# CPSC-354 Report

Sarah Yoon
Chapman University

December 1, 2024

**Abstract**

## Contents



# 1 Introduction

# 2 Week by Week

## 2.1 Week 1

### Notes

Learned about some tactics and theorems

rfl: a tactic that proves theorems that take the form of X = X
rw: a tactic that rewrites a proof
one_eq_succ_zero: a theorem that proves 1 = succ 0 (there are also other similar existing theorems like

two_eq_succ_one and so on)
add_zero: a theorem that proves a + 0 = a.
add_succ: a theorem that proves a + succ b = succ(a + b)
succ_eq_add_one: a theorem that proves succ a = a + 1

**Homework**

Problem 5:
a b c are in the set of natural numbers.
Prove that both sides are equal to each other.
a + (b + 0) + (c + 0) = a + b + c
rw [add_zero] - uses the add_zero theorem to prove that b + 0 = b
This is rewritten to:
a + b + (c + 0) = a + b + c
rw [add_zero] - this is done again to prove that c + 0 = c
This is rewritten to:
a + b + c = a + b + c
rfl - this proves that both sides that look the same are equal to each other

Problem 6:
This is the same problem as 5 but will be approached in a different manner.
a + (b + 0) + (c + 0) = a + b + c
rw [add_zero c] - specifically applies the add_zero theorem to c, making c + 0 into c
This is rewritten to:
a + (b + 0) + c = a + b + c
rw [add_zero b] - specifically applies the add_zero theorem to b, making b + 0 into b
This is rewritten to:
a + b + c = a + b + c
rfl - this proves that both sides that look the same are equal to each other

Problem 7:
n is in the set of natural numbers.
Prove that both sides are equal to each other.
succ n = n + 1 rw[one_eq_succ_zero] - rewrite 1 into successor 0 This is rewritten to:
succ n = n + succ 0
rw[add_succ] - uses the add_succ theorem to change n + succ 0 into succ(n + 0)
This is rewritten to:
succ n = succ (n + 0)
rw[add_zero] - uses the add_zero theorem to prove that n + 0 = n
This is rewritten to:
succ n = succ n
rfl - this proves that both sides that look the same are equal to each other

Problem 8:
Prove that both sides are equal to each other.
2 + 2 = 4
rw[two_eq_succ_one] - rewrites 2 into succ 1
This is rewritten to:
succ 1 + succ 1 = 4

rw[one_eq_succ_zero] - rewrites 1 into succ 0
This is rewritten to:
succ (succ 0) + succ (succ 0) = 4
rw[four_eq_succ_three] - rewrites 4 into succ 3
This is rewritten to:
succ (succ 0) + succ (succ 0) = succ 3
rw[three_eq_succ_two] - rewrites 3 into succ 2
This is rewritten to:
succ (succ 0) + succ (succ 0) = succ (succ 2)
rw[two_eq_succ_one] - rewrites 2 into succ 1
This is rewritten to:
succ (succ 0) + succ (succ 0) = succ (succ (succ 1))
rw[one_eq_succ_zero] - rewrites 1 into succ 0
This is rewritten to:
succ (succ 0) + succ (succ 0) = succ (succ (succ (succ 0)))
rw[add_succ] - changes succ (succ 0) + succ (succ 0) into succ (succ (succ 0) + succ 0)
This is rewritten to:
succ (succ (succ 0) + succ 0) = succ (succ (succ (succ 0)))
rw[add_succ] - changes succ (succ (succ 0) + succ 0) into succ (succ (succ (succ 0) + 0))
This is rewritten to:
succ (succ (succ (succ 0) + 0)) = succ (succ (succ (succ 0)))
rw[add_zero] - changes succ (succ 0) + 0 into
This is rewritten to:
succ (succ (succ (succ 0))) = succ (succ (succ (succ 0)))
rfl - this proves that both sides that look the same are equal to each other

For level 5: add_zero is a Lean proof that a + 0 = a (a representing any number). In mathematics, there are laws for arithemic. One of them is called the identity which applies to addition and multiplication. For addition, it states that m + 0 = m = 0 + m. This is the exact same as the Lean proof, a + 0 = a, which can also be written as a = 0 + a.

