CPSC-354 Report

Ethan Tapia
Chapman University

October 5, 2025

Abstract

Contents

1 Introduction

2 Week by Week

2.1 Week 1

Lecture Summary

We introduced *formal systems* and worked with Hofstadter's MIU-system as a rule–based rewriting game. Alphabet: $\Sigma = \{M, I, U\}$. Axiom (start string): MI. Production rules:

(R1) If a string ends in I, append U: $xI \Rightarrow xIU$.

(R2) If a string is Mx, duplicate x: $Mx \Rightarrow Mxx$.

(R3) Replace any III by U: $xIIIy \Rightarrow xUy$.

(R4) Delete any UU: $xUUy \Rightarrow xy$.

Key idea: reason about *invariants* that rules preserve, instead of searching blindly through derivations.

Homework: The MU-puzzle

Definition 2.1 (I–count and residue)**.** For a string w, let $\#_I(w)$ be the number of I's in w, and define the residue

$$\varphi(w) \;=\; \#_I(w) \bmod 3 \in \{0,1,2\}.$$

Lemma 2.2 (Effect of each rule on $\#_I$)**.** *For any string w:*

1. ***(R1)*** *and* ***(R4)*** *do not change $\#_I$.*
2. ***(R2)*** *doubles the number of I's* after *the initial M, so φ is multiplied by 2 modulo 3.*
3. ***(R3)*** *decreases $\#_I$ by 3, so φ is unchanged.*

Proposition 2.3 (Invariant modulo 3)**.** *Every string derivable from MI has $\varphi \in \{1,2\}$. In particular, no derivable string has $\varphi = 0$.*

Proof. We use induction on the length of a derivation from MI.

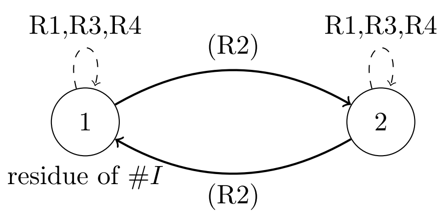
Base. $\varphi(MI) = 1$.

Step. Assume $\varphi \in \{1,2\}$ for some derivable w. By Lemma 2.2, rules (R1), (R3), and (R4) keep φ unchanged, and rule (R2) maps $1 \leftrightarrow 2$ modulo 3. None of these operations yields 0 from a value in $\{1,2\}$. Therefore the next string also has $\varphi \in \{1,2\}$. ☐

Theorem 2.4 (MU is unreachable)**.** *MU cannot be derived from MI in the MIU-system.*

Proof. MU contains zero I's, hence $\varphi(MU) = 0$. By Proposition 2.3, every derivable string has residue 1 or 2. Thus MU is not derivable. ☐

Conclusion. Starting from MI we can toggle the residue $1 \leftrightarrow 2$ with (R2) and otherwise keep it fixed with (R1), (R3), (R4). We never reach residue 0, so no sequence of legal rule applications yields MU.



Question: If the MU-puzzle shows that some goals are unreachable due to invariants (like the mod-3 property of I's), how does this idea connect to undecidability in programming languages?

2.2 Week 2

Lecture Summary
We introduced *Abstract Reduction Systems (ARS)*: a pair (A, R) with one-step reduction $R \subseteq A \times A$. Key notions: reducible/normal form, joinability, confluence, termination, and unique normal forms.

Homework Part 2: The 8 Combinations

We provide an example ARS for each combination of $(\text{confluent}, \text{terminating}, \text{unique NFs})$. If a row is impossible, we explain why.

Confluent	**Terminating**	**Unique NFs**	**Example**
True	True	True	$A = \{a\},\ R = \emptyset$ (Fig. 1)
True	True	False	*Impossible*
True	False	True	$A = \{a, b\},\ R = \{(a, a), (a, b)\}$ (Fig. 2)
True	False	False	$A = \{a\},\ R = \{(a, a)\}$ (Fig. 3)
False	True	True	*Impossible*
False	True	False	$A = \{a, b, c\},\ R = \{(a, b), (a, c)\}$ (Fig. 4)
False	False	True	*Impossible*
False	False	False	$A = \{a, b, c\},\ R = \{(a, b), (a, c), (b, b), (c, c)\}$ (Fig. 5)

Why some rows are impossible. If an ARS has unique normal forms, it must be confluent. If an ARS is both confluent and terminating, then every element reduces to a unique normal form. Therefore the rows $(\mathrm{T}, \mathrm{T}, \mathrm{F})$, $(\mathrm{F}, \mathrm{T}, \mathrm{T})$, and $(\mathrm{F}, \mathrm{F}, \mathrm{T})$ cannot occur.

