String Rewriting Exercises (2025)

Gabriel Giancarlo

September 11, 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 3

The rewrite rules are:

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

(a) **Why does the ARS *not* terminate?**

It doesn't terminate because of the two swapping rules $\texttt{ab} \to \texttt{ba}$ and $\texttt{ba} \to \texttt{ab}$. These two rules can be applied back and forth forever on the substring ab. For example

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

so there is an infinite rewrite sequence. The presence of such an endless swap means the system as a whole is not terminating (even though the shortening rules $\texttt{aa} \to \texttt{a}$ and $\texttt{bb} \to \texttt{b}$ would sometimes shorten words).

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

A normal form is a word with no left-hand-side of any rule as a substring. Since the four left-hand sides are exactly the four possible length-2 pairs $\{aa, bb, ab, ba\}$, every word of length ≥ 2 contains some reducible pair and so is not irreducible. Thus the only irreducible words (normal forms) are the words of length 0 or 1:

$$\varepsilon, \ \texttt{a}, \ \texttt{b}.$$

(Here ε is the empty word.)

Note: even though ε, a, b are the only irreducible words, not every input actually reaches one of them by rewriting, because you can get stuck in infinite swapping instead of ever applying the shortening rules.

(c) **Modify the ARS so it is terminating and has unique normal forms, while keeping the same equivalence relation.**

A simple fix is to remove one of the swapping rules so swaps only go in one direction. For example, keep

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

but *drop* the rule $\texttt{ab} \to \texttt{ba}$.

Why this works:

- Every step either shortens the word (the aa or bb rules remove one letter) or strictly decreases the number of inversions (the $\texttt{ba} \to \texttt{ab}$ step moves an a left of a b, decreasing the inversion count). Combining length and inversion gives a well-founded measure that strictly decreases with every rewrite step, so there can be no infinite rewrite sequences — the modified system is terminating.

- Because $\texttt{ba} \to \texttt{ab}$ moves all $\texttt{a}$'s to the left of $\texttt{b}$'s and the $\texttt{aa}, \texttt{bb}$ rules collapse repeated letters, every word reduces to one of the four short canonical forms

  $$\varepsilon,\ \texttt{a},\ \texttt{b},\ \texttt{ab},$$

  and in fact the last two-letter form is always $\texttt{ab}$ (not $\texttt{ba}$) because swaps are directed. After collapsing duplicates, each letter appears at most once, and all $\texttt{a}$'s end up left of all $\texttt{b}$'s, so the unique normal form is determined by whether the word contains an $\texttt{a}$ and/or a $\texttt{b}$.
- The equivalence relation generated by the original rules (the smallest equivalence containing the original $\to$) already makes $\texttt{ab}$ and $\texttt{ba}$ equivalent, because $\texttt{ba} \to \texttt{ab}$ is one of the original rules (and equivalence closure makes relations symmetric). Thus removing $\texttt{ab} \to \texttt{ba}$ does not change the equivalence relation generated by the rules: the equivalence classes are still the same, but the modified oriented system gives a terminating, confluent presentation with one canonical representative per class.

(d) **Describe the specification implemented by the ARS (student level).**

The system is basically deciding which letters appear in the input at least once. In plain terms:

- If the input has no $\texttt{a}$ and no $\texttt{b}$ (the empty word), the output is ε.
- If the input contains at least one $\texttt{a}$ but no $\texttt{b}$, the output is $\texttt{a}$.
- If it contains at least one $\texttt{b}$ but no $\texttt{a}$, the output is $\texttt{b}$.
- If it contains both letters, the output is $\texttt{ab}$ (we pick $\texttt{ab}$ as the canonical representative).

So the algorithm computes the *set of letters present* in the word (represented as a short canonical string). Another way to say this: the ARS collapses each long word to a compact indicator of which of the two letters occur at least once.

Short example (illustration). Start with $\texttt{aabbaa}$. Under the modified rules:

$$\texttt{aabbaa} \xrightarrow{\texttt{aa}\to\texttt{a}} \texttt{abbaa} \xrightarrow{\texttt{bb}\to\texttt{b}} \texttt{abaa} \xrightarrow{\texttt{ba}\to\texttt{ab}} \texttt{aaba} \xrightarrow{\texttt{aa}\to\texttt{a}} \texttt{aba} \xrightarrow{\texttt{ba}\to\texttt{ab}} \texttt{aab} \xrightarrow{\texttt{aa}\to\texttt{a}} \texttt{ab},$$

and $\texttt{ab}$ is the unique normal form (meaning the input contains both letters).

