String Rewriting Exercises (2025)

Gabriel Giancarlo

September 15, 2025

Introduction

Term Rewriting refers to rewriting abstract syntax trees (typically without binders). *String Rewriting* is the special case where we only rewrite strings (as opposed to trees). Normal forms, confluence, termination, invariants can all be studied in this simpler setting. Everything we will learn will also transfer to the generalisations of string rewriting. (And, btw, string rewriting is already Turing complete (Why?).)

In all of the following exercises the task is to analyse a so-called *abstract reduction system* (ARS). An ARS $(A, \to)$ consists of a set A of words (finite lists, strings) and $\to$ is a relation on A. When we specify a rewrite rule

$$w \to v$$

we understand that it can be applied inside any longer word. For example, the rewrite rule

$$ba \to ab$$

can be applied to rewrite the word *cbad* to *cabd*.

Footnote. The math behind the exercises requires the material on equivalence relations, abstract reduction systems, termination and invariants, but it may be good to first try the exercises and then learn the math.

The exercises in this section come in form of puzzles. Think of each ARS as the **implementation of an algorithm**. The puzzle for you is to find out the **input-output behaviour** of the algorithm, that is, you have to find out what the algorithm is meant to compute. In each case, the task is to explain without mentioning the rules what specification the algorithm/ARS implements.

Definition. An **abstract characterization** or a **specification** of an ARS is a description of its input/output behaviour that does not refer to the rules of the ARS (the implementation).

The Task

Roughly speaking, the task is as follows:

- Show that the ARS is an algorithm (that is, the ARS is terminating and every input computes to a unique result = the ARS has unique normal forms).
- Find the specification the ARS implements. This usually requires describing the equality (equivalence relation) generated by the rewrite relation independently of the rewrite relation itself by an invariant.

The Roadmap

(Skip this at first reading.)

Here is a roadmap that you may find useful:

- The basic question is how many equivalence classes there are and how we can recognise in which equivalence class a given word is.
- Collect basic facts and observations. Are there normal forms? What are they? Do all elements reduce to a normal form? Are normal forms unique?
- Does the system terminate?
- Can we characterise equivalence classes by unique normal forms?
- Can we characterise equivalence classes by invariants?
- Derive a specification from the invariant.

Exercises (The Method)

The purpose of these exercises is not so much to practice problem solving but rather to learn the method of decidability via rewriting to normal form and the method of invariants. In particular, the relationship between $\longrightarrow$ and $\overset{*}{\longleftrightarrow}$ is important here.

Exercise 1

The rewrite rule is:

$$ba \to ab$$

Why does the ARS terminate?

The system always terminates because every time we apply the rule, the letters get closer to being in the correct order. There are only a limited number of ways to reorder a finite string, so eventually no more rules can be applied.

What is the result of a computation (the normal form)?

The normal form is the string where all the a's come before all the b's. For example, starting with baba we eventually reach aabb.

Show that the result is unique (the ARS is confluent).

Yes, the result is unique. No matter how we choose to apply the rule, we always end up with the same final string: all the a's on the left and all the b's on the right. This shows the system is confluent.

What specification does this algorithm implement?

This algorithm basically sorts the string by moving all the a's to the left and the b's to the right. In other words, it implements a simple sorting process.

Exercise 2

The rewrite rules are:

$$\texttt{aa} \to \texttt{a}, \qquad \texttt{bb} \to \texttt{a}, \qquad \texttt{ab} \to \texttt{b}, \qquad \texttt{ba} \to \texttt{b}.$$

(a) **Why does the ARS terminate?**

Every rule replaces two adjacent letters by a single letter, so each rewrite step strictly decreases the length of the word by exactly 1. Since words are finite, you can't keep shortening forever. Therefore every rewrite sequence must stop after finitely many steps, so the ARS terminates.

