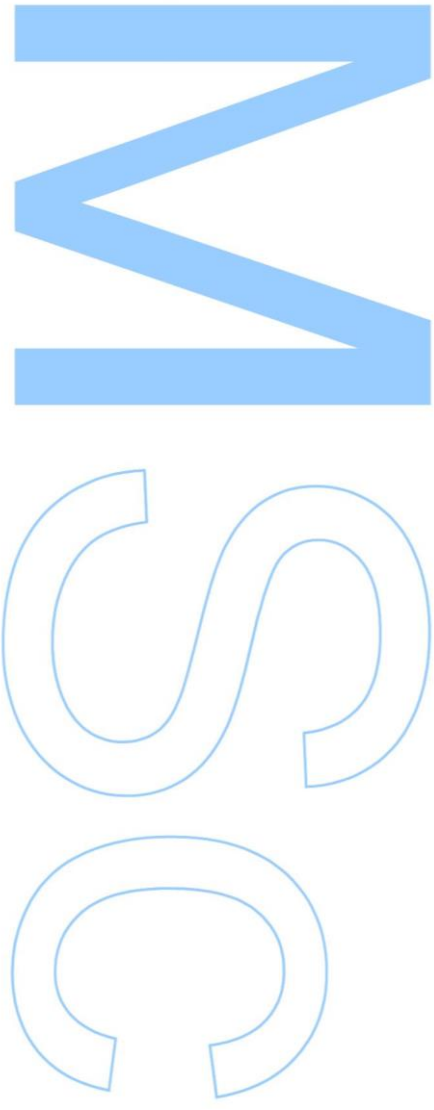
# Transducers and 2D Regular Expressions

João Rebelo Grifo Pires
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2018

**Orientador**
Nelma Resende Araújo Moreira, Professora Auxiliar,
Faculdade de Ciências da Universidade do Porto

**Coorientador**
Rogério Ventura Lages dos Santos Reis, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto



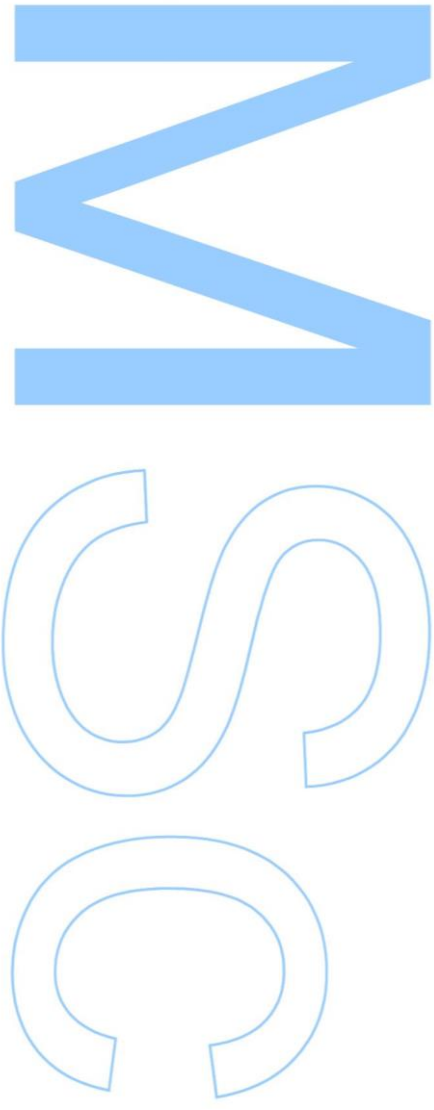
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



# Abstract

Regular expressions are a popular research field in Computer Science, which provide, among other things, a compact way for representing regular languages. With that in mind, we propose a definition of regular expressions for rational relations, which can be represented by finite transducers.

We present some important results, like an upper-bound for the transducer equivalent to a given two-dimensional regular expression.

We present extensions of some classical automata and regular expressions to transducers and two-dimensional regular expressions, for instance the Thompson's method and the state elimination method.

On the other hand, partial derivatives have numerous applications in regular expressions, which include the word problem. The word problem has already been defined and solved for transducers, but we define partial derivatives for two-dimensional regular expressions and present an algorithm for deciding this, without the need to convert the regular expression to an equivalent transducer.

Furthermore, we present a method for deciding if two functional two-dimensional regular expressions are equivalent, using the method we know for transducers. This implies the usage of the presented methods for converting a two-dimensional regular expression to an equivalent transducer.

Finally, we implement these new representations by the means of two-dimensional regular expressions in FAdo system, as well as the subsequent algorithms and methods described.

**Keywords:** algorithms, automata, FAdo, partial derivatives, regular expressions, transducers, two-dimensional regular expressions.

# Resumo

Expressões regulares são um popular campo de investigação em Ciência de Computadores, que providenciam, entre outras coisas, uma forma compacta de representar linguagens regulares. Tendo isso em vista, propomos uma definição para expressões regulares de relações racionais, que podem ser representadas por transdutores finitos.

Apresentamos alguns resultados importantes, como um majorante para o número de estados do transdutor equivalente a uma dada expressão regular a duas dimensões.

Apresentamos extensões de métodos clássicos em autómatos e expressões regulares a transdutores e expressões regulares a duas dimensões, por exemplo o método de Thompson e o método de eliminação de estados.

Por outro lado, derivadas parciais têm numerosas aplicações em expressões regulares, o que inclui o problema da palavra. Este problema estava já definido e resolvido para transdutores, mas definimos neste trabalho derivadas parciais para expressões regulares a duas dimensões e apresentamos um algoritmo para o problema da palavra, sem a necessidade de converter a expressão regular para um transdutor equivalente.

Para além disso, apresentamos um método para decidir se duas expressões regulares a duas dimensões funcionais são equivalentes, usando o método que conhecemos para transdutores. Isto implica a utilização dos métodos apresentados para converter uma expressão regular a duas dimensões num transdutor equivalente.

Finalmente, implementamos estas novas representações por forma de expressões regulares a duas dimensões no sistema FAdo, bem como os algoritmos e métodos subsequentes.

**Palavras-chave:** algoritmos, autómatos, derivadas parciais, expressões regulares, expressões regulares a duas dimensões, FAdo, transdutores.

# Acknowledgments

First of all, I would like to deeply thank Professors Nelma Moreira and Rogério Reis, who provided the tools for this work, for all the patience, guidance and hard work throughout the last years. I would also like to thank Professor Stavros Konstantinidis for his ideas and directions for research.

I can not forget to thank my family and friends. In particular, my mom, Maria, for supporting my studies, as well as my sister, Eloísa, for all the support and guidance. I would also like to thank Pedro and Gonçalo for their wisdom.

Last, but not least, I want to thank the research project UID-MAT-00144-2013.

**Para a avó Adelina**

# Contents

# List of Figures

# Listings

# Chapter 1

# Introduction

Regular expressions and automata theory are two popular fields in computer science, with hundreds of papers and work over the years.

Transducers, which have also been studied in the last few decades, are a way to characterise rational relations between words. Various formulations for transducers have been presented, for instance Berstel [3], Choffrut [9] and Sakarovitch [26]. This work is motivated by the premise of finding more compact ways to represent transducers, like there is for automata, namely, regular expressions.

With that in mind, we present two models for such type of representation, a standard and a general form. One can see the first as changing the alphabet from symbols to pairs of symbols, and the second one as having for atoms an input and an output regular expression, that is, where any word that can be generated from the regular expression over the input alphabet can produce as output any word from the regular expression over the output alphabet.

We also present some classical automata theory methods, as well as some non-classic ones. Thus, we analyse the works of Antimirov [2], Brzozowski [6] and Thompson [27], among others, who have contributed with important methods that we try to extend to transducers, which include the state elimination method and Thompson's method. We take into consideration the work of Antimirov on partial derivatives, and we define the corresponding notions to two-dimensional regular expressions, where the designation refers to the fact that we have some notion of input and output.

We also present one implementation of the model that we propose in the FAdo system [12], a Python system that offers various tools for manipulating regular languages. Other systems include Grail/Grail+ [15, 23], Vaucanson [28], OpenFST [25] and JFLAP [30]. Some of these systems allow one to manipulate such objects within simple script environments. FAdo provides a set of methods for manipulating automata, regular expressions, transducers and, as of this work, two-dimensional regular expressions, on a Python shell.

With that in mind, this work also extends the capabilities of the FAdo system by implementing

important methods for both transducers and two-dimensional regular expressions. More specifically, we present an implementation for finding a sequential transducer equivalent to a given one, and two new implementations for testing the functionality of a transducer a witness of non-functionality, as well as methods related to two-dimensional regular expressions, like calculating the linear form and the set of partial derivatives with respect to a pair of symbols. This work is founded on the theory of rational relations and transducers [3, 26].

The structure of the thesis is as follows. In the first two chapters, we present some important background. Namely, in Chapter 2, we review some notions of classical automata theory and regular expressions. In Chapter 3, on the other hand, we define binary relations over words and transducers, and we present a new implementation of an algorithm for testing the functionality of a transducer, as well as a method based on this algorithm to obtain a witness of non-functionality of a given transducer. Finally, we present an algorithm for obtaining a sequential transducer equivalent to a given functional transducer.

In Chapter 4, we present the main focus of this work. First, we start by defining two-dimensional regular expressions, as well as some important characteristics. We continue by analysing in more depth the two main models we propose, the standard form and the general form. We finish this chapter by presenting methods for conversion between transducers and two-dimensional regular expressions. We also define the word problem for transducers, and we present a method for deciding this on the regular expression level.

In Chapter 5, we discuss the implementation in FAdo system of the definitions, methods and algorithms presented in Chapter 4, as well as the newly implemented methods discussed in Chapter 3.

We finish this thesis with Chapter 6, where we conclude the work and present some directions for future research.

# Chapter 2

# Background

We start by introducing some basic knowledge [17]. Let $S$ be a set. We denote the cardinality of $S$ by $|S|$ and the set of all subsets of $S$ by $2^S$. An *alphabet* is a finite nonempty set of symbols. The Greek letters $\Sigma$ and $\Delta$ denote any two arbitrary alphabets. A *word* over $\Sigma$ is a finite sequence of symbols taken from $\Sigma$. The *empty word* is the word consisting in zero symbols and is denoted by $\varepsilon$. For example, $\sigma$ and $\tau$ are the symbols of the alphabet $\Sigma = \{\sigma, \tau\}$, and $\varepsilon$, $\sigma$ and $\sigma\tau\tau\sigma$ are words over $\Sigma$.

The *set of all words*, or word, including the empty word, over an alphabet $\Sigma$ is denoted by $\Sigma^*$. For example, if $\Sigma = \{\sigma, \tau\}$,

$$\Sigma^* = \{\varepsilon, \sigma, \tau, \sigma\sigma, \sigma\tau, \tau\sigma, \tau\tau, \sigma\sigma\sigma, \ldots\}.$$

The *concatenation* of two words $w$ and $w'$ over $\Sigma$, denoted $w \cdot w'$ or simply $ww'$, is the word obtained by juxtaposing the word $w'$ at the end of the word $w$. For instance, the concatenation of $\sigma\tau$ and $\tau\sigma$ is $\sigma\tau\tau\sigma$. The concatenation of two words $w$ and $w'$ can be defined recursively on the structure of $w'$ as follows

$$\begin{aligned} w\varepsilon &= w \\ w\,(w'\sigma) &= (ww')\,\sigma, \text{ where } \sigma \in \Sigma. \end{aligned}$$

Therefore, the concatenation is an associative operation with identity element $\varepsilon$, that is, $w\varepsilon = \varepsilon w = w$. Thus, the set $\Sigma^*$ with the concatenation operation is a monoid with the empty word as identity.

The length of a word $w$, denoted by $|w|$, is the number of $\Sigma$-symbols in $w$. For instance, $\varepsilon$, $\sigma$ and $\sigma\tau\tau\sigma$ have lengths 0, 1 and 4, respectively. The length of a word $w \in \Sigma^*$ can be defined recursively on the structure of $w$ as follows

$$\begin{aligned} |\varepsilon| &= 0 \\ |w\sigma| &= |w| + 1, \text{ where } \sigma \in \Sigma. \end{aligned}$$

Let $w$ be a word of the form $uvx$, where $u, v, x \in \Sigma^*$. We say that $u$ is a *prefix* of $w$, $v$ is an *infix* and $x$ is a *suffix*. If $u \neq w$, we say that $u$ is a *proper* prefix of $w$. Similarly, if $v \neq w$ and $x \neq w$, $v$ is a proper infix and $x$ is a proper suffix of $w$, respectively. We say that two words, $w$ and $w'$ are conjugate if they are cyclic shifts of each other, that is, if there are words $u, v \in \Sigma^*$ such that $w = uv$ and $w' = vu$.

## 2.1 Formal Languages

Given an alphabet $\Sigma$, a *language* over $\Sigma$ is a subset of $\Sigma^*$. The *empty language* is the empty set, and is thus denoted by $\emptyset$. The set $\Sigma^*$ is called the *universal language*. The *cardinality* of a language $\mathcal{L}$ is denoted by $|\mathcal{L}|$.

On the set of languages, the usual set operations are defined, such as union, intersection and difference. The *concatenation* of two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ is denoted by $\mathcal{L}_1 \cdot \mathcal{L}_2$, where the $\cdot$ operator is often omitted, and is the language

$$\mathcal{L}_1\mathcal{L}_2 = \{w_1w_2 \mid w_1 \in \mathcal{L}_1 \wedge w_2 \in \mathcal{L}_2\}.$$

The *n-th power* of a language $\mathcal{L}$, denoted by $\mathcal{L}^n$ is defined recursively as

$$\begin{aligned} \mathcal{L}^0 &= \{\varepsilon\}, \\ \mathcal{L}^n &= \mathcal{L}^{n-1}\mathcal{L}, \text{ for } n \geq 1. \end{aligned}$$

The *Kleene closure* of $\mathcal{L}$ is denoted by $\mathcal{L}^*$ and is the set

$$\mathcal{L}^* = \bigcup_{i \geq 0} \mathcal{L}^i,$$

and the *positive closure* of $\mathcal{L}$, denoted by $\mathcal{L}^+$, is the set

$$\mathcal{L}^+ = \bigcup_{i \geq 1} \mathcal{L}^i = \mathcal{L}\mathcal{L}^*.$$

The *complement* of $\mathcal{L}$ is denoted by $\overline{\mathcal{L}}$ and is the set

$$\overline{\mathcal{L}} = \Sigma^* \setminus \mathcal{L}.$$

The *class of regular languages* over $\Sigma$ is the smallest set that contains $\emptyset$ and the singletons $\{\sigma\}$, for each $\sigma \in \Sigma$, and that is closed under union, concatenation and Kleene star. The class of regular languages is also closed under the intersection and complement operations.

Given a language $\mathcal{L}$, the *constant part* of $\mathcal{L}$, denoted by $\epsilon(\mathcal{L})$, is defined by

$$\epsilon(\mathcal{L}) = \begin{cases} \varepsilon, \text{ if } \varepsilon \in \mathcal{L}, \\ \emptyset, \text{ otherwise.} \end{cases}$$

## 2.2 Finite Automata

Languages are usually represented by (finite) models. Finite automata are important such models. In this section, we define two types of finite automata: deterministic finite automata and nondeterministic finite automata.

### 2.2.1 Deterministic Finite Automata

**Definition 2.1.** A *deterministic finite automaton* (DFA) over an alphabet $\Sigma$ is a quintuple

$$\dot{a} = (Q, \Sigma, \delta, i, F),$$

where $Q$ is the finite set of *states*, $\delta : Q \times \Sigma \to Q$ is the finite set of *transitions*, $i \in Q$ is the *initial state* and $F \subseteq Q$ is the set of *final states*.

We can graphically represent a DFA using a *transition diagram*, which is a directed graph. The vertices of the graph correspond to the states in $Q$ and are represented by a circle. Each transition from state $p$ to state $q$, labeled by $\sigma$, $\delta(p, \sigma) = q$, corresponds to an edge from $p$ to $q$ labeled with $\sigma$. The initial state $i$ is represented by a circle with an unlabeled *incoming* edge. Final states are represented by double circles.

**Example 2.2.** Let $\dot{a}_{2.2} = (Q, \Sigma, \delta, i, F)$ be the DFA represented by the diagram in Figure 2.1, where the set of states is $Q = \{0, 1, 2\}$, the alphabet is $\Sigma = \{\sigma, \tau\}$, the initial state is $i = 0$, the set of final states is $F = \{2\}$, and the transition function is defined by

$$\begin{aligned} &\delta(0, \sigma) = 2, && \delta(1, \sigma) = 2, && \delta(2, \sigma) = 2, \\ &\delta(0, \tau) = 1, && \delta(1, \tau) = 2, && \delta(2, \tau) = 1. \end{aligned}$$

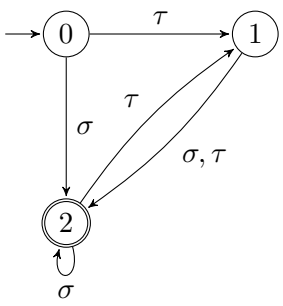



Figure 2.1: Diagram of the DFA $\dot{a}_{2.2}$.

Let $\delta : Q \times \Sigma^* \to Q$ be an extension of the transition function to words. This extension is recursively defined by

$$\begin{aligned} \delta(p, \varepsilon) &= p, \\ \delta(p, \sigma w) &= \delta(\delta(p, \sigma), w), \text{ where } w \in \Sigma^*, \sigma \in \Sigma. \end{aligned}$$

Let $\dot{a}$ be a DFA and $(p, \sigma, q)$ a transition of $\dot{a}$. We say that $\sigma$ is the *label of the transition*, and we say that $p$ has an *outgoing* transition (with label $\sigma$). A *path* of $\dot{a}$ is a finite sequence $(p_0, \sigma_1, p_1, \ldots, \sigma_\ell, p_\ell)$, for some nonnegative integer $\ell$, such that each triplet $(p_{i-1}, \sigma_i, p_i)$, where $i \geq 1$, is a transition of $\dot{a}$. We naturally define $\sigma_1 \cdots \sigma_\ell$ as the *label of the path*. An *accepting path* is a path where $p_0$ is the initial state and $p_\ell$ is a final state. The *language accepted* by $\dot{a}$ is the set of the labels of all the accepting paths of $\dot{a}$ and is denoted by $\mathcal{L}(\dot{a})$. A state $q \in Q$ is *accessible* if there is a path from an initial state to $q$ and *coaccessible* if there is a path from $q$ to a final state. The DFA $\dot{a}$ is called *trim* if every state is accessible and coaccessible, that is, if it appears in at least one accepting path of $\dot{a}$. Finally, we say that $\dot{a}$ *accepts* $w$ if $w$ is the label of an accepting path, that is, if $\delta(p_0, w) = p_\ell$, where $p_0$ is the initial state and $p_\ell \in F$.