Exercise 4

The rewrite rules are

$$\texttt{ab} \to \texttt{ba}, \qquad \texttt{ba} \to \texttt{ab}.$$

(a) **Why does the ARS not terminate?**

This system doesn't terminate because the two rules allow you to swap an `a` and a `b` back and forth forever. The simplest example is the word `ab`:

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

So there are infinite rewrite sequences, and therefore the ARS is not terminating.

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

A normal form is a word that has no left-hand side of any rule as a substring. Here the left-hand sides are exactly `ab` and `ba`, i.e. any adjacent pair of different letters is reducible. So the only words with no reducible pairs are the words where every adjacent pair is the same letter — in other words, the constant words

$$\varepsilon,\ \texttt{a}^n,\ \texttt{b}^n \quad (n \geq 1).$$

Concretely: the empty word and all words consisting entirely of `a`'s or entirely of `b`'s are the irreducible words. Every word that contains both letters has at least one `ab` or `ba` and so is not in normal form.

Note however that most words containing both letters never reduce to one of these normal forms under the given rules, because the rules only swap letters and never remove them.

(c) **Is the ARS confluent?**

Yes — in a simple sense. Because each rule is the inverse of the other, the one-step relation $\to$ is symmetric, and so its reflexive-transitive closure $\to^*$ is an equivalence relation (it is reflexive, symmetric and transitive). If from a word w you can reach x and you can also reach y, then by symmetry and transitivity you can go from x to y (via w), so x and y have a common descendant (for instance y itself). That meets the usual definition of confluence: any two descendants of a common ancestor have a common descendant.

So the system is confluent, even though it is not terminating.

(d) **What does this ARS 'mean' (what specification does it implement)?**

The rules only swap neighboring `a` and `b` letters, so they never change how many `a`'s and how many `b`'s a word has. The natural invariant is the pair $(|w|_{\texttt{a}}, |w|_{\texttt{b}})$ (the counts of `a` and `b`). The equivalence relation generated by the rules therefore groups together exactly those words that have the same number of `a`'s and the same number of `b`'s — i.e. the words that are anagrams of each other.

The system doesn't delete or create letters, it only permutes them, so two words are considered equivalent iff they contain the same multiset of letters (same counts of `a` and `b`).

(e) **How to make it terminating with the same equivalence relation?**

A common trick is to orient the swap in one direction only, for example keep

$$\texttt{ba} \to \texttt{ab}$$

but drop $\texttt{ab} \to \texttt{ba}$. With only $\texttt{ba} \to \texttt{ab}$ the system becomes exactly like the bubble-sort style system from Exercise 1:

- It terminates because every application of $\texttt{ba} \to \texttt{ab}$ reduces the number of inversions (a finite natural measure), so you can't keep doing steps forever.
- It has unique normal forms $a^{|w|_a} b^{|w|_b}$ (all a's left, all b's right).
- The equivalence relation generated by the original (two-way) swap is the same as the one generated by this one-way swap once you take symmetric, transitive closure: both say "letters can be permuted", i.e. words with the same letter counts are equivalent. So orienting the swap yields a terminating, confluent presentation whose normal forms are the sorted representatives of the same equivalence classes.

So the modification gives a terminating ARS with unique normal forms while preserving the abstract meaning (same equivalence classes).

(f) **Example (small illustration).** Starting from baba under the original two-way rules you can keep swapping:

$$\texttt{baba} \longrightarrow \texttt{abba} \longrightarrow \texttt{a b b a} \longrightarrow \cdots$$

(you can always find swaps, and you can go back and forth.) Under the modified one-way rule $\texttt{ba} \to \texttt{ab}$ you would instead push the a's left and eventually get

$$\texttt{baba} \xrightarrow{\texttt{ba}\to\texttt{ab}} \texttt{abba} \xrightarrow{\texttt{ba}\to\texttt{ab}} \texttt{a b b a} \xrightarrow{\text{repeat}} \texttt{aabb},$$

and then collapse (if collapsing rules are present) to the canonical sorted form $a^{|w|_a} b^{|w|_b}$.

Exercise 5

The rewrite rules are

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

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

(i) Some sample reductions.

- Start with abba.

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

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

- Start with bababa.

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

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

(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 $\texttt{ab} \to \texttt{ba}$ and $\texttt{ba} \to \texttt{ab}$, you can cycle forever on any alternating adjacent pair. The shortest example is ab:

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

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

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

They are not equivalent because no rule can turn a into ε: the only erasing rules are $\texttt{aa} \to \varepsilon$ (needs two a's) and $\texttt{b} \to \varepsilon$ (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 $\texttt{ab} \leftrightarrow \texttt{ba}$ never change the counts of a's or b's. The rules $\texttt{aa} \to \varepsilon$ removes two a's, and $\texttt{b} \to \varepsilon$ removes one b. Therefore the *parity* of the number of a's,

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

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 $\#\mathtt{a}$. 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:

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

So every rewrite step strictly decreases the lexicographically ordered pair $(\mathrm{inv}, |w|)$, 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 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* `a`*'s?* The ARS reduces the input to ε iff the answer is "yes".
2. *Does the input string have an odd number of* `a`*'s?* The ARS reduces the input to `a` iff the answer is "yes".

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

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

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

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

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

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

Final short summary (student voice).

- 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. $\mathtt{ba} \to \mathtt{ab}$) 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 8

The rewrite rules are

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

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 $\texttt{ab} \to \texttt{cc}$: (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 $\texttt{ac} \to \texttt{bb}$: $(a, b, c) \mapsto (a - 1, b + 2, c - 1)$. So

  $$(a - 1) - (b + 2) = a - b - 3,$$

  i.e. $a - b$ changes by -3.

- For $\texttt{bc} \to \texttt{aa}$: $(a, b, c) \mapsto (a + 2, b - 1, c - 1)$. 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).

Exercise 9

We work over the alphabet $\{O, R, K\}$. Write x for an arbitrary string over $\{O, R, K\}^*$. The four rules in a precise ARS form are:

(1)	$uR \longrightarrow uRK$	(if the last letter is R you may add a K at the end)
(2)	$Ox \longrightarrow Oxx$	(if a word begins with O and has tail x, duplicate the tail)
(3)	$u\,RRR\,v \longrightarrow uKv$	(replace any occurrence of RRR by K)
(4)	$u\,KK\,v \longrightarrow uv$	(erase any occurrence of KK)

Here u, v are arbitrary context strings in $\{O, R, K\}^*$ and rule (2) is intended to apply when the whole word has the form Ox (i.e. O is the first letter). This describes the ARS $(A, \to)$ where $A = \{O, R, K\}^*$ and $\to$ is the relation generated by the four schemes above.

Some sample reductions. I'll show a few short reductions so you get a feel for the rules.

(a) $OK \xrightarrow{\text{(2) with } x=K} OKK \xrightarrow{(4)} O$.

(b) $OR \xrightarrow{(1)} ORK \xrightarrow{\text{(2) with } x=RK} ORKRK \xrightarrow{\text{(2) with } x=RKRK} ORKRKRKRK$ (and so on, tails can blow up)

(c) $ORRR \xrightarrow{(3)} OK$ (because the substring RRR becomes K).

Note (c): if you already have three consecutive R's anywhere you can collapse them to a single K.

Question: Can we reduce `OK` to `OR`?
No. The easy reason is that `OK` contains zero R's, and none of the four rules can create an R out of nothing:

- Rule (1) only appends a K when there is already an R at the end — it does not create new R's.
- Rule (2) duplicates the tail x, so it only copies letters already present; it does not introduce new letters of types not already in x.
- Rule (3) replaces three R's by a single K (it removes R's, never adds them).
- Rule (4) erases KK (it only removes K's).

So starting from OK the number of R's is always 0. Since `OR` has one R, `OK` $\not\to^*$ `OR`.

Question: Can we reduce `OR` to `OK`?
No — and here a simple invariant proves impossibility.
Let $r(w)$ denote the number of R's in a word w. Track how r changes under each rule:

- (1) $uR \to uRK$: r is unchanged.
- (2) $Ox \to Oxx$: if the word is Ox and x contains r_x many R's, then after the rule the word is Oxx and the number of R's becomes $2r_x$. So rule (2) multiplies the count of R's in the tail

Exercise 10

Setup. We have a box containing some black and white balls. A step is: remove any one ball; if the removed ball is black then (after removing it) you may add any finite number of white balls. We need to show that every possible sequence of such steps eventually stops (i.e. the process always terminates).

Idea (student-style). The only thing that can be added to the box are white balls, and the only way to change the number of black balls is to remove them (you never add black balls). Intuitively that suggests the number of black balls will eventually run out if we keep removing them often enough; but we must also handle the fact that removing a black can add arbitrarily many whites, which could allow infinitely many further removals of whites. The right way to make this rigorous is to pick a measure (an ordering) on configurations that always strictly decreases with each move and that has no infinite descending chains. A lexicographic order on the pair $(\#\text{black}, \#\text{white})$ does exactly that.

Formal argument.

Describe a configuration by the pair (B, W) where B is the number of black balls and W the number of white balls (both are natural numbers).

Order these pairs lexicographically, with B the primary coordinate and W the secondary one:

$$(B_1, W_1) \;>\; (B_2, W_2) \quad\Longleftrightarrow\quad \big(B_1 > B_2\big) \;\text{ or }\; \big(B_1 = B_2 \text{ and } W_1 > W_2\big).$$

(So we compare first by how many black balls there are; if those are equal we compare by how many white balls there are.)

This order is well-founded: any strictly decreasing sequence $(B_0, W_0) > (B_1, W_1) > (B_2, W_2) > \cdots$ must have strictly decreasing B's eventually (because B cannot stay constant forever while W strictly decreases indefinitely as W is a natural number). Since B is a natural number it cannot decrease infinitely many times, so infinite descending chains are impossible.

Now check that every allowed move strictly decreases (B, W) in this lexicographic order:

- If you remove a white ball, then B stays the same and W decreases by 1. Thus (B, W) goes to $(B, W-1)$, which is strictly smaller (same B, smaller W).

– If you remove a black ball, then B decreases by 1. After removing the black you may add some finite number $k \geq 0$ of white balls, so the new configuration is $(B-1, W+k)$. Even though $W+k$ could be larger than the old W, the first coordinate $B-1$ is strictly smaller than B. Hence $(B-1, W+k)$ is strictly smaller than (B, W) in the lexicographic order (because the primary coordinate decreased).

So every possible move produces a strictly smaller pair (B', W') under our well-founded order. Therefore no infinite sequence of moves is possible and every sequence of moves must eventually terminate.

Conclusion. Using the lexicographic order on the pair (#black, #white) as a measure shows that every allowed move strictly decreases the configuration and this order has no infinite descending chains. Hence the process always terminates.

Remark / example. If we start with $(B, W) = (3, 0)$, possible moves might be

$(3,0) \to (2,5)$ (removed a black, added 5 whites) $\to (2,4) \to (2,0) \to (1,0) \to (0,0)$,

and each arrow is a decrease in the lexicographic order (first coordinate falls from 3 to 2, then whites drop while blacks stay 2, etc.), so the chain must stop.

Exercise 10

Setup. We have a box containing some black and white balls. A step is: remove any one ball; if the removed ball is black then (after removing it) you may add any finite number of white balls. We need to show that every possible sequence of such steps eventually stops (i.e. the process always terminates).

Idea. The only thing that can be added to the box are white balls, and the only way to change the number of black balls is to remove them (you never add black balls). Intuitively that suggests the number of black balls will eventually run out if we keep removing them often enough; but we must also handle the fact that removing a black can add arbitrarily many whites, which could allow infinitely many further removals of whites. The right way to make this rigorous is to pick a measure (an ordering) on configurations that always strictly decreases with each move and that has no infinite descending chains. A lexicographic order on the pair (#black, #white) does exactly that.

Conclusion. Using the order on the pair (#black, #white) as a measure shows that every allowed move strictly decreases the configuration and this order has no infinite descending chains. Hence the process always terminates.

Remark / example. If we start with $(B, W) = (3, 0)$, possible moves might be

$(3,0) \to (2,5)$ (removed a black, added 5 whites) $\to (2,4) \to (2,0) \to (1,0) \to (0,0)$,

and each arrow is a decrease in the lexicographic order (first coordinate falls from 3 to 2, then whites drop while blacks stay 2, etc.), so the chain must stop.