(b) **What are the normal forms?**

Because each step reduces length by 1, any normal form must be a word that cannot be shortened further. The only words of length 1 are a and b, and they contain no length-2 substring to rewrite, so they are normal. There are no other normal forms (every word of length ≥ 2 has some adjacent pair and so admits a rewrite), hence the normal forms are exactly

$$\texttt{a} \quad \text{and} \quad \texttt{b}.$$

(c) **Is there a string s that reduces to both a and b?**

No. Intuitively, the rules preserve whether the number of b's is even or odd (see part (d)), and a has zero b's (even) while b has one b (odd). So a given input cannot end up as both a and b. Concretely: since the system terminates and every input has at least one normal form, and because an invariant (parity of #b's) distinguishes a from b, no string can reduce to both.

(d) **Show that the ARS is confluent.**

We use the invariant "number of b's modulo 2" to argue confluence together with termination.

- Check the invariant: each rule changes the string locally but does not change the parity of the number of b's.
  - aa → a: number of b's unchanged (both sides have 0 b's).
  - bb → a: two b's are removed, so #b decreases by 2 (parity unchanged).
  - ab → b and ba → b: before there is exactly one b, after there is one b (parity unchanged).
- By termination, every word rewrites in finitely many steps to some normal form (either a or b). Because parity of #b is invariant, a word with even #b cannot reach b (which has odd #b) and a word with odd #b cannot reach a. So each input has exactly one possible normal form determined by that parity.

Termination plus the fact that every input has a unique normal form implies confluence (there can't be two different normal forms reachable from the same input). So the ARS is confluent.

(e) **Which words become equal if we replace '→' by '=‘?**

If we let '=‘ be the equivalence relation generated by the rewrite rules, then two words are equivalent exactly when they have the same parity of b's. In other words:

$$u = v \quad \Longleftrightarrow \quad |u|_{\mathtt{b}} \equiv |v|_{\mathtt{b}} \pmod 2.$$

So there are exactly two equivalence classes: the class of words with an even number of b's (these are all equivalent to a) and the class of words with an odd number of b's (these are all equivalent to b).

(f) **Characterise the equality abstractly / using modular arithmetic / final specification.**

An abstract (implementation-free) description is: the system computes the parity of the number of b's in the input word. If the number of b's is even, the output is a; if it is odd, the output is b.

A modular-arithmetic formulation: identify a with 0 and b with 1. For a word $w = w_1 \cdots w_n$ set

$$F(w) \;=\; \sum_{i=1}^{n} \mathbf{1}_{\{w_i = \mathtt{b}\}} \pmod 2.$$

Then the normal form is a when $F(w) = 0$ and b when $F(w) = 1$.

Specification: the algorithm takes a word over $\{\mathtt{a}, \mathtt{b}\}$ and returns a single letter that tells you the parity of the number of b's: a for even parity, b for odd parity. Equivalently, it computes the XOR (parity) of the letters when a=0 and b=1.

Example (work shown). Start with baba (it has two b's, so parity is even, we expect a):

$$\mathtt{baba} \xrightarrow{\mathtt{ba}\to\mathtt{b}} \mathtt{bba} \xrightarrow{\mathtt{bb}\to\mathtt{a}} \mathtt{aa} \xrightarrow{\mathtt{aa}\to\mathtt{a}} \mathtt{a}.$$

No matter which valid rewrites we choose at each step, we end up with a; that matches the parity-based specification above.

Exercise 5

The rewrite rules are

$$\mathtt{ab} \longrightarrow \mathtt{ba}, \qquad \mathtt{ba} \longrightarrow \mathtt{ab}, \qquad \mathtt{aa} \longrightarrow \varepsilon, \qquad \mathtt{b} \longrightarrow \varepsilon.$$

I will answer each bullet carefully and give short, clear justifications.

(i) Some sample reductions.