In Example 2.2, the path $(0, \tau, 1, \sigma, 2, \sigma, 2)$ is an accepting path, but the path $(1, \sigma, 2, \sigma, 2)$ is not. The label of the first path is $\tau\sigma\sigma$ and the label of the second path is $\sigma\sigma$.

### 2.2.2 Nondeterministic Finite Automata

A *nondeterministic finite automaton* (NFA) is an extension of the DFA where each transition can be from a state to a set of states instead of just one state. Moreover, a NFA can have more than one initial state. An $\varepsilon$-NFA is defined exactly as a NFA except that transitions can be labeled by $\varepsilon$. We often say NFA regardless of whether it accepts $\varepsilon$ transitions or not.

**Definition 2.3.** A *nondeterministic finite automaton* (NFA) over an alphabet $\Sigma$ is a quintuple

$$\dot{a} = (Q, \Sigma, \delta, I, F),$$

where $Q$, $\Sigma$ and $F$ are defined as in Definition 2.1, $\delta \subseteq Q \times \Sigma \times Q$ is the finite set of *transitions* and $I \subseteq Q$ is the set of *initial states*.

We often right $\delta(p, \sigma) = q$ instead of $\delta(p, \sigma) = \{q\}$ when there is no risk of ambiguity.

**Example 2.4.** Let $\dot{a}_{2.4} = (Q, \Sigma, \delta, I, F)$ be the NFA represented by the diagram in Figure 2.2, where the set of states is $Q = \{0, 1, 2\}$, the alphabet is $\Sigma = \{\sigma, \tau\}$, the set of initial states is $I = \{0\}$, the set of final states is $F = \{2\}$, and the transition function is defined by

$$\begin{aligned} &\delta(0, \sigma) = \{0, 2\}, && \delta(1, \sigma) = \{1, 2\}, && \delta(2, \sigma) = 2, \\ &\delta(0, \tau) = 1, && \delta(1, \tau) = 2, && \delta(2, \tau) = 1. \end{aligned}$$

As we did for DFA, we can extend the transition function $\delta$ to $\delta \subseteq Q \times \Sigma^* \times Q$. This extension is recursively defined by

$$\begin{aligned} \delta(p, \varepsilon) &= \{p\}, \\ \delta(p, \sigma w) &= \delta(\delta(p, \sigma), w), \text{ where } w \in \Sigma^*, \sigma \in \Sigma. \end{aligned}$$

Since $\delta(p, w)$ can be a set, we need to extend $\delta$ to $\delta \subseteq 2^Q \times \Sigma^* \times Q$. We can do this by

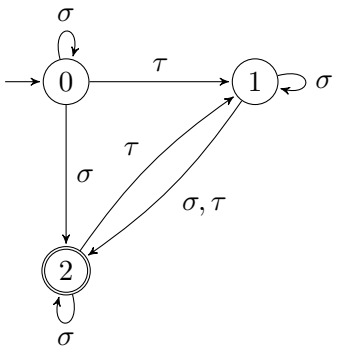



Figure 2.2: Diagram of the NFA $\dot{a}_{2.4}$.

$$\delta(P, w) = \bigcup_{p \in P} \delta(p, w),$$

where $P \subseteq Q$.

Let $\dot{a}$ be a NFA and $(p, \sigma, q)$ a transition of $\dot{a}$. As for DFA, we say that $\sigma$ is the *label of the transition*, and we say that $p$ has an *outgoing* transition (with label $\sigma$). A *path* of $\dot{a}$ is a finite sequence $(p_0, \sigma_1, p_1, \ldots, \sigma_\ell, p_\ell)$, for some nonnegative integer $\ell$, such that each triplet $(p_{i-1}, \sigma_i, p_i)$, where $i \geq 1$, is a transition of $\dot{a}$. We naturally define $\sigma_1 \cdots \sigma_\ell$ as the *label of the path*. An *accepting path* is a path where $p_0$ is the initial state and $p_\ell$ is a final state. The *language accepted* by $\dot{a}$ is the set of the labels of all the accepting paths of $\dot{a}$ and is denoted by $\mathcal{L}(\dot{a})$. The NFA $\dot{a}$ is called *trim* if every state is accessible and coaccessible. Finally, we say that $\dot{a}$ *accepts* $w$ if there is an accepting path labeled by $w$, that is, if $p_\ell \in \delta(p_0, w)$, where $p_0 \in I$ and $p_\ell \in F$.

**Definition 2.5.** Given a NFA $\dot{a}$ and a state $q \in Q$, the *right language* of $q$ is $\mathcal{L}_q(\dot{a}) = \{w \in \Sigma^* \mid \delta(q, w) \cap F \neq \emptyset\}$.

We can see the language $\mathcal{L}(\dot{a})$ of a NFA $\dot{a} = (Q, \Sigma, \delta, I, F)$ as the union

$$\bigcup_{q \in I} \mathcal{L}_q(\dot{a}).$$

### 2.2.3 Equivalence Between DFAs and NFAs

We say that two finite automata $\dot{a}_1$ and $\dot{a}_2$ are *equivalent* if and only if $\mathcal{L}(\dot{a}_1) = \mathcal{L}(\dot{a}_2)$. It is clear that DFAs are a special case of NFAs, where for every state, no two outgoing transitions have the same label, and the set of initial states is a singleton. On the other hand, every NFA can be converted into an equivalent DFA, as we will see later in this subsection.

Therefore, the class of languages accepted by DFAs is the same as the class of languages accepted by NFAs. This class is exactly the class of regular languages. Proof of this fact and

that the class of languages accepted by finite automata are closed under union, concatenation and Kleene star can be found in [17].

Every NFA can be simulated by a DFA, using the *subset construction*.

**Definition 2.6** (Subset Construction)**.** Let $\dot{a} = (Q, \Sigma, \delta, I, F)$ be an NFA. A DFA that simulates $\dot{a}$ is defined by $\mathcal{D}(\dot{a}) = \left(Q^{\mathcal{D}}, \Sigma, \delta^{\mathcal{D}}, i^{\mathcal{D}}, F^{\mathcal{D}}\right)$, where

- $Q^{\mathcal{D}} = \{P_i \mid P_i = \delta(I, w) \wedge w \in \Sigma^*\} \subseteq 2^Q$;
- $\delta^{\mathcal{D}}(P_i, \sigma) = \delta(P_i, \sigma)$, where $\sigma \in \Sigma$;
- $i^{\mathcal{D}} = I$;
- $F^{\mathcal{D}} = \left\{P_i \in Q^{\mathcal{D}} \mid P_i \cap F \neq \emptyset\right\}$.

In the worst case, given a NFA $\dot{a}$ with set of states $Q$, if $Q^{\mathcal{D}} = 2^Q$, then $\mathcal{D}(\dot{a})$ has $2^{|Q|}$ states, thus being exponentially larger than $\dot{a}$. We now give an example of this using the Lupanov's family of automata [13].

**Example 2.7** (Lupanov)**.** Let $\dot{a}_{2.7} = (Q, \Sigma, \delta, I, F)$ be the NFA represented by the diagram in Figure 2.3. We call this NFA *Lupanov's minimal 3-state NFA*. The minimal DFA that simulates $\dot{a}_{2.7}$ has 8 states and is represented by the diagram in Figure 2.4.

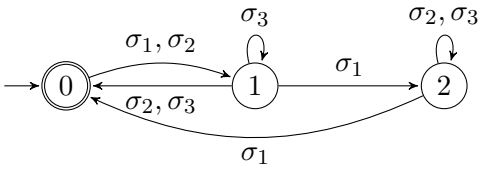



Figure 2.3: Diagram of the Lupanov's NFA $\dot{a}_{2.7}$.

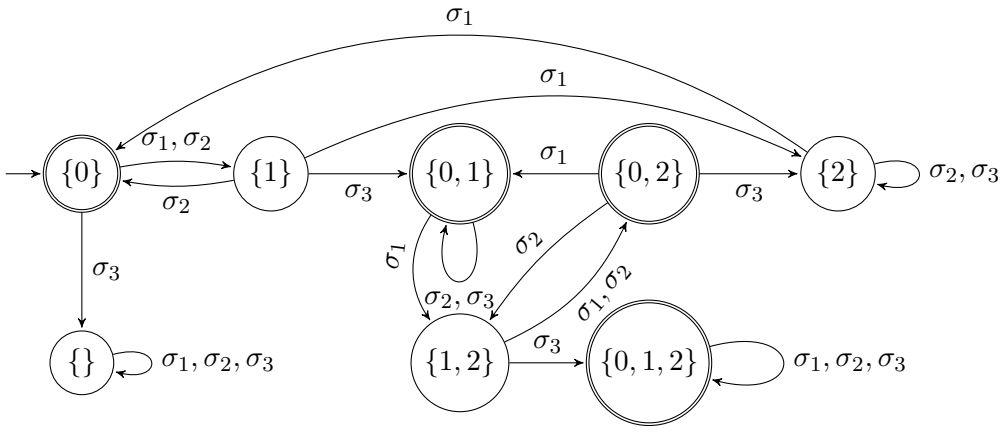



Figure 2.4: Diagram of the minimal DFA that simulates $\dot{a}_{2.7}$.

A DFA is called *minimal* if there is no other equivalent DFA with fewer states. For every DFA, there is an equivalent minimal DFA and the minimal DFA is unique up to renaming its states. Minimization algorithms can be found in [22], [16] and [5].

## 2.3 Regular Expressions

*Regular expressions* are another model of defining regular languages that could be more compact than the model of finite automata.

**Definition 2.8.** Let $\Sigma$ be an alphabet, and let $\varepsilon$ be the empty word over $\Sigma$. An ordinary *regular expression* (over $\Sigma$) (*RE* for short) is either $\emptyset$ or an expression that can be defined by the following grammar

$$r \to \varepsilon \mid \sigma \in \Sigma \mid (r + r) \mid (r \cdot r) \mid (r^*),$$

where the $*$ operator has precedence over both $+$ and $\cdot$ operators, and $\cdot$ has higher precedence than $+$. Thus, we can omit parenthesis. The operator $\cdot$ (concatenation) is often omitted. The *language* associated with a regular expression $r$, defined by $\mathcal{L}(r)$, is recursively defined by

$$\begin{aligned} \mathcal{L}(\emptyset) &= \emptyset, & \mathcal{L}(\varepsilon) &= \{\varepsilon\}, & \mathcal{L}(\sigma) &= \{\sigma\}, \\ \mathcal{L}(r_1 + r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2), & \mathcal{L}(r_1 r_2) &= \mathcal{L}(r_1)\mathcal{L}(r_2), & \mathcal{L}(r_1^*) &= \mathcal{L}(r_1)^*. \end{aligned}$$

Moreover, the set of *all regular languages* is denoted by *RE*.

**Example 2.9.** The language $\mathcal{L}(\dot{a}_{2.2})$ can be defined by the regular expression

$$r = (\sigma + \tau\,(\sigma + \tau))\,(\sigma + \tau\,(\sigma + \tau))^*.$$

Given a finite set $S \subseteq RE$ of REs, the *language* of $S$ is defined by $\mathcal{L}(S) = \bigcup_{r \in S} \mathcal{L}(r) = \mathcal{L}\left(\sum_{r \in S} r\right)$, that is, the language of a set of regular expressions is the same as the language of the disjunction of the expressions in the set.

**Definition 2.10.** Let $r_1$ and $r_2$ be two REs over $\Sigma$. Then $r_1$ is *equivalent* to $r_2$, denoted $r_1 \sim r_2$, if $\mathcal{L}(r_1) = \mathcal{L}(r_2)$.

Let $r_1$ and $r_2$ be two REs over $\Sigma$. We write $r_1 \equiv r_2$ when $r_1$ and $r_2$ are syntactically identical.

It is known that regular expressions form a Kleene Algebra [19]. In particular, we have the unrolling property.

*Remark* 2.11 (Unrolling). For any RE $r$ over $\Sigma$, we have that $r^* \sim rr^* + \varepsilon$. This is called *unrolling* the regular expression $r$.

**Definition 2.12.** Let $r$ be a regular expression over $\Sigma$.

1. The *number of concatenations* $|r|_\cdot$ of $r$ is the number of symbols $\cdot$ in $r$.
2. The *number of disjunctions* $|r|_+$ of $r$ is the number of symbols $+$ in $r$.
3. The *number of Kleene stars* $|r|_*$ of $r$ is the number of symbols $*$ in $r$.

**Example 2.13.** Consider the expression $r = (\sigma + \tau(\sigma + \tau))(\sigma + \tau(\sigma + \tau))^*$ in Example 2.9. Then $|r|_\cdot = 3$, $|r|_+ = 4$ and $|r|_* = 1$.

**Definition 2.14.** Let $r$ be a regular expression over $\Sigma$.

1. The *size* $|r|$ of $r$ is the number of symbols in $r$ (disregarding parenthesis).
2. The *alphabetic size* $|r|_\Sigma$ of $r$ is the number of letters in $r$.

**Example 2.15.** Consider the expression $r = (\sigma + \tau(\sigma + \tau))(\sigma + \tau(\sigma + \tau))^*$ in Example 2.9. Then $|r| = 16$ and $|r|_\Sigma = 8$.

**Definition 2.16.** Let $r$ be a regular expression over $\Sigma$. The *constant part* of $r$, denoted by $\epsilon(r)$, is defined as follows:

$$\epsilon(r) = \begin{cases} \varepsilon, & \text{if } \varepsilon \in \mathcal{L}(r); \\ \emptyset, & \text{otherwise.} \end{cases}$$

If $\varepsilon = \epsilon(r)$ we also say that $r$ has the *empty word property* (e.w.p.). The constant part of $r$ can also be recursively defined by

$$\begin{aligned} \epsilon(\emptyset) &= \emptyset, \\ \epsilon(\varepsilon) &= \varepsilon, \\ \epsilon(\sigma) &= \emptyset, \ \sigma \in \Sigma, \\ \epsilon(r_1{+}r_2) &= \begin{cases} \emptyset, & \text{if } \epsilon(r_1) = \epsilon(r_2) = \emptyset, \\ \varepsilon, & \text{otherwise,} \end{cases} \\ \epsilon(r_1 r_2) &= \begin{cases} \varepsilon, & \text{if } \epsilon(r_1) = \epsilon(r_2) = \varepsilon, \\ \emptyset, & \text{otherwise,} \end{cases} \\ \epsilon(r_1^*) &= \varepsilon. \end{aligned}$$

### 2.3.1 Derivatives

The *derivatives* of a regular expression were first introduced by Brzozowski in [6]. In the same article, the author presents a method for obtaining a DFA from a given regular expression using derivatives.

**Definition 2.17** (Derivative)**.** Let $r$ be a regular expression over $\Sigma$. The *derivative* of $r$ w.r.t. a symbol $\sigma \in \Sigma$, denoted by $d_\sigma(r)$, is the regular expression recursively defined by

$$\begin{aligned} d_\sigma(\emptyset) &= d_\sigma(\varepsilon) = \emptyset, \\ d_\sigma(\sigma') &= \begin{cases} \varepsilon, & \text{if } \sigma' = \sigma, \\ \emptyset, & \text{otherwise,} \end{cases} \\ d_\sigma(r_1 + r_2) &= d_\sigma(r_1) + d_\sigma(r_2) \\ d_\sigma(r_1 r_2) &= \begin{cases} d_\sigma(r_1) r_2 + d_\sigma(r_2), & \text{if } \epsilon(r_1) = \varepsilon, \\ d_\sigma(r_1) r_2, & \text{otherwise,} \end{cases} \\ d_\sigma(r_1^*) &= d_\sigma(r_1) r_1^*, \end{aligned}$$

where $\sigma' \in \Sigma$ and $r_1, r_2$ are also REs over $\Sigma$.

**Example 2.18.** Consider the regular expression $r = (\sigma\tau + \tau)^* \tau$ over $\Sigma = \{\sigma, \tau\}$. Then the derivative of $r$ w.r.t. $\sigma$ is

$$\begin{aligned} d_\sigma((\sigma\tau + \tau)^* \tau) &= d_\sigma((\sigma\tau + \tau)^*)\tau + d_\sigma(\tau) \\ &= d_\sigma(\sigma\tau + \tau)\,(\sigma\tau + \tau)^* \tau + \emptyset \\ &= (d_\sigma(\sigma\tau) + d_\sigma(\tau))\,(\sigma\tau + \tau)^* \tau + \emptyset \\ &= (\tau + \emptyset)\,(\sigma\tau + \tau)^* \tau + \emptyset \\ &\sim \tau\,(\sigma\tau + \tau)^* \tau. \end{aligned}$$

The definition of derivative of a regular expression $r$ can be extended to a word $w \in \Sigma^*$ on the structure of $w$ by

$$\begin{aligned} d_\varepsilon(r) &= r, \\ d_{w\sigma}(r) &= d_\sigma(d_w(r)). \end{aligned}$$

The language of the derivative of a regular expression w.r.t. a symbol $\sigma$ is the language $\mathcal{L}(d_\sigma(r)) = \{w \mid \sigma w \in \mathcal{L}(r)\}$. We have that $w \in \mathcal{L}(r)$ if and only if $\epsilon(d_w(r)) = \varepsilon$, that is, if and only if $\varepsilon \in \mathcal{L}(d_w(r))$.

**Example 2.19.** Let $r = (\sigma\tau + \tau)^* \tau$ be the regular expression in Example 2.18. Then we have that $\tau\tau \in \mathcal{L}((\sigma\tau + \tau)^* \tau)$, since $\varepsilon \in \mathcal{L}(d_{\tau\tau}((\sigma\tau + \tau)^* \tau))$.

Brzozowski proved that every regular expression has a finite number of distinct derivatives, considering the set of regular expressions module ACI-equivalences and $r\varepsilon \sim \varepsilon r \sim r$, for every RE $r$ over $\Sigma$.

