

CPSC 354 — Report

Ray Hettleman `rhettleman@chapman.edu`

October 5, 2025

Contents

1	Assignment 1: MIU System — Can we derive MIII from MI?	1
2	Assignment 2: ARS Pictures & Properties	2
3	Assignment 3: Exercises 5 and 5b	3
4	Assignment 4: Termination Exercises	4
5	Assignment 5: Lambda Calculus Workout	5

1 Assignment 1: MIU System — Can we derive MIII from MI?

Problem (restated)

Starting from MI, and using the MIU rules:

- I. If a string ends with I, you may append U.
- II. From Mx you may infer Mxx.
- III. Replace any occurrence of III by U.
- IV. Delete any occurrence of UU.

Determine whether MIII is derivable. Explain your reasoning.

Solution

Idea. Track the count of I's. Rule I and Rule IV do not change that count. Rule II doubles it; Rule III removes 3 but can only apply if 3 consecutive I's already exist.

Starting from MI (one I), the only growth is doubling. So after the first step the number of I's is always even. Removing triples never gives exactly three. So MIII cannot appear.

Conclusion

It is **impossible** to derive MIII from MI. Reason: the number of I's is never equal to three under these rules.


2 Assignment 2: ARS Pictures & Properties


Task (restated)

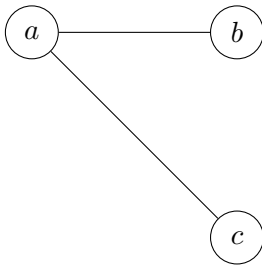
For each given abstract reduction system (ARS) (A, R) : draw the graph, decide whether it is terminating, confluent, and whether it has unique normal forms (UNFs).


Instances

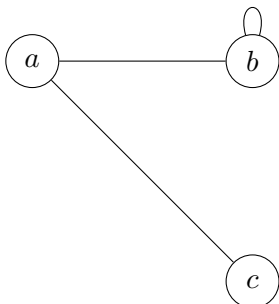
1. $A = \emptyset$ No nodes/edges. *Terminating*: True. *Confluent*: True. *UNFs*: True.

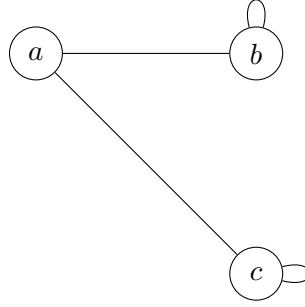
2. $A = \{a\}$, $R = \emptyset$ 
Normal forms: a . *Terminating*: True. *Confluent*: True. *UNFs*: True.

3. $A = \{a\}$, $R = \{(a, a)\}$ 
Infinite $a \rightarrow a \rightarrow \dots$: *Terminating*: **False**. *Confluent*: **True**. *UNFs*: **False**.

4. $A = \{a, b, c\}$, $R = \{(a, b), (a, c)\}$ 
 b, c are normal; from a two distinct NFs. *Terminating*: **True**. *Confluent*: **False**. *UNFs*: **False**.

5. $A = \{a, b\}$, $R = \{(a, a), (a, b)\}$ 
 b is normal; every element has the (unique) NF b . *Terminating*: **False**. *Confluent*: **True**. *UNFs*: **True**.

6. $A = \{a, b, c\}$, $R = \{(a, b), (b, b), (a, c)\}$ 
Non-terminating via $b \rightarrow b$; c is normal, b has no NF. *Confluent*: **False**. *Terminating*: **False**. *UNFs*: **False**.



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

Both b and c loop; no NFs reachable from a . Terminating: **False**. Confluent: **False**. UNFs: **False**.