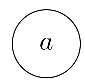


Figure 1: Combination (True, True, True). Terminating, confluent, unique NF.

Figure 2: Combination (True, False, True). Non-terminating, confluent, unique NF b.

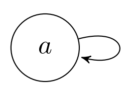


Figure 3: Combination (True, False, False). Non-terminating, confluent, no normal form.

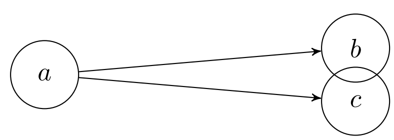


Figure 4: Combination (False, True, False). Terminating, not confluent; two distinct normal forms b, c are not joinable.

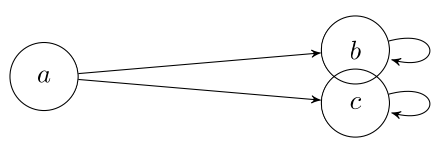


Figure 5: Combination (False, False, False). Non-terminating (loops), not confluent, no unique normal forms.

Conclusion. The MU-puzzle illustrates how invariants prove impossibility in a formal system. The ARS framework provides the general language to study rewrite systems via termination, confluence, and normal forms. The 8-combination analysis shows which behaviors are possible and which are structurally impossible.

Question: Could there be a general framework that unifies invariants with confluence and termination, so that impossibility and determinism appear as two sides of the same rewriting theory?

2.3 Week 3

Lecture Summary
TBD

Homework 3

Exercise 5 Consider an ARS with [A = a,b* $= \varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots,]andrewriterules[ab \to ba, \quad ba \to ab, \quad aa \to \varepsilon, \quad b \to \varepsilon.]$

Reduce some example strings such as $abba$ **and** $bababa$**.**

$$abba \to aa \to \varepsilon, \; bababa \qquad\qquad \to aaa \to a. \tag{1}$$

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

These have different normal forms and cannot be transformed into each other.

How many equivalence classes does $\overset{*}{\longleftrightarrow}$ **have? What are the normal forms?** There are two equivalence classes:

(a) Strings whose normal form is ε,

(b) Strings whose normal form is a.

The class is determined by the parity of the number of a's in the string.

Can you modify the ARS so that it becomes terminating without changing its equivalence classes? Yes. Remove one of the first two rules. They only permute a and b and do not affect equivalence classes, but having both makes the system non-terminating.

Question:
If I remove all the b's from a string, does the remaining word reduce to a or to ε?" This can be answered using the ARS because $b \to \varepsilon$ always deletes b's, and the final result depends only on whether the number of a's left is odd or even. Odd $\mapsto a$, even $\mapsto \varepsilon$.

Exercise 5b Now replace the rule $aa \to \varepsilon$ with $aa \to a$.

1. **Reduce some example strings such as** $abba$ **and** $bababa$**.**

$$abba \to aa \to a, \; bababa \qquad\qquad \to aaa \to aa \to a. \tag{2}$$

2. **Find two strings that are not equivalent.**

   - ε
   - a

How many equivalence classes are there? What are the normal forms? There are two equivalence classes:

(a) Strings with no a's ;→; normal form ε,

(b) Strings with at least one a ;→; normal form a.

Modify the ARS to make it terminating. As above, remove one of the two swapping rules $ab \leftrightarrow ba$.

Question: Is the system confluent? That is, if a string can be reduced in two different ways, do the reductions always lead to the same normal form?

2.4 Week 4

Lecture Summary
An *invariant* is a function or property that remains unchanged under the rewriting relation of an ARS. They are central tools across science (e.g. conservation laws in physics, chemistry, and biology) and mathematics. Formally, $P : A \to B$ is an invariant if $a \to b \Rightarrow P(a) = P(b)$. Strong invariants preserve exact equality, while weak invariants preserve truth of properties. Invariants induce partitions on A, often serving as abstractions of the equivalence relation $\leftrightarrow^*$. They can be used to prove impossibility (show $P(a) = \text{true}$, $P(b) = \text{false}$) and to build *complete invariants*, which fully classify equivalence classes. Examples include letter counts in string rewriting systems and parity arguments in puzzles (domino tilings, sliding puzzles). In programming, invariants explain correctness of while-loops and recursion, while measure functions guarantee termination.

