

CPSC-354 Report

Jeffrey Bok
Chapman University

October 21, 2025

Abstract

Contents

1	Introduction	2
2	Week by Week	2
2.1	Week 1	2
2.1.1	Homework	2
2.1.2	Exploration	2
2.1.3	Questions	3
2.2	Week 2	3
2.2.1	Homework	3
2.2.2	Exploration	6
2.2.3	Questions	7
2.3	Week 3	7
2.3.1	Homework	7
2.3.2	Sample Reductions	7
2.3.3	Analysis	7
2.4	Exercise 5b	8
2.4.1	Modified Rewrite Rules	8
2.4.2	Sample Reductions	8
2.4.3	Analysis	9
2.4.4	Exploration	9
2.4.5	Questions	9
2.5	Week 4	10
2.5.1	Homework	10
2.5.2	Exploration	10
2.5.3	Questions	10
2.6	Week 5	10
2.6.1	Homework	10
2.6.2	Exploration	10
2.6.3	Questions	10
2.7	Week 6	10
2.7.1	Homework	10
2.7.2	Exploration	10
2.7.3	Questions	10
2.8	Week 7	10

2.8.1	Homework	10
2.8.2	Exploration	10
2.8.3	Questions	10
2.9	Week 8	10
2.9.1	Homework	10
2.9.2	Natural Language Proof	11
2.9.3	Exploration	12
2.9.4	Questions	12
2.10	Week 9	13
2.10.1	Homework	13
2.10.2	Exploration	13
2.10.3	Questions	13
2.11	Week 10	13
2.11.1	Homework	13
2.11.2	Exploration	13
2.11.3	Questions	13
2.12	Week 11	13
2.12.1	Homework	13
2.12.2	Exploration	13
2.12.3	Questions	13
3	Synthesis	13
4	Evidence of Participation	13
5	Conclusion	13

1 Introduction

2 Week by Week

2.1 Week 1

2.1.1 Homework

What is the MU Puzzle and how do you "solve" it?:

The MU puzzle is a logic puzzle created by Douglas Hofstadter in his 1979 book "Gödel, Escher, Bach: An Eternal Golden Braid." It's designed to illustrate concepts about formal systems, computability, and the limits of rule-based reasoning. The rules are below:

Rule I: If a string ends in I, you can add U to the end ($xI \rightarrow xIU$)

Rule II: If you have Mx, you can make Mxx (double everything after M)

Rule III: If you find III anywhere in your string, you can replace it with U ($xIIIy \rightarrow xUy$)

Rule IV: If you find UU anywhere in your string, you can remove it ($xUUY \rightarrow xy$)

To "solve" the puzzle, you try to apply a combination of rules step by step, creating new strings. Eventually, you'll find that MU can never be reached because the rules never allow you to remove the odd number of I's needed to get zero.

2.1.2 Exploration

Hofstadter used this puzzle to demonstrate how formal systems can have inherent limitations - some statements that seem like they should be provable within a system are actually unprovable. This connects to

Gödel's incompleteness theorems and fundamental questions about the nature of mathematical truth and computation.

Programming languages are formal systems, just like the MU puzzle. They have:

- Syntax rules (what constitutes valid code)
- Transformation rules (how expressions evaluate)
- Semantic constraints (what programs can actually compute)

The MU puzzle demonstrates that even simple rule sets can have hidden limitations - similarly, programming languages have inherent computational boundaries.

2.1.3 Questions

1. The impossibility of reaching "MU" from "MI" is provable, yet someone working within the system might not realize this. How does this relate to the halting problem and undecidable questions in programming?

2.2 Week 2

2.2.1 Homework

Consider the following list of ARSs:

1. $A = \{\}$.
2. $A = \{a\}$ and $R = \{\}$.
3. $A = \{a\}$ and $R = \{(a, a)\}$.
4. $A = \{a, b, c\}$ and $R = \{(a, b), (a, c)\}$.
5. $A = \{a, b\}$ and $R = \{(a, a), (a, b)\}$.
6. $A = \{a, b, c\}$ and $R = \{(a, b), (b, b), (a, c)\}$.
7. $A = \{a, b, c\}$ and $R = \{(a, b), (b, b), (a, c), (c, c)\}$.

