CPSC-354 Report

John Robert Mulhern
Chapman University

December 1, 2024

Abstract

This document will contain the various assignments completed by John Mulhern over the course of the CPSC 354 Programming Languages course. For any questions, comments, or concerns in this document, feel free to reach out to John Mulhern at his email, mulhern@chapman.edu, or by phone number: (208)-451-3484.

Contents

1 Introduction

This report will document my learning throughout the course. It will be a collection of my notes, homework solutions, and critical reflections on the content of the course. Something in between a semester-long take home exam and my own lecture notes.[1]

To modify this template I would need to modify the source `report.tex` which is available in the course repo. For guidance on how to do this read both the source and the pdf of `latex-example.tex` which is also available in the repo. Also check out the usual resources (Google, Stackoverflow, LLM, etc). It was never as easy as now to learn a new programming lanugage (which, btw, LaTeX is).

For writing LaTeX with VSCode use the LaTeX Workshop extension.

There will be deadlines during the semester, graded mostly for completeness. That means that I will get the points if I submit in time and are on the right track, independently of whether the solutions are technically correct. I will have the opportunity to revise my work for the final submission of the full report.

The full report is due at the end of the finals week. It will be graded according to the following guidelines.

Grading guidelines (see also below):

- Is typesetting and layout professional?
- Is the technical content, in particular the homework, correct?
- Did I find interesting references [BLA] 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.

[1] One purpose of giving the report the form of lecture notes is that self-explanation is a technique proven to help with learning, see Chapter 6 of Craig Barton, How I Wish I'd Taught Maths, and references therein. In fact, the report can lead you from self-explanation (which is what you do for the weekly deadline) to explaining to others (which is what you do for the final submission). Another purpose is to help those of you who want to go on to graduate school to develop some basic writing skills. A report that you could proudly add to your application to graduate school (or a job application in industry) would give you full points.

2 Week by Week

2.1 Week 1

Tuesday: Orientation and introduction to the course.
Thursday: First Lab on Tuesday's content.

2.1.1 Notes and Homework

Our homework for this week was to finish levels 5-8 of the tutorial world inside the Natural Numbers Game provided to us in class. On Tuesday of week 1, we went over the game in basic detail, covering levels 1-4 as to become used to the website so we could begin our first challenge. The solutions to each of the worlds can be found below in an itemized format.

- `World 5 Solution: rw[add_zero] , rw[add_zero] , rfl.`
- `World 6 Solution: rw[add_zero c] , rw[add_zero b] , rfl.`
- `World 7 Solution: rw[one_eq_succ_zero] , rw[add_succ] , rw[add_zero] , rfl.`
- `World 8 Solution: rw[two_eq_succ_one, one_eq_succ_zero] , rw[add_succ] , rw[add_zero] , rw[<- one_eq_succ_zero, <- two_eq_succ_one, <- three_eq_succ_two, <- four_eq_succ_three]`

World 5's solutition is to rewrite the equation by adding zero onto a variable **b** or **c**, and as we know from a descrete math proof described as $\forall n \in N, n + 0 = n$, adding zero to any number provides the same equivalent number. This becomes useful later in sections seven and eight when we begin dealing with adding zero to successor values.

2.1.2 Comments and Questions

Looking at Discrete Math over the past few days and getting a refresher on the course since I took it a few semesters ago has been very interesting. I have enjoyed the tutorial levels of the Natural Numbers Game, and I particularly enjoyed their explanations for the proofs and early concepts for Discrete Math. Had I known about this website when I was taking the course, I have a feeling it would have been a great resource to support my understanding of those proofs and other concepts.

My question then in relation to discrete mathematics comes more so with how we utilize those proofs on paper verses when they are used in a computational environment. For example, writing a Discrete Math proof can often take a significant amount of time and paper to create, depending of course on the operation. For something as simple as addition or multiplication, using the proofs outlined in Discrete Math can turn a simple problem, such as $2 * (3 + 2 + 4)$, into a massive multi-page proof. However, when a computer runs such a problem, it concludes the correct answer in an astoundingly short amount of time.

