CPSC-354 Report

Jeffrey Bok
Chapman University

September 13, 2025

Abstract

Contents

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

$$\begin{tikzcd}
a \arrow[loop above]
\end{tikzcd}$$

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

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

$$\begin{tikzcd}
& a \arrow[dl] \arrow[dr] & \\
b & & c
\end{tikzcd}$$

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

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

$$\begin{tikzcd}
a \arrow[loop left] \arrow[d] \\
b
\end{tikzcd}$$

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

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

$$\begin{tikzcd}
& a \arrow[dl] \arrow[dr] & \\
b \arrow[loop left] & & c
\end{tikzcd}$$

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

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

$$\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & \\
b \arrow[loop left] & & c \arrow[loop right]
\end{tikzcd}$$

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 $A = \{a\}$, $R = \{\}$

$$\begin{tikzcd}
a
\end{tikzcd}$$

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

Empty graph

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

$$\begin{tikzcd}
a \arrow[r] & b \arrow[loop right]
\end{tikzcd}$$

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

$$\begin{tikzcd}
a \arrow[loop above]
\end{tikzcd}$$

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

$$\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & \\
b & & c \arrow[d] \\
 & & d
\end{tikzcd}$$

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

$$\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & \\
b & & c
\end{tikzcd}$$

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

$$\begin{tikzcd}
 & a \arrow[dl, bend left] \arrow[dr, bend right] & \\
b \arrow[ur, bend left] & & c \arrow[ul, bend right]
\end{tikzcd}$$

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

$$\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & \\
b \arrow[loop left] & & c
\end{tikzcd}$$

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 $\to$ ba
- ba $\to$ ab
- aa $\to$ (empty string)
- b $\to$ (empty string)

2.3.2 Sample Reductions

Reducing abba:

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

baba $\to$ bbaa (using ab $\to$ ba)

bbaa $\to$ baa (using b $\to$ empty)

baa $\to$ aa (using b $\to$ empty)

aa $\to$ empty (using aa $\to$ empty)

Reducing bababa:

bababa $\to$ bbaaaba (using ab $\to$ ba twice)

bbaaaba $\to$ baaaba (using b $\to$ empty)

baaaba $\to$ aaaba (using b $\to$ empty)

aaaba $\to$ aba (using aa $\to$ empty)

aba $\to$ baa (using ab $\to$ ba)

baa $\to$ aa (using b $\to$ empty)

aa $\to$ empty (using aa $\to$ empty)

2.3.3 Analysis

Why is the ARS not terminating?

The first two rules ab $\to$ ba and ba $\to$ 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 → empty to aa → a:

- ab → ba
- ba → ab
- aa → a (pairs of a become single a)
- b → (empty string)

2.4.2 Sample Reductions

Reducing abba:

abba → baba (using ab → ba)

baba → bbaa (using ab → ba)

bbaa → baa (using b → empty)

baa → aa (using b → empty)

aa → a (using aa → a)

Reducing bababa:

bababa → bbaaaba (using ab → ba twice)

bbaaaba → baaaba (using b → empty)

baaaba → aaaba (using b → empty)

aaaba → aaba (using aa → a)

aaba → abaa (using ab → ba)

abaa → baaa (using ab → ba)

baaa → aaa (using b → 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 $\rightarrow$ 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

2.9.2 Exploration

2.9.3 Questions

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