### 2.3.2 Partial Derivatives

The notion of *partial derivative* of a regular expression w.r.t. a symbol was first introduced by Antimirov [2]. In the same article, the author presents a method for obtaining an NFA (instead

of a DFA like Brzozowski) from a given regular expression using partial derivatives.

We can extend the concatenation operation from regular expressions to sets of regular expressions. Let $\cdot : 2^{RE} \times RE \to 2^{RE}$ be such extension. The operation is recursively defined by

$$\begin{aligned} S \cdot \emptyset &= \emptyset, \\ S \cdot \varepsilon &= S, \\ \emptyset \cdot r_2 &= \emptyset, \\ \{\varepsilon\} \cdot r_2 &= \{r_2\}, \\ \{r_1\} \cdot r_2 &= \{r_1 r_2\}, \\ (S \cup S') \cdot r_2 &= (S \cdot r_2) \cup (S' \cdot r_2), \end{aligned}$$

where $S, S' \subseteq 2^{RE}$, $r_1 \in RE \setminus \{\emptyset\}$ and $r_2 \in RE \setminus \{\emptyset, \varepsilon\}$. We will often omit the $\cdot$ operator.

**Definition 2.20** (Partial Derivatives)**.** Let $r$ be a regular expression over $\Sigma$. The set of *partial derivatives* of $r$ w.r.t. a symbol $\sigma \in \Sigma$, denoted $\partial_\sigma(r)$, is defined recursively on the structure of $r$ as follows:

$$\begin{aligned} \partial_\sigma(\emptyset) &= \partial_\sigma(\varepsilon) = \emptyset, \\ \partial_\sigma(\sigma') &= \begin{cases} \{\varepsilon\}, & \text{if } \sigma' = \sigma, \\ \emptyset, & \text{otherwise,} \end{cases} \\ \partial_\sigma(r_1 + r_2) &= \partial_\sigma(r_1) \cup \partial_\sigma(r_2), \\ \partial_\sigma(r_1 r_2) &= \begin{cases} \partial_\sigma(r_1) r_2 \cup \partial_\sigma(r_2), & \text{if } \epsilon(r_1) = \varepsilon, \\ \partial_\sigma(r_1) r_2, & \text{otherwise,} \end{cases} \\ \partial_\sigma(r_1^*) &= \partial_\sigma(r_1) r_1^*, \end{aligned}$$

where $r_2$ is also a regular expression over $\Sigma$.

**Example 2.21.** Consider the expression $r = (\sigma\tau + \tau)^* \tau$ in Example 2.18. The set of partial derivatives of $r$ w.r.t. $\sigma$ is

$$\begin{aligned} \partial_\sigma((\sigma\tau + \tau)^* \tau) &= \partial_\sigma((\sigma\tau + \tau)^*)\tau \cup \epsilon((\sigma\tau + \tau)^*)\partial_\sigma(\tau) \\ &= \partial_\sigma(\sigma\tau + \tau)\,(\sigma\tau + \tau)^* \tau \cup \varepsilon \partial_\sigma(\tau) \\ &= (\partial_\sigma(\sigma\tau) \cup \partial_\sigma(\tau))\,(\sigma\tau + \tau)^* \tau \cup \varepsilon\emptyset \\ &= (\partial_\sigma(\sigma)\tau \cup \epsilon(\sigma)\partial_\sigma(\tau) \cup \emptyset)\,(\sigma\tau + \tau)^* \tau \\ &= (\{\varepsilon\}\tau \cup \emptyset\emptyset)\,(\sigma\tau + \tau)^* \tau \\ &= \{\tau\}\,(\sigma\tau + \tau)^* \tau \\ &= \{\tau\,(\sigma\tau + \tau)^* \tau\} \end{aligned}$$

The set of partial derivatives of a regular expression $r$ over $\Sigma$ w.r.t. a word $w \in \Sigma^*$ can be

inductively defined by

$$\partial_\varepsilon(r) = \{r\},$$
$$\partial_{w\sigma}(r) = \partial_\sigma(\partial_w(r)),$$

where given a set $S \subseteq RE$, $\partial_\sigma(S) = \bigcup_{r\in S} \partial_\sigma(r)$.

The *language* induced by partial derivatives is defined by $\mathcal{L}(\partial_\sigma(r)) = \{w \mid \sigma w \in \mathcal{L}(r)\}$.

*Remark* 2.22. Let $r$ be a regular expression over $\Sigma$ and $w \in \Sigma^*$. Then $\mathcal{L}(d_w(r)) = \mathcal{L}(\partial_w(r))$.

**Definition 2.23.** The set of *all partial derivatives* of a regular expression $r$ over $\Sigma$ w.r.t. words is denoted by $\mathcal{PD}(r)$ and is such that

$$\mathcal{PD}(r) = \bigcup_{w\in\Sigma^*} \partial_w(r).$$

**Example 2.24.** Consider the expression $r = (\sigma\tau + \tau)^* \tau$ in Example 2.18. The set of all partial derivatives of $r$ w.r.t. words is

$$\mathcal{PD}((\sigma\tau + \tau)^* \tau) = \{(\sigma\tau + \tau)^* \tau, \tau (\sigma\tau + \tau)^* \tau, \varepsilon\}.$$

The *set of all partial derivatives excluding the derivative by the empty word* of a regular expression $r$ over $\Sigma$ is denoted by $\partial^+(r)$, that is,

$$\partial^+(r) = \bigcup_{w\in\Sigma^+} \partial_w(r).$$

*Remark* 2.25. For every regular expression $r$ over $\Sigma$, $\mathcal{PD}(r) = \{r\} \cup \partial^+(r)$.

Let us take the regular expression $r = (\sigma\tau + \tau)^* \tau$ from Example 2.18. Since $r \in \partial^+(r)$, we have that $\partial^+((\sigma\tau + \tau)^* \tau) = \mathcal{PD}((\sigma\tau + \tau)^* \tau) = \{(\sigma\tau + \tau)^* \tau, \tau (\sigma\tau + \tau)^* \tau, \varepsilon\}$.

**Proposition 2.26.** *The set of all partial derivatives excluding the derivative by empty word, $\partial^+$, satisfies the following equalities*

$$\partial^+(\emptyset) = \partial^+(\varepsilon) = \emptyset, \tag{2.1}$$
$$\partial^+(\sigma) = \{\varepsilon\}, \ (\sigma \in \Sigma), \tag{2.2}$$
$$\partial^+(r_1 + r_2) = \partial^+(r_1) \cup \partial^+(r_2), \tag{2.3}$$
$$\partial^+(r_1 r_2) = \partial^+(r_1) r_2 \cup \partial^+(r_2), \tag{2.4}$$
$$\partial^+(r^*) = \partial^+(r) r^*. \tag{2.5}$$

*Proof.* The proof follows by induction on the structure of $r$. It is clear that (2.1) and (2.2) hold. In the remaining cases, to prove that an inclusion $\partial^+(r) \subseteq E$ holds, for some RE $r$, we show by induction on the length of $w$ that $\partial_w(r) \subseteq E$, for every $w \in \Sigma^+$. Thus, we only need to prove $\partial_\sigma(r) \subseteq E$ and $\partial_{w\sigma}(r) \subseteq E$, for $\sigma \in \Sigma$ and $w \in \Sigma^*$.

Note that for any RE $r$ over $\Sigma$ and $\sigma \in \Sigma$, one has $\partial_\sigma(\partial^+(r)) \subseteq \partial^+(r)$ and $\partial_\sigma(r) \subseteq \partial^+(r)$.

Now suppose that the claim is true for some REs $r_1$ and $r_2$. We need to consider three induction cases.

1. **Case** $r_1 + r_2$.

   By definition, we have that $\partial_\sigma(r_1 + r_2) = \partial_\sigma(r_1) \cup \partial_\sigma(r_2)$, therefore one has $\partial_\sigma(r_1 + r_2) \subseteq \partial^+(r_1) \cup \partial^+(r_2)$. Also by definition, we have that $\partial_{w\sigma}(r_1 + r_2) = \partial_\sigma(\partial_w(r_1 + r_2))$ and $\partial_\sigma(\partial_w(r_1 + r_2)) \underset{\text{I.H.}}{\subseteq} \partial_\sigma(\partial^+(r_1) \cup \partial^+(r_2))$. Finally,

   $$\partial_\sigma(\partial^+(r_1) \cup \partial^+(r_2)) = \partial_\sigma(\partial^+(r_1)) \cup \partial_\sigma(\partial^+(r_2)) \subseteq \partial^+(r_1) \cup \partial^+(r_2).$$

   Therefore, $\partial_{w\sigma}(r_1 + r_2) \subseteq \partial^+(r_1) \cup \partial^+(r_2)$.

   Similarly, one proves that for every $w \in \Sigma^+$, $\partial_w(r_1) \subseteq \partial^+(r_1 + r_2)$ and $\partial_w(r_2) \subseteq \partial^+(r_1 + r_2)$.

2. **Case** $r_1^*$.

   By definition, we have that $\partial_\sigma(r_1^*) = \partial_\sigma(r_1)r_1^*$, and since $\partial_\sigma(r_1) \subseteq \partial^+(r_1)$, we have that $\partial_\sigma(r_1)r_1^* \subseteq \partial^+(r_1)r_1^*$, that is, $\partial_\sigma(r_1^*) \subseteq \partial^+(r_1)r_1^*$. Also by definition, $\partial_{w\sigma}(r_1^*) = \partial_\sigma(\partial_w(r_1^*)) \underset{\text{I.H.}}{\subseteq} \partial_\sigma(\partial^+(r_1)r_1^*) = \partial_\sigma(\partial^+(r_1))r_1^*$.

   But we also have that

   $$\begin{aligned}\partial_\sigma\Big(\partial^+(r_1)\Big)r_1^* \cup \partial_\sigma(r_1^*) &\subseteq \partial^+(r_1)r_1^* \cup \partial_\sigma(r_1^*) \\ &= \partial^+(r_1)r_1^* \cup \partial_\sigma(r_1)r_1^* \\ &\subseteq \partial^+(r_1)r_1^* \cup \partial^+(r_1)r_1^* \\ &= \partial^+(r_1)r_1^*.\end{aligned}$$

   Therefore, $\partial_{w\sigma}(r_1^*) \subseteq \partial^+(r_1)r_1^*$. Conversely, first note that $\partial_\sigma(r_1)r_1^* = \partial_\sigma(r_1^*) \subseteq \partial^+(r_1^*)$.

   On the other hand,

   $$\begin{aligned}\partial_{w\sigma}(r_1)r_1^* &= \partial_\sigma(\partial_w(r_1))r_1^* \\ &\subseteq \partial_\sigma(\partial_w(r_1))r_1^* \cup \epsilon(\partial_w(r_1))\partial_\sigma(r_1^*) \\ &= \partial_\sigma(\partial_w(r_1)r_1^*) \underset{\text{I.H.}}{\subseteq} \partial_\sigma\Big(\partial^+(r_1^*)\Big) \subseteq \partial^+(r_1^*).\end{aligned}$$

   Therefore, $\partial_{w\sigma}(r_1)r_1^* \subseteq \partial^+(r_1^*)$.

3. **Case** $r_1 r_2$.

   By definition, we have

   $$\begin{aligned}\partial_\sigma(r_1 r_2) &= \partial_\sigma(r_1)r_2 \cup \epsilon(r_1)\partial_\sigma(r_2) \\ &\subseteq \partial_\sigma(r_1)r_2 \cup \partial_\sigma(r_2) \\ &\subseteq \partial^+(r_1)r_2 \cup \partial^+(r_2).\end{aligned}$$

Also by definition,

$$\begin{aligned}
\partial_{w\sigma}(r_1 r_2) &= \partial_\sigma(\partial_w(r_1 r_2)) \\
&\underset{\text{I.H.}}{\subseteq} \partial_\sigma\Big(\partial^+(r_1)r_2 \cup \partial^+(r_2)\Big) \\
&= \partial_\sigma\Big(\partial^+(r_1)r_2\Big) \cup \partial_\sigma\Big(\partial^+(r_2)\Big) \\
&= \partial_\sigma\Big(\partial^+(r_1)\Big)r_2 \cup \epsilon\Big(\partial^+(r_1)\Big)\partial_\sigma(r_2) \cup \partial_\sigma\Big(\partial^+(r_2)\Big) \\
&\subseteq \partial_\sigma\Big(\partial^+(r_1)\Big)r_2 \cup \partial_\sigma(r_2) \cup \partial_\sigma\Big(\partial^+(r_2)\Big) \\
&\subseteq \partial^+(r_1)r_2 \cup \partial^+(r_2) \cup \partial^+(r_2) \\
&= \partial^+(r_1)r_2 \cup \partial^+(r_2).
\end{aligned}$$

On the other hand, we have that $\partial_\sigma(r_1)r_2 \subseteq \partial_\sigma(r_1 r_2) \subseteq \partial^+(r_1 r_2)$ and $\partial_{w\sigma}(r_1)r_2 = \partial_\sigma(\partial_w(r_1))r_2 \subseteq \partial_\sigma(\partial_w(r_1)r_2) \subseteq \partial_\sigma(\partial_w(r_1 r_2)) \underset{\text{I.H.}}{\subseteq} \partial_\sigma(\partial^+(r_1 r_2)) \subseteq \partial^+(r_1 r_2)$.

Finally, if $\epsilon(r_1) = \varepsilon$, then $\partial_\sigma(r_2) \subseteq \partial_\sigma(r_1 r_2)$ and $\partial_{w\sigma}(r_2) = \partial_\sigma(\partial_w(r_2)) \subseteq \partial_\sigma(\partial_w(r_1 r_2)) = \partial_{w\sigma}(r_1 r_2)$. Thus, $\partial_w(r_2) \subseteq \partial_w(r_1 r_2)$ for all $w \in \Sigma^+$, and therefore $\partial^+(r_2) \subseteq \partial^+(r_1 r_2)$.

On the other hand, if $\epsilon(r_1) = \emptyset$, we have that $\partial^+(r_1) \neq \emptyset$. We also have that there is some $r_0 \in \partial^+(r_1)$ such that $\epsilon(r_0) = \varepsilon$, since $w \in \mathcal{L}(r)$ if and only if $\epsilon(d_w(r)) = \varepsilon$ and $\mathcal{L}(d_w(r)) = \mathcal{L}(\partial_w(r))$. This implies that $\partial_w(r_2) \subseteq \partial_w(r_0 r_2)$ for all $w \in \Sigma^+$. We have shown that $\partial^+(r_1)r_2 \subseteq \partial^+(r_1 r_2)$. In particular, $r_0 r_2 \in \partial^+(r_1 r_2)$. From these two facts we conclude that $\partial_w(r_2) \subseteq \partial_w(r_0 r_2) \subseteq \partial_w(\partial^+(r_1 r_2)) \subseteq \partial^+(r_1 r_2)$.

This concludes the proof. □

Antimirov [2] proved that the following inequalities hold

$$\left|\partial^+(r)\right| \leq |r|_\Sigma, \tag{2.6}$$

$$|\mathcal{PD}(r)| \leq |r|_\Sigma + 1. \tag{2.7}$$

Proposition 2.26 cam be used to prove the above inequalities via induction on the structure of $r$. We can also conclude from these inequalities that the set of partial derivatives is finite.

### 2.3.3 Linear Form

The notion of *linear form* was introduced by Antimirov [2]. The linear form of a regular expression allows one to compute the set of all partial derivatives w.r.t every symbol in the alphabet in a single step.

**Definition 2.27.** Let $r$ be a regular expression over $\Sigma$. The *linear form* of $r$, denoted $\mathrm{lf}(r)$ is

defined recursively on the structure of $r$ as follows

$$\begin{aligned}
\mathrm{lf}(\emptyset) &= \emptyset;\\
\mathrm{lf}(\sigma) &= \{(\sigma, \varepsilon)\}, \text{ for } \sigma \in \Sigma;\\
\mathrm{lf}(r_1 + r_2) &= \mathrm{lf}(r_1) \cup \mathrm{lf}(r_2);\\
\mathrm{lf}(r_1 \cdot r_2) &= \begin{cases} \mathrm{lf}(r_1) \cdot r_2, & \text{if } \epsilon(r_1) = \emptyset;\\ \mathrm{lf}(r_1) \cdot r_2 \cup \mathrm{lf}(r_2), & \text{otherwise;} \end{cases}\\
\mathrm{lf}(r_1^*) &= \mathrm{lf}(r_1) \cdot r_1^*;
\end{aligned}$$

where $r_1$ and $r_2$ are also REs over $\Sigma$, $\mathrm{lf}(r_1) \cdot r_2 = \{(\sigma, r \cdot r_2) \mid (\sigma, r) \in \mathrm{lf}(r_1)\}$ and $\sigma \in \Sigma$.

*Remark* 2.28. Note that $\mathrm{lf}(r_1 \cdot r_2)$ can also be defined as $\mathrm{lf}(r_1 \cdot r_2) = \mathrm{lf}(r_1) r_2 \cup \epsilon(r_1)\mathrm{lf}(r_2)$.

**Example 2.29.** Consider the expression $r = (\sigma\tau + \tau)^* \tau$ in Example 2.18. The linear form of $r$ is

$$\begin{aligned}
\mathrm{lf}(r) &= \mathrm{lf}((\sigma\tau + \tau)^* \tau)\\
&= \mathrm{lf}((\sigma\tau + \tau)^*)\tau \cup \mathrm{lf}(\tau)\\
&= \mathrm{lf}(\sigma\tau + \tau)\,(\sigma\tau + \tau)^* \tau \cup \{(\tau, \varepsilon)\}\\
&= (\mathrm{lf}(\sigma\tau) \cup \mathrm{lf}(\tau))\,(\sigma\tau + \tau)^* \tau \cup \{(\tau, \varepsilon)\}\\
&= (\mathrm{lf}(\sigma)\tau \cup \{(\tau, \varepsilon)\})\,(\sigma\tau + \tau)^* \tau \cup \{(\tau, \varepsilon)\}\\
&= (\{(\sigma, \varepsilon)\}\,\tau \cup \{(\tau, \varepsilon)\})\,(\sigma\tau + \tau)^* \tau \cup \{(\tau, \varepsilon)\}\\
&= \{(\sigma, \tau), (\tau, \varepsilon)\}\,(\sigma\tau + \tau)^* \tau \cup \{(\tau, \varepsilon)\}\\
&= \{(\sigma, \tau\,(\sigma\tau + \tau)^* \tau), (\tau, (\sigma\tau + \tau)^* \tau), (\tau, \varepsilon)\}.
\end{aligned}$$

**Proposition 2.30.** *For any RE $r$, $lf(r)$ is such that*

$$r \sim \bigcup_{(\sigma, r') \in lf(r)} \sigma r' \cup \epsilon(r).$$

*Proof.* The proof follows by induction on the structure of $r$. The base cases $\emptyset$ and $\sigma$ are trivial and left to the reader.

