# CPSC-354 Report

Gabriel Giancarlo
Chapman University

October 13, 2025

**Abstract**

This report documents my work on string rewriting exercises, exploring abstract reduction systems (ARS) and their properties. Through various string transformation exercises, I learned about termination, confluence, and the use of invariants to characterize equivalence classes.

## Contents



## 1 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).

# 2 Week by Week

## 2.1 Week 1: String Rewriting Systems

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.

### 2.1.1 Exercise 1: Basic Sorting

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.

### 2.1.2 Exercise 2: Parity Computation

The rewrite rules are:

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

[label=(b)]

1. **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.

2. **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 $\geq 2$ has some adjacent pair and so admits a rewrite), hence the normal forms are exactly

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

3. **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.

4. **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` $\to$ `a`: number of `b`'s unchanged (both sides have 0 `b`'s).
     - `bb` $\to$ `a`: two `b`'s are removed, so #`b` decreases by 2 (parity unchanged).
     - `ab` $\to$ `b` and `ba` $\to$ `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.

5. **Which words become equal if we replace '$\to$' 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:

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

   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`).

6. **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

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

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

   **Specification:** the algorithm takes a word over $\{\texttt{a}, \texttt{b}\}$ 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`):

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

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

# 3 Essay

Working through these string rewriting exercises was both challenging and enlightening. The most fascinating aspect was discovering how seemingly simple string transformation rules can implement complex algorithms.

The key insight that emerged was the power of invariants. In Exercise 2, realizing that the parity of the number of `b`'s was preserved by all rewrite rules was the breakthrough that made everything else fall into place. This showed me how mathematical properties can be used to prove correctness and understand the behavior of algorithms without having to trace through every possible execution path.

The concept of confluence was particularly interesting. The idea that different sequences of rewrite steps can lead to the same final result demonstrates the robustness of these systems. It's reassuring to know that the order of operations doesn't affect the final outcome, which is crucial for parallel and distributed computing.

These exercises also highlighted the connection between formal systems and practical algorithms. The string rewriting systems we studied are essentially abstract machines that compute specific functions. Understanding this connection helps bridge the gap between theoretical computer science and practical programming.

# 4 Evidence of Participation

I completed all the string rewriting exercises, including:

- Exercise 1: Analysis of the sorting algorithm with rule $ba \to ab$
- Exercise 2: Complete analysis of the parity computation system with four rewrite rules
- Detailed proofs of termination, confluence, and invariant properties
- Mathematical characterization of equivalence classes using modular arithmetic
- Worked examples showing the reduction process step-by-step

Each exercise was solved with careful mathematical reasoning, including formal proofs where appropriate. The solutions demonstrate understanding of abstract reduction systems, termination analysis, and the use of invariants to characterize algorithm behavior.

# 5 Conclusion

These string rewriting exercises provided valuable insight into the mathematical foundations of computation. The exercises demonstrated how:

- Simple rewrite rules can implement complex algorithms
- Invariants provide powerful tools for proving correctness
- Mathematical abstraction helps understand algorithm behavior
- Formal systems can serve as models for computation

The experience of working through these puzzles has improved my ability to think formally about computational problems and to construct rigorous arguments about program behavior. These skills will be valuable as I continue to study computer science and work with different programming paradigms.

## References

[ARS] Franz Baader and Tobias Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[Term] Nachum Dershowitz and Jean-Pierre Jouannaud, Rewrite Systems, Handbook of Theoretical Computer Science, 1990.