Summary Table

#	(A, R)	confluent	terminating	unique normal forms
1	(\emptyset, \emptyset)	True	True	True
2	$(\{a\}, \emptyset)$	True	True	True
3	$(\{a\}, \{(a, a)\})$	True	False	False
4	$(\{a, b, c\}, \{(a, b), (a, c)\})$	False	True	False
5	$(\{a, b\}, \{(a, a), (a, b)\})$	True	False	True
6	$(\{a, b, c\}, \{(a, b), (b, b), (a, c)\})$	False	False	False
7	$(\{a, b, c\}, \{(a, b), (b, b), (a, c), (c, c)\})$	False	False	False

All 8 combinations

confluent	terminating	unique NFs	example
True	True	True	e.g. ARS 2 (or 1)
True	True	False	<i>Impossible</i>
True	False	True	ARS 5
True	False	False	ARS 3
False	True	True	<i>Impossible</i>
False	True	False	ARS 4
False	False	True	<i>Impossible</i>
False	False	False	ARS 6 (or 7)

3 Assignment 3: Exercises 5 and 5b

Problem (restated)

Exercise 5 asks us to analyse a given ARS and check for termination, confluence, and unique normal forms. Exercise 5b asks us to consider a small variation and explain the difference.

Solution to Exercise 5

For the ARS with $A = \{a, b\}$ and rules $a \rightarrow a$, $a \rightarrow b$:

- There is a loop $a \rightarrow a$, so the system is **not terminating**.

- From a we can still reach b , and b is normal. Every path from a eventually has the option to reach b , and once at b no rules apply.
- Thus the ARS is **confluent**: all reductions can be joined at b .
- Normal forms are unique: everything reduces to b .

Solution to Exercise 5b

Now suppose we add c with $a \rightarrow c$ and $c \rightarrow c$:

- Again, a can reduce to both b and c .
- But c loops forever and never reaches b .
- That means the system is no longer confluent: starting from a you can end in either the normal form b or the non-terminating loop on c .
- Unique normal forms are therefore lost.

Conclusion

Exercise 5 shows a non-terminating but confluent system (everything has the unique NF b). Exercise 5b shows that adding another looping branch breaks confluence, since from a different outcomes are possible.

4 Assignment 4: Termination Exercises

HW 4.1

Consider the algorithm:

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

This is the Euclidean algorithm for gcd. It terminates whenever $b > 0$, since the measure function $m(a, b) = b$ decreases strictly at each step and never goes negative.

HW 4.2

Consider 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)
```

We can take as measure function

$$\varphi(left, right) = right - left + 1.$$

This is the size of the interval being sorted. Each recursive call splits the interval into smaller subintervals, so the measure decreases until it reaches 1, at which point the recursion stops.

5 Assignment 5: Lambda Calculus Workout

Note

As suggested, I also checked the step sequence in VS Code with Haskell syntax highlighting (file saved with `.hs`), which helps match parentheses.

Problem (restated)

Evaluate

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

Solution

Step 1 (parentheses). Application associates left; abstractions extend right:

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

Step 2 (β). $(\lambda v. M) N \mapsto M[N/v]$ with $v = f$:

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

Step 3 (α then β inside). Rename the inner abstraction's bound x to y to avoid clashes, then apply β :

$$(\lambda f. \lambda y. f(f(f(y)))) x \mapsto \lambda y. x(x(x(y))).$$

Step 4 (β again).

$$\lambda x. ((\lambda f. \lambda y. f(f(f(y))))(\lambda y. x(x(x(y)))) \mapsto \lambda x. \lambda y. (\lambda y. x(x(x(y)))) (\lambda y. x(x(x(y)))) (\lambda y. x(x(x(y))))).$$

Step 5 (pattern). This denotes nine nested applications of x :

$$\mapsto \lambda x. \lambda y. x^9(y).$$

Conclusion

We practiced β -reduction, careful substitution, and α -conversion; the composition doubles the triple into a ninefold application.

6 Assignment 6: Fixed Point Combinator — Computing fact 3

Problem (restated)

Compute the result of

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

using only the computation rules provided in class for **fix**, **let**, and **let rec**, together with simple rules for booleans, conditionals, and arithmetic (as suggested on the sheet).

Rules we use (from lecture)

$\text{fix } F \mapsto (F (\text{fix } F))$	$\langle \text{rule for fix} \rangle$
$\text{let } x = e1 \text{ in } e2 \mapsto ((\lambda x. e2) e1)$	$\langle \text{def of let} \rangle$
$\text{let rec } f = e1 \text{ in } e2 \mapsto \text{let } f = (\text{fix } (\lambda f. e1)) \text{ in } e2$	$\langle \text{def of let rec} \rangle$

For the arithmetic/boolean side (as recommended):

$0=0 \mapsto \text{True}$, $1=0 \mapsto \text{False}$, $\text{if True then } A \text{ else } B \mapsto A$, $\text{if False then } A \text{ else } B \mapsto B$,
 $2*1 \mapsto 2$, $3*2 \mapsto 6$, $3-1 \mapsto 2$, $2-1 \mapsto 1$, etc.

Abbreviation

For readability let

$$F \equiv \backslash \text{fact}. \backslash n. \text{ if } n=0 \text{ then } 1 \text{ else } n * \text{fact } (n-1).$$

Computation

We now reduce, labeling every step with the rule name (matching the sheet).