My question is then, what is the largest, hardest, and most difficult proof a person could do by hand that can be done in seconds by a machine?[2]

2.2 Week 2

Tuesday: What is a proposition? Covering Eric Villanueva's question "I wonder how the computer or code implements the logic we have in understanding discrete mathematics to make computtations. How do we define the idea of successors so that the computer knows how to carry out calculations?"

[2]It is important to learn to ask *interesting* questions. There is no precise way of defining what is meant by interesting. You can only learn this by doing. An interesting question comes typically in two parts. Part 1 (one or two sentences) sets the scene. Part 2 (one or two sentences) asks the question. A good question strikes the right balance between being specific and technical on the one hand and open ended on the other hand. A question that can be answered with yes/no is not an interesing question.

2.2.1 Notes and Homework

- World 1 Solution: `Induction n with d, rw[add_zero], rfl, rw[succ_eq_add_one], rw[one_eq_succ_zero] rw[add_zero], rw[add_succ], rw[n_ih], rfl.`
- World 2 Solution: `Induction b with d hd, rw[add_zero, add_zero], rfl, rw[add_succ, add_succ], rw[hd], rfl.`
- World 3 Solution: `Induction b with b hb, rw[add_zero, zero_add], rfl, rw[add_succ, succ_add, hb], rfl.`
- World 4 Solution: `Induction c with c hc, rw[add_zero], rw[add_zero], rfl, rw[add_succ], rw[add_succ], rw[hc], rfl.`
- World 5 Solution: `Induction c with c hc, rw[add_zero, add_zero], rfl, rw[add_succ, add_succ], rw[succ_add], rw[hc], rfl.`

World 5's solution is to discover the add_right_comm function through other theorems we are already familier with. Utilizing several Lean theorems, such as `rw[add_succ]`, `rw[add_zero]`, and proving by indcution, we can rewrite $a + b + c = a + c + b$ into $\text{succ}(a + c + b) = \text{succ}(a + c + b)$. Proving this fact using standard mathematics follows a similar set of rules as well. Should you aim to prove by induction, you can then add zeros and use reflexivity to reconstruct an equation to prove add_right_comm.

2.2.2 Comments & Questions

In reference to the additional reading for this week, the dialogue *Little Harmonic Labrinth*, it describes recursion in a much more palatable format. Essentially, you have an item operating within itself endlessly. This concept is very useful as a time saver when it comes to certain coding tasks, but my question in relation to recursion is; How can recursion be optimized in a computer environment as to be effective without running out of resources in relation to large tasks?