- Start with abba.

$$\mathtt{abba} \xrightarrow{\mathtt{b}\to\varepsilon} \mathtt{aba} \xrightarrow{\mathtt{ba}\to\mathtt{ab}} \mathtt{aab} \xrightarrow{\mathtt{aa}\to\varepsilon} \mathtt{b} \xrightarrow{\mathtt{b}\to\varepsilon} \varepsilon.$$

  So $\mathtt{abba} \to^* \varepsilon$.

- Start with bababa.

$$\mathtt{bababa} \xrightarrow{\mathtt{b}\to\varepsilon} \mathtt{ababa} \xrightarrow{\mathtt{b}\to\varepsilon} \mathtt{aaba} \xrightarrow{\mathtt{aa}\to\varepsilon} \mathtt{ba} \xrightarrow{\mathtt{b}\to\varepsilon} \mathtt{a}.$$

  So $\mathtt{bababa} \to^* \mathtt{a}$.

(These choices of rewrite steps are not unique; other valid choices lead to the same equivalence-class representatives — see the invariant-based description below.)

(ii) Why is the ARS not terminating?

Because of the two swap rules $\mathtt{ab} \to \mathtt{ba}$ and $\mathtt{ba} \to \mathtt{ab}$, you can cycle forever on any alternating adjacent pair. The shortest example is ab:

$$\mathtt{ab} \longrightarrow \mathtt{ba} \longrightarrow \mathtt{ab} \longrightarrow \cdots$$

This gives an infinite rewrite sequence, so the system is not terminating. (The erasing rules exist, but they don't prevent the existence of infinite swapping sequences when you choose only swaps.)

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

A simple pair of non-equivalent strings is

$$\mathtt{a} \quad \text{and} \quad \varepsilon.$$

They are not equivalent because no rule can turn a into ε: the only erasing rules are $\mathtt{aa} \to \varepsilon$ (needs two a's) and $\mathtt{b} \to \varepsilon$ (erases b's). In particular, the parity of the number of a's (even vs odd) cannot be changed by any step, so a single a (odd number of a's) cannot become ε (zero a's, even).

How many non-equivalent strings can we find? If we ask for *pairwise non-equivalent* representatives, the ARS only distinguishes two equivalence classes (see next part). So up to equivalence there are only two distinct outcomes: one represented by ε and one represented

by a. Of course there are infinitely many distinct strings as concrete syntactic objects, but they fall into just two equivalence classes.

(iv) How many equivalence classes does $\overset{*}{\longleftrightarrow}$ have? Describe them; what are the normal forms?

Invariant. Swaps $\mathtt{ab} \leftrightarrow \mathtt{ba}$ never change the counts of a's or b's. The rules $\mathtt{aa} \to \varepsilon$ removes two a's, and $\mathtt{b} \to \varepsilon$ removes one b. Therefore the *parity* of the number of a's,

$$|w|_{\mathtt{a}} \pmod 2,$$

is preserved by every rule. (Each step either removes 0, 2, or an even number of a's.) So parity of a's is an invariant.

Completeness / reachability to representatives. Using swaps we can reorder letters arbitrarily (because the two-way swaps generate all permutations of positions), so we can gather all a's together. Then repeatedly apply $\mathtt{aa} \to \varepsilon$ to cancel a's in pairs; use $\mathtt{b} \to \varepsilon$ to erase any b's. After these reductions every string is reduced to either

ε (if the original had an even number of a's), or $\mathtt{a}$ (if the original had an odd number of a's).

So there are exactly two equivalence classes, determined by the parity of #a. The two normal forms (irreducible representatives) are ε and a. (They are irreducible since none of the four left-hand sides occurs in them.)

Thus $\overset{*}{\longleftrightarrow}$ partitions all strings into two classes:

$$\{w \mid |w|_{\mathtt{a}} \text{ is even}\} \quad \text{and} \quad \{w \mid |w|_{\mathtt{a}} \text{ is odd}\},$$

with canonical normal forms ε and a respectively.