Draw a picture for each of the ARSs above. Are the ARSs terminating? Are they confluent? Do they have unique normal forms?

Try to find an example of an ARS for each of the possible 8 combinations. Draw pictures of these examples.

ARS 1: $A = \{\}$

Empty graph (no nodes, no edges)

Terminating: YES **Confluent:** YES **Unique Normal Forms:** YES

ARS 2: $A = \{a\}, R = \{\}$



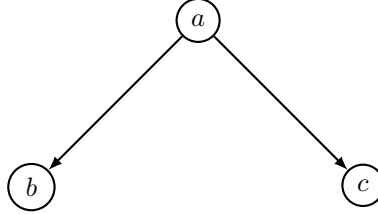
Terminating: YES **Confluent:** YES **Unique Normal Forms:** YES

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



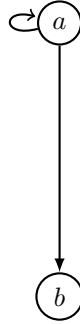
Terminating: NO **Confluent:** YES **Unique Normal Forms:** NO

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



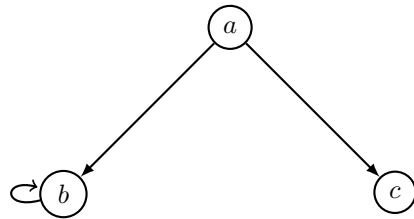
Terminating: YES **Confluent:** NO **Unique Normal Forms:** NO

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



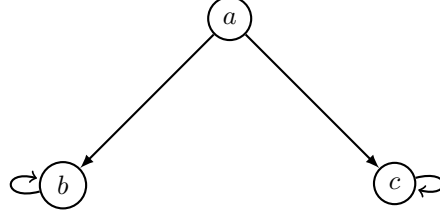
Terminating: NO **Confluent:** NO **Unique Normal Forms:** NO

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



Terminating: NO **Confluent:** NO **Unique Normal Forms:** NO

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



Terminating: NO **Confluent:** NO **Unique Normal Forms:** NO

8 Combinations Table

Confluent	Terminating	Unique NF	Example
True	True	True	$A = \{a\}, R = \{\}$
True	True	False	$A = \{\}, R = \{\}$
True	False	True	$A = \{a, b\}, R = \{(a, b), (b, b)\}$
True	False	False	$A = \{a\}, R = \{(a, a)\}$
False	True	True	$A = \{a, b, c, d\}, R = \{(a, b), (a, c), (c, d)\}$
False	True	False	$A = \{a, b, c\}, R = \{(a, b), (a, c)\}$
False	False	True	$A = \{a, b, c\}, R = \{(a, b), (b, a), (a, c), (c, a)\}$
False	False	False	$A = \{a, b, c\}, R = \{(a, b), (b, b), (a, c)\}$

Examples for 8 Combinations

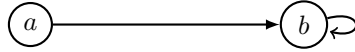
Example 1: Confluent=T, Terminating=T, Unique Normal Forms=T $A = \{a\}, R = \{\}$



Example 2: Confluent=T, Terminating=T, Unique Normal Forms=F $A = \{\}, R = \{\}$

Empty graph

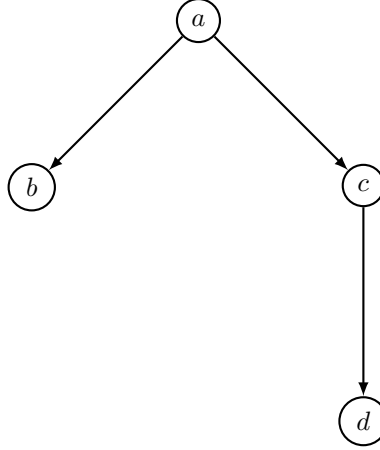
Example 3: Confluent=T, Terminating=F, Unique Normal Forms=T $A = \{a, b\}, R = \{(a, b), (b, b)\}$



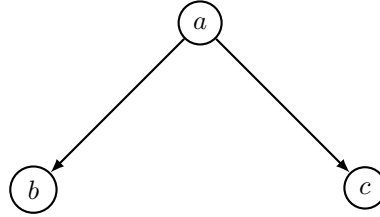
Example 4: Confluent=T, Terminating=F, Unique Normal Forms=F $A = \{a\}, R = \{(a, a)\}$



