

# CPSC-354 Report

Mitchell Toney  
Chapman University

October 5, 2025

## Abstract

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Week by Week</b>	<b>1</b>
2.1	Week 1: HW1 . . . . .	1
2.2	Week 2: HW2 . . . . .	2
2.3	Week 3: HW3 . . . . .	3
2.3.1	Exercise 5 . . . . .	3
2.3.2	Semantic question . . . . .	4
2.4	Week 4: HW4 . . . . .	5
2.4.1	HW4.1: Termination Proof for GCD . . . . .	5
2.4.2	HW4.2: Termination Proof for Merge Sort . . . . .	6
2.5	Week 5: HW5 . . . . .	6
2.5.1	Lambda Calculus Workout Evaluation (Corrected) . . . . .	6
2.6	Week 6: HW6 . . . . .	7
2.6.1	Fixed Point Combinator Exercise . . . . .	7
<b>3</b>	<b>Essay</b>	<b>8</b>
<b>4</b>	<b>Evidence of Participation</b>	<b>8</b>
<b>5</b>	<b>Conclusion</b>	<b>8</b>

## 1 Introduction

## 2 Week by Week

### 2.1 Week 1: HW1

The *MU* puzzle is a puzzle created by Douglas Hofstadter. It consists of four rules that can be applied to a string *MI*.

1.  $xI \rightarrow xIU$
2.  $Mx \rightarrow Mxx$
3.  $xIIIy \rightarrow xUy$
4.  $xUUy \rightarrow xy$

When first approaching this puzzle, the first strategy that came to mind was to take advantage of rule number 2 to keep duplicating the I's until there is a multiple of three, then using rules 3 and 4 to get rid of the I's and leave a remaining U.

The issue with this is that  $2^n \bmod 3$  will never equal 0, it infinitely cycles between equaling 1 and 2, and without being able to get rid of all the I's, which would require them being a multiple of 3, you will never be able to get MU.

Thus, the puzzle is not solvable.

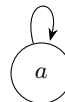
## 2.2 Week 2: HW2



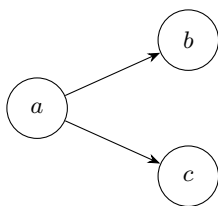
1.  $A = \emptyset, R = \emptyset$



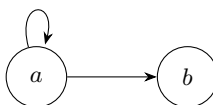
2.  $A = \{a\}, R = \emptyset$



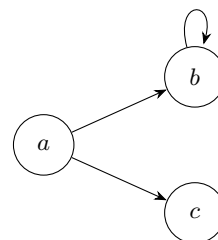
3.  $A = \{a\}, R = \{(a, a)\}$



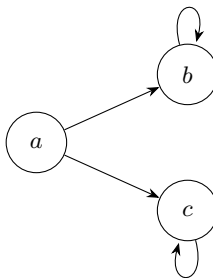
4.  $A = \{a, b, c\}, R = \{(a, b), (a, c)\}$



5.  $A = \{a, b\}, R = \{(a, a), (a, b)\}$



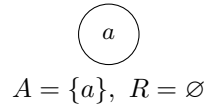
6.  $A = \{a, b, c\}, R = \{(a, b), (b, b), (a, c)\}$



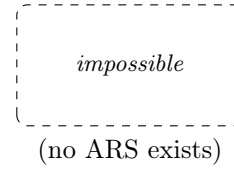
7.  $A = \{a, b, c\}, R = \{(a, b), (b, b), (a, c), (c, c)\}$

#	Terminating	Confluent	Unique NFs
1	Yes	Yes	Yes
2	Yes	Yes	Yes
3	No	Yes	No
4	Yes	No	No
5	No	Yes	Yes
6	No	No	No
7	No	No	No

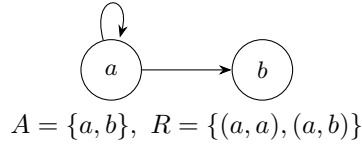
**Confluent True, Terminating True,  
Unique NFs True**



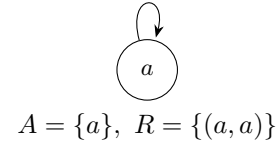
**Confluent True, Terminating True,  
Unique NFs False**



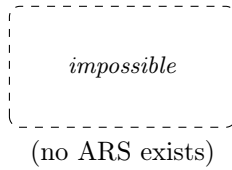
**Confluent True, Terminating False,  
Unique NFs True**



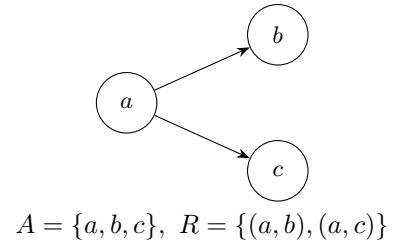
**Confluent True, Terminating False,  
Unique NFs False**



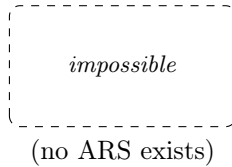
**Confluent False, Terminating True,  
Unique NFs True**



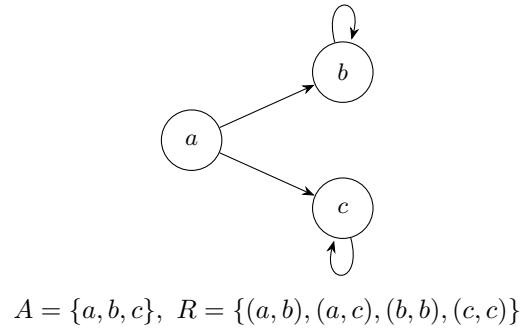
**Confluent False, Terminating True,  
Unique NFs False**



**Confluent False, Terminating False,  
Unique NFs True**



**Confluent False, Terminating False,  
Unique NFs False**



## 2.3 Week 3: HW3

### 2.3.1 Exercise 5

Consider rewrite rules:

$ab \rightarrow ba$   
 $ba \rightarrow ab$   
 $aa \rightarrow$   
 $b \rightarrow$

## Example Reductions

Reducing **abba**:

$$\begin{aligned} abba &\rightarrow baba && \text{(using } ab \rightarrow ba) \\ baba &\rightarrow bbaa && \text{(using } ba \rightarrow ab) \\ bbaa &\rightarrow baa && \text{(using } b \rightarrow \varepsilon) \\ baa &\rightarrow aba && \text{(using } ba \rightarrow ab) \\ aba &\rightarrow baa && \text{(using } ab \rightarrow ba) \end{aligned}$$

There is an infinite loop between **aba** and **baa**.

Reducing **bababa**:

$$\begin{aligned} bababa &\rightarrow ababab && \text{(using } ba \rightarrow ab) \\ ababab &\rightarrow baabab && \text{(using } ab \rightarrow ba) \\ baabab &\rightarrow ababab && \text{(using } ba \rightarrow ab) \end{aligned}$$

This is an infinite loop between **ababab** and **baabab**.

**Why the ARS is not terminating** The ARS is not terminating because the rules  $ab \rightarrow ba$  and  $ba \rightarrow ab$  create infinite cycles. These rules allow us to swap adjacent  $a$  and  $b$  characters indefinitely, leading to non-terminating reduction sequences.

**Non-equivalent strings** Two strings that are not equivalent: **a** and **aa**.

The string **a** cannot be reduced further, while **aa** reduces to nothing using the rule  $aa \rightarrow \varepsilon$ . Since *nothing*  $\neq a$ , these strings are in different equivalence classes.

**Equivalence classes** The equivalence relation  $\leftrightarrow^*$  has infinitely many equivalence classes. Each equivalence class can be characterized by the number of  $a$ 's modulo 2 and the number of  $b$ 's modulo 1.

The equivalence classes are:

- $[\varepsilon]$ : strings with even number of  $a$ 's and no  $b$ 's
- $[a]$ : strings with odd number of  $a$ 's and no  $b$ 's
- $[b]$ : strings with any number of  $a$ 's and at least one  $b$

The normal forms are:  $\varepsilon$ ,  $a$ , and  $b$ .

**Modifying the ARS to be terminating** To make the ARS terminating without changing equivalence classes, we can remove the symmetric rules and keep only one direction:

$$\begin{aligned} ba &\rightarrow ab \\ aa &\rightarrow \varepsilon \\ b &\rightarrow \varepsilon \end{aligned}$$

This eliminates the infinite cycles while preserving the same equivalence relation.

### 2.3.2 Semantic question

**Parity of  $a$ 's:** "Does this string contain an odd number of  $a$ 's?"

Answer: Yes if the normal form is  $a$ , No if the normal form is  $\varepsilon$ .

### Exercise 5b

Consider rewrite rules:

$$\begin{aligned}
ab &\rightarrow ba \\
ba &\rightarrow ab \\
aa &\rightarrow a \\
b &\rightarrow
\end{aligned}$$

**Reducing abba:**

$$\begin{aligned}
abba &\rightarrow baba \\
baba &\rightarrow abba
\end{aligned}$$

**Reducing bababa:**

$$\begin{aligned}
bababa &\rightarrow ababab \\
ababab &\rightarrow bababa
\end{aligned}$$

**Why not terminating.** The symmetric swaps  $ab \leftrightarrow ba$  allow infinite rewriting.

**Non-equivalent strings.**  $a$  and  $\varepsilon$  are not equivalent.

**Equivalence classes.** Exactly two:  $[\varepsilon]$  (no  $a$ 's; all  $b$ 's delete) and  $[a]$  (at least one  $a$ ; since  $aa \sim a$ ). Normal forms (under a terminating orientation):  $\varepsilon$  and  $a$ .

**Terminating variant.**

$$\begin{aligned}
ba &\rightarrow ab \\
aa &\rightarrow a \\
b &\rightarrow
\end{aligned}$$

This gives a complete semantics to the ARS: it computes the invariant for any input string.

## 2.4 Week 4: HW4

### 2.4.1 HW4.1: Termination Proof for GCD

Consider the following algorithm:

```

while b != 0:
    temp = b
    b = a mod b
    a = temp
return a

```

**Assume:** Work over integers with Euclidean division: inputs  $a \in \mathbb{Z}$ ,  $b \in \mathbb{N}$ ; if  $b \neq 0$  then  $0 \leq a \bmod b < b$ .

**Model:** States  $A = \mathbb{Z} \times \mathbb{N}$ . One step  $(a, b) \rightarrow (a', b')$  is one loop iteration.

**Measure:**  $\phi : A \rightarrow \mathbb{N}$ ,  $\phi(a, b) = b$ .

**Show:**  $(a, b) \rightarrow (a', b') \Rightarrow \phi(a', b') < \phi(a, b)$ .

If  $b \neq 0$ , the update sets  $b' = a \bmod b$  with  $0 \leq b' < b$ ; hence  $\phi(a', b') = b' < b = \phi(a, b)$ .

$\phi$  strictly decreases in  $\mathbb{N}$ , so there is no infinite  $\rightarrow$ -chain; eventually  $b = 0$  and the loop stops. Thus the algorithm terminates under the stated conditions.

### 2.4.2 HW4.2: Termination Proof for Merge Sort

Consider the following fragment of an implementation of merge sort:

```
function merge_sort(arr, left, right):
    if left >= right:
        return
    mid = (left + right) / 2
    merge_sort(arr, left, mid)
    merge_sort(arr, mid+1, right)
    merge(arr, left, mid, right)
```

Prove that

$$\phi(\text{left}, \text{right}) = \text{right} - \text{left} + 1$$

is a measure function for `merge_sort`.

**Show:** For

```
merge_sort(arr, left, right):
    if left >= right: return
    mid = floor((left + right)/2)
    merge_sort(arr, left, mid)
    merge_sort(arr, mid+1, right)
    merge(arr, left, mid, right)
```

the function  $\phi(\text{left}, \text{right}) = \text{right} - \text{left} + 1$  is a strictly decreasing measure, hence `merge_sort` terminates.

**Assume:**  $\text{left}, \text{right} \in \mathbb{Z}$  with  $0 \leq \text{left} \leq \text{right} < |\text{arr}|$ . Division for `mid` is integer. If  $\text{left} < \text{right}$ , then  $\text{left} \leq \text{mid} < \text{right}$ . `merge` makes no recursive calls.

**Model:** States are intervals  $A = \{(l, r) \in \mathbb{Z}^2 \mid l \leq r\}$ . A "step" is a recursive edge from  $(l, r)$  (with  $l < r$ ) to each child  $(l, \text{mid})$  and  $(\text{mid} + 1, r)$ .

**Measure:**  $\phi : A \rightarrow \mathbb{N}$ ,  $\phi(l, r) = r - l + 1$ .

Let  $l < r$  and  $\text{mid} = \lfloor (l + r)/2 \rfloor$  so  $l \leq \text{mid} < r$ .

First child:  $\phi(l, \text{mid}) = \text{mid} - l + 1 \leq (r - 1) - l + 1 = r - l = \phi(l, r) - 1 < \phi(l, r)$ .

Second child:  $\phi(\text{mid} + 1, r) = r - \text{mid} \leq r - l = \phi(l, r) - 1 < \phi(l, r)$ .

Every recursive edge strictly decreases  $\phi$  in  $\mathbb{N}$ , which is well-founded. Thus no infinite recursion is possible; the calls bottom out at states with  $\phi \in \{0, 1\}$  (i.e.,  $\text{left} \geq \text{right}$ ), where the function returns. Therefore  $\phi(\text{left}, \text{right}) = \text{right} - \text{left} + 1$  is a valid measure and `merge_sort` terminates under the stated conditions.