(v) Can you modify the ARS so that it becomes terminating without changing its equivalence classes?

Yes. A standard trick is to orient the swapping in one direction only (so swaps become a "sorting" operation) while keeping the erasing rules. For instance, replace the two-way swaps by a single directed rule

$$\mathtt{ba} \longrightarrow \mathtt{ab}$$

(only move a's left). Keep $\mathtt{aa} \to \varepsilon$ and $\mathtt{b} \to \varepsilon$. Call this the modified ARS.

Why this preserves equivalence classes: the reflexive–symmetric–transitive closure of the original two-way swaps is the same as the closure generated by the one-way swap once you allow symmetric closure (permutations). In other words, the original equivalence relation said "letters can be permuted arbitrarily" — orienting swaps only gives a terminating presentation (a canonical way to permute) but does not change the equivalence relation when you take the equivalence closure.

Why the modified system terminates: use the pair

$$\big(\mathrm{inv}(w),\ |w|\big)$$

ordered lexicographically, where

$$\mathrm{inv}(w) = \#\{(i < j) \mid w_i = \mathtt{b},\ w_j = \mathtt{a}\}$$

is the number of "inversions" (a b before an a). Check each rule:

- $\texttt{ba} \to \texttt{ab}$ strictly decreases $\text{inv}(w)$ by at least 1, length unchanged.
- $\texttt{aa} \to \varepsilon$ strictly decreases $|w|$ (by 2), while $\text{inv}(w)$ stays the same (there are no $\texttt{b}$'s involved in that pair).
- $\texttt{b} \to \varepsilon$ strictly decreases $|w|$ (by 1), $\text{inv}(w)$ may decrease but at worst stays the same.

So every rewrite step strictly decreases the lexicographically ordered pair $(\text{inv}, |w|)$, which is a well-founded order on finite strings. Thus there are no infinite rewrite sequences in the modified system, i.e. it terminates. Because the reachable normal forms under the modified system are still exactly ε and $\texttt{a}$, the equivalence classes are unchanged.

(vi) A couple of natural questions about strings (a specification) that this ARS answers.

Think of the ARS as an algorithm that reduces a given input string to a canonical representative. Two natural questions (specifications) that are decided by this ARS are:

1. *Does the input string have an even number of* $\texttt{a}$*'s?* The ARS reduces the input to ε iff the answer is "yes".
2. *Does the input string have an odd number of* $\texttt{a}$*'s?* The ARS reduces the input to $\texttt{a}$ iff the answer is "yes".

These are good specifications because they are complete invariants: the parity of $\#\texttt{a}$ completely characterizes the equivalence class of any string (independent of the rewrite rules).

Remark / answer to Exse 5b (change $\texttt{aa} \to \varepsilon$ into $\texttt{aa} \to \texttt{a}$).
Replace the rule $\texttt{aa} \to \varepsilon$ by $\texttt{aa} \to \texttt{a}$ and keep the rest. Then:

- Any positive number of $\texttt{a}$'s collapses to a single $\texttt{a}$ (repeatedly use $\texttt{aa} \to \texttt{a}$). All $\texttt{b}$'s still erase by $\texttt{b} \to \varepsilon$.
- The relevant invariant is no longer parity; instead the invariant that classifies strings is whether the string contains at least one $\texttt{a}$ or not.
- Therefore the equivalence classes become:

  $\{w \mid |w|_{\texttt{a}} = 0\}$ (all these are equivalent to ε), $\{w \mid |w|_{\texttt{a}} \geq 1\}$ (all equivalent to $\texttt{a}$).

- Orienting swaps as above (say $\texttt{ba} \to \texttt{ab}$) again gives a terminating presentation with the same classes; a suitable measure is the same lexicographic pair $(\text{inv}(w), |w|)$ or simply $(\text{inv}(w), \#\texttt{a} > 0,\ |w|)$ where the boolean $\#\texttt{a} > 0$ is treated in the order before $|w|$.