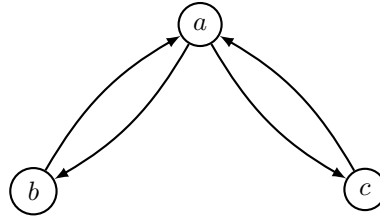
Example 5: Confluent=F, Terminating=T, Unique Normal Forms=T $A = \{a, b, c, d\}, R = \{(a, b), (a, c), (c, d)\}$



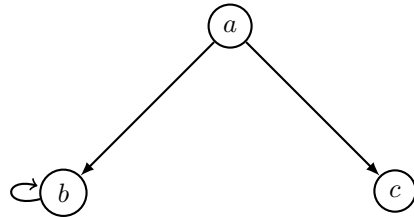
Example 6: Confluent=F, Terminating=T, Unique Normal Forms=F $A = \{a, b, c\}$, $R = \{(a, b), (a, c)\}$



Example 7: Confluent=F, Terminating=F, Unique Normal Forms=T $A = \{a, b, c\}$, $R = \{(a, b), (b, a), (a, c), (c, a)\}$



Example 8: Confluent=F, Terminating=F, Unique Normal Forms=F $A = \{a, b, c\}$, $R = \{(a, b), (b, b), (a, c)\}$



2.2.2 Exploration

Abstract Reduction Systems provide a mathematical foundation for understanding computation and rewriting. The properties of termination, confluence, and unique normal forms are fundamental to understanding how programming languages behave:

- **Termination** ensures that computations eventually halt
- **Confluence** guarantees that the order of operations doesn't affect the final result

- **Unique Normal Forms** means every expression has a single, well-defined simplified form

These concepts directly apply to programming language design, where we want predictable evaluation strategies and guaranteed termination for certain classes of programs.

2.2.3 Questions

1. How do the termination properties of ARSs relate to the halting problem in computation?
2. Why might a programming language designer prefer confluent systems over non-confluent ones?

2.3 Week 3

2.3.1 Homework

Consider the rewrite rules:

- $ab \rightarrow ba$
- $ba \rightarrow ab$
- $aa \rightarrow (\text{empty string})$
- $b \rightarrow (\text{empty string})$

2.3.2 Sample Reductions

Reducing abba:

$abba \rightarrow baba$ (using $ab \rightarrow ba$)
 $baba \rightarrow bbba$ (using $ab \rightarrow ba$)
 $bbba \rightarrow bba$ (using $b \rightarrow \text{empty}$)
 $bba \rightarrow aa$ (using $b \rightarrow \text{empty}$)
 $aa \rightarrow \text{empty}$ (using $aa \rightarrow \text{empty}$)

Reducing bababa:

$bababa \rightarrow bbaaaba$ (using $ab \rightarrow ba$ twice)
 $bbaaaba \rightarrow baaaba$ (using $b \rightarrow \text{empty}$)
 $baaaba \rightarrow aaaba$ (using $b \rightarrow \text{empty}$)
 $aaaba \rightarrow aba$ (using $aa \rightarrow \text{empty}$)
 $aba \rightarrow baa$ (using $ab \rightarrow ba$)
 $baa \rightarrow aa$ (using $b \rightarrow \text{empty}$)
 $aa \rightarrow \text{empty}$ (using $aa \rightarrow \text{empty}$)

2.3.3 Analysis

Why is the ARS not terminating?

The first two rules $ab \rightarrow ba$ and $ba \rightarrow ab$ create cycles. You can apply these rules forever, going back and forth between ab and ba .

Find two strings that are not equivalent. How many non-equivalent strings can you find?

Two strings that are not equivalent: "a" and "empty string". The string "a" cannot be reduced further, while other strings can reduce to empty.

Equivalence classes and normal forms: There are exactly 2 equivalence classes:

1. Strings that reduce to empty string
2. Strings that reduce to "a"

The normal forms are: empty string and "a"

Modified terminating ARS: To make it terminating, always eliminate b's first, then eliminate aa's, then do swapping only if needed.

Questions about strings that can be answered using the ARS:

1. "Given a string, does it contain an even number of a's?"
2. "Given a string, does it contain an odd number of a's?"
3. "Are two strings equivalent under this rewrite system?"