Homework 4.1
Algorithm

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

Conditions under which it always terminates. Assume $a, b \in \mathbb{N}$ with $b \geq 0$. If $b = 0$ the loop does not run and the program returns immediately. If $b > 0$ then each loop iteration is well defined and yields a strictly smaller nonnegative b because $a \bmod b \in \{0, 1, \ldots, b-1\}$. Thus the loop must terminate. (Equivalently: Euclid's algorithm terminates for all nonnegative integers, not both zero.)

Measure function and proof. Let the state be the pair $(a, b) \in \mathbb{N}^2$. Define

$$\phi(a, b) = b.$$

Suppose the guard holds, so $b > 0$. One loop step computes

$$(a', b') = (b,\ a \bmod b).$$

Then $0 \leq b' < b$, hence $\phi(a', b') = b' < b = \phi(a, b)$. Therefore ϕ strictly decreases on every iteration while staying in $\mathbb{N}$. Since $>$ on $\mathbb{N}$ is well founded, no infinite descent exists, so the loop terminates.

Homework 4.2
Fragment

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

Claim. $\phi(left, right) = right - left + 1$ is a measure function for the recursive calls of `merge_sort`.

Proof. We reason about the domain $D = \{(l, r) \in \mathbb{Z}^2 \mid l \leq r\}$ with the measure $\phi(l, r) = r - l + 1 \in \mathbb{N}$.

If $left \geq right$ then $\phi(left, right) \in \{0, 1\}$ and the function returns, so there is no recursive descent.

Assume $left < right$. Let $mid = \lfloor (left + right)/2 \rfloor$. Standard bounds give

$$left \leq mid < right \quad \text{and} \quad left < mid + 1 \leq right.$$

Hence both subranges are valid:

$$(left, mid) \in D, \qquad (mid + 1, right) \in D.$$

Their measures satisfy

$$\phi(left, mid) = mid - left + 1 \leq \left\lfloor \frac{left + right}{2} \right\rfloor - left + 1 < \frac{left + right}{2} - left + 1 = \frac{right - left + 2}{2} \leq right - left,$$

so $\phi(left, mid) \leq right - left < right - left + 1 = \phi(left, right)$. Similarly,

$$\phi(mid+1, right) = right - (mid+1) + 1 = right - mid \leq right - \left\lfloor \frac{left + right}{2} \right\rfloor < right - \frac{left + right}{2} = \frac{right - left}{2} < right$$

Thus each recursive argument strictly decreases the measure ϕ. Since ϕ takes values in $\mathbb{N}$ and strictly decreases along every recursion chain, the recursion is well founded and `merge_sort` terminates.

Question:
We can discovered that Euclid's algorithm always stops. But how could you use an invariant to also show that it actually gives the greatest common divisor, not just any number?

2.5 Week 5

Lecture Summary
Lambda calculus is a minimal but Turing-complete language with only three constructs: abstraction ($\lambda x.e$ defines a function), application ($e_1\, e_2$ applies a function to an argument), and variables (simple names without assignment). Application associates to the left and abstraction chains naturally. Computation is substitution: $(\lambda x.M)\, N \rightsquigarrow M[N/x]$ (the β-rule), with bound variables freely renamable (α-equivalence) to avoid capture. Functions can return functions (currying), and using Church encodings, numbers and arithmetic can be represented purely by substitution.

Homework 5: Lambda Calculus Reduction
We Evaluate:

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

Step 1: Rename the boud variables of the second term to avoid clashes

$$(\lambda f.\, \lambda x.\, f(f(x)))\; (\lambda g.\, \lambda y.\, g(g(g(y))))$$

Step 2: Apply the outer function to its argument

$$\lambda x.\, (\lambda g.\, \lambda y.\, g(g(g(y))))\big((\lambda g.\, \lambda y.\, g(g(g(y))))\, x\big)$$

Step 3: Reduce the inner application

$$\lambda x.\, (\lambda g.\, \lambda y.\, g(g(g(y))))\; (\lambda y.\, x(x(xy)))$$

Step 4: Apply again

$$\lambda x.\, \lambda y.\, (\lambda y.\, x(x(xy)))\big((\lambda y.\, x(x(xy)))\,((\lambda y.\, x(x(xy)))\,y)\big)$$

Step 5: Evaluate the nested calls

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

Final result. This is the Church numeral

$$\lambda f.\, \lambda x.\, f^9(x)$$

This is the number 9 in Church encoding.

