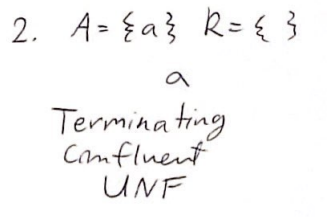
2.10.2 Homework

I drew pictures of 7 ARSs as shown below:

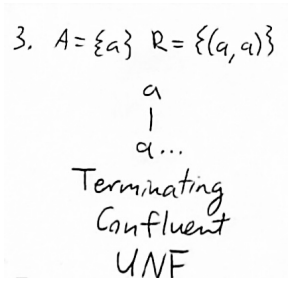
1. A = []



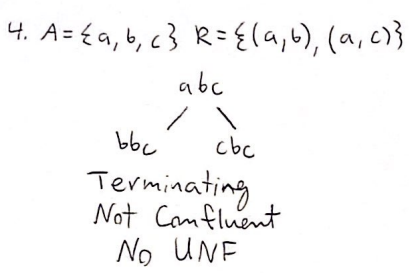
2. $\mathbf{A} = [a]$ **and** $\mathbf{R} = []$



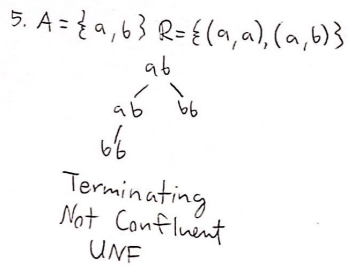
3. $\mathbf{A} = [a]$ **and** $\mathbf{R} = [(a, a)]$



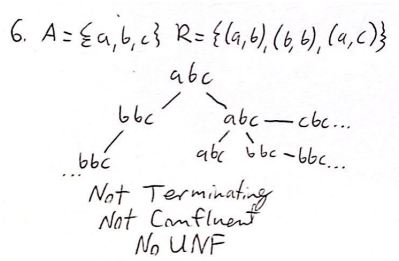
4. $\mathbf{A} = [a, b, c]$ **and** $\mathbf{R} = [(a, b), (a, c)]$



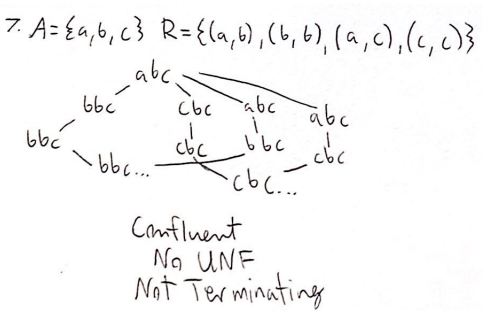
5. $\mathbf{A} = [a, b]$ **and** $\mathbf{R} = [(a, a), (a, b)]$



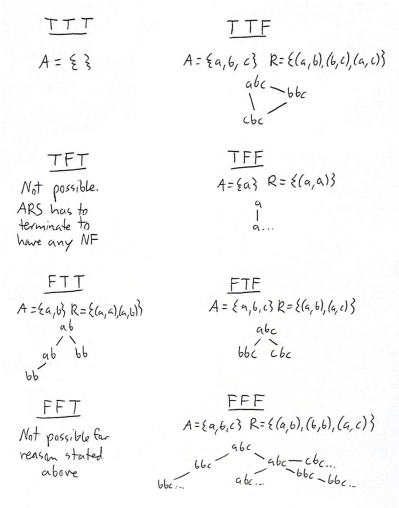
6. $\mathbf{A} = [a, b, c]$ **and** $\mathbf{R} = [(a, b), (b, b), (a, c)]$



7. $\mathbf{A} = [a, b, c]$ **and** $\mathbf{R} = [(a, b), (b, b), (a, c), (c, c)]$



I also drew examples of ARSs for each quality combination, whether it was confluent, terminating, or had a unique normal form:



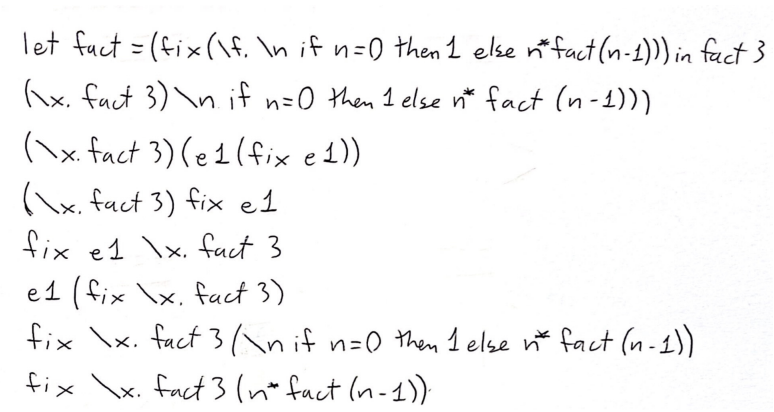
2.10.3 Comments and Questions

Could there be any possible theorem or representation of a reduction system that allows normal forms to exist without termination? Is this impossible only due to the rules mathematicians have defined or is it just naturally and scientifically impossible?

2.11 Week 12

2.11.1 Homework

For week 12 we covered fixed point conversions where we worked on one reduction for homework. My reduction for the problem "let rec fact = \n. if n=0 then 1 else n * fact (n-1) in fact 3", following the given template, was as follows:



2.11.2 Comments and Questions

I found these lectures helpful for getting the basic concept, but in order to understand it on a deeper level in order to implementing the let, letrec, and fix grammar rules, I looked for some resources online. I found an interesting source explaining the concept of the Y-combinator. Essentially, the concept is to add two numbers by "repeatedly taking one away from argument y and adding it to argument x until there is no more y to take" [5]. This explanation helped me understand the concept How complex can these expression reductions get before we can no longer create an algorithm that can handle them?

2.12 Week 13

2.12.1 Notes

We talked about differences between programming languages, the different design decisions we make, and the criteria for those decisions, that being simplicity and naturality.

We also went over induction riddles like the prisoner hat riddle, where you have to connect the past and future in order to correctly guess the hat colors. Since only nine people need to guess correctly, the first person can use their answer to instruct the rest of the people in line what number of a certain hat color they should be looking for, and base their own answer on the previous answer given (the past), as well as their observations of the hats in front of them and how their answer will inform the rest of the prisoners (the future).

This riddle illustrates the concept of "nested conditioning" that can open the way for breakthroughs in fields like AI and game theory [6].

2.12.2 Homework

We continued to add to our programming language project for homework. I completed exercise 7 of the String Rewriting Exercises:

Considering the rules:

ab -> a

bb -> b
aa -> b

and the rule that the letter order does not matter, I answered the question of what the last color would remain in an urn of 198 black "b" balls and 99 white "a" balls.

To solve this system, I would have to figure out the invariant, the property of the ruleset that does not change. This would be the parity of white balls, since the ab -> a rule flips the parity whenever applied, and the other rules keep the parity the same. But since we start with an odd number of white balls, then assuming that the first rule is applied at least once, the parity of white balls will end up odd. And since there is an odd number of white balls, and the black balls are always reduced in pairs, we will get to a point where there is a single black ball remaining at the end. Therefore, the last ball would be black.

When you consider this system with an unknown number of black and white balls, the color of the last ball would depend on the initial parity of the number of white balls. As explained above, starting with an odd number would result in one black ball remaining, but starting with an even number would result in one white ball remaining.

2.12.3 Comments and Questions

Is there a limited number of scenarios to which these algorithms can be applied? What do we do when we encounter a problem that seemingly cannot be solved with an algorithm but complex enough that we could not solve it by hand (if such a problem exists)?

3 Lessons from the Assignments

3.1 Introduction

I worked on each project and milestone on my own, with the exception of using Cursor AI and getting some help from the professor during office hours.

It hit me just how much more complicated the process of making something like a calculator would be. When trying to make it from scratch, I could only manage to create a system that took a specific type of input, a pair of characters with the first being a number and the second being an operator symbol. If the input did not match this specific order then it would not work. Without the concepts we discussed in class, it was difficult to visualize a system that performed specific actions with a given input while still being versatile enough to take many types of input in vastly different orders.

3.2 Parsing Input

That is why I think the system of translating input into a tree is so interesting and so versatile. Addmittedly it took a while for me to understand how the program worked but once I figured it out I realized how helpful of a concept it is. Being able to parse input so that it can handle any format so long as it meets a grammar rule, allows for a much more versatile program. Inputs do not have to be taken in specific pairs in the way I wrote my calculator, and it has been very easy to build onto this program. Rather than having to create an entirely new program, the way I had to when I went from taking just three characters to multiple, is crucial for any longterm program, especially ones that receive frequent software updates. So I found it very helpful to learn how to parse inputs properly.

3.3 Lark Grammar

I found the lark grammar rules to be incredibly confusing at first, but once I figured out how they work I realized just how efficient and versatile they are. The way that you can instruct the parser on how to parse input using this grammar, and how to enforce the precedent of operations was helpful in constructing an

efficient programming language without having to create countless amounts of functions like I would have if if this project were built with a higher-level language.