Now let us assume that the claim is true for $r_1$ and $r_2$ over $\Sigma$.

1. **Case** $r_1 + r_2$.

   We have by definition that $\mathrm{lf}(r_1 + r_2) = \mathrm{lf}(r_1) \cup \mathrm{lf}(r_2)$. By induction hypothesis, we have that

$$r_1 \sim \bigcup_{(\sigma_1, r_1') \in \mathrm{lf}(r_1)} \sigma_1 r_1' \cup \epsilon(r_1) \text{ and}$$

$$r_2 \sim \bigcup_{(\sigma_2, r_2') \in \mathrm{lf}(r_2)} \sigma_2 r_2' \cup \epsilon(r_2).$$

Therefore,

$$\begin{aligned} r_1 + r_2 &\sim \bigcup_{(\sigma_1, r_1') \in \mathrm{lf}(r_1)} \sigma_1 r_1' \cup \epsilon(r_1) \cup \bigcup_{(\sigma_2, r_2') \in \mathrm{lf}(r_2)} \sigma_2 r_2' \cup \epsilon(r_2) \\ &\sim \bigcup_{(\sigma_1, r_1') \in \mathrm{lf}(r_1)} \sigma_1 r_1' \cup \bigcup_{(\sigma_2, r_2') \in \mathrm{lf}(r_2)} \sigma_2 r_2' \cup \epsilon(r_1) \cup \epsilon(r_2) \\ &\sim \bigcup_{(\sigma_1, r_1') \in \mathrm{lf}(r_1)} \sigma_1 r_1' \cup \bigcup_{(\sigma_2, r_2') \in \mathrm{lf}(r_2)} \sigma_2 r_2' \cup \epsilon(r_1 + r_2) \\ &\sim \bigcup_{(\sigma, r) \in \mathrm{lf}(r_1 + r_2)} \sigma r \cup \epsilon(r_1 + r_2). \end{aligned}$$

2. **Case** $r_1 r_2$.

   Note that from Remark 2.28 we have that $\mathrm{lf}(r_1 r_2) = \mathrm{lf}(r_1) r_2 \cup \epsilon(r_1)\mathrm{lf}(r_2)$. By induction hypothesis, we also have that

$$\begin{aligned} r_1 r_2 &\sim \left( \bigcup_{(\sigma_1, r_1') \in \mathrm{lf}(r_1)} \sigma_1 r_1' \cup \epsilon(r_1) \right) r_2 \\ &\sim \bigcup_{(\sigma_1, r_1') \in \mathrm{lf}(r_1)} \sigma_1 r_1' r_2 \cup \epsilon(r_1) \left( \bigcup_{(\sigma_2, r_2') \in \mathrm{lf}(r_2)} \sigma_2 r_2' \cup \epsilon(r_2) \right) \\ &\sim \bigcup_{(\sigma_1, r_1') \in \mathrm{lf}(r_1)} \sigma_1 r_1' r_2 \cup \epsilon(r_1) \bigcup_{(\sigma_2, r_2') \in \mathrm{lf}(r_2)} \sigma_2 r_2' \cup \epsilon(r_1)\epsilon(r_2) \\ &\sim \bigcup_{(\sigma_1, r_1') \in \mathrm{lf}(r_1)} \sigma_1 r_1' r_2 \cup \epsilon(r_1) \bigcup_{(\sigma_2, r_2') \in \mathrm{lf}(r_2)} \sigma_2 r_2' \cup \epsilon(r_1 r_2) \end{aligned}$$

   and since $\mathrm{lf}(r_1) r_2 \cup \epsilon(r_1)\mathrm{lf}(r_2) = \mathrm{lf}(r_1 r_2)$, we have that

$$r_1 r_2 \sim \bigcup_{(\sigma, r) \in \mathrm{lf}(r_1 r_2)} \sigma r \cup \epsilon(r_1 r_2).$$

3. **Case** $r_1^*$.

   From Remark 2.11, we have that $r_1^* \sim r_1 r_1^* + \varepsilon$, and we can suppose that $\epsilon(r_1) = \emptyset$. By induction hypothesis, we have that

$$\begin{aligned} r_1^* &\sim r_1 r_1^* + \varepsilon \\ &\sim \bigcup_{(\sigma, r) \in \mathrm{lf}(r_1)} \sigma r r_1^* \cup \epsilon(r_1) r_1^* \cup \{\varepsilon\}. \end{aligned}$$

   But $\mathrm{lf}(r_1^*) = \mathrm{lf}(r_1) r_1^*$, $\epsilon(r_1) = \emptyset$ and $\epsilon(r_1^*) = \varepsilon$. Therefore,

$$r_1^* \sim \bigcup_{(\sigma, r) \in \mathrm{lf}(r_1^*)} \sigma r \cup \epsilon(r_1^*).$$

This concludes the proof. □

## 2.4 Equivalence Between REs and FAs

In this section, we present some methods for obtaining a regular expression from a given finite automaton and vice-versa. The first method, the state elimination method, provides us a way of obtaining a regular expression from a given finite automaton. The second method is the Thompson's method, which allows us to obtain an $\varepsilon$-NFA from a given RE. The third and forth methods, due to Brzozowski and Antimirov, respectively, provide a way of obtaining a DFA and a NFA from a given regular expression, respectively.

### 2.4.1 State Elimination Method

The method that we present here is adapted from the one presented by Hopcroft and Ullman in [17].

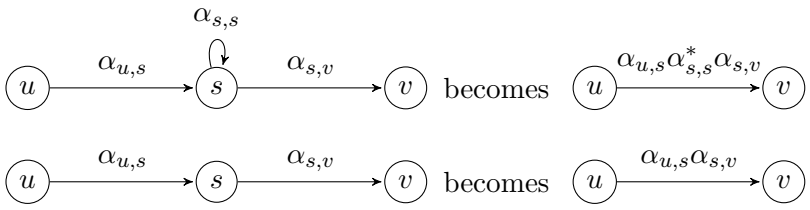
**Definition 2.31.** A *generalized finite automaton* is a quintuple

$$\dot{a} = (Q, \Sigma, \delta, I, F)$$

where $Q$, $I$, $F$, $\Sigma$ are exactly the same as those defined for FAs, and $\delta \subseteq Q \times RE \times Q$ is the finite set of transitions, where $RE$ is the set of REs.

The state elimination method works as follows

- If the FA has more than one initial state, $i_1, \ldots, i_n$, add a new initial state $i$ and $\varepsilon$ transitions from $i$ to $i_1, \ldots, i_n$;
- Remove all the states that are not accessible or coaccessible;
- If there are no final states left, return as result $\emptyset$ and terminate;
- If the initial state $s_0$ has in-degree not 0, add a new initial state $i$ and an $\varepsilon$ transition from $i$ to $s_0$;
- If the automaton has several final states, $f_1, \ldots, f_n$, add a new final state $f$ which will be the only final state, with $\varepsilon$ transitions from $f_1, \ldots, f_n$ to $f$;
- If the final state $f$ has out-degree not 0, add a new final state $f'$, which will be the only final state, and an $\varepsilon$ transition from $f$ to $f'$;
- Convert the FA into a generalized one, that is, replace the labels of the transitions with REs. This means that when a transition from a state to another has more than one symbol as its label, namely $\sigma_1, \ldots, \sigma_n$, the label of the transition becomes $\sum_{i=1}^{n} \sigma_i$;
- Delete the states, one by one, with exception to the initial and final states. To delete a state $s$, replace each transitions $(u, \alpha_{u,s}, s)$ and $(s, \alpha_{s,v}, v)$, with $u \neq s$ and $s \neq v$ by a new transiton from $u$ to $v$ which label is as follows




If a transition from $u$ to $v$ already exists, with label $\alpha_{u,v}$, replace it with $\left(\alpha_{u,v} + \alpha_{u,s}\alpha^*_{s,s}\alpha_{s,v}\right)$ or $(\alpha_{u,v} + \alpha_{u,s}\alpha_{s,v})$, respectively.

This method terminates when the automaton has only two states, the initial and final states. The regular expression equivalent to the automaton is the expression that labels the only transition from the initial state to the final state.

### 2.4.2 Thompson's Method

The Thompson's automaton was introduced by Thompson [27] and allows us to obtain a $\varepsilon$-NFA from a given RE. Here we will present the construction proposed by Sheng Yu [31], which allows the number of final states to be greater than 1.

**Definition 2.32.** Let $r$ be a regular expression over $\Sigma$. The Thompson's automaton is constructed recursively as follows

- **Case** $\emptyset$.

  The Thompson's automaton is given by $\dot{a}_\emptyset = (\{q\}, \Sigma, \delta, q, \emptyset)$, where $\delta(q, \tau) = \emptyset$, for each $\tau \in \Sigma \cup \{\varepsilon\}$.

- **Case** $\varepsilon$.

  The Thompson's automaton is given by $\dot{a}_\varepsilon = (\{q\}, \Sigma, \delta, q, \{q\})$, where $\delta(q, \tau) = \emptyset$, for each $\tau \in \Sigma \cup \{\varepsilon\}$.

- **Case** $\sigma$.

  The Thompson's automaton is given by $\dot{a}_\sigma = (\{q, f\}, \Sigma, \delta, q, \{f\})$, where $\delta(q, \sigma) = f$ and this is the only defined transition.

Now let $r_1$ and $r_2$ be two REs over $\Sigma$ with Thompson's automata $\dot{a}_{r_1} = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $\dot{a}_{r_2} = (Q_2, \Sigma, \delta_2, q_2, F_2)$, respectively.

- **Case** $r_1 r_2$.

  The Thompson's automaton is given by $\dot{a}_{r_1 r_2} = (Q_1 \cup Q_2, \Sigma, \delta, q_1, F_2)$, where $\mathcal{L}(\dot{a}_{r_1 r_2}) = \mathcal{L}(\dot{a}_{r_1})\mathcal{L}(\dot{a}_{r_2})$ and

$$\begin{aligned}
&\delta(s, \tau) = \delta_1(s, \tau), \text{ if } s \in Q_1 \text{ and } \tau \in \Sigma, \text{ or } s \in Q_1 \setminus F_1 \text{ and } \tau = \varepsilon,\\
&\delta(s, \varepsilon) = \delta_1(s, \varepsilon) \cup \{q_2\}, \text{ if } s \in F_1,\\
&\delta(s, \tau) = \delta_2(s, \sigma), \text{ if } s \in Q_2 \text{ and } \tau \in \Sigma \cup \{\varepsilon\}.
\end{aligned}$$

- **Case** $r_1 + r_2$.

  The Thompson's automaton is given by $\dot{a}_{r_1+r_2} = (Q_1 \cup Q_2 \cup \{q\}, \Sigma, \delta, q, F_1 \cup F_2)$, where $q \notin Q_1 \cup Q_2$, $\mathcal{L}(\dot{a}_{r_1+r_2}) = \mathcal{L}(\dot{a}_{r_1}) \cup \mathcal{L}(\dot{a}_{r_2})$ and

$$\begin{aligned}
&\delta(q, \varepsilon) = \{q_1, q_2\}, \\
&\delta(s, \tau) = \delta_1(s, \tau), \text{ if } s \in Q_1 \text{ and } \tau \in \Sigma \cup \{\varepsilon\}, \\
&\delta(s, \tau) = \delta_2(s, \tau), \text{ if } s \in Q_2 \text{ and } \tau \in \Sigma \cup \{\varepsilon\}.
\end{aligned}$$

- **Case** $r_1^*$.

  The Thompson's automaton is given by $\dot{a}_{r_1^*} = (Q_1 \cup \{q\}, \Sigma, \delta, q, F_1 \cup \{q\})$, where $q \notin Q_1$, $\mathcal{L}\left(\dot{a}_{r_1^*}\right) = \mathcal{L}(\dot{a}_{r_1})^*$ and

$$\begin{aligned}
&\delta(q, \varepsilon) = \{q_1\}, \\
&\delta(s, \varepsilon) = \delta_1(s, \varepsilon) \cup \{q_1\}, \text{ if } s \in F_1, \\
&\delta(s, \tau) = \delta_1(s, \tau), \text{ if } s \in Q_1 \text{ and } \tau \in \Sigma, \text{ or } s \in Q_1 \setminus F_1 \text{ and } \tau = \varepsilon.
\end{aligned}$$

The automaton constructed for $r$ accepts the language $\mathcal{L}(r)$.

### 2.4.3 Brzozowski's Method

Brzozowski [6] presented a method for obtaining a DFA equivalent to a given regular expression using derivatives. The method builds an automaton where the states are derivatives of some regular expression and the transitions from a state by a symbol are given by the derivative of the RE from that state w.r.t. that symbol. Remember that every regular expression has a finite number of distinct derivatives, considering the set of regular expressions module ACI-equivalences and the fact that $r\varepsilon \sim \varepsilon r \sim r$, for every RE $r$ over $\Sigma$, which we will denote by $D_{\mathrm{ACI}_+}(r)$. This fact is used in the method, in order to produce an automaton with a finite number of states.

**Definition 2.33.** Let $r$ be a regular expression over $\Sigma$. The *Brzozowski's automaton* for $r$ is the DFA $\dot{a} = (Q, \Sigma, \delta, i, F)$, where the set of states is $Q = D_{\mathrm{ACI}_+}(r)$, the initial state is $i = r$, the transition function is defined as $\delta(q, \sigma) = d_\sigma(q)$, for all $\sigma \in \Sigma$ and $q \in D_{\mathrm{ACI}_+}(r)$, and the set of final states is $F = \{q \in D_{\mathrm{ACI}_+}(r) \mid \epsilon(q) = \varepsilon\}$. The DFA $\dot{a}$ accepts the language $\mathcal{L}(r)$.

### 2.4.4 Antimirov's Method

Antimirov [2] presented a method for obtaining a NFA equivalent to a given regular expression $r$ over $\Sigma$ using partial derivatives. This automaton can be seen as a nondeterministic version of the Brzozowski's DFA. This NFA has at most $|r|_\Sigma + 1$ states, as proved by Antimirov.

**Definition 2.34.** Let $r$ be a regular expression over $\Sigma$. The partial derivative's automaton for $r$ is given by the NFA $\dot{a} = (Q, \Sigma, \delta, i, F)$, where the set of states is $Q = \mathcal{PD}(r)$, the initial state is $i = r$, the set of final states is $F = \{q \in \mathcal{PD}(r) \mid \epsilon(q) = \varepsilon\}$ and the transition function is defined as $\delta(q, \sigma) = \partial_\sigma(q)$. The NFA $\dot{a}$ accepts the language $\mathcal{L}(r)$.

Champarnaud and Ziadi [8] proved that Mirkin's prebases [21] and partial derivatives lead to identical NFAs. If we denote the Mirkin prebase of $r$ by $\mathcal{P}(r)$, since the set of states of the partial derivative's automaton is $Q = \mathcal{PD}(r)$, then we have that $\mathcal{PD}(r) = \mathcal{P}(r) \cup \{r\}$.

Since the linear form is an efficient way to compute the partial derivatives of a given RE $r$ over $\Sigma$, the transition function of the partial derivative's automaton of $r$ can be computed using the linear form, more specifically

$$\delta(r', \sigma) = \{r'' \mid (\sigma, r'') \in \mathrm{lf}(r')\}, \text{ for all } r' \in \mathcal{P}(r) \text{ and } \sigma \in \Sigma.$$

# Chapter 3

# Transducers

Transducers were first introduced by Elgot and Mezei [11]. Ginsberg [14] uses the term "$a$-transducer", where "$a$" emphasises the presence of accepting states. Before introducing transducers, we need to introduce relations over two alphabets [20].

## 3.1 Binary Relations Over Words

**Definition 3.1.** A *relation* $\rho$ over $\Sigma$ and $\Delta$ is a subset of $\Sigma^* \times \Delta^*$, that is,

$$\rho \subseteq \{(x, y) \mid x \in \Sigma^*, y \in \Delta^*\}.$$

**Example 3.2.** The *identity* relation over an alphabet $\Sigma$ is defined by

$$Id_\Sigma = \{(x, x) \mid x \in \Sigma^*\}.$$

Since relations are sets, we have defined for relations the boolean operations like union and intersection.

**Definition 3.3** (Concatenation, Union, Kleene Closure)**.** Let $\rho_1, \rho_2$ be two relations over $\Sigma$ and $\Delta$.

1. The *concatenation* of the two relations $\rho_1$ and $\rho_2$, denoted by $\rho_1 \cdot \rho_2$, is a subset of $\Sigma^* \times \Delta^*$ such that $\rho_1 \cdot \rho_2 = \{(xx', yy') \mid (x, y) \in \rho_1 \wedge (x', y') \in \rho_2\}$. The operator $\cdot$ is often omitted.
2. The *union* of the two relations $\rho_1$ and $\rho_2$, denoted by $\rho_1 \cup \rho_2$, is a subset of $\Sigma^* \times \Delta^*$ and is defined by $\rho_1 \cup \rho_2 = \{(w, w') \mid (w, w') \in \rho_1 \vee (w, w') \in \rho_2\}$.
3. The *Kleene closure* of the relation $\rho$, $\rho^*$, is a subset of $\Sigma^* \times \Delta^*$ such that $\rho^* = \bigcup_n \rho^n$, where $\rho^n$ is the result of concatenating $\rho$ with itself $n$ times.