2.4 Exercise 5b

2.4.1 Modified Rewrite Rules

Same as Exercise 5, but change $aa \rightarrow \text{empty}$ to $aa \rightarrow a$:

- $ab \rightarrow ba$
- $ba \rightarrow ab$
- $aa \rightarrow a$ (pairs of a become single a)
- $b \rightarrow (\text{empty string})$

2.4.2 Sample Reductions

Reducing abba:

$abba \rightarrow baba$ (using $ab \rightarrow ba$)

$baba \rightarrow bbba$ (using $ab \rightarrow ba$)

$bbba \rightarrow bba$ (using $b \rightarrow \text{empty}$)

$bba \rightarrow aa$ (using $b \rightarrow \text{empty}$)

$aa \rightarrow a$ (using $aa \rightarrow a$)

Reducing bababa:

$bababa \rightarrow bbbaaba$ (using $ab \rightarrow ba$ twice)

$bbbaaba \rightarrow baaaba$ (using $b \rightarrow \text{empty}$)

$baaaba \rightarrow aaaba$ (using $b \rightarrow \text{empty}$)

$aaaba \rightarrow aaba$ (using $aa \rightarrow a$)

$aaba \rightarrow abaa$ (using $ab \rightarrow ba$)

$abaa \rightarrow baaa$ (using $ab \rightarrow ba$)

$baaa \rightarrow aaa$ (using $b \rightarrow \text{empty}$)

$aaa \rightarrow aa$ (using $aa \rightarrow a$)

$aa \rightarrow a$ (using $aa \rightarrow a$)

2.4.3 Analysis

Why the ARS is not terminating: Same as Exercise 5 - the rules $ab \rightarrow ba$ and $ba \rightarrow ab$ create infinite cycles.

Non-equivalent strings: Two strings that are not equivalent: "a" and "empty string". We can find exactly 2 non-equivalent strings.

Equivalence classes and normal forms: There are exactly 2 equivalence classes:

1. Strings with even number of a's \rightarrow reduce to empty
2. Strings with odd number of a's \rightarrow reduce to "a"

The normal forms are: empty string and "a"

Modified terminating ARS: To make it terminating, use the same priority as Exercise 5:

1. $b \rightarrow \text{empty}$ (eliminate all b's first)
2. $aa \rightarrow a$ (reduce pairs of a's)
3. $ab \rightarrow ba$ (only if needed)

Questions about strings that can be answered using the ARS:

1. "Given a string, does it contain an even number of a's?"
2. "Given a string, does it contain an odd number of a's?"
3. "Are two strings equivalent under this rewrite system?"

2.4.4 Exploration

2.4.5 Questions

1. If two completely different sets of rewrite rules (Exercise 5 vs 5b) produce the same equivalence classes, what does this tell us about the relationship between implementation and specification in computer science?
2. If "abab" and "bbaa" are equivalent under this system, but clearly different as strings, what does "equivalence" really mean? Is mathematical equivalence different from everyday sameness?

2.5 Week 4

2.5.1 Homework

2.5.2 Exploration

2.5.3 Questions

2.6 Week 5

2.6.1 Homework

2.6.2 Exploration

2.6.3 Questions

2.7 Week 6

2.7.1 Homework

2.7.2 Exploration

2.7.3 Questions

2.8 Week 7

2.8.1 Homework

2.8.2 Exploration

2.8.3 Questions

2.9 Week 8

2.9.1 Homework

Natural Number Game - Tutorial World: Levels 5-8

Level 5: Simplifying with add_zero Prove that $a + (b + 0) + (c + 0) = a + b + c$.

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

Level 6: Targeted rewriting Prove that $a + (b + 0) + (c + 0) = a + b + c$ using explicit arguments.

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

Level 7: succ_eq_add_one Prove that for all natural numbers n , $\text{succ}(n) = n + 1$.

```
rw [one_eq_succ_zero]
rw [add_succ]
rw [add_zero]
rfl
```

Level 8: Proving $2 + 2 = 4$ Prove that $2 + 2 = 4$.

```
nth_rewrite 2 [two_eq_succ_one]
rw [add_succ]
rw [one_eq_succ_zero]
rw [add_succ]
rw [add_zero]
rw [<- three_eq_succ_two]
rw [<- four_eq_succ_three]
rfl
```

2.9.2 Natural Language Proof

Level 8: Proving $2 + 2 = 4$

Proof. We want to prove that $2 + 2 = 4$ using only the Peano axioms and previously established theorems about natural numbers.