I found the ARS trees we drew in Week 4 to be especially helpful for understanding the basic concept of the grammar, as well as how to visualize it. It took me a little longer to understand the syntax of the grammar once we started adding different levels, but

3.4 Combinator

I used the concept of the Y-combinator, with the help of the Sookocheff's [5] explaination to understand the concept of letrec in order to implement it. Since we are writing this program only using Lambda calculus concepts, having this combinator is helpful for allowing a lambda function to effectively call itself. With this, I figured out how to get it to recursively refer to itself until there is no "Y" portion left to refer to. Thus, we can have recursion in a programming langauges without needing to create specific recursive functions.

3.5 Operator Binding

When adding the rules for creating lists, I considered the way the operators associate with or "bind" to their operands. The higher the precedent level, the more closely operators bind. So I considered how tightly I would want each of these operations to bind to operands in a string of input. Since the sequencing operator is a high-level operation, it has to get one of the highest precedents. Nil and cons are keywords like var and num so they share the same precedent level. The remaining rules should be able to nest deeply so they would get a lower level of precedence. It was easier to visualize this concept when reflecting on the exercises where we practiced implementing levels of precedence. Knowing how to create rules that can properly nest low level input within higher levels of input like parentheses was crucial for moving on to things like sequencers.

3.6 Beta Reduction

Another concept that was helpful for understanding and implenting the substitute() function was the concept of beta reduction that we practiced. Knowing what should be reduced and how was helpful for constructing the substitute() function, especially when considering the later rules that we add.The concept of capture-avoiding substitution is relevant here because it is important that the meaning of the initial AST should not change when substituting new values into the tree.

3.7 Normal Forms

Another concept we learned concerning reduction was the concept of a normal form. I think it was helpful to keep in mind how to reduce an AST down to its simplest form. Taking the exercises we did in class and for homework, I found that it was easier to understand the behavior of the evaluate(), substitute(), and linearize() functions when thinking about them as expanded applications of reduction. Viewing the whole program as one large reduction system with the aim of taking an input and using the grammar rules to substitute and evaluate each data type until a normal form, the final output, defintely helped me with how to approach the program as a whole.

3.8 String Reduction

In addition to normal AST reduction, practicing string reduction was helpful for getting used to reduction systems and being able to analyze them without the additional data creating a distraction. I found that I could understand concepts like confluence much easier when the system was laid out as simply as possible.

4 Conclusion

I found it interesting just how many layers there are to programming. We are taught very early on about concepts like high- and low- level programming languages, but we're never given a good idea of the foundation of programming languages. This course has given me a better understanding of the theories and concepts that make up all these languages that we use. I think knowing these details is helpful if we were ever interested in creating languages of our own or even modifying existing ones.

Additionally, from our lessons and from my research I saw how something as simple as lambda calculus could actually be beneficial for breaking down complex problems into their most basic elements to get a better understanding of them – something higher-level programming languages would have a harder time achieving. So even if we were not planning on creating a whole new language, we could still apply the principles we learned to solve very complex problems that may come up in our future as computer scientists and software engineers.

I found it interesting when concepts I was already famililar with, like operator precedent, were explained in greater detail. The trees we drew helped me visualize the way that languages read and parse input. I found that visuals like the drawings, the hanoi game, and the videos we watched made it much easier to understand the concepts we were learning, and gave me a more concrete way to think about otherwise abstract concepts.

One improvement to the course would be to make clearer connections between each new lesson we learn. In hindsight, I can see how one lecture leads into the next, but experiencing the content week by week left me a little lost. I think focusing on reinforcing the material we have already learned while showing what problems or concepts lead to the next topic would be helpful. Or, having an overview of what we are covering before jumping in so we know what to expect ahead of time would also be helpful. Another improvement would be putting the concepts we learn in the scope of the programming we are familiar. While I understood the concepts individually, I still had trouble connecting them to the languages I am familiar with, so putting some more emphasis on that would help me ground these concepts better in my understanding of computer science.

References

[1] M. Krichen, The Interplay Between Mathematics and Computer Science, Scholarly Community Encyclopedia, 2023.

[2] Uri Levy, The Magnetic Tower of Hanoi, Atlantium Technologies, no date.

[3] Author unknown, COS 441- Syntax - Feb 6, 1996, Princeton, 1996.

[4] Ugo de'Liguoro and Riccardo Treglia, From semantics to types: The case of the imperative λ-calculus, ScienceDirect, 2023.

[5] Kevin Sookocheff, Recursive Lambda Functions the Y-Combinator, Self-published, 2018.

[6] A. Stuhlmüller and N.D. Goodman, Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs, ScienceDirect, 2014.