Let $\rho$ be a relation over $\Sigma$ and $\Delta$ and let $(x_1, y_1), (x_2, y_2) \in \rho$. The *concatenation* of $(x_1, y_1)$ and $(x_2, y_2)$, denoted by the operator $\cdot$, is defined by $(x_1, y_1) \cdot (x_2, y_2) = (x_1 \cdot x_2, y_1 \cdot y_2)$. The operator $\cdot$ is often omitted. The *inverse* of $\rho$, denoted by $\rho^{-1}$, is defined by $\rho^{-1} = \{(y, x) \mid (x, y) \in \rho\}$. The *composition*, denoted by the operator $\circ$, of a relation $\rho_1$ over $\Sigma_1$ and $\Sigma_2$ and a relation $\rho_2$ over $\Sigma_2$ and $\Sigma_3$ is the relation $\rho_1 \circ \rho_2 = \{(x, z) \mid \exists y \in \Sigma_2 \, (x, y) \in \rho_2 \wedge (y, z) \in \rho_1\}$.

**Definition 3.4.** Let $\rho$ be a relation over $\Sigma$ and $\Delta$. We say that $\rho$ is *rational* if is a rational subset of $\Sigma^* \times \Delta^*$, that is, if it can be obtained by taking a finite number of finite subsets of $\Sigma^* \times \Delta^*$ and applying the union, concatenation and Kleene star operations a finite number of times.

## 3.2 Transducers

Rational relations can be defined by models. Rational relations can be represented by models, which include finite transducers. From now on, we will always refer to rational relations omitting the word rational.

### 3.2.1 Finite Transducers

**Definition 3.5.** A (finite) *transducer* over $\Sigma$ and $\Delta$ is a sextuple

$$\dot{t} = (Q, \Sigma, \Delta, \delta, I, F),$$

where $Q$ is the set of *states*, $\delta \subseteq Q \times \Sigma^* \times \Delta^* \times Q$ is the finite set of *transitions*, $I$ is the set of *initial* states and $F$ is the set of *final* states. We say that $\Sigma$ is the *input* alphabet and $\Delta$ is the *output* alphabet.

We can graphically represent a finite transducer using a *transition diagram*, which is a directed graph. The vertices of the graph correspond to the states in $Q$ and are represented by a circle. Each transition from state $p$ to state $q$, labeled by $(w, w')$, corresponds to an edge from $p$ to $q$ labeled with $(w, w')$. The initial state $i$ is represented by a circle with an unlabeled *incoming* edge. Final states are represented by double circles.

We often write $\delta(p, (w, w')) = q$ instead of $\delta(p, (w, w')) = \{q\}$ when there is no risk of ambiguity.

*Remark* 3.6. By convention, all states $q$ of a transducer $\dot{t}$ over $\Sigma$ and $\Delta$ have an implicit transition from $q$ to $q$ labeled by $(\varepsilon, \varepsilon)$.

**Example 3.7.** Let $\dot{t}_{3.7}$ be the finite transducer represented by the diagram in Figure 3.1, where the set of states is $Q = \{0, 1\}$, the input alphabet is $\Sigma = \{\sigma, \tau\}$, the output alphabet is $\Delta = \{\sigma, \tau\}$, the set of initial states is $I = \{0\}$, the set of final states is $F = \{0\}$, and the transition function

is defined by

$$\delta(0,(\sigma,\sigma)) = 0, \qquad \delta(0,(\sigma,\tau)) = 1,$$
$$\delta(1,(\tau,\tau)) = 1, \qquad \delta(1,(\tau,\sigma)) = 0.$$

This transducer represents the left circular shift of words that begin with $\sigma$. One example of a pair of words in the relation realised by $\dot{t}_{3.7}$ is the pair $(\sigma\sigma\tau\tau\tau, \sigma\tau\tau\tau\sigma)$.

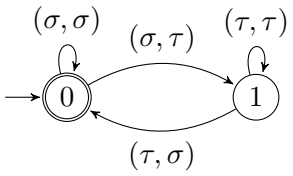



Figure 3.1: Diagram of the finite transducer $\dot{t}_{3.7}$.

Let $\dot{t}$ be a finite transducer and $(p, w, w', q)$ a transition of $\dot{t}$. We say that $(w, w')$ is the *label of the transition*, and we say that $p$ has an *outgoing* transition (with label $(w, w')$). A *path* of $\dot{t}$ is a finite sequence $(p_0, x_1, y_1, p_1, \ldots, x_\ell, y_\ell, p_\ell)$, for some nonnegative integer $\ell$, such that each tuple $(p_{i-1}, x_i, y_i, p_i)$, where $i \geq 1$, is a transition of $\dot{t}$. We naturally define $(x_1 \cdots x_\ell, y_1 \cdots y_\ell)$ as the *label of the path*, where $x_1 \cdots x_\ell$ is the *input label* and $y_1 \cdots y_\ell$ is the *output label*. We also write $p_0 \overset{(x_1\cdots x_\ell, y_1\cdots y_\ell)}{\rightsquigarrow} p_\ell$ when there is a path from $p_0$ to $p_\ell$ labeled by $(x_1 \cdots x_\ell, y_1 \cdots y_\ell)$. An *accepting path* is a path where $p_0$ is a initial state and $p_\ell$ is a final state. The *relation realized* by $\dot{t}$, denoted by $R(\dot{t})$, is the set of labels of all the accepting paths of $\dot{t}$. The set of all *possible outputs* of $\dot{t}$ when given the input word $w$, denoted by $\dot{t}(w)$, is the set $\dot{t}(w) = \{w' \mid (w, w') \in R(\dot{t})\}$. The *domain* of $\dot{t}$, denoted by $Dom(\dot{t})$, is the set set of all words $w$ over $\Sigma^*$ such that $\dot{t}(w) \neq \emptyset$. The *inverse* of $\dot{t}$, denoted by $\dot{t}^{-1}$, is the transducer that results from switching both the input and output alphabets and the input and output labels of each transition of $\dot{t}$. Given another transducer $\dot{s}$, the *composition* of $\dot{t}$ and $\dot{s}$ is the transducer that realizes the relation $R(\dot{t}) \circ R(\dot{s})$. The transducer $\dot{t}$ is called *trim* if every state is accessible and coaccessible, that is, if it appears in at least one accepting path of $\dot{t}$.

*Remark* 3.8. It follows by definition that for every transducer $\dot{t}$, its inverse $\dot{t}^{-1}$ realizes the inverse of the inverse of the relation realized by $\dot{t}$.

We can extend $\delta$ to $\delta \subseteq 2^Q \times \Sigma^* \times \Delta^* \times Q$ by

$$\delta(P, (w, w')) = \bigcup_{p \in P} \delta(p, (w, w')),$$

where $P \subseteq Q$.

**Definition 3.9.** Let $\dot{t}_1$ and $\dot{t}_2$ be two transducers over $\Sigma$ and $\Delta$. We say that $\dot{t}_1$ is *equivalent* to $\dot{t}_2$, denoted by $\dot{t}_1 \equiv \dot{t}_2$, if $R(\dot{t}_1) = R(\dot{t}_2)$.

### 3.2.2 Standard and Normal Form Transducers

**Definition 3.10.** Let $\dot{t} = (Q, \Sigma, \Delta, \delta, I, F)$ be a transducer over $\Sigma$ and $\Delta$. We say that $\dot{t}$ is a *standard form transducer* (SFT, for short), or say that $\dot{t}$ is in standard form, if the transition function, $\delta$, is such that

$$\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \times Q,$$

that is, if for every transition $(p, \tau, \tau', q)$, we have that $\tau \in \Sigma \cup \{\varepsilon\}$ and $\tau' \in \Sigma \cup \{\varepsilon\}$.

**Example 3.11.** The transducer in Example 3.7 is a transducer in standard form.

Note that a SFT is a finite transducer where we restrict the labels of the transitions to either symbols or the empty word. Therefore, everything that is valid for finite transducers is also valid for SFTs.

*Remark* 3.12. Every transducer $\dot{t}$ over $\Sigma$ and $\Delta$ has a SFT equivalent to it.

**Definition 3.13.** Let $\dot{t} = (Q, \Sigma, \Delta, \delta, I, F)$ be a transducer over $\Sigma$ and $\Delta$. We say that $\dot{t}$ is a *normal form transducer* (NFT, for short), or say that $\dot{t}$ is in normal form, if the transition function, $\delta$, is such that

$$\delta \subseteq Q \times ((\Sigma \cup \{\varepsilon\}) \times \{\varepsilon\} \cup \{\varepsilon\} \times (\Delta \cup \{\varepsilon\})) \times Q,$$

that is, if for every transition $(p, \tau, \tau', q)$, we have that either $\tau$ is a symbol and $\tau'$ is $\varepsilon$, $\tau$ is $\varepsilon$ and $\tau'$ is a symbol, or both $\tau$ and $\tau'$ are $\varepsilon$.

**Example 3.14.** Let $\dot{t}_{3.14}$ be the NFT represented by the diagram in Figure 3.2, where the set of states is $Q = \{0, 1, 2, 3, 4, 5\}$, the input alphabet is $\Sigma = \{\sigma, \tau\}$, the output alphabet is $\Delta = \{\sigma, \tau\}$, the set of initial states is $I = \{0\}$, the set of final states is $F = \{0\}$, and the transition function is defined by

$$\begin{aligned} &\delta(0, (\sigma, \varepsilon)) = \{2, 3\}, && \delta(1, (\tau, \varepsilon)) = \{4, 5\}, && \delta(2, (\varepsilon, \sigma)) = 0, \\ &\delta(3, (\varepsilon, \tau)) = 1, && \delta(4, (\varepsilon, \tau)) = 1, && \delta(5, (\varepsilon, \sigma)) = 0. \end{aligned}$$

Note that a NFT is a SFT where the labels of the transitions are such that at least the input symbol or the output symbol is $\varepsilon$. Therefore, everything that is valid for SFTs is also valid for NFTs.

*Remark* 3.15. Every SFT $\dot{t}_1$ over $\Sigma$ and $\Delta$ has a NFT equivalent to it. Therefore, every transducer $\dot{t}_2$ over $\Sigma$ and $\Delta$ has a NFT equivalent to it.

The NFT in Example 3.14 is a NFT equivalent to the transducer in Example 3.7.

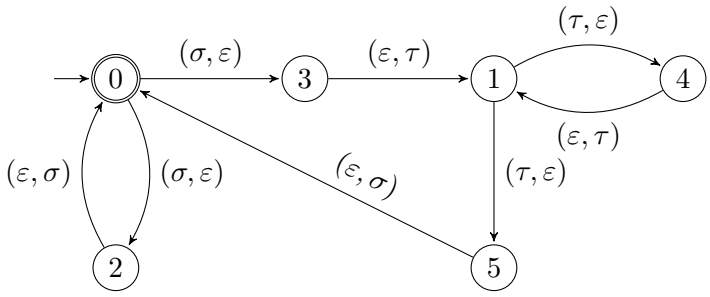



Figure 3.2: Diagram of the NFT $\dot{t}_{3.14}$.

## 3.3 Sequentialization of a Transducer

Contrary to ordinary automata, it is not true that any finite transducer is equivalent to a deterministic one, as explained by Lothaire in [20]. In fact, the notion of determinism in transducers is different than the notion of determinism in regular automata. In this section we discuss the notion of sequential transducers and the fact that a transducer has a equivalent sequential transducer if it has the functionality property.

### 3.3.1 Sequential Transducers

**Definition 3.16.** A (finite) *sequential* transducer over $\Sigma$ and $\Delta$ is a septuple

$$\dot{t} = (Q, \Sigma, \Delta, \delta, i, F, T)$$

where $Q$, $\Sigma$, $\Delta$, and $F$ are exactly the same as in Definition 3.5, i is the single *initial* state, $\delta \subseteq Q \times \Sigma \to \Delta^* \times Q$ is the *transition* function and $T : Q \to \Delta^*$ a function called the *terminal* function. Thus, a sequential transducer is a transducer that is *input deterministic*, that is, for every $q \in Q$, there are no two outgoing transitions from $q$ with the same input label.

Sequential transducers are represented exactly like finite transducers. The terminal function is represented by an outgoing edge from $q$ to no other state labeled by the word $T(q)$. If $T(q) = \varepsilon$, we often omit the edge that represents $T(q)$. If $T(q) = w$, where $w \in \Delta^*$, that means that we can add the word $w$ to all outputs in that state. A relation over $\Sigma$ and $\Delta$ that is realized by a sequential transducer is called a *sequential function*.

**Example 3.17.** Let $\dot{t}_{3.17}$ be the sequential transducer represented by the diagram in Figure 3.3, where the set of states is $Q = \{0, 1\}$, the input alphabet is $\Sigma = \{\sigma, \tau\}$, the output alphabet is $\Delta = \{\sigma, \tau\}$, the initial state is 0, the set of final states is $F = \{0, 1\}$, the transition function is defined by

$$\delta(0, (\sigma, \varepsilon)) = 1, \qquad \delta(1, (\sigma, \sigma)) = 1, \qquad \delta(1, (\tau, \tau)) = 1,$$

and the terminal function is defined by

$$T(0) = \varepsilon, \qquad T(1) = \sigma.$$

As we will see later, this sequential transducer is equivalent to the transducer in Example 3.7.

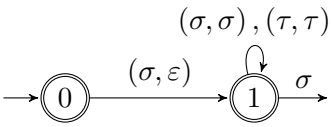



Figure 3.3: Diagram of the sequential transducer $\dot{t}_{3.17}$.

### 3.3.2 Functional Transducers

**Definition 3.18.** Let $\dot{t} = (Q, \Sigma, \Delta, \delta, I, F)$ be a transducer over $\Sigma$ and $\Delta$, and let

$$i \overset{(w_1, w_1')}{\rightsquigarrow} q \overset{(w_2, w_2')}{\rightsquigarrow} q \text{ and}$$
$$i' \overset{(w_1, w_1'')}{\rightsquigarrow} q' \overset{(w_2, w_2'')}{\rightsquigarrow} q'$$

be two paths with the same input label, where $i, i' \in I$. These two paths are called *twin*. We say that $\dot{t}$ has the *twins property* if for any pair of twin paths, the output is such that $w_2'$ and $w_2''$ are conjugate and $w_1' w_2' w_1' \cdots = w_1'' w_2'' w_2'' \cdots$.

The twins property was first formulated for transducers in Choffrut [9] (see also Berstel [3]).

**Definition 3.19.** Let $\dot{t}$ be a transducer over $\Sigma$ and $\Delta$. We say that $\dot{t}$ is *functional* if for any $w \in \Sigma^*$, $\dot{t}(w)$ is either empty or a singleton.

As proved by Lothaire in [20], a transducer has an equivalent sequential transducer if and only if it satisfies the twins property.

*Remark* 3.20. We can observe that a transducer $\dot{t}$ over $\Sigma$ and $\Delta$ has an equivalent sequential transducer if it is functional.

### 3.3.3 Functionality Test

In this subsection we will discuss an algorithm to determine if a transducer is functional, which is an adaptation of the algorithm presented by Allauzen and Mohri [1] and we revert to that article for more detailed informations. Another algorithm was introduced by Béal, Carton, Prieur and Sakarovitch [7].

**Definition 3.21.** Let $\dot{t}$ be a transducer over $\Sigma$ and $\Delta$ and $x, y \in \Sigma^*$. The *residue* of $x$ by $y$ is defined by $y^{-1}x$, that is, if $x = wu$ and $y = wv$, then $y^{-1}x = u$, where $w, u, v \in \Sigma^*$ and $w$ is the longest common prefix of $x$ and $y$.

Let $\dot{t}$ be a transducer over $\Sigma$ and $\Delta$. The definition of residues can be extended to paths by considering the input and output labels of the path, that is, if $\pi = (p_0, w_1, w'_1, p_1, \ldots, w_\ell, w'_\ell, p_\ell)$ is a path and $(w, w') = (w_1 \cdots w_\ell, w'_1 \cdots w'_\ell)$ then the residue of $\pi$ is ${w'}^{-1}w$.

**Lemma 3.22.** *A transducer $\dot{t}$ over $\Sigma$ and $\Delta$ is functional if and only if*

$$R\left(\dot{t}^{-1}\right) \circ R(\dot{t}) = Id_{Dom\left(\dot{t}^{-1} \circ \dot{t}\right)}.$$

*Proof.* By definition of $\dot{t}^{-1}$, $w' \in \left(\dot{t}^{-1} \circ \dot{t}\right)(w)$ if and only if there exists $w''$ such that $w \in \dot{t}(w'')$ and $w' \in \dot{t}(w'')$. If $\dot{t}$ is functional, then $w = w'$, and thus $R(\dot{t}^{-1}) \circ R(\dot{t}) = Id_{Dom\left(\dot{t}^{-1} \circ \dot{t}\right)}$. On the other hand, if $R(\dot{t}^{-1}) \circ R(\dot{t}) = Id_{Dom\left(\dot{t}^{-1} \circ \dot{t}\right)}$, then $w = w'$, therefore $\dot{t}$ is functional. □

Lemma 3.22 leads to an efficient method for checking whether a transducer is functional or not. This method consists of checking if for a given transducer $\dot{t}$, one has that $R(\dot{t}^{-1}) \circ R(\dot{t})$ is the identity relation and is described in Theorem 3.23.

**Theorem 3.23.** *Let $\dot{t} = (Q, \Sigma, \Delta, \delta, I, F)$ be a transducer over $\Sigma$ and $\Delta$. There exists an algorithm for testing if $\dot{t}$ realizes the identity relation in $\mathcal{O}(|\delta| + |\Delta|\,|Q|)$ time.*

*Proof.* First, we assume that $\dot{t}$ has only one intial state. If it has more than one initial state, a new initial state $i$ is added with $(\varepsilon, \varepsilon)$ transitions from $i$ to every $q \in I$ and make $i$ as the only initial state. Next, for every state we compute the residue of any path from the initial state to such state. Since we are only interested in states that are both accessible and coaccessible, we trim the transducer. It is clear that if we have a transition $(p, w, w', q)$, then the residue of $q$ is given by $R[q] = w^{-1}R[p]w'$, where $R[p]$ is the residue of $p$. Trivially, the residue of the initial state $i$ is $\varepsilon$. Thus, we only need to compute the residue of a state once, which can be done in linear time [1]. The algorithm returns `FALSE` if at some point, for a state that has already a residue found, a residue from another path is different from that residue. On the other hand, for every $f \in F$ we have that if the relation is the identity relation, then $R[f] = \varepsilon$. Thus, if there is any $f \in F$ such that $R[f] \neq \varepsilon$, then the algorithm also returns `FALSE`. If neither of these contradictions hold, the algorithm returns `TRUE` after every residue is computed. □

Theorem 3.23 together with Lemma 3.22 gives an efficient algorithm for testing if a transducer is functional.