Note: The workout shows that $2\,3 = 9$ for Church numerals. In general, Church numerals encode repeated function application, and application corresponds to multiplication.

Question: If variable names don't matter in λ -calculus, what does that suggest about how meaning can exist independently of representation?

2.6 Week 6

Lecture Summary

This lecture introduced recursion in the λ-calculus via the *fixed point combinator*. We learned that recursion can be encoded without special syntax by defining `fix` such that $\texttt{fix}\ F \to F(\texttt{fix}\ F)$. Using this, one can define recursive functions like factorial. We also reviewed the definitions of `let` and `let rec`, which expand into λ-abstractions and applications of `fix`. The key point is that recursion in functional languages comes from self-application and fixed points, with the famous Y-combinator as a canonical construction.

Homework 6: Fixed Points and Recursion

Rules:

$$\begin{aligned} \texttt{fix}\ F \ &\to\ F\,(\texttt{fix}\ F) && \textbf{(def of fix)} \\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \ &\to\ (\lambda x.\, e_2)\ e_1 && \textbf{(def of let)} \\ \texttt{let rec}\ f = e_1\ \texttt{in}\ e_2 \ &\to\ \texttt{let}\ f = (\texttt{fix}\ (\lambda f.\, e_1))\ \texttt{in}\ e_2 && \textbf{(def of let rec)} \end{aligned}$$

Abbreviation. For readability set

$$G \ \equiv\ \lambda f.\, \lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f(n-1).$$

Goal term.

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

Derivation

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

$\to \texttt{let fact} = \texttt{fix}\, G\ \texttt{in fact}\ 3$ <**def of let rec**>

$\to (\lambda \texttt{fact}.\,\texttt{fact}\ 3)\ (\texttt{fix}\, G)$ <**def of let**>

$\to (\texttt{fix}\, G)\ 3$ <**β-rule**>

$\to (G(\texttt{fix}\, G))\ 3$ <**def of fix**>

$\to (\lambda n.\,\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\, G)(n-1))\ 3$ <**β-rule**>

$\to \texttt{if}\ 3 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3 * (\texttt{fix}\, G)(2)$ <**β-rule**>

$\to 3 * (\texttt{fix}\, G)(2)$ <**def of if**>

$\to 3 * \big(G(\texttt{fix}\, G)\big)\ 2$ <**def of fix**>

$\to 3 * (\lambda n.\,\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\, G)(n-1))\ 2$ <**β-rule**>

$\to 3 * (\texttt{if}\ 2 = 0\ \texttt{then}\ 1\ \texttt{else}\ 2 * (\texttt{fix}\, G)(1))$ <**β-rule**>

$\to 3 * (2 * (\texttt{fix}\, G)(1))$ <**def of if**>

$\to 3 * (2 * (G(\texttt{fix}\, G))\ 1)$ <**def of fix**>

$\to 3 * (2 * (\lambda n.\,\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\, G)(n-1))\ 1)$ <**β-rule**>

$\to 3 * (2 * (\texttt{if}\ 1 = 0\ \texttt{then}\ 1\ \texttt{else}\ 1 * (\texttt{fix}\, G)(0)))$ <**β-rule**>

$\to 3 * (2 * (1 * (\texttt{fix}\, G)(0)))$ <**def of if**>

$\to 3 * (2 * (1 * (G(\texttt{fix}\, G))\ 0))$ <**def of fix**>

$\to 3 * (2 * (1 * (\lambda n.\,\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\, G)(n-1))\ 0))$ <**β-rule**>

$\to 3 * (2 * (1 * (\texttt{if}\ 0 = 0\ \texttt{then}\ 1\ \texttt{else}\ 0 * (\texttt{fix}\, G)(-1))))$ <**β-rule**>

$\to 3 * (2 * (1 * 1))$ <**def of if**>

$\to 3 * (2 * 1)$ <**arith**>

$\to 3 * 2$ <**arith**>

$\to 6$ <**arith**>

Result: `fact 3` reduces to 6, each step justified by **def of let rec**, **def of let**, β-**rule**, **def of fix**, **def of if**, and **arith**.

Question: Since the fixed point combinator allows functions to call themselves without being named, what does this suggest about the nature of recursion and whether naming is essential for defining self-reference?

3 Evidence of Participation

4 Conclusion

References

[BLA] Author, Title, Publisher, Year.