2.3 Week 3

2.3.1 Notes & Homework

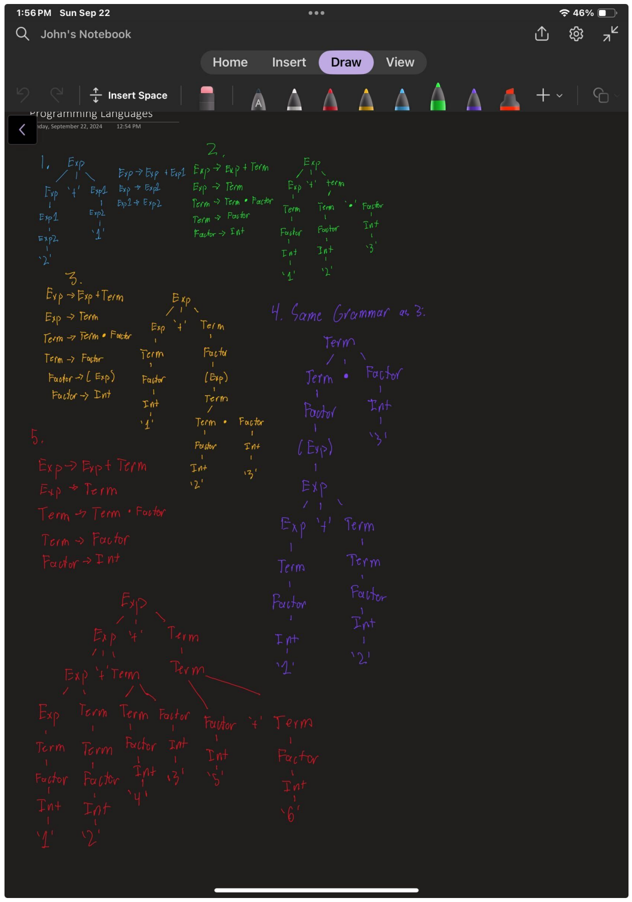
2.3.2 Comments & Questions

2.4 Week 4

2.4.1 Notes & Homework

Tuesday: Covering individuals projects from last week.
Thursday: Learning to use Cursor and its various features. Progress on Assignment 1 and associated lab work.

Homework: Solve homework exercises in relation to parsing trees. Below, questions 1-5 are completed with specific solutions being in differing colors.



2.4.2 Comments & Questions

My question for this week centers around the ideas of parsing trees. How are parsing trees best utilized when in a coding environment? They seem very useful for mathematical equations and logical reasoning, but I am unsure of how they would be useful when programming.

2.5 Week 5

2.5.1 Notes & Homework

Tuesday: Working on Assignment 2, due Wednesday.
Thursday: Begin $\wedge$ Tutorial: Party Invites. Solutions to levels 1-8 are listed below.

- `World 1 Solution:` exact $todo\_list$.
- `World 2 Solution:` exact$\langle p, s\rangle$.
- `World 3 Solution:` have $ai := and\_introai, haveou := and\_introou, exact\langle ai, ou\rangle$.
- `World 4 Solution:` exact and_left vm .
- `World 5 Solution:` exact and_right h .
- `World 6 Solution:` exact$\langle h1.left, h2.right\rangle$.
- `World 7 Solution:` have $h1 := h.left$, have $h2 := h.right$, have $h3 := h2.left$, have $h4 := h3.left$, exact $h4.right$.

- `World 8 Solution:` have $h1 := h.left$, have $a := h1.right$, have $b := h1.left$, have $h2 := h.right$, have $h3 := h2.right$, have $h4 := h3.left$, have $c := h4.left$, exact$\langle a, b, c\rangle$.

The solution for world 8 written in mathematical logic goes as follows:

- Assume $h = ((P \land S) \land A) \land \not I \land (C \land \not O) \land \not U$
- $h1 = (P \land S) \land A$ and_left on h (1)
- $a = A$ and_right on h1 (2)
- $b = P \land S$, and_left on h1 (3)
- $h2 = \not I \land (C \land \not O) \land \not U$ and_right on h (4)
- $h3 = (C \land \not O) \land \not U$, and_right on h2 (5)
- $h4 = C \land \not O$, and_left on h3 (6)
- $c = C$, and_left on h4 (7)
- $A \land C \land P \land S$ (8)

2.5.2 Comments & Questions

My question for this week pertains to the complexity of lean. As we progress through different worlds of the game, the harder and more complex the problems we have to solve, especially on boss levels, where combinations from prior levels are brought together to make one much more challenging level. In math however, items that are significantly more complex than simple logic proofs or basic arithmetic exist. How then does Lean simplify complex math down into assumed proofs for much larger and complex equations or logic proofs, and what problems take longer to solve using a Lean logic rather than standard math or coding logic?

2.6 Week 6

2.6.1 Notes & Homework

Tuesday: We began to cover the ideas of lambda calculus, introducing some ideas for lambda parsing, as well as the Lean Logic levels on implications.
Thursday: We continued to cover lambda calculus, specifically lambda abstraction and application.

- `World 1 Solution:` exact $backery\_servicep$.
- `World 2 Solution:` exact $\lambda(h : C) \mapsto h$.
- `World 3 Solution:` exact $\lambda h : I \land S \mapsto$ and_intro(and_right h) $h.left$.
- `World 4 Solution:` exact $h1 >> h2$.
- `World 5 Solution:` have $step1 := h1 >> h3$.
- `World 6 Solution:` exact $\lambda c \mapsto$ and_intro $c >> h$.
- `World 7 Solution:` exact $\lambda(cd : C \land D) \mapsto h\ cd.left\ cd.right$.
- `World 8 Solution:` exact $\lambda(s : S) \mapsto$ and_intro$(h.left\ s)(h.right\ s)$.
- `World 9 Solution:` have $sr := \lambda(r : R)(_{:}S) \mapsto r$ have $nsr := \lambda(r : R)(_{:}\not S) \mapsto r$ exact $\lambda r \mapsto \langle srr, nsrr\rangle$.

2.6.2 Comments & Questions

My question this week since we delved deeper into the world of lambda calculus is this: Since lambda calculus is a tool primarily used for function abstraction and fuction applications, is there a limit to what can be encoded within lambda calculus? Is there a point in which it struggles or, given enough time, could lambda calculus parse anything given the proper conditions and rules? I would assume the latter given infinite time, but if there is a more efficient method for certain calculations, I'd be interested to know how the computer parses those problems and what tools are used in the analyzing of such a function.

2.7 Week 7

2.7.1 Notes & Homework

Tuesday: We continued to cover ideas of lambda calculus and covered last weeks questions on discord.
Thursday: We introduced the idea of church numerals, covered the weekly homework, and covered more lambda calculus calculations in class.

The secondary aspect of the homework this week asked a question to explain the function on natural numbers $(\lambda m.\lambda n.mn)$ implements. It takes two functions m and n and applies m to n. In relation to church numerals, the function represents addition between two listed functions.

Week 7 Homework:

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

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

↓

\n. (\f.\x.f (fx)) n

↓

(\n.(\f.\x.f (fx)n) where we sub n for:

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

↓

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

↓

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

↓

looking at we can sub x in for f,

resulting in:

\x . x (x(xx))

which makes the final reduced term

equal to the church numeral for 3!

2.7.2 Comments & Questions

With how we've covered church numerals in this homework and in class, my question this week is how quickly can these complex types of church numeral equations be calculated in a computer environment, and how high do equations like these tend to scale? Is there a better more preferential option for more complex equations or can this type of calculation be scaled to meet even the most complex forms of calculus?

2.8 Week 8 & 9

2.8.1 Notes & Homework

Tuesday: We introduced the third homework assignment, covering the premise as well as assigning partners for the project. We continued on with lambda calculus examples, furthering our knowledge of substitution

and other important functions.
Thursday: We continued work on the code provided to us by our professors, executing steps one through nine as provided on Canvas.

- Step 1: Not required for HW.
- Step 2: I added these functions to the test.lc file:
  $\lambda x.\lambda y.x$
  $\lambda x.\lambda y.y$
  $\lambda x.\lambda y.\lambda z.xz$
  $\lambda x.xx$
  $(\lambda x.xx)(\lambda x.xx)$
  In addition to these new functions though, to answer the second part of step 2, the reason the expression $a\ b\ c\ d$ reduces to $(((a\ b)\ c)\ d)$ is because association between variables is always prioritized to the left!
- Step 3: Capture avoid substitution works due to the interesting principle that variables in lambda calculus can be substituted for any other variable name so long as they are not bound in an equation. For instance, a funtion of x means the same thing as a function of a or a function of b. The variable's name has no extreme relevance unless the function dictates so through a different set of circumstances.

  Imagine for a moment the equation $(\lambda x.(\lambda y.yx))$. One could rewrite this equation to be $(\lambda a.(\lambda b.ba))$, and this would be a functional form of substitution. However, if you took a closer look at the internal expression $(\lambda y.yx)$, one could say that this smaller function has a free variable x, meaning it isn't associated with any other lambda values in the expression.

  Now though, if we took that internal expression and changed the y variable to x such as this, $(\lambda x.xx)$, our originally free variable x has now become captured by our variable name change, completely altering the expression. Therefore when substituting variables, we must perform Capture Avoiding Substitution as to avoid capturing these free variables and fundamentally altering the expression.

  In our code provided to us, this is done with the function `NameGenerator`. Using variables passed to the function, it uses a counter to perpetually increase the number of variables passed through the function allowing for an incredible amount of increasing random variables attached to the string `Var`. When a function is passed into the interpreter and subsitution is neccesary, the output will more than likely contain the values `Var[i]`, where i is the number of times the `NameGenerator` function has been called.
- Step 4: The answer to this question is interesting because it almost feels like it was slightly spoiled by Step 5, but the answer is no. I do not always get what I expected when I input values due to functions that cannot be reduced to a normal form.
- Step 5: The MWE, or Minimum Working Expression value that I found is $(\lambda x.xx)(\lambda x.xx)$, which was also a function provided to us in the Church Encodings information file we covered in class. When executed, the function will return itself over and over and over again in an infinite loop of recursion until a check is made stopping the function from running infinitly.
- Step 6: This step is not required for the HW.
- Step 7: Upon the execution of the function $((\lambda m.\lambda n.mn)(\lambda f.\lambda x.f(fx)))(\lambda f.\lambda x.f(f(fx)))$, the output $(\lambda Var3.(\lambda Var5.(Var3(Var3(Var3(Var3(Var3(Var3(Var3(Var3(Var3Var5))))))))))$ is produced.

  For my output that I recieved from looking at the debugger for steps 7 & 8, I wondered if I would recieve different output due to my modifications to my `evaluate` function prior to completing these steps. I beleive that I do have very different output, but following the instructions to the best of my ability, below are the responses I recieved for Step 7:

  - $('lam','Var1',('app',('lam','f',('lam','x',('app',('var','f'),('app',('var','f'),('var','x'))))),('var','Var1')))$
  - $('lam','Var2',('app',('var','Var1'),('app',('var','Var1'),('var','Var2'))))$
  - $('lam','Var1',('app',('lam','f',('lam','x',('app',('var','f'),('app',('var','f'),('var','x'))))),('var','Var1')))$
  - $('lam','Var3',('app',('lam','f',('lam','x',('app',('var','f'),('app',('var','f'),('app',('var','f'),('var','x')))))),$
    $('app',('lam','f',('lam','x',('app',('var','f'),('app',('var','f'),('app',('var','f'),('var','x')))))),('var','Var3')))$
  - $('lam','Var4',('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),('var','Var4')))))$
  - $('lam','Var5',('app',('lam','Var4',('app',('var','Var3'),('app',('var','Var3'),$
    $('app',('var','Var3'),('var','Var4'))))),$
    $('app',('lam','Var4',('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),('var','Var4'))))),$
    $('app',('lam','Var4',('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),('var','Var4'))))),$
    $('var','Var5')))))$
  - $('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),('var','Var5'))))$
  - $('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),$
    $('app',('var','Var3'),('var','Var5')))))))$
  - $('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),$
    $('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),('var','Var5'))))))))))$
  - $('lam','Var5',('app',('lam','Var4',('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),$
    $('var','Var4'))))),('app',('lam','Var4',('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),$
    $('var','Var4'))))),('app',('lam','Var4',('app',('var','Var3'),('app',('var','Var3'),('app',('var','Var3'),$
    $('var','Var4'))))),('var','Var5')))))$
  - $('lam','Var3',('app',('lam','f',('lam','x',('app',('var','f'),('app',('var','f'),('app',('var','f'),('var','x')))))),$
    $('app',('lam','f',('lam','x',('app',('var','f'),('app',('var','f'),('app',('var','f'),('var','x')))))),$
    $('var','Var3'))))$
  - `Result:` $(\lambda Var3.(\lambda Var5.(Var3(Var3(Var3(Var3(Var3(Var3(Var3(Var3(Var3\,Var5))))))))))$