## 2.5 Week 5: HW5

### 2.5.1 Lambda Calculus Workout Evaluation (Corrected)

Evaluate:  $(\lambda f. \lambda x. f(f x))(\lambda g. \lambda y. g(g(y)))$

Use  $\alpha$ -renaming to avoid capture.

$$\begin{aligned} (\lambda f. \lambda x. f(f x))(\lambda g. \lambda y. g(g(y))) &\rightarrow \lambda x. (\lambda g. \lambda y. g(g(y)))((\lambda g. \lambda y. g(g(y))) x) \\ &\quad (\lambda g. \lambda y. g(g(y))) x \rightarrow \lambda y. x(x y) \quad (\alpha\text{-rename inner } y) \\ &\Rightarrow \lambda x. (\lambda g. \lambda y. g(g(y))) (\lambda y. x(x y)) \rightarrow \lambda x. \lambda y. x^9 y. \end{aligned}$$

Hence the normal form is  $\lambda f. \lambda x. f^9 x$  (Church numeral 9).

## 2.6 Week 6: HW6

### 2.6.1 Fixed Point Combinator Exercise

Compute `fact 3` following the computation rules for `fix`, `let`, and `let rec`:

Given:

$$E_0 = \text{let rec fact} = \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact } (n - 1) \text{ in fact } 3$$

Abbreviations:

$$F \stackrel{\text{def}}{=} \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n - 1) \quad \text{FACT} \stackrel{\text{def}}{=} \text{fix } F$$

Computation:

$$\begin{aligned}
 E_0 &\xrightarrow{\text{def of let rec}} \text{let fact} = \text{fix}(\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n - 1)) \text{ in fact } 3 \\
 &\xrightarrow{\text{def of let}} (\lambda \text{fact}. \text{fact } 3) \text{ fix}(\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n - 1)) \\
 &\xrightarrow{\beta} (\text{fix}(\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n - 1))) 3 \\
 &\equiv \text{FACT } 3 \\
 &\xrightarrow{\text{def of fix}} (F \text{ FACT}) 3 \\
 &\xrightarrow{\beta} (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{FACT } (n - 1)) 3 \\
 &\xrightarrow{\beta} \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 * \text{FACT } (3 - 1) \\
 &\xrightarrow{\text{def of if}} 3 * \text{FACT } (3 - 1) \\
 &\xrightarrow{\text{arith}} 3 * \text{FACT } 2 \\
 &\xrightarrow{\text{def of fix}} 3 * (F \text{ FACT}) 2 \\
 &\xrightarrow{\beta} 3 * (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{FACT } (n - 1)) 2 \\
 &\xrightarrow{\beta} 3 * \text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * \text{FACT } (2 - 1) \\
 &\xrightarrow{\text{def of if}} 3 * (2 * \text{FACT } (2 - 1)) \\
 &\xrightarrow{\text{arith}} 3 * (2 * \text{FACT } 1) \\
 &\xrightarrow{\text{def of fix}} 3 * (2 * (F \text{ FACT}) 1) \\
 &\xrightarrow{\beta} 3 * (2 * (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{FACT } (n - 1)) 1) \\
 &\xrightarrow{\beta} 3 * (2 * \text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * \text{FACT } (1 - 1)) \\
 &\xrightarrow{\text{def of if}} 3 * (2 * (1 * \text{FACT } (1 - 1))) \\
 &\xrightarrow{\text{arith}} 3 * (2 * (1 * \text{FACT } 0)) \\
 &\xrightarrow{\text{def of fix}} 3 * (2 * (1 * (F \text{ FACT}) 0)) \\
 &\xrightarrow{\beta} 3 * (2 * (1 * (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{FACT } (n - 1)) 0)) \\
 &\xrightarrow{\beta} 3 * (2 * (1 * \text{if } 0 = 0 \text{ then } 1 \text{ else } 0 * \text{FACT } (0 - 1))) \\
 &\xrightarrow{\text{def of if}} 3 * (2 * (1 * 1)) \\
 &\xrightarrow{\text{arith}} 3 * (2 * 1) \xrightarrow{\text{arith}} 3 * 2 \xrightarrow{\text{arith}} 6.
 \end{aligned}$$

**Result:** fact  $3 = 6$

### **3 Essay**

### **4 Evidence of Participation**

### **5 Conclusion**

### **References**

[BLA] Author, [Title](#), Publisher, Year.