#### Comments and Questions

Learning the root of mathematics is very eye-opening, and I am confident it will be the same for programming languages. It provides another perspective for elementary functions like 2 + 2 equals 4, which is different from just knowing it through memorization. I feel as though this is why people have been able to expand mathematically. This makes me wonder: how can looking through the core of programming help us better current languages (e.g. python, rust)?

### 2.2 Week 2

#### Notes

Recursion as a concept using the Towers of Hanoi: It is broken down into: moving a tower of n disks from x to y moving a tower of n+1 disks when it is already known how to move a tower of n disks The algorithm is made up of a bunch of "pushs" and "pops" The logic overall is a bunch of back and forth movement of the disks

Lean: induction proof with: induction n with d hd succ_add: proves that succ a + b = succ (a + b) add_comm x y: proves that x + y = y + x add_assoc: proves that a + b + c = a + (b + c) add_right_comm a b c: proves that a + b + c = a + c + b

**Homework**

Problem 1:
n is in the natural number set
Prove 0 + n = n.
induction n with d hd - starting a proof by induction
Now our first goal is:
0 + 0 = 0
rw[add_zero] - proves that 0 + 0 = 0
This is rewritten to:
0 = 0
rfl - this proves that both sides that look the same are equal to each other
Now, we prove our second goal
hd: 0 + d = d
0 + succ d = succ d
rw[add_succ] - proves that 0 + succ d = succ (0 + d)
This is rewritten to:
succ (0 + d) = succ d
rw[hd] - this replaces 0 + d with d
This is rewritten to:
succ d = succ d
rfl - this proves that both sides that look the same are equal to each other

Problem 2:
a b is in the set of natural numbers
Prove succ a + b = succ (a + b)
inductin b with d hd - starting a proof by induction
Now our first goal is:
succ a + 0 = succ (a + 0)
rw[add_zero] - proves that succ a + 0 = succ a
This is rewritten to:
succ a = succ(a + 0)
rw[add_zero] - proves that succ (a + 0) = succ a
This is rewritten to:
succ a = succ a
rfl - this proves that both sides that look the same are equal to each other
Now, we prove our second goal
hd: succ a + d = succ (a + d)
succ a + succ d = succ (a + succ d)
rw[add_succ] - proves that succ a + succ d = succ (succ a + d)
This is rewritten to:
succ (succ a + d) = succ (a + succ d)
rw[hd] - this replaces succ a + d with succ (a + d)
This is rewritten to:
succ (succ (a + d)) = succ (a + succ d)
rw[add_succ] - proves that succ (a + succ d) = succ (succ (a + d))
This is rewritten to:
succ (succ (a + d)) = succ (succ (a + d))
rfl - this proves that both sides that look the same are equal to each other

Problem 3:

a b is in the set of natural numbers
Prove a + b = b + a
induction b with hd - starting a proof by induction
Now our first goal is:
a + 0 = 0 + a
rw[zero_add] - proves that 0 + a = a
This is rewritten to:
a + 0 = a
rw[add_zero] - proves that a + 0 = a
This is rewritten to:
a = a
rfl - this proves that both sides that look the same are equal to each other
Now, we prove our second goal
n_ih: a + hd = hd + a
a + succ hd = succ hd + a
rw[add_succ] - proves that a + succ hd = succ (a + hd)
This is rewritten to:
succ (a + hd) = succ hd + a
rw[succ_add] - proves that succ hd + a = succ (hd + a)
This is rewritten to:
succ (a + hd) = succ (hd + a)
rw[n_ih] - replaces succ (a + hd) with succ (hd + a)
This is rewritten to:
succ (hd + a) = succ (hd + a)
rfl - this proves that both sides that look the same are equal to each other