- Step 8: Similar to step seven, due to my alterations to the `evaluate` function, my code below is split into my two instances of the `evaluate` function and their associated outputs.
  - `e1 =` $('var','m')$`, e2 =` $('var','n')$
  - `e1 =` $('var','f')$`, e2 =` $('var','x')$
  - `e1 =` $('lam','m',('lam','n',('app',('var','m'),('var','n'))))$`,`
    `e2 =` $('lam','f',('lam','x',('app',('var','f'),('app',('var','f'),('var','x')))))$
  - `e1 =` $('var','Var1')$`, e2 =` $('var','Var2')$
  - `e1 =` $('lam','m',('lam','n',('app',('var','m'),('var','n'))))$`,`
    `e2 =` $('lam','f',('lam','x',('app',('var','f'),('app',('var','f'),('var','x')))))$
  - `e1 =` $('lam','Var1',('lam','Var2',('app',('var','Var1'),('app',('var','Var1'),('var','Var2')))))$`,`
    `e2 =` $('lam','f',('lam','x',('app',('var','f'),('var','x'))))$
  - `e1 =` $('var','f')$`, e2` $('var','x')$
  - `e1 =` $('lam','f',('lam','x',('app',('var','f'),('var','x'))))$`, e2 =` $('var','Var3')$
  - `e1 =` $('var','Var3')$`, e2 =` $('var','Var4')$
  - `e1 =` $('lam','f',('lam','x',('app',('var','f'),('var','x'))))$`,`
    `e2 =` $('lam','Var4',('app',('var','Var3'),('var','Var4')))$

- `e1 =` $('var', 'Var3')$, `e2 =` $('var', 'Var4')$
- `e1 =` $('lam', 'Var4', ('app', ('var', 'Var3'), ('var', 'Var4')))$, `e2 =` $('var', 'Var5')$
- `e1 =` $('var', 'Var3')$, `e2 =` $('var', 'Var5')$
- `e1 =` $('lam', 'Var1', ('lam', 'Var2', ('app', ('var', 'Var1'), ('app', ('var', 'Var1'), ('var', 'Var2')))))$,
  `e2 =` $('lam', 'f', ('lam', 'x', ('app', ('var', 'f'), ('var', 'x'))))$
- `Result:` $(\lambda Var3.(\lambda Var5.(Var3\ Var5)))$

2.8.2 Comments & Questions

My question for week 8 is how have the use cases for lambda calculus evolved over its existence? What was it originally designed for? What is it primaraly used for now? Are those answers different, and if so, when did they branch?

My Question for week 9 is given that there are certain equations that do not reduce to a normal form like the MWE, what are other examples of functions that do not reduce to a normal form, and what do we do with said functions? What do they represent in a math context similar to standard calculus?

2.9 Week 10

2.9.1 Notes

Tuesday: As we were granted an extension on the homework and programming assignment for weeks 8 & 9, we continued learning about lambda calculus and its various types of functions.
Thursday: I was unfortunetly absent from the class, but I continued working on the programming assignments and assoicated homework assignments. Looking at the posted notes, what was covered in class was further examples of ARS termination as well as an introduction to sorting algorithms through Bubble Sort experimentation.

2.9.2 Homework

What I found most challenging about working through homeworks 8 & 9 was the pieces of the homework regarding debugging. Since I did the homework a little out of order, I was unfortunetly graced with a significantly larger debugging task than I would have otherwise encountered due to my recursive setup to solve the Assignment3 premise.

My solution to said problem was discovered after some experimentation with the evaluate function. After some initial trial and error, I was only able to successfully reduce half of a large inputed expression to a desired normal form. It was then I went down the thought path of beta-reducing both halves of a large expression in order to calculate a properly reduced outcome, which yielded correct results upon several tested executions.

I really enjoyed working on this programming assignment as it proved to me once again that I tend to get in my own head with certain challenges. This programming assignment was one that I though would be significantly difficult, especially due to the fact that I was working alone. However, upon discovering a possible method of execution for the programming assignment, I found that I was able to create a solid solution in a timely manner. This assignment is a nice reminder to myself that I know more than I tend to give myself credit for, and to be confident in the material that I know.