```
let rec fact = \n. if n=0 then 1 else n * fact (n-1) in 3
→ let fact = (fix (\fact. \n. if n=0 then 1 else n * fact (n-1))) in 3  ⟨def of let rec⟩
→ ((\fact. 3) (fix (\fact. \n. if n=0 then 1 else n * fact (n-1))))  ⟨def of let⟩
→ 3 with an environment binding fact := (fix (\fact. \n. if n=0 then 1 else n * fact (n-1))).  ⟨β⟩
```

But to *evaluate* fact 3 we unfold **fix** as needed:

```
fact = fix (\fact. \n. if n=0 then 1 else n * fact (n-1))
      = fix F
      ↦ (F (fix F))  ⟨rule for fix⟩
      = (\fact. \n. if n=0 then 1 else n * fact (n-1)) (fix F)
      ↦ \n. if n=0 then 1 else n * (fix F) (n-1)  ⟨β⟩
```

Now apply this to 3:

<code>fact 3</code>	\mapsto	<code>if 3=0 then 1 else 3 * (fix F) (3-1)</code>	$\langle\beta\rangle$
	\mapsto	<code>if False then 1 else 3 * (fix F) 2</code>	$\langle 3 = 0 \mapsto \text{False} \rangle$
	\mapsto	<code>3 * (fix F) 2</code>	$\langle \text{if False} \rangle$

Unfold `fix` again to compute `(fix F) 2`:

<code>(fix F)</code>	\mapsto	<code>(F (fix F))</code>	$\langle \text{rule for fix} \rangle$
	\mapsto	<code>\n. if n=0 then 1 else n * (fix F) (n-1)</code>	$\langle\beta\rangle$

So

<code>(fix F) 2</code>	\mapsto	<code>if 2=0 then 1 else 2 * (fix F) (2-1)</code>	$\langle\beta\rangle$
	\mapsto	<code>2 * (fix F) 1</code>	$\langle 2 = 0 \mapsto \text{False}, \text{ if False} \rangle$

Again unfold for the 1 case:

<code>(fix F) 1</code>	\mapsto	<code>if 1=0 then 1 else 1 * (fix F) (1-1)</code>	$\langle\beta\rangle$
	\mapsto	<code>if False then 1 else 1 * (fix F) 0</code>	$\langle 1 = 0 \mapsto \text{False} \rangle$
	\mapsto	<code>1 * (fix F) 0</code>	$\langle \text{if False} \rangle$

Base case:

<code>(fix F) 0</code>	\mapsto	<code>if 0=0 then 1 else 0 * (fix F) (-1)</code>	$\langle\beta\rangle$
	\mapsto	<code>1</code>	$\langle 0 = 0 \mapsto \text{True}, \text{ if True} \rangle$

So back-substituting:

`(fix F) 1` \mapsto `1 * 1` \mapsto `1`.

Then

`(fix F) 2` \mapsto `2 * 1` \mapsto `2`.

Finally

`fact 3` \mapsto `3 * 2` \mapsto `6`.

Conclusion

Following only the lecture's computation rules and the suggested simple arithmetic/boolean reductions, the program

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

reduces to 6. The trace shows the pattern of unfolding `fix` exactly when needed and using β -reductions together with the conditional rules. (I also mirrored the lecture style of naming each rule beside the step, and verified parentheses in VS Code with Haskell highlighting.)