So Exse 5b is similar in spirit but the invariant that captures meaning changes from "parity of $\texttt{a}$" to "presence of at least one $\texttt{a}$".

Final short summary .

- The original system is not terminating because swaps can cycle.
- There are exactly two equivalence classes: words with an even number of a's (class represented by ε) and words with an odd number of a's (class represented by a).
- A terminating presentation that preserves the same equivalence relation is obtained by orienting swaps (e.g. ba $\to$ ab) and using the measure $(\mathrm{inv}(w), |w|)$ to prove termination.
- The ARS answers a clear question: "Is the number of a's even or odd?" (or in Exse 5b: "Does the string contain any a's at all?").

Exercise 7
Consider the rewrite rules
ab -¿ a bb -¿ b aa -¿ b
plus rules saying that the order of letters does not matter.
- Think of 'a' and 'b' as colours and an [urn](https://en.wikipedia.org/wiki/Urn$_p$*roblem*)*thathasballsofco*
*Ifyoustartwith*198*blackballsand*99*whiteballs*, *whatisthecolouroftheLastballremaining*?−*Answertheques*
Usetheinvarianttoshowthatthesystemhasuniquenormalforms.−*Ifyoustartwith*n*blackballsand*m*whiteba*

Exercise 8

The rewrite rules are

$$\mathtt{ab} \to \mathtt{cc}, \qquad \mathtt{ac} \to \mathtt{bb}, \qquad \mathtt{bc} \to \mathtt{aa},$$

and we are allowed to permute letters (order doesn't matter), so we can think of configurations just as multisets or triples of counts (a, b, c). Start: $(a, b, c) = (15, 14, 13)$. Total letters $N = 42$ is preserved by every step (each rule consumes two letters and produces two), so any reachable configuration must still sum to 42.

We want to know whether we can reach a configuration that has only one kind of letter, i.e. $(42, 0, 0)$, $(0, 42, 0)$ or $(0, 0, 42)$.

Key invariant) Look at the difference $a - b$ modulo 3. Check how this changes under each rule:

- For ab $\to$ cc: (a, b, c) goes to $(a - 1, b - 1, c + 2)$. So
$$(a - 1) - (b - 1) = a - b,$$
i.e. $a - b$ does not change at all.
- For ac $\to$ bb: $(a, b, c) \mapsto (a - 1, b + 2, c - 1)$. So
$$(a - 1) - (b + 2) = a - b - 3,$$
i.e. $a - b$ changes by -3.
- For bc $\to$ aa: $(a, b, c) \mapsto (a + 2, b - 1, c - 1)$. So
$$(a + 2) - (b - 1) = a - b + 3,$$
i.e. $a - b$ changes by $+3$.

So every rule changes $a - b$ by a multiple of 3. That means the value of $a - b$ modulo 3 is an invariant of the system.

Apply the invariant to the start and targets. Compute the invariant for the start:

$$a - b = 15 - 14 = 1 \equiv 1 \pmod 3.$$

For the three all-one-letter targets we have

$$(42, 0, 0) : \ a-b = 42 \equiv 0 \pmod 3, \quad (0, 42, 0) : \ a-b = -42 \equiv 0 \pmod 3, \quad (0, 0, 42) : \ a-b = 0 \equiv 0 \quad (\text{m}$$

Every all-one-letter configuration has $a - b \equiv 0 \pmod 3$, but our start has $a - b \equiv 1 \pmod 3$. Since $a - b \pmod 3$ is invariant, it is impossible to reach any of the all-one-letter targets from $(15, 14, 13)$.

Conclusion. No — starting from 15 `a`'s, 14 `b`'s and 13 `c`'s you *cannot* reach a configuration with only `a`'s, only `b`'s, or only `c`'s. The invariant $a - b \pmod 3$ (which equals 1 at the start) rules those targets out (they all have value 0 modulo 3).