2.9.3 Comments & Questions

My question for this week revoles around the bubble sort method covered last session on Thursday. For all the sorted values leading to successfully calculated normal forms, is there any way to affect the efficiency of the sorting algorithm? Assuming that a successful conversion of a function to normal form is a success, how can we guarantee mass conversion rates to normal form to boost a positive efficiency rate?

2.10 Week 11

2.10.1 Notes & Homework

Tuesday: We began identifying ARS's and constructing various forms of them under certain criteria of either confluent, terminating, or possesing a unique normal form. Below is my rendition of the homework for this week.

confluent	terminating	has unique normal forms	example
True	True	True	4.
True	True	False	1.
True	False	True	3.
True	False	False	5.
False	True	True	5.
False	True	False	2.
False	False	True	3.
False	False	False	7.

2.10.2 Comments & Questions

My question for this week in relation to ARS's is how can they be used in practical computation? Are they used as a simplification method or are they used in calculations or other forms of connectivity?

2.11 Week 12

2.11.1 Notes

Tuesday: We continued review on ARS's, specifically with rules of confluence, termination, and unique normal forms with regards to rewriting rules.
Thursday: We were introduced to milestone 2 of the fourth programming assingment as well as the continuation of learning about ARS's.

2.11.2 Homework

- (Exse 1) The rewrite rule is

```
ba -> ab
```

  - Why does the ARS terminate? Because there is a singular non-repeating normal form.
  - What is the result of a computation (the *normal form*)? The transition of ba -> ab.
  - Show that the result is unique (the ARS is *confluent*). As there are no other arrows, there is no other possible computation to yield ab.
  - What specification does this algorithm implement?
    Confluence, termination, and a unique normal form.

- (Exse 2). Rewrite rules are

```
aa -> a
bb -> a
ab -> b
ba -> b
```

  - Why does the ARS terminate? There is a final possible reduction.
  - What are the normal forms? a and b.
  - Is there a string `s` that reduces to both `a` and `b`? abba
  - Show that the ARS is confluent. While there isn't a unique normal form, the multiple items still reduce to a single term.
  - The next questions have all essentially the same answer:
    - Replacing `->` by `=`, which words become equal? [6] aa = bb, ab = ba
    - Can you describe the equality `=` **without making reference to the four rules above?** Two instances of a letter equal the same unit just as two different letters in any order equal their own unit.
    - Can you repeat the last item using modular arithmetic? $a \cdot b = b$; $a = 1$.
    - Which specification does the algorithm implement?
      Confluence and termination.

- (Exse 3) Rewrite rules are

- (Exse 3) Rewrite rules are

```
aa -> a
bb -> b
ba -> ab
ab -> ba
```

  - Why does the ARS not terminate? ba and ab loop forever.
  - What are the normal forms? a, b
  - Modify the ARS so that it is terminating, has unique normal forms (and still the same equivalence relation). $aa \to a$ $ba \to ab$ $bb \to b$ By dropping the last term, you can still maintain equivalence relations through the use of rewriting variables.
  - Describe the specification implemented by the ARS.
    Confluence

- (Exse 4) Rewrite rules are

```
ab -> ba
ba -> ab
```

Same questions as above. (This is a variation of Exse 1.)
This does not terminate as ab and ba loop forever. There are no normal forms

- (Exse 5) Consider the rewrite rules

```
ab -> ba
ba -> ab
aa ->
b ->
```

  - Reduce some example strings such as `abba` and `bababa`. for both 5 and 5b. non-terminating.
  - Why is the ARS not terminating? ab and ba loop forever.
  - How many equivalence classes does $\overset{*}{\longleftrightarrow}$ have? Can you describe them in a nice way? What are the normal forms? Equivalence classes are categorized by the counts of [Hint: It may be easier to first answer the next question.] a's and b's in the string. There are infinitely many equivalence classes. one for each pair (a, b) where a, b ≥ 0.
  - Can you change the rules so that the ARS becomes terminating without changing its equivalence classes? Yes, you can.
  - Write down a question or two about strings[7] that can be answered using the ARS. Think about whether this amounts to giving a semantics to the ARS. [Hint: The best answers are likely to involve a complete invariant.]

- **Remark:** A characterisation of the equivalence classes that mentions the reduction relation is not interesting. Question 1: Does a provided string reduce to the empty string? Question 2: How many a's and b's would remain after reducing a certain string?

- (Exse 5b) As Exse 5, but change `aa ->` to `aa -> a`.
  ab, ba still loop forever, equivalence classes are still governed
- (Exse 6) Consider the rewrite rules by infinitely many pairs of a and b.

2.11.3 Comments & Questions

My question for this week is: What are ways to reduce the likelyhood of there being a non-terminating result when creating an ARS, and how common is it when creating an ARS to find a terminating result where one wasn't expected?

2.12 Week 13

2.12.1 Notes & Homework

Tuesday: We began to investigate the idea of method functions, functions of assigning such as let, functions to help compute recursion, as well as if and else statements.
Thursday: We were assigned homework 13 to compute the factorial of 3 utilizng lambda calculus. We spend the majority of the class time working on the homework.

- $let\ rec\ fact = \lambda n.ifn = 0\ then\ 1\ else\ n * fact(n-1)\ in\ fact\ 3$
- $->let\ fact\ =\ fix\ (\lambda fact.\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n\ *\ fact\ (n-1))\ in\ fact\ 3$
- $->(fix\ (\lambda fact.\lambda n.\ if\ n\ =\ 0\ then\ 1\ else\ n\ *\ fact\ (n-1)))\ 3$
- $->(\lambda fact.\lambda n.if\ n = 0\ then\ 1\ else\ n * fact\ (n-1))\ (fix\ (\lambda fact.\lambda n.if\ n = 0\ then\ 1\ else\ n * fact\ (n-1)))\ 3$
- $->(\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n * (fix\ (\lambda fact.\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n\ *\ fact\ (n-1)))\ (n-1))\ 3$
- $->if\ 3\ =\ 0\ then\ 1\ else\ 3\ *\ (fix\ (\lambda fact.\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n\ *\ fact\ (n-1)))\ (3-1)$
- $->3\ *\ (fix\ (\lambda fact.\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n\ *\ fact\ (n-1)))\ 2$
- $->3 * (\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n * (fix\ (\lambda fact.\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n * fact\ (n-1)))\ (n-1))\ 2$
- $->3\ *\ (if\ 2\ =\ 0\ then\ 1\ else\ 2\ *\ (fix\ (\lambda fact.\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n\ *\ fact\ (n-1)))\ (2-1))$
- $->3\ *\ (2\ *\ (fix\ (\lambda fact.\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n\ *\ fact\ (n-1)))\ 1)$
- $->3\ *\ (2\ *\ (\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n\ *\ (fix\ (\lambda fact.\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n\ *\ fact\ (n-1)))\ (n-1))\ 1)$
- $->3\ * (2\ *\ (1\ *\ (fix\ (\lambda fact.\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n\ *\ fact\ (n-1)))\ 0))$
- $->3\ *\ (2\ *\ (1\ *\ (if\ 0\ =\ 0\ then\ 1\ else\ 0\ *\ (fix\ (\lambda fact.\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n\ *\ fact\ (n-1)))\ (0-1))))$
- $->3\ *\ (2\ *\ (1\ *\ 1))$
- $->3\ *\ 2$
- $->\ 6$

2.12.2 Comments & Questions

My question for this week: Given the emphasis on recursion in the homework, as functions tend to expand as they did in reference to the factorial function we needed to compute, they become increasingly harder and harder to keep track of with the human eye, but computers do not have this problem as far as I am aware. If there is such a threshold where computers begin to experience problems with keeping track of such large expressions, what defines that point? Is it standard accross all computers or is it based on specific components that would be diffrent across computers?

2.13 ...

...

3 Lessons from the Assignments

4 Conclusion

References

[BLA] Author, Title, Publisher, Year.