We begin by expanding the second 2 in the left-hand side using its definition. By the theorem `two_eq_succ_one`, we know that $2 = \text{succ}(1)$. Rewriting the second occurrence of 2, we obtain:

$$2 + \text{succ}(1) = 4$$

Next, we apply the fundamental recursion axiom for addition, `add_succ`, which states that for any natural numbers a and b , we have $a + \text{succ}(b) = \text{succ}(a + b)$. Applying this axiom gives us:

$$\text{succ}(2 + 1) = 4$$

Now we expand 1 using its definition. By `one_eq_succ_zero`, we know that $1 = \text{succ}(0)$. Substituting this yields:

$$\text{succ}(2 + \text{succ}(0)) = 4$$

We apply `add_succ` again to the inner addition:

$$\text{succ}(\text{succ}(2 + 0)) = 4$$

By the base case axiom for addition, `add_zero`, which states that $n + 0 = n$ for any natural number n , we can simplify $2 + 0$ to 2:

$$\text{succ}(\text{succ}(2)) = 4$$

Now we recognize this expression in terms of known number definitions. By the definition `three_eq_succ_two`, we know that $3 = \text{succ}(2)$. Using this in reverse (indicated by the backwards arrow in the formal proof), we can rewrite $\text{succ}(2)$ as 3:

$$\text{succ}(3) = 4$$

Finally, by the definition `four_eq_succ_three`, we know that $4 = \text{succ}(3)$. Using this in reverse, we obtain:

$$4 = 4$$

This is true by the reflexivity of equality: any object is equal to itself.

Therefore, $2 + 2 = 4$. □

2.9.3 Exploration

The proof of $2 + 2 = 4$ is a remarkable example of how even the simplest arithmetic facts require rigorous justification when building mathematics from first principles. Several profound insights emerge from this exercise:

- **Numbers as constructions:** In the Peano axioms, numbers are not primitive objects but are constructed iteratively from zero using the successor function. The number 2 is defined as $\text{succ}(\text{succ}(0))$, 3 as $\text{succ}(2)$, and 4 as $\text{succ}(3)$. This constructive approach ensures that all natural numbers can be built systematically.
- **Addition as recursion:** Addition is not defined by a lookup table but by two recursive rules: the base case $n + 0 = n$ and the recursive case $n + \text{succ}(m) = \text{succ}(n + m)$. The proof of $2 + 2 = 4$ essentially "executes" this recursive definition step by step.
- **The role of definitions:** Much of the proof consists of unfolding and refolding definitions. We expand 2 into $\text{succ}(1)$, then 1 into $\text{succ}(0)$, perform the addition, and finally recognize the result as 3 and then 4. This shows that definitions are not just abbreviations but active components of reasoning.
- **Computational content of proofs:** This proof has a computational interpretation. Each rewrite step corresponds to a computation step, and the entire proof traces the execution of the addition algorithm. This connection between proofs and programs is central to the Curry-Howard correspondence.
- **Nothing is obvious in formal systems:** What seems trivial in everyday mathematics ($2 + 2 = 4$) requires multiple logical steps when formalized. This explicitness is both a strength (eliminates ambiguity and hidden assumptions) and a weakness (can obscure high-level mathematical intuition).

This exercise bridges the gap between our intuitive understanding of arithmetic and the formal foundations required for computer-verified mathematics and programming language semantics.

2.9.4 Questions

1. Why does proving $2 + 2 = 4$ require eight steps when it seems inherently true?
2. When should a programming language prioritize precision over simplicity?

2.10 Week 9

2.10.1 Homework

2.10.2 Exploration

2.10.3 Questions

2.11 Week 10

2.11.1 Homework

2.11.2 Exploration

2.11.3 Questions

2.12 Week 11

2.12.1 Homework

2.12.2 Exploration

2.12.3 Questions

3 Synthesis

4 Evidence of Participation

5 Conclusion

References