Problem 4:
a b c is in the set of natural numbers
Prove a + b + c = a + (b + c)
induction a with hd - starting a proof by induction
Now our first goal is:
0 + b + c = 0 + (b + c)
rw[zero_add] - proves that 0 + b = b
This is rewritten to:
b + c = 0 + (b + c)
rw[zero_add] - proves that 0 + (b + c) = b + c
This is rewritten to:
b + c = b + c
rfl - this proves that both sides that look the same are equal to each other
Now, we prove our second goal
n_ih: hd + b + c = hd + (b + c)
succ hd + b + c = succ hd + (b + c)
rw[succ_add] - proves that succ hd + b + c = succ (hd + b) + c
This is rewritten to:
succ (hd + b) + c = succ hd + (b + c)
rw[succ_add] - proves that succ (hd + b) + c = succ (hd + b + c)
This is rewritten to:
succ (hd + b + c) = succ hd + (b + c)
rw[n_ih] - replaces succ (hd + b + c) with succ (hd + (b + c))
This is rewritten to:
succ (hd + (b + c)) = succ hd + (b + c)

rw[succ_add] - proves succ hd + (b + c) = succ (hd + (b + c))
This is rewritten to:
succ (hd + (b + c)) = succ (hd + (b + c))
rfl - this proves that both sides that look the same are equal to each other

Problem 5:
a b c is in the set of natural numbers
Prove a + b + c = a + c + b
induction c with hd - starting a proof by induction
Now our first goal is:
a + b + 0 = a + 0 + b
rw[add_zero] - proves that b + 0 = b
This is rewritten to:
a + b = a + 0 + b
rw[add_zero] 0 proves that a + 0 = a
This is rewritten to:
a + b = a + b
rfl - this proves that both sides that look the same are equal to each other
Now, we prove our second goal
n_ih: a + b + hd = a + hd + b
a + b + succ hd = a + succ hd + b
rw[add_succ] - proves that a + b + succ hd = succ (a + b + hd)
This is rewritten to:
succ (a + b + hd) = a + succ hd + b
rw[add_succ] - proves that a + succ hd + b = succ (a + hd) + b
This is rewritten to:
succ (a + b + hd) = succ (a + hd) + b
rw[succ_add] - proves that succ (a + hd) + b = succ (a + hd + b)
This is rewritten to:
succ (a + b + hd) = succ (a + hd + b)
rw[n_ih] - replaces a + b + hd with a + hd + b
This is rewritten to
succ (a + hd + b) = succ (a + hd + b)
rfl - this proves that both sides that look the same are equal to each other

Problem 5 Proof in Mathematics:
a + b + c = a + (b + c)
0 + b + c = 0 + (b + c) - Basis
b + c = 0 + (b + c) - Addition Identity
b + c = b + c - Addition Identity
Inductive Step:
k + b + c = k + (b + c)
The goal is to prove that Sk + b + c = Sk + (b + c)
S(k + b + c) = Sk + (b + c) - Definition of Addition
S(k + b + c) = S(k + (b + c)) - Definition of Addition
S(k + (b + c)) = S(k + (b + c)) - Inductive Hypothesis
Therefore, by the Axiom of induction a + b + c = a + (b + c) for all a in the natural numbers set

Math to Lean
Basis: induction a with hd

Addition Identity: zero_add
Definition of Addition: succ_add
Inductive Hypothesis: n_ih

**Comments and Questions**

The Towers of Hanoi reminded me of solving certain problems by simply using recursion. I also remember applying this method to the Fibonacci sequence. This makes me wonder how it transfers to math. How does recursion appear in mathematics or, specifically, in Lean?

## 2.3 Week 3

**Homework**

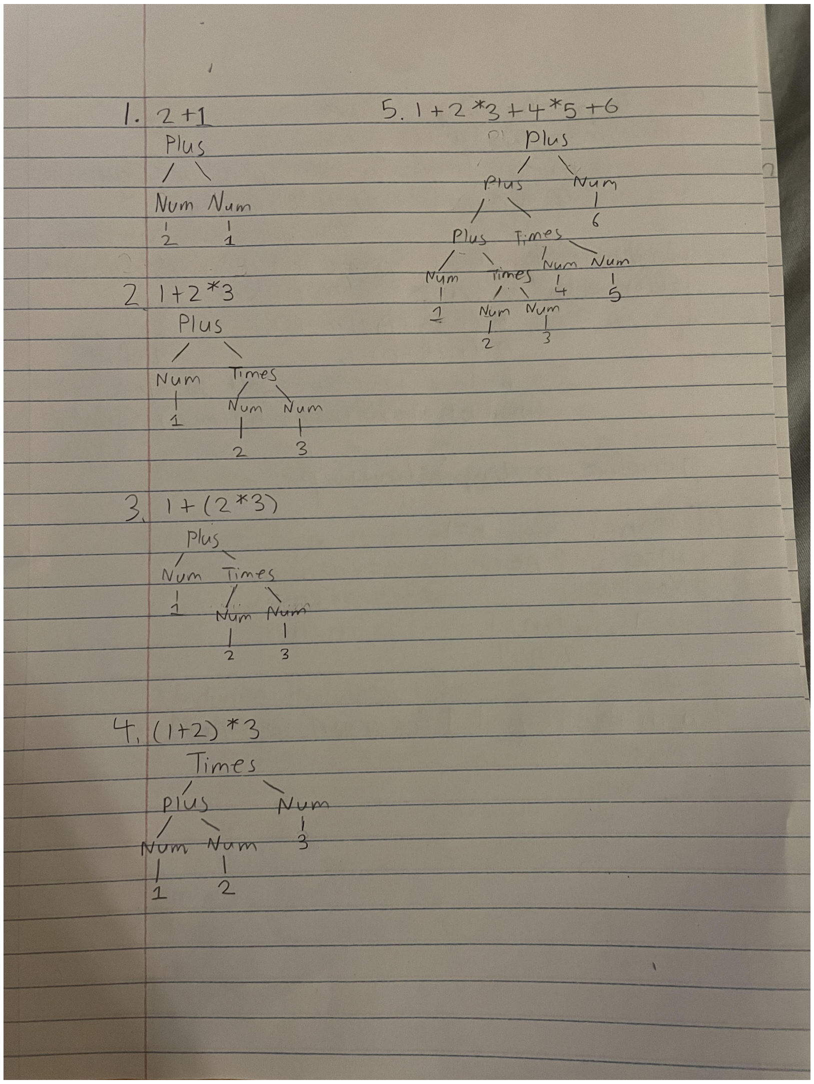
Week 3 Assignment

**Comments and Questions**

My literature review is about the cause for many different programming languages, the abstraction of them in the future, and what needs future ones would need to fulfill. I found that languages evolved depending on the different needs and users over time. For instance, Domain-specific languages were created to meet specific needs. SQL, being one of them, was created to interact with databases. Within the topic of abstraction, PLs are bound to become higher level to not worry about lower-level implementation. Examples of the current progress towards abstraction would be AI-assisted programming and declarative programming. However, abstraction will not change the usage of current languages like Java, Python, and C++ in the foreseeable future. These languages have made a huge impact and it is shown through their extensive ecosystems and sheer amount of existing codebases. On the other hand, some specific areas use newer languages like Rust and Kotlin. They fulfill more modern needs like memory safety and developer productivity. Future programming languages would need to deal with challenges like security, AI, and concurrency.

## 2.4 Week 4

### Homework




### Comments and Questions

Parsing seemed to work really great with expressions. In general, is parsing a good strategy when it comes to breaking things down?

## 2.5 Week 5

### Notes

exact -conclusion of a proof
Operator: $\wedge$ -Logical And
Example: $A \wedge B$ -A and B
and_into -takes two pieces of evidence and combines it into one
have -adds new assumptions to the proof

**Homework**

Problem 1:
exact todo_list

Problem 2:
exact and_intro p s

Problem 3:
exact ⟨⟨a,i⟩,⟨o,u⟩⟩

Problem 4:
have p := and_left vmS
exact p

Problem 5:
have q := and_right h
exact q

Problem 6:
have a := and_left h1
have u := and_right h2
exact ⟨a, u⟩

Problem 7:
have h1 := h.left
have h2 := h1.right
have h3 := h2.left
have h4 := h3.left
have h5 := h4.right
exact h5

Problem 8:
have h1 := h.left
have a := h1.right
have h2 := h1.left
have p := h2.left
have s := h2.right
have h3 := h.right
have h4 := h3.right
have h5 := h4.left
have c := h5.left

In mathmatical proof:
If $((P \wedge S) \wedge A) \wedge I \wedge (C \wedge O) \wedge U$ then $A \wedge C \wedge P \wedge S$.
Proof:
(1)
$((P \wedge S) \wedge A) \wedge I \wedge (C \wedge O) \wedge U$

assumption

(2)
$(P \wedge S) \wedge A$
and_left (1)

(3)
A
and_right (2)

(4)
$(P \wedge S)$
and_left (2)

(5)
P
and_left (4)

(6)
S
and_right (4)

(7)
$I \wedge (C \wedge O) \wedge U$
and_right (1)

(8)
$(C \wedge O) \wedge U$
and_right (7)

(9)
$C \wedge O$
and_left (8)

(10)
C
and_left (9)

(11)
$A \wedge C \wedge P \wedge S$
and_intro (3)(10)(5)(6)

**Comments and Questions**

How does lean help with problem solving when it comes to programming?

## 2.6 Week 6

### Homework

Problem 1:
have b := bakery_service p
exact b

Problem 2:
exact fun h : $C$ => h

Problem 3:
exact fun $h$ => and_intro h.right h.left

Problem 4:
exact fun h : $C$ => have a : A := h1 h; h2 a

Problem 5:
have q : Q := h1 p
have t : T := h3 q
exact h5 t

Problem 6:
exact fun c : $C$ =>
fun d : $D$ =>
have cd : $C \wedge D$ := ⟨c, d⟩;
h cd

Couldn't finish...

### Comments and Questions

Solving lean logic game is a simple view into proving theorems and the foundations of programming languages. When it comes to the lambda problems specifically, how do real-world challenges complicate the applications of lambda?

## 2.7 Week 7

### Homework

1.

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

$$((\lambda m.\lambda n.m\,n)\,(\lambda f.\lambda x.f\,(f\,x))) \to \lambda n.(\lambda f.\lambda x.f\,(f\,x))\,n$$

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

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

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

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

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

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

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

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

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

2.
The lambda term implements addition on natural numbers. It is done by combing two Church numerals. It carries out addition by counting the the total number of times the function is being applied.

#### Comments and Questions

Can church numerals be used to represent recursive functions/processes?

### 2.8 Week 8

#### Homework

Answers on week 9

#### Comments and Questions

When it comes to evaluation strategies in lambda calculus, what are the trade-offs? How do these strategies affect performance and accuracy in a practical implementation?

### 2.9 Week 9

#### Homework

2. a is applied to b, which leads to (ab). Then, (ab) is applied to c, resulting in ((ab)c). Finally, ((ab)c) is applied to d, which leads to (((ab)c)d). (a) reduces to a it's already in its simplest form.

3.When substituting a variable, wall instances of that variable are replaced. However, if the variable being substituted for is also bound, the meaning of the expression is changed. This is implemented by traversing the AST of the expression, checking for bound variables, and renaming them.

4. No, not everything is the expected result.

5.
$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

**Comments and Questions**

How could different evaluation strategies impact the trace output produced by an interpreter?

## 2.10 Week 10

**Homework**

1. The challenge of working through Homework 8/9 and Assignment 3 was figuring out why the interpreter wouldn't completely evaluate everything.

2. The key insight for Assignment 3 was just looking through the evaluation function and looking at where ecxactly it stopped evaluating the function and just returned whatever it found.

3. The most interesting take away was from the homework and assignment was using the debugger to see how far the program would evaluate a lambda function. I never really use the debugger, so it was interesting to break things apart.

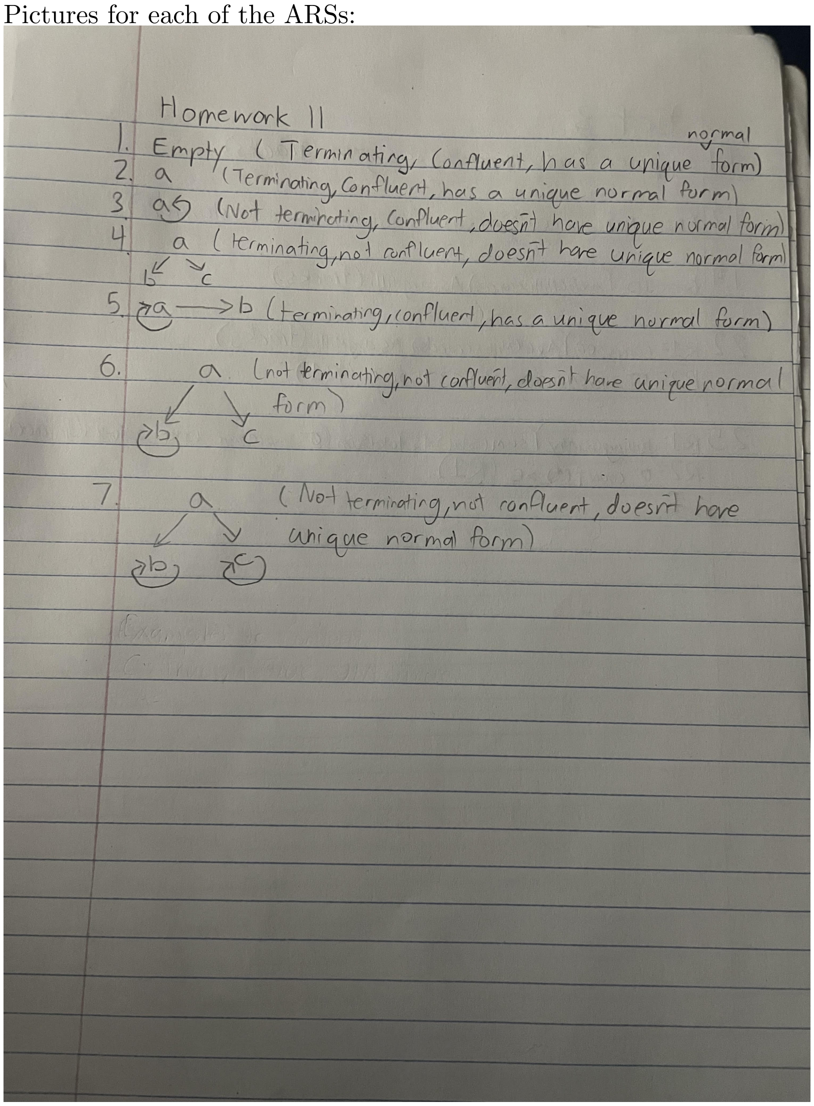
**Comments and Questions**

How can the interpreter be expanded upon to handle more complex structures?

## 2.11 Week 11

### Homework

Pictures for each of the ARSs:




Example of ARS for each of the possible 8 combinations

Confluent = True, Terminating: True, Has A Unique Normal Form: True

$$A = a, b, R = (a, b)$$

Confluent = True, Terminating: True, Has A Unique Normal Form: False

$$A = a, b, c, R = (a, b), (a, c)$$

Confluent = True, Terminating: False, Has A Unique Normal Form: True

$$A = a, R = (a, a)$$

Confluent = True, Terminating: False, Has A Unique Normal Form: False

$$A = a, b, R = (a, a), (a, b)$$

Confluent = False, Terminating: True, Has A Unique Normal Form: True

$$A = a, b, c, R = (a, b), (b, b)$$

Confluent = False, Terminating: True, Has A Unique Normal Form: True

$$A = a, b, c, R = (a, b), (a, c)$$

Confluent = False, Terminating: False, Has A Unique Normal Form: True

$$A = a, b, R = (a, b), (b, a)$$

Confluent = False, Terminating: False, Has A Unique Normal Form: False

$$A = a, b, c, R = (a, b), (b, b), (a, c), (c, c)$$

**Comments and Questions**

What characteristics of an ARS (confluency, termination, etc) is important when it comes to programming languages and why?

### 2.12 Week 12

**Homework**

1. $ba- > ab$
The number of substrings is finite. Therefore, the ARS must terminate.
The normal form is where b characters appear before a characters.
It is confluent because it only shifts the positions of the characters. It does not make a new substring.
The ARS sorts the string so that the b characters appear before the a characters.
2.
$aa- > a$
$bb- > a$
$ab- > b$
$ba- > b$
The ARS terminates because the rules reduces the length of the string.
The normal forms are either a or b.
There is not string that reduces to both a and b.
It's confluent because each rewrite reduces the length of the string, and the choice of application does not affect the final result.
All the strings become become equal to their normal forms by switching from $- >$ to $=$.
$=$ is determined by whether the string has an even or odd number of a's and b's.
If the mod is 2, then the answer is a. if the mod is 1, then the answer is b.
The ARS computes the parity of the number of a's and b's.
3.
$aa- > a$
$bb- > b$
$ba- > ab$
$ab- > ba$

The ARS does not terminate because the rewrites alternate and end up looping.
There are no normal forms.
New Rules: $ab->b$ AND $ba->a$
The ARS computes a simplified form.
4.
$ab->ba$
$ba->ab$
The ARS does not terminate because it constantly switches between ab and ba.
There are no normal forms.
New Rules: $ab->a$ AND $ba->b$
The ARS sorts the characters deterministically.
5.
$ab->ba$
$ba->ab$
$aa->$
$b->$
Reduce Examples:
$abba->aabb->ab->ba$ (loops)
$bababa->$ infinite alternations
The ARS is not terminating because the $ab->ba$ causes infinite alternations
There are two equivalence classes:
- string with a equal number of a's and b's are equal.
- strings with an unequal number of a's and b's can be reduced.
New Rules: $ab->a$ AND $ba->b$
Question: Does a string reduce to an empty one?
Answer: The string will be empty depending on the number of a's and b's.
5b.
Rewrite: $aa->a$ AND $b->$
The equivalence class is not changed.

#### Comments and Questions

How does modularity infuence the properties of an ARS?

## 2.13 Week 13

#### Homework

Compute fact 3:

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

fact = fix $F$ in fact 3 (def of let rec)

$(fix\ F)$ 3 (def of let)

$(\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x)))F$ 3 (def of fix)

$(\lambda x.F(x\ x))(\lambda x.F(x\ x))$ 3 (beta rule)

$F((\lambda x.F(x\ x))(\lambda x.F(x\ x)))$ 3 (beta rule)

$(\lambda fact.\lambda n.$ if $n = 0$ then 1 else $n * fact(n-1))((\lambda x.F(x\ x))(\lambda x.F(x\ x)))$ 3 (beta rule)

$(\lambda n.$ if $n = 0$ then 1 else $n * ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))(n-1))$ 3 (beta rule)

if $3 = 0$ then 1 else $3 * ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))(3-1)$ (beta rule)

$3 * ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))(3-1)$ (def of if)

$3 * ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ 2$ (arithmetic)

$3 * (2 * ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))(2-1))$ (beta rule and arithmetic)

$3 * (2 * ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ 1)$ (arithmetic)

$3 * (2 * (1 * ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))(1-1)))$ (beta rule and arithmetic)

$3 * (2 * (1 * ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ 0))$ (arithmetic)

$3 * (2 * (1 * 1))$ (def of if)

$3 * (2 * 1)$

$3 * 2$

6 (arithmetic)

$fact\ 3 = 6$

**Comments and Questions**

I noticed while computing fact 3, there was a repetition of substitution and expansion of the fixed point combinator. Is there a way to make this process more efficient?

# 3 Lessons from the Assignments

# 4 Conclusion

# References

[BLA] Author, Title, Publisher, Year.