**Corollary 3.24.** *Let $\dot{t} = (Q, \Sigma, \Delta, \delta, I, F)$ be a transducer over $\Sigma$ and $\Delta$. There exists an algorithm for testing the functionality of $\dot{t}$ in $\mathcal{O}\left(|\delta|^2 + |\Delta|\,|Q|^2\right)$ time.*

*Proof.* The proof follows directly from Theorem 3.23, Lemma 3.22 and the fact that the number of states of $\dot{t}^{-1} \circ \dot{t}$ is in $\mathcal{O}\left(|Q|^2\right)$ and the number of transitions in $\mathcal{O}\left(|\delta|^2\right)$. □

**Example 3.25.** Consider the transducers in Figure 3.4. On the left side, we have a transducer $\dot{t}$ that is not functional. On the right, we have the transducer that realizes the relation $R(\dot{t}^{-1}) \circ R(\dot{t})$.

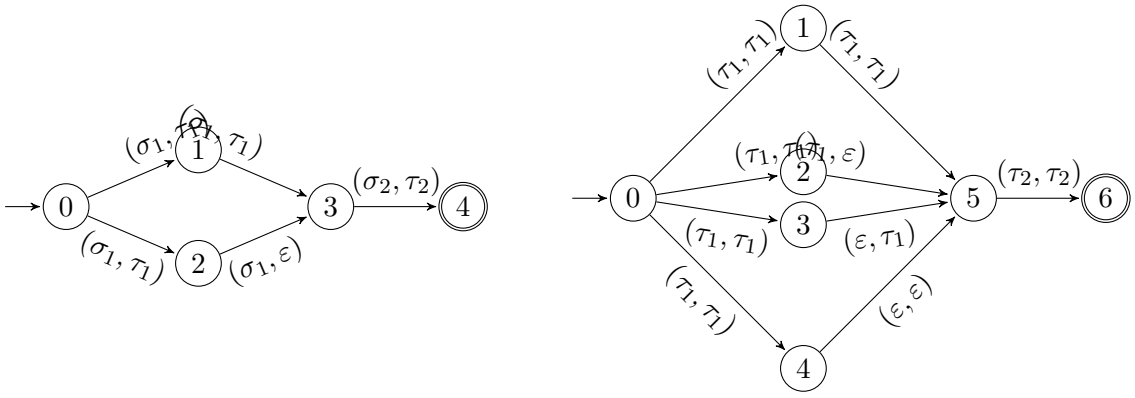



Figure 3.4: Application of the functionality algorithm.

### 3.3.4 Witness of Non-Functionality

In this subsection we discuss a modification of the algorithm presented in the previous section in order to produce a new method for obtaining a witness of non-functionality of a non-functional transducer.

**Definition 3.26.** Let $\dot{t}$ be a non-functional transducer over $\Sigma$ and $\Delta$. A *witness* of non-functionality of $\dot{t}$ is a triple $(w, w', w'')$, where $w \in \Sigma^*$ and $w', w'' \in \dot{t}(w)$.

**Example 3.27.** Consider the transducer $\dot{t}$ on the left of Figure 3.4. A witness of non-functionality of $\dot{t}$ is the triple $(\sigma_1\sigma_1\sigma_2, \tau_1\tau_1\tau_2, \tau_1\tau_2)$.

The algorithm that we present relies on the assumption that when the algorithm presented in Theorem 3.23 finds a different residue for some state, then there are two paths from the initial state to that state that take the same word as input but produce different outputs. Thus, we only have to run a depth-first search [10] from that state, stopping when we reach a final state.

### 3.3.5 Sequentialization of a Functional Transducer

As we saw in Subsection 3.3.2, a transducer has an equivalent sequential transducer if it is functional. We now present an algorithm adapted from the one presented by Allauzen and Mohri [1] and is a variant of the determinization algorithm for automata. The main difference is that the algorithm would fail to terminate if the transducer provided was non-functional. To prevent this, we test if the transducer is functional beforehand.

**Definition 3.28.** Let $\dot{t} = (Q, \Sigma, \Delta, \delta, I, F)$ be a transducer over $\Sigma$ and $\Delta$. A pair $(w', q) \in \Delta^* \times Q$ is called a *half-edge*.

The algorithm that was implemented is based on finding sets of half-edges (which will be the states of the transducer) and consists of a set of methods. We will only describe the algorithm. For further information, we refer to [1].

We start with a transducer that is in standard form and trim. If the transducer is not functional, the algorithm simply returns `None`, since there is no sequential transducer equivalent to the transducer given. If the transducer is functional, then we start computing an equivalent sequential transducer. As referred above, the states of the equivalent sequential transducer are sets of half-edges. The states are computed by using a function that given a set of half-edges $S$ and an input letter $\sigma$ returns the union, for $(u, q) \in S$, of the set of half-edges $(uvw, r)$ such that there are, in the transducer, an edge $(p, \sigma, v, q)$ and a path $(q, \varepsilon, w, r)$. As an auxiliary step, we compute, for a set of half-edges $U$, the longest common prefix of the words $u$ such that there is a pair $(u, q) \in U$. After that, we erase that prefix from the half-edges $(u, q) \in U$.

We now have the conditions to compute the sequential transducer. We use the method described to find every state, using the longest common prefix determined for the output labels.

**Example 3.29.** Consider the transducers in Figure 3.5. On the left side, we have the transducer that realizes the circular left shift of words starting with a $\sigma$. In the right, we have an equivalent sequential transducer obtained by applying the method described.


Figure 3.5: The sequentialization algorithm.

# Chapter 4

# 2D-Regular Expressions

In this chapter, we discuss the notions of two dimensional (2D) regular expressions. The idea is to define transductions in a more compact form.

**Definition 4.1.** An *atomic 2D regular expression* (or A2D-RE for short) over $\Sigma \times \Delta$, where $\Sigma$ and $\Delta$ are alphabets, is an expression of the type $(r_1, r_2)$, where $r_1$ and $r_2$ are regular expressions over $\Sigma$ and $\Delta$, respectively.

**Definition 4.2.** A *(general) 2D regular expression* (or G2D-RE, or 2D-RE, for short) over $\Sigma \times \Delta$, where $\Sigma$ and $\Delta$ are alphabets, is an expression that is either $\emptyset$ or can be defined by the grammar

$$g \to (r_1, r_2) \mid (g \oplus g) \mid (g \odot g) \mid (g^{\circledast}) \,,$$

where $r_1, r_2$ are regular expressions over $\Sigma$ and $\Delta$, respectively. The operator $\odot$ is often omitted. We often use $+$, $\cdot$ and $*$ instead of $\oplus$, $\odot$ and $\circledast$, respectively, when there is no risk of ambiguity.

We denote by *G2D-RE* the set of all G2D-REs.

**Example 4.3.** Let $\Sigma = \{\sigma\}$ and $\Delta = \{\tau\}$. Then $g = ((\sigma, \tau) + (\varepsilon, \tau)^*)\,(\sigma, \varepsilon) + (\varepsilon, \tau)$ is a 2D-RE over $\Sigma \times \Delta$.

*Remark* 4.4. Every A2D-RE is also a G2D-RE.

**Definition 4.5.** Let $g$ be a 2D-RE over $\Sigma \times \Delta$. The *relation* defined by a 2D-RE, denoted by $T(g)$, is inductively defined on the structure of $g$ by

$$\begin{aligned}
T(\emptyset) &= \emptyset, \\
T((r_1, r_2)) &= \mathcal{L}(r_1) \times \mathcal{L}(r_2), \\
T(g_1 g_2) &= \{(xx', yy') \mid (x, y) \in T(g_1) \wedge (x', y') \in T(g_2)\} = T(g_1)T(g_2), \\
T(g_1 + g_2) &= T(g_1) \cup T(g_2), \\
T(g_1^*) &= \bigcup_n T(g_1^n) = (T(g_1))^* \,,
\end{aligned}$$

where $r_1, r_2$ are regular expressions over $\Sigma$ and $\Delta$, respectively.

Let $G$ be a set of G2D-REs over $\Sigma \times \Delta$. We have that $T(G) = \bigcup_{g \in G} T(g)$.

**Definition 4.6.** Let $g_1, g_2$ be two 2D-RE over $\Sigma \times \Delta$. We say that $g_1$ and $g_2$ are *equivalent*, denoted by $g_1 \sim g_2$, if $T(g_1) = T(g_2)$.

**Lemma 4.7.** *Let $g$ be a 2D-REs over $\Sigma \times \Delta$. The following equivalences hold*

$$(\varepsilon, \varepsilon)\, g \sim g\, (\varepsilon, \varepsilon) \sim g, \tag{4.1}$$

$$\emptyset g \sim g \emptyset \sim \emptyset, \tag{4.2}$$

$$\emptyset + g \sim g + \emptyset \sim g, \tag{4.3}$$

$$\emptyset^* \sim (\varepsilon, \varepsilon)\,. \tag{4.4}$$

*Proof.* (4.1) Note that, by definition,

$$\begin{aligned} T((\varepsilon, \varepsilon)\, g) &= \{(w_1 w_2, w_1' w_2') \mid (w_1, w_1') \in T((\varepsilon, \varepsilon)) \wedge (w_2, w_2') \in T(g)\} \\ &= \{(\varepsilon w_2, \varepsilon w_2') \mid (w_2, w_2') \in T(g)\} \\ &= \{(w_2, w_2') \mid (w_2, w_2') \in T(g)\} \\ &= T(g). \end{aligned}$$

Analogously, one proves that $g\, (\varepsilon, \varepsilon) \sim g$.

(4.2) We have, by definition, $T(\emptyset g) = \{(w_1 w_2, w_1' w_2') \mid (w_1, w_1') \in T(\emptyset) \wedge (w_2, w_2') \in T(g)\} = \emptyset = T(\emptyset)$. Analogously, one proves that $g\emptyset \sim \emptyset$.

(4.3) We have, by definition, $T(\emptyset + g) = T(\emptyset) \cup T(g) = \emptyset \cup T(g) = T(g)$. Analogously, one proves that $g + \emptyset \sim g$.

(4.4) We have, by definition, $T(\emptyset^*) = \bigcup_n T(\emptyset^n) = T(\emptyset^0) = \emptyset^0 = \{(\varepsilon, \varepsilon)\} = T((\varepsilon, \varepsilon))$. □

**Definition 4.8.** A *standard 2D regular expression* (or S2D-RE for short) over $\Sigma \times \Delta$, where $\Sigma$ and $\Delta$ are alphabets, is an expression that is either $\emptyset$ or can be defined by the grammar

$$\begin{aligned} s &\to (a_1, a_2) \mid (s \oplus s) \mid (s \odot s) \mid (s^{\circledast})\,, \\ a_1 &\to \varepsilon \mid \sigma \in \Sigma, \\ a_2 &\to \varepsilon \mid \sigma \in \Delta, \end{aligned}$$

where the operator $\odot$ is often omitted. We often use $+$, $\cdot$ and $*$ instead of $\oplus$, $\odot$ and $\circledast$, respectively, when there is no risk of ambiguity.

*Remark* 4.9. Any S2D-RE is also a G2D-RE.

We denote by *S2D-RE* the set of all S2D-REs.

**Example 4.10.** Let $g = ((\sigma, \tau) + (\varepsilon, \tau)^*)\, (\sigma, \varepsilon) + (\varepsilon, \tau)$ be the 2D-RE in Example 4.3. We have that $g$ is also a S2D-RE.

**Definition 4.11.** Let $g$ be a 2D-RE over $\Sigma \times \Delta$ and $op \in \{\cdot, +, *, \odot, \oplus, \circledast\}$ be an operator. The number of occurrences of $op$ in $g$ is denoted by $|g|_{op}$.

**Example 4.12.** Let $g = ((\sigma, \tau) + (\varepsilon, \tau)^*)(\sigma, \varepsilon) + (\varepsilon, \tau)$ be the 2D-RE in Example 4.3. Then, we have

1. $|g|_{\cdot} = 0$.
2. $|g|_{+} = 0$.
3. $|g|_{*} = 0$.
4. $|g|_{\odot} = 1$.
5. $|g|_{\oplus} = 2$.
6. $|g|_{\circledast} = 1$.

**Definition 4.13.** Let $g$ be a G2D-RE over $\Sigma \times \Delta$.

1. The *input-alphabetic size* $|g|_{\Sigma}$ of $g$ is the number of letters on the input parts of $g$.
2. The *output-alphabetic size* $|g|_{\Delta}$ of $g$ is the number of letters on the output parts of $g$.
3. The *number of non-$\varepsilon$ atoms* $|g|_{\Sigma\times\Delta}$ of $g$ is the number of atoms $(r_1, r_2)$ in $g$ where $r_1$, $r_2$ are REs over $\Sigma$ and $\Delta$, respectively, and neither $r_1$ nor $r_2$ is $\varepsilon$.
4. The *number of atoms* $|g|_{\overline{(\varepsilon,\varepsilon)}}$ of $s$ is the number of atoms $(r_1, r_2)$ in $g$, where $r_1$ and $r_2$ are regular expressions over $\Sigma$ and $\Delta$, respectively, and either $r_1$ or $r_2$ can be $\varepsilon$, but not both.
5. The *number of* $(\varepsilon, \varepsilon)$ $|g|_{(\varepsilon,\varepsilon)}$ in $g$ is the number of $(\varepsilon, \varepsilon)$ in $g$.
6. The *alphabetic size* $|g|_{\Sigma\cup\Delta}$ of $g$ is the number of letters in $g$, that is, $|g|_{\Sigma\cup\Delta} = |g|_{\Sigma} + |g|_{\Delta}$.
7. The *size* $|g|$ of $g$ is the number of symbols in $g$, disregarding parenthesis.

**Example 4.14.** Let $g = ((\sigma, \tau) + (\varepsilon, \tau)^*)(\sigma, \varepsilon) + (\varepsilon, \tau)$ be the 2D-RE in Example 4.3. Then, we have

1. $|g|_{\Sigma} = 2$, since there are two $\sigma$'s in $g$.
2. $|g|_{\Delta} = 3$, since there are three $\tau$'s in $g$.
3. $|g|_{\Sigma\times\Delta} = 1$.
4. $|g|_{\overline{(\varepsilon,\varepsilon)}} = 4$.
5. $|g|_{\Sigma\cup\Delta} = 2 + 3 = 5$, since there are five letters in $g$.
6. $|g| = |g|_{\cdot} + |g|_{+} + |g|_{*} + |g|_{\odot} + |g|_{\oplus} + |g|_{\circledast} + |g|_{\overline{(\varepsilon,\varepsilon)}} + |g|_{(\varepsilon,\varepsilon)} + |(\sigma, \tau)| + |(\varepsilon, \tau)| + |(\sigma, \varepsilon)| + |(\varepsilon, \tau)| = |g|_{\cdot} + |g|_{+} + |g|_{*} + |g|_{\odot} + |g|_{\oplus} + |g|_{\circledast} + |g|_{\overline{(\varepsilon,\varepsilon)}} + |g|_{(\varepsilon,\varepsilon)} + (|\sigma| + |\tau|) + (|\varepsilon| + |\tau|) + (|\sigma| + |\varepsilon|) + (|\varepsilon| + |\tau|) = 0 + 0 + 0 + 1 + 2 + 1 + 4 + 0 + (1 + 1) + (1 + 1) + (1 + 1) + (1 + 1) = 16$.

## 4.1 Standard 2D-Regular Expressions

In this section, we will discuss some definitions and results particular to S2D-REs.

**Definition 4.15.** Let $s$ be a S2D-RE over $\Sigma \times \Delta$. Then $s$ is said to be *normalised* if every atom is either $(\sigma, \varepsilon)$ or $(\varepsilon, \sigma')$.

**Definition 4.16.** Let $s$ be a S2D-RE over $\Sigma \times \Delta$. The *S2D constant part* of $s$, denoted by $\epsilon(s)$, is defined by

$$\epsilon(s) = \begin{cases} (\varepsilon, \varepsilon), & \text{if } (\varepsilon, \varepsilon) \in T(s); \\ \emptyset, & \text{otherwise.} \end{cases}$$

We can extend this definition to a set of S2D-REs $S$ by

$$\epsilon(S) = \begin{cases} (\varepsilon, \varepsilon), & \text{if } \exists s \in S : \epsilon(s) = (\varepsilon, \varepsilon), \\ \emptyset, & \text{otherwise.} \end{cases}$$

We also need to consider the cases where only input (output) is necessarily $\varepsilon$, that is, we need to consider expressions $s$ over $\Sigma \times \Delta$ such that $\exists w' \in \Delta^* \, (\varepsilon, w') \in T(s)$ $(\exists w \in \Sigma^* \, (w, \varepsilon) \in T(s))$. More formally, we need to consider the cases where we have a restriction of $T(s)$ where the input (output) is $\varepsilon$.

**Definition 4.17.** Let $s$ be a S2D-RE over $\Sigma \times \Delta$. The *S2D input-constant part* of $s$, denoted by $\epsilon_{\text{in}}(s)$, is defined recursively on the structure of $s$ by

$$\begin{aligned} \epsilon_{\text{in}}(\emptyset) &= \emptyset, \\ \epsilon_{\text{in}}((\varepsilon, \varepsilon)) &= (\varepsilon, \varepsilon), \\ \epsilon_{\text{in}}((\sigma, \varepsilon)) &= \emptyset, \\ \epsilon_{\text{in}}((\varepsilon, \sigma')) &= (\varepsilon, \sigma'), \\ \epsilon_{\text{in}}((\sigma, \sigma')) &= \emptyset, \\ \epsilon_{\text{in}}(s_1 + s_2) &= \epsilon_{\text{in}}(s_1) + \epsilon_{\text{in}}(s_2), \\ \epsilon_{\text{in}}(s_1 s_2) &= \epsilon_{\text{in}}(s_1)\epsilon_{\text{in}}(s_2), \\ \epsilon_{\text{in}}(s^*) &= (\epsilon_{\text{in}}(s))^*, \end{aligned}$$

for every $\sigma \in \Sigma$, $\sigma' \in \Delta$.

The *S2D output-constant part* of $s$, denoted by $\epsilon_{\text{out}}(s)$, is defined in the exact same way as $\epsilon_{\text{in}}(s)$, except that $\epsilon_{\text{out}}((\varepsilon, \sigma')) = \emptyset$ and $\epsilon_{\text{out}}((\sigma, \varepsilon)) = (\sigma, \varepsilon)$.

**Example 4.18.** Let $g = ((\sigma, \tau) + (\varepsilon, \tau)^*)\,(\sigma, \varepsilon) + (\varepsilon, \tau)$ be the 2D-RE in Example 4.3. We have

that

$$\begin{aligned}
\epsilon_{\text{in}}(g) &= \epsilon_{\text{in}}(((\sigma,\tau)+(\varepsilon,\tau)^*)\,(\sigma,\varepsilon)+(\varepsilon,\tau)) \\
&= \epsilon_{\text{in}}(((\sigma,\tau)+(\varepsilon,\tau)^*)\,(\sigma,\varepsilon)) + \epsilon_{\text{in}}((\varepsilon,\tau)) \\
&= \epsilon_{\text{in}}((\sigma,\tau)+(\varepsilon,\tau)^*)\epsilon_{\text{in}}((\sigma,\varepsilon)) + (\varepsilon,\tau) \\
&= (\epsilon_{\text{in}}((\sigma,\tau)) + \epsilon_{\text{in}}((\varepsilon,\tau)^*))\,\emptyset + (\varepsilon,\tau) \\
&= (\emptyset + \epsilon_{\text{in}}((\varepsilon,\tau))^*)\,\emptyset + (\varepsilon,\tau) \\
&= (\emptyset + (\varepsilon,\tau)^*)\,\emptyset + (\varepsilon,\tau) \\
&\sim (\varepsilon,\tau)^*\,\emptyset + (\varepsilon,\tau) \\
&\sim (\varepsilon,\tau)\,,
\end{aligned}$$

and

$$\begin{aligned}
\epsilon_{\text{out}}(g) &= \epsilon_{\text{out}}(((\sigma,\tau)+(\varepsilon,\tau)^*)\,(\sigma,\varepsilon)+(\varepsilon,\tau)) \\
&= \epsilon_{\text{out}}(((\sigma,\tau)+(\varepsilon,\tau)^*)\,(\sigma,\varepsilon)) + \epsilon_{\text{out}}((\varepsilon,\tau)) \\
&= \epsilon_{\text{out}}((\sigma,\tau)+(\varepsilon,\tau)^*)\epsilon_{\text{out}}((\sigma,\varepsilon)) + \emptyset \\
&= (\epsilon_{\text{out}}((\sigma,\tau)) + \epsilon_{\text{out}}((\varepsilon,\tau)^*))\,(\sigma,\varepsilon) + \emptyset \\
&= (\emptyset + \epsilon_{\text{out}}((\varepsilon,\tau))^*)\,(\sigma,\varepsilon) + \emptyset \\
&= (\emptyset + \emptyset^*)\,(\sigma,\varepsilon) + \emptyset \\
&\sim (\emptyset + \emptyset^*)\,(\sigma,\varepsilon) \\
&\sim \emptyset^*\,(\sigma,\varepsilon) \\
&\sim (\varepsilon,\varepsilon)\,(\sigma,\varepsilon) \\
&\sim (\sigma,\varepsilon)\,.
\end{aligned}$$

**Lemma 4.19.** *Let $s$ be a S2D-RE over $\Sigma \times \Delta$. Then $T(\epsilon_{in}(s)) = T(s) \cap (\{\varepsilon\} \times \Delta^*)$ and $T(\epsilon_{out}(s)) = T(s) \cap (\Sigma^* \times \{\varepsilon\})$.*

*Proof.* We only prove the first equality, since the second follows from the first one by symmetry. We proceed by induction on the structure of $s$. The base cases $\emptyset$, $(\sigma,\varepsilon)$, $(\varepsilon,\sigma')$ and $(\sigma,\sigma')$ are trivial. Now let us suppose that we have that the claim is true for S2D-REs $s_1$ and $s_2$ over $\Sigma \times \Delta$. The case $s_1+s_2$ is also trivial, since $T(\epsilon_{\text{in}}(s_1+s_2)) = T(\epsilon_{\text{in}}(s_1) + \epsilon_{\text{in}}(s_1)) = T(\epsilon_{\text{in}}(s_1)) \cup T(\epsilon_{\text{in}}(s_2))$. For the concatenation case, we have that

$$\begin{aligned}
T(\epsilon_{\text{in}}(s_1 s_2)) &= T(\epsilon_{\text{in}}(s_1)\epsilon_{\text{in}}(s_2)) = T(\epsilon_{\text{in}}(s_1))T(\epsilon_{\text{in}}(s_2)) \\
&\underset{\text{I.H.}}{=} (T(s_1) \cap (\{\varepsilon\} \times \Delta^*))\,(T(s_2) \cap (\{\varepsilon\} \times \Delta^*)) \\
&= \{(\varepsilon,w_1) \mid (\varepsilon,w_1) \in T(s_1)\}\,\{(\varepsilon,w_2) \mid (\varepsilon,w_2) \in T(s_2)\} \\
&= \{(\varepsilon,w_1)\,(\varepsilon,w_2) \mid (\varepsilon,w_1) \in T(s_1) \wedge (\varepsilon,w_2) \in T(s_2)\} \\
&= \{(\varepsilon,w_1)\,(\varepsilon,w_2) \mid (\varepsilon,w_1)\,(\varepsilon,w_2) \in T(s_1)T(s_2)\} \\
&= \{(\varepsilon,w_1 w_2) \mid (\varepsilon,w_1 w_2) \in T(s_1 s_2)\} \\
&= T(s_1 s_2) \cap (\{\varepsilon\} \times \Delta^*)\,.
\end{aligned}$$

On the other hand, since for all $n \geq 0$ we have that $(T(s) \cap (\{\varepsilon\} \times \Delta^*))^n = T(s)^n \cap (\{\varepsilon\} \times \Delta^*)$, the result follows for $s_1^*$. □

*Remark* 4.20. We have from Lemma 4.19 that

$$T(\epsilon_{\text{in}}(s)) = \{(\varepsilon, w') \mid (\varepsilon, w') \in T(s)\}$$
$$\text{and } T(\epsilon_{\text{out}}(s)) = \{(w, \varepsilon) \mid (w, \varepsilon) \in T(s)\}.$$

### 4.1.1 Partial Derivatives

We can extend the concatenation operation from S2D-REs to sets of S2D-REs. Let $\cdot : 2^{S2D\text{-}RE} \times S2D\text{-}RE \to 2^{S2D\text{-}RE}$ be such extension. The operation is recursively defined by

$$\begin{aligned} S \cdot \emptyset &= \emptyset, \\ S \cdot (\varepsilon, \varepsilon) &= S, \\ \emptyset \cdot s_2 &= \emptyset, \\ \{(\varepsilon, \varepsilon)\} \cdot s_2 &= \{s_2\}, \\ \{s_1\} \cdot s_2 &= \{s_1 s_2\}, \\ (S \cup S') \cdot s_2 &= (S \cdot s_2) \cup (S' \cdot s_2), \end{aligned}$$

where $S, S' \subseteq 2^{S2D\text{-}RE}$, $s_1 \in S2D\text{-}RE \setminus \{\emptyset\}$ and $s_2 \in S2D\text{-}RE \setminus \{\emptyset, (\varepsilon, \varepsilon)\}$. We will often omit the $\cdot$ operator.

**Definition 4.21.** Let $s$ be a S2D-RE over $\Sigma \times \Delta$. The set of *partial derivatives* of $s$ with respect to a pair $(\sigma_1, \varepsilon)$, where $\sigma_1 \in \Sigma$ ($(\varepsilon, \sigma_2)$, where $\sigma_2 \in \Delta$), denoted by $\partial_{(\sigma_1,\varepsilon)}(s)$ ($\partial_{(\varepsilon,\sigma_2)}(s)$), is

defined recursively on the structure of $s$ by

$$\begin{aligned}
\partial_{(\sigma_1,\varepsilon)}(\emptyset) &= \partial_{(\varepsilon,\sigma_2)}(\emptyset) = \emptyset,\\
\partial_{(\sigma_1,\varepsilon)}((\varepsilon,\varepsilon)) &= \partial_{(\varepsilon,\sigma_2)}((\varepsilon,\varepsilon)) = \emptyset,\\
\partial_{(\sigma_1,\varepsilon)}((\varepsilon,\sigma_2')) &= \partial_{(\varepsilon,\sigma_2)}((\sigma_1',\varepsilon)) = \emptyset,\\
\partial_{(\sigma_1,\varepsilon)}((\sigma_1',\varepsilon)) &= \begin{cases} \{(\varepsilon,\varepsilon)\}, & \text{if } \sigma_1 = \sigma_1',\\ \emptyset, & \text{otherwise,}\end{cases}\\
\partial_{(\varepsilon,\sigma_2)}((\varepsilon,\sigma_2')) &= \begin{cases} \{(\varepsilon,\varepsilon)\}, & \text{if } \sigma_2 = \sigma_2',\\ \emptyset, & \text{otherwise,}\end{cases}\\
\partial_{(\sigma_1,\varepsilon)}((\sigma_1',\sigma_2')) &= \begin{cases} \{(\varepsilon,\sigma_2')\}, & \text{if } \sigma_1 = \sigma_1',\\ \emptyset, & \text{otherwise,}\end{cases}\\
\partial_{(\varepsilon,\sigma_2)}((\sigma_1',\sigma_2')) &= \begin{cases} \{(\sigma_1',\varepsilon)\}, & \text{if } \sigma_2 = \sigma_2',\\ \emptyset, & \text{otherwise,}\end{cases}\\
\partial_{(\sigma_1,\varepsilon)}(s_1 + s_2) &= \partial_{(\sigma_1,\varepsilon)}(s_1) \cup \partial_{(\sigma_1,\varepsilon)}(s_2),\\
\partial_{(\varepsilon,\sigma_2)}(s_1 + s_2) &= \partial_{(\varepsilon,\sigma_2)}(s_1) \cup \partial_{(\varepsilon,\sigma_2)}(s_2),\\
\partial_{(\sigma_1,\varepsilon)}(s_1 s_2) &= \partial_{(\sigma_1,\varepsilon)}(s_1)s_2 \cup \epsilon_{\text{in}}(s_1)\partial_{(\sigma_1,\varepsilon)}(s_2),\\
\partial_{(\varepsilon,\sigma_2)}(s_1 s_2) &= \partial_{(\varepsilon,\sigma_2)}(s_1)s_2 \cup \epsilon_{\text{out}}(s_1)\partial_{(\varepsilon,\sigma_2)}(s_2),\\
\partial_{(\sigma_1,\varepsilon)}(s_1^*) &= \partial_{(\sigma_1,\varepsilon)}(s_1)s_1^*,\\
\partial_{(\varepsilon,\sigma_2)}(s_1^*) &= \partial_{(\varepsilon,\sigma_2)}(s_1)s_1^*,
\end{aligned}$$

where $s_2$ is also a S2D-RE over $\Sigma \times \Delta$.

**Lemma 4.22.** *Let $s$ be a S2D-RE over $\Sigma \times \Delta$. The relations defined by $\partial_{(\sigma,\varepsilon)}(s)$ and $\partial_{(\varepsilon,\sigma')}(s)$, denoted by $T\left(\partial_{(\sigma,\varepsilon)}(s)\right)$ and $T\left(\partial_{(\varepsilon,\sigma')}(s)\right)$, respectively, verify the following equalities*

$$\begin{aligned}
T\left(\partial_{(\sigma,\varepsilon)}(s)\right) &= \{(w,w') \mid (\sigma w, w') \in T(s)\},\\
T\left(\partial_{(\varepsilon,\sigma')}(s)\right) &= \{(w,w') \mid (w, \sigma' w') \in T(s)\}.
\end{aligned}$$

*Proof.* Since the proof for both equalities is analogous by symmetry, we will only prove the first one. The proof follows by induction on the structure of $s$.

The base cases $\partial_{(\sigma,\varepsilon)}(\emptyset)$, $\partial_{(\sigma,\varepsilon)}((\varepsilon,\varepsilon))$, $\partial_{(\sigma,\varepsilon)}((\sigma_1,\varepsilon))$, $\partial_{(\sigma,\varepsilon)}((\varepsilon,\sigma_2))$ and $\partial_{(\sigma_1,\varepsilon)}((\sigma_1',\sigma_2'))$ hold trivially.

Now let us suppose that the claim is true for S2D-REs $s_1$ and $s_2$.

1. **Case** $s_1 + s_2$.

$$\begin{aligned} T\Big(\partial_{(\sigma,\varepsilon)}(s_1 + s_2)\Big) &= T\Big(\partial_{(\sigma,\varepsilon)}(s_1) \cup \partial_{(\sigma,\varepsilon)}(s_2)\Big) = T\Big(\partial_{(\sigma,\varepsilon)}(s_1)\Big) \cup T\Big(\partial_{(\sigma,\varepsilon)}(s_2)\Big) \\ &\underset{\text{I.H.}}{=} \{(w, w') \mid (\sigma w, w') \in T(s_1)\} \cup \{(w, w') \mid (\sigma w, w') \in T(s_2)\} \\ &= \{(w, w') \mid (\sigma w, w') \in T(s_1) \vee (\sigma w, w') \in T(s_2)\} \\ &= \{(w, w') \mid (\sigma w, w') \in (T(s_1) \cup T(s_2))\} \\ &= \{(w, w') \mid (\sigma w, w') \in T(s_1 + s_2)\}. \end{aligned}$$

2. **Case** $s_1 s_2$.

We have that $T\Big(\partial_{(\sigma,\varepsilon)}(s_1 s_2)\Big) = T\Big(\partial_{(\sigma,\varepsilon)}(s_1)s_2 \cup \epsilon_{\text{in}}(s_1)\partial_{(\sigma,\varepsilon)}(s_2)\Big) = T\Big(\partial_{(\sigma,\varepsilon)}(s_1)s_2\Big) \cup T\Big(\epsilon_{\text{in}}(s_1)\partial_{(\sigma,\varepsilon)}(s_2)\Big)$. We now need to consider two cases, the case when $\epsilon_{\text{in}}(s_1) = \emptyset$ and the case when $\epsilon_{\text{in}}(s_1) \neq \emptyset$.

If $\epsilon_{\text{in}}(s_1) = \emptyset$, we have that

$$\begin{aligned} T\Big(\partial_{(\sigma,\varepsilon)}(s_1 s_2)\Big) &= T\Big(\partial_{(\sigma,\varepsilon)}(s_1)s_2\Big) \cup \emptyset \\ &= T\Big(\partial_{(\sigma,\varepsilon)}(s_1)\Big)T(s_2) \\ &\underset{\text{I.H.}}{=} \{(w_1, w_1') \mid (\sigma w_1, w_1') \in T(s_1)\}\{(w_2, w_2') \mid (w_2, w_2') \in T(s_2)\} \\ &= \{(w_1 w_2, w_1' w_2') \mid (\sigma w_1, w_1') \in T(s_1) \wedge (w_2, w_2') \in T(s_2)\} \\ &= \{(w, w') \mid (\sigma w, w') \in T(s_1 s_2)\}. \end{aligned}$$

If $\epsilon_{\text{in}}(s_1) \neq \emptyset$, we have that

$$\begin{aligned} T\Big(\partial_{(\sigma,\varepsilon)}(s_1 s_2)\Big) =& T\Big(\partial_{(\sigma,\varepsilon)}(s_1)s_2\Big) \cup T\Big(\epsilon_{\text{in}}(s_1)\partial_{(\sigma,\varepsilon)}(s_2)\Big) \\ \underset{\text{I.H.}}{=}& \{(w, w') \mid (\sigma w, w') \in T(s_1 s_2)\} \\ &\cup \{(\varepsilon, w_1') \mid (\varepsilon, w_1') \in T(s_1)\}\{(w_2, w_2') \mid (\sigma w_2, w_2') \in T(s_2)\} \\ =& \{(w, w') \mid (\sigma w, w') \in T(s_1 s_2)\} \\ &\cup \{(w_2, w_1' w_2') \mid (\varepsilon, w_1') \in T(s_1) \wedge (\sigma w_2, w_2') \in T(s_2)\}. \end{aligned}$$

But $\{(w_2, w_1' w_2') \mid (\varepsilon, w_1') \in T(s_1) \wedge (\sigma w_2, w_2') \in T(s_2)\} \subseteq \{(w, w') \mid (\sigma w, w') \in T(s_1 s_2)\}$.
Therefore, $T\Big(\partial_{(\sigma,\varepsilon)}(s_1 s_2)\Big) = \{(w, w') \mid (\sigma w, w') \in T(s_1 s_2)\}$.

3. **Case** $s_1^*$.

$$\begin{aligned} T\Big(\partial_{(\sigma,\varepsilon)}(s_1^*)\Big) &= T\Big(\partial_{(\sigma,\varepsilon)}(s_1)s_1^*\Big) \\ &\underset{\text{I.H.}}{=} \{(w, w') \mid (\sigma w, w') \in T(s_1)\}\, T(s_1^*) \\ &= \{(w_1, w_2)\,(w_1', w_2') \mid (\sigma w_1, w_2) \in T(s_1) \wedge (w_1', w_2') \in T(s_1^*)\} \\ &= \{(w_1 w_1', w_2 w_2') \mid (\sigma w_1 w_1', w_2 w_2') \in T(s_1 s_1^*)\}. \end{aligned}$$

Since $T(s_1^*) = T(s_1 s_1^* + (\varepsilon,\varepsilon)) = T(s_1 s_1^*) \cup T((\varepsilon,\varepsilon)) = T(s_1 s_1^*) \cup \{(\varepsilon,\varepsilon)\}$, and since $\sigma$ is not $\varepsilon$,

$$\begin{aligned}\partial_{(\sigma,\varepsilon)}(s_1^*) &= \{(w_1 w_1', w_2 w_2') \mid (\sigma w_1 w_1', w_2 w_2') \in T(s_1^*)\} \\ &= \{(w, w') \in T(s_1^*)\}\end{aligned}$$

This concludes the proof. □

The set of partial derivatives of a S2D-RE over $\Sigma \times \Delta$ w.r.t. a pair $(\sigma_1, \varepsilon)$ extends to the partial derivatives of a set of S2D-REs by

$$\partial_{(\sigma,\varepsilon)}(S) = \bigcup_{s \in S} \partial_{(\sigma,\varepsilon)}(s),$$

where $S \subseteq$ *S2D-RE*.

**Lemma 4.23.** *Let $s$ be a S2D-RE over $\Sigma \times \Delta$. The following equalities hold*

$$\begin{aligned}\epsilon_{in}(\epsilon_{out}(s)) &= \epsilon_{out}(\epsilon_{in}(s)) = \epsilon(s), \\ \epsilon_{in}\Big(\partial_{(\varepsilon,\sigma')}(s)\Big) &= \partial_{(\varepsilon,\sigma')}(\epsilon_{in}(s)), \\ \epsilon_{out}\Big(\partial_{(\sigma,\varepsilon)}(s)\Big) &= \partial_{(\sigma,\varepsilon)}(\epsilon_{out}(s)).\end{aligned}$$

*Proof.* For the first equality, we will only prove that $\epsilon_{\text{out}}(\epsilon_{\text{in}}(s)) = \epsilon(s)$ since the proof for $\epsilon_{\text{in}}(\epsilon_{\text{out}}(s)) = \epsilon(s)$ is analogous. We have that $T(\epsilon_{\text{out}}(\epsilon_{\text{in}}(s))) = \{(w,\varepsilon) \mid (w,\varepsilon) \in T(\epsilon_{\text{in}}(s))\}$, and since $T(\epsilon_{\text{in}}(s)) = \{(\varepsilon, w') \mid (\varepsilon, w') \in T(s)\}$, we have that $T(\epsilon_{\text{out}}(\epsilon_{\text{in}}(s))) = \{(\varepsilon,\varepsilon)\}$ if $T(\epsilon_{\text{in}}(s)) \neq \emptyset$ and $T(\epsilon_{\text{out}}(s)) \neq \emptyset$, which happens exactly when $(\varepsilon,\varepsilon) \in T(s)$, that is, if $\epsilon(s) = (\varepsilon,\varepsilon)$. Otherwise, we have that either $T(\epsilon_{\text{in}}(s)) = \emptyset$ or $T(\epsilon_{\text{out}}(s)) = \emptyset$, that is, $\epsilon(s) = \emptyset$. Thus, $\epsilon_{\text{out}}(\epsilon_{\text{in}}(s)) = \epsilon(s)$.

The second and third equalities have analogous proofs. Therefore, we will only explicitly prove $\epsilon_{\text{in}}\Big(\partial_{(\varepsilon,\sigma')}(s)\Big) = \partial_{(\varepsilon,\sigma')}(\epsilon_{\text{in}}(s))$. We have that

$$\begin{aligned}T\Big(\epsilon_{\text{in}}\Big(\partial_{(\varepsilon,\sigma')}(s)\Big)\Big) &= \Big\{(\varepsilon, w') \mid (\varepsilon, w') \in T\Big(\partial_{(\varepsilon,\sigma')}(s)\Big)\Big\} \\ &= \{(\varepsilon, w') \mid (\varepsilon, \sigma' w') \in T(s)\} \\ &= \{(w, w') \mid (w, \sigma' w') \in T(\epsilon_{\text{in}}(s))\} \\ &= T\Big(\partial_{(\varepsilon,\sigma')}(\epsilon_{\text{in}}(s))\Big).\end{aligned}$$

This concludes the proof. □

The set of partial derivatives of a S2D-RE $s$ over $\Sigma \times \Delta$ w.r.t. a pair $(w,\varepsilon)$, where $w \in \Sigma^*$, can be inductively defined by

$$\begin{aligned}\partial_{(\varepsilon,\varepsilon)}(s) &= \{s\}, \\ \partial_{(\sigma w,\varepsilon)}(s) &= \partial_{(w,\varepsilon)}\Big(\partial_{(\sigma,\varepsilon)}(s)\Big).\end{aligned}$$

Similarly, one can define the set of partial derivatives of a S2D-RE $s$ over $\Sigma \times \Delta$ w.r.t. a pair $(\varepsilon, w')$, where $w' \in \Delta^*$.

**Proposition 4.24.** *For any S2D-RE $s$ over $\Sigma \times \Delta$, the following equalities hold*

$$T\Big(\partial_{(w,\varepsilon)}(s)\Big) = \{(w_1, w') \mid (ww_1, w') \in T(s)\},$$
$$T\Big(\partial_{(\varepsilon,w')}(s)\Big) = \{(w, w'_1) \mid (w, w'w'_1) \in T(s)\}.$$

*Proof.* By induction on $|w|$. The first and second equalities are analogously proved, thus we will only explicitly prove the first one. The base case, when $w = \varepsilon$ is trivial. Now let us suppose the claim is true for some $w \in \Sigma^*$. Then, we have that

$$\begin{aligned} T\Big(\partial_{(\sigma w,\varepsilon)}(s)\Big) &= \Big\{(w_1, w'_1) \mid (\sigma w_1, w'_1) \in T\Big(\partial_{(w,\varepsilon)}(s)\Big)\Big\} \\ &\underset{\text{I.H.}}{=} \{(w_1, w'_1) \mid (\sigma w_1, w') \in \{(w_2, w') \mid (ww_2, w') \in T(s)\}\} \\ &= \{(w_1, w'_1) \mid (\sigma w w_1, w') \in T(s)\}. \end{aligned}$$

□

We can also extend the set of partial derivatives of a S2D-RE $s$ over $\Sigma \times \Delta$ w.r.t. a pair $(w, w') \in \Sigma^* \times \Delta^*$ by

$$\partial_{(w,w')}(s) = \partial_{(w,\varepsilon)}\Big(\partial_{(\varepsilon,w')}(s)\Big).$$

**Proposition 4.25.** *For any S2D-RE $s$ over $\Sigma \times \Delta$, we have that*

$$T\Big(\partial_{(w,\varepsilon)}\Big(\partial_{(\varepsilon,w')}(s)\Big)\Big) = \{(w_1, w'_1) \mid (ww_1, w'w'_1) \in T(s)\}.$$

*Proof.* Follows directly from Proposition 4.24. □

**Corollary 4.26.** *For any S2D-RE $s$ over $\Sigma\times\Delta$ and $(w, w') \in \Sigma^*\times\Delta^*$, we have that $(w, w') \in T(s)$ if and only if $\epsilon\Big(\partial_{(w,w')}(s)\Big) = (\varepsilon, \varepsilon)$.*

*Proof.* Follows directly from Proposition 4.25. □

### 4.1.2 Linear Form

We will now define the linear form of a S2D-RE $s$. Later in this chapter we will present a method, using this notion, to obtain a transducer that realizes the same relation as $s$ that on average has less states than the result using Thompson's method.

**Definition 4.27.** Let $s$ be a S2D-RE over $\Sigma \times \Delta$. The *linear form* of $s$, denoted $\mathrm{lf}(s)$ is defined

recursively on the structure of $s$ as follows

$$\begin{aligned}
\mathrm{lf}(\emptyset) = \mathrm{lf}((\varepsilon,\varepsilon)) &= \emptyset,\\
\mathrm{lf}((\sigma,\varepsilon)) &= \{((\sigma,\varepsilon),(\varepsilon,\varepsilon))\},\\
\mathrm{lf}((\varepsilon,\sigma')) &= \{((\varepsilon,\sigma'),(\varepsilon,\varepsilon))\},\\
\mathrm{lf}((\sigma,\sigma')) &= \{((\sigma,\sigma'),(\varepsilon,\varepsilon))\},\\
\mathrm{lf}(s_1 + s_2) &= \mathrm{lf}(s_1) \cup \mathrm{lf}(s_2),\\
\mathrm{lf}(s_1 \cdot s_2) &= \begin{cases} \mathrm{lf}(s_1)\cdot s_2, & \text{if } \epsilon(s_1) = \emptyset,\\ \mathrm{lf}(s_1)\cdot s_2 \cup \mathrm{lf}(s_2), & \text{otherwise,}\end{cases}\\
\mathrm{lf}(s_1^*) &= \mathrm{lf}(s_1)\cdot s_1^*,
\end{aligned}$$

where $\mathrm{lf}(s_1)\cdot s_2 = \{(p, s\cdot s_2) \mid (p,s) \in \mathrm{lf}(s_1)\}$ and $s_1$ and $s_2$ are also S2D-REs over $\Sigma\times\Delta$, $\sigma\in\Sigma$, $\sigma'\in\Delta$. To the first element in each pair of lf we call *head* and we call *tail* to the second element.

The case for $(\sigma,\sigma')$ is not necessary, since we have that $(\sigma,\sigma') = (\sigma,\varepsilon)\,(\varepsilon,\sigma')$, but this produces a more succinct set. Thus, and in order to obtain all possible linear forms for $(\sigma,\sigma')$, we can also define the *extended linear form*, denoted $\mathrm{lf}_{\mathrm{ext}}(s)$ in the exact same way as we defined the linear form, except that

$$\mathrm{lf}_{\mathrm{ext}}((\sigma,\sigma')) = \{((\sigma,\varepsilon),(\varepsilon,\sigma')),((\varepsilon,\sigma'),(\sigma,\varepsilon)),((\sigma,\sigma'),(\varepsilon,\varepsilon))\}, \text{ and}$$

$$\mathrm{lf}_{\mathrm{ext}}((\sigma,\sigma')\cdot s_1) = \{((\sigma,\varepsilon),(\varepsilon,\sigma')\,s_1),((\varepsilon,\sigma'),(\sigma,\varepsilon)\,s_1),((\sigma,\sigma'),s_1)\}.$$

**Proposition 4.28.** *For any S2D-RE $s$ over $\Sigma\times\Delta$, $lf(s)$ is such that*

$$s \sim \bigcup_{((\tau,\tau'),s')\in lf(s)} (\tau,\tau')\,s' \cup \epsilon(s),$$

*where $\tau\in\Sigma\cup\{\varepsilon\}$ and $\tau'\in\Delta\cup\{\varepsilon\}$.*

*Proof.* The proof follows by induction on the structure of $s$. The base cases $\emptyset$, $(\sigma,\varepsilon)$, $(\varepsilon,\sigma')$, $(\sigma,\sigma')$ trivially hold. Let us now suppose that the claim is true for some $s_1$ and $s_2$.

1. **Case** $s_1 + s_2$.

   We have that $\mathrm{lf}(s_1+s_2) = \mathrm{lf}(s_1)\cup\mathrm{lf}(s_2)$. By induction hypothesis, we know that

$$s_1 \sim \bigcup_{((\tau_1,\tau_1'),s_1')\in \mathrm{lf}(s_1)} (\tau_1,\tau_1')\,s_1' \cup \epsilon(s_1) \text{ and}$$

$$s_2 \sim \bigcup_{((\tau_2,\tau_2'),s_2')\in \mathrm{lf}(s_2)} (\tau_2,\tau_2')\,s_2' \cup \epsilon(s_2).$$

Therefore,

$$\begin{aligned} s_1 + s_2 &\sim \bigcup_{((\tau_1,\tau_1'),s_1')\in \mathrm{lf}(s_1)} (\tau_1,\tau_1')\, s_1' \cup \epsilon(s_1) \cup \bigcup_{((\tau_2,\tau_2'),s_2')\in \mathrm{lf}(s_2)} (\tau_2,\tau_2')\, s_2' \cup \epsilon(s_2) \\ &= \bigcup_{((\tau_1,\tau_1'),s_1')\in \mathrm{lf}(s_1)} (\tau_1,\tau_1')\, s_1' \cup \bigcup_{((\tau_2,\tau_2'),s_2')\in \mathrm{lf}(s_2)} (\tau_2,\tau_2')\, s_2' \cup \epsilon(s_1) \cup \epsilon(s_2) \\ &= \bigcup_{((\tau_1,\tau_1'),s_1')\in \mathrm{lf}(s_1)} (\tau_1,\tau_1')\, s_1' \cup \bigcup_{((\tau_2,\tau_2'),s_2')\in \mathrm{lf}(s_2)} (\tau_2,\tau_2')\, s_2' \cup \epsilon(s_1 + s_2) \\ &= \bigcup_{((\tau,\tau'),s')\in \mathrm{lf}(s_1+s_2)} (\tau,\tau')\, s' \cup \epsilon(s_1 + s_2). \end{aligned}$$

2. **Case** $s_1 s_2$.

   Note that $\mathrm{lf}(s_1 s_2)$ can also be defined as $\mathrm{lf}(s_1)s_2 \cup \epsilon(s_1)\mathrm{lf}(s_2)$. By induction hypothesis, we have that

$$\begin{aligned} s_1 \cdot s_2 &\sim \left( \bigcup_{((\tau_1,\tau_1'),s_1')\in \mathrm{lf}(s_1)} (\tau_1,\tau_1')\, s_1' \cup \epsilon(s_1) \right) \cdot \left( \bigcup_{((\tau_2,\tau_2'),s_2')\in \mathrm{lf}(s_2)} (\tau_2,\tau_2')\, s_2' \cup \epsilon(s_2) \right) \\ &\sim \bigcup_{((\tau_1,\tau_1'),s_1')\in \mathrm{lf}(s_1)} (\tau_1,\tau_1')\, s_1' s_2 \cup \epsilon(s_1) \bigcup_{((\tau_2,\tau_2'),s_2')\in \mathrm{lf}(s_2)} (\tau_2,\tau_2')\, s_2' \cup \epsilon(s_1)\epsilon(s_2) \\ &= \bigcup_{((\tau_1,\tau_1'),s_1')\in \mathrm{lf}(s_1)} (\tau_1,\tau_1')\, s_1' s_2 \cup \epsilon(s_1) \bigcup_{((\tau_2,\tau_2'),s_2')\in \mathrm{lf}(s_2)} (\tau_2,\tau_2')\, s_2' \cup \epsilon(s_1 s_2), \end{aligned}$$

   and since $\mathrm{lf}(s_1)s_2 \cup \epsilon(s_1)\mathrm{lf}(s_2) = \mathrm{lf}(s_1 s_2)$, we have that

$$s_1 s_2 \sim \bigcup_{((\tau,\tau'),s')\in \mathrm{lf}(s_1 s_2)} (\tau,\tau')\, s' \cup \epsilon(s_1 s_2).$$

3. **Case** $s_1^*$.

   We have that $s_1^* \sim s_1 s_1^* + (\varepsilon,\varepsilon)$, and we can suppose that $\epsilon(s_1) = \emptyset$. By induction hypothesis, we have that

$$\begin{aligned} s_1^* &\sim s_1 s_1^* + (\varepsilon,\varepsilon) \\ &\sim \bigcup_{((\tau,\tau'),s')\in \mathrm{lf}(s_1)} (\tau,\tau')\, s' s_1^* \cup \epsilon(s_1) s_1^* \cup \{(\varepsilon,\varepsilon)\}. \end{aligned}$$

   But $\mathrm{lf}(s_1^*) = \mathrm{lf}(s_1)s_1^*$, $\epsilon(s_1) = \emptyset$ and $\epsilon(s_1^*) = (\varepsilon,\varepsilon)$, thus

$$s_1^* \sim \bigcup_{((\tau,\tau'),s')\in \mathrm{lf}(s_1^*)} (\tau,\tau')\, s' \cup \epsilon(s_1^*).$$

This concludes the proof. □

## 4.2 General 2D-Regular Expressions

In this section, we will discuss some definitions and results particular to G2D-REs. The definition of S2D-constant part naturally extends to G2D-REs, where for the base case $(r_1, r_2)$, we have

$$\epsilon((r_1, r_2)) = \begin{cases} (\varepsilon, \varepsilon)\,, \text{ if } \epsilon(r_1) = \epsilon(r_2) = \varepsilon, \\ \emptyset, \text{ otherwise.} \end{cases}$$

We can extend this definition to a set of G2D-REs $G$ by

$$\epsilon(G) = \begin{cases} (\varepsilon, \varepsilon)\,, \text{ if } \exists g \in G : \epsilon(g) = (\varepsilon, \varepsilon)\,, \\ \emptyset, \text{ otherwise.} \end{cases}$$

**Definition 4.29.** Let $g$ be a G2D-RE over $\Sigma \times \Delta$. The *G2D input-constant part* of $g$, denoted by $\epsilon_{\text{in}}(g)$, is defined recursively on the structure of $g$ as follows:

$$\begin{aligned} \epsilon_{\text{in}}(\emptyset) &= \emptyset; \\ \epsilon_{\text{in}}((r_1, r_2)) &= \begin{cases} (\varepsilon, r_2)\,, \text{ if } \epsilon(r_1) = \varepsilon; \\ \emptyset, \text{ otherwise;} \end{cases} \\ \epsilon_{\text{in}}(g_1 + g_2) &= \epsilon_{\text{in}}(g_1) + \epsilon_{\text{in}}(g_2); \\ \epsilon_{\text{in}}(g_1 g_2) &= \epsilon_{\text{in}}(g_1)\epsilon_{\text{in}}(g_2); \\ \epsilon_{\text{in}}(g^*) &= (\epsilon_{\text{in}}(g))^*\,, \end{aligned}$$

for every $r_1$ over $\Sigma$ and $r_2$ over $\Delta$.

One can also define the *G2D-RE output-constant part* of $g$, in the exact same way as $\epsilon_{\text{in}}(g)$, except that

$$\epsilon_{\text{out}}((r_1, r_2)) = \begin{cases} (r_1, \varepsilon)\,, \text{ if } \epsilon(r_2) = \varepsilon, \\ \emptyset, \text{ otherwise.} \end{cases}$$

**Lemma 4.30.** *Let $g$ be a G2D-RE over $\Sigma \times \Delta$. Then $T(\epsilon_{in}(g)) = T(g) \cap (\{\varepsilon\} \times \Delta^*)$ and $T(\epsilon_{out}(g)) = T(g) \cap (\Sigma^* \times \{\varepsilon\})$.*

*Proof.* The proofs for the base case $\emptyset$ and the concatenation, union and Kleene star cases follow the same structure as the proof of Lemma 4.19.

The base case $(r_1, r_2)$ follows directly from the definition of $\epsilon_{\text{in}}((r_1, r_2))$. □

### 4.2.1 Partial Derivatives

We can extend the concatenation operation from G2D-REs to sets of G2D-REs, like we did for S2D-REs.

**Definition 4.31.** Let $g$ be a G2D-RE over $\Sigma \times \Delta$. The set of *partial derivatives* of $g$ with respect to a pair $(\sigma, \varepsilon)$, where $\sigma \in \Sigma$ ($(\varepsilon, \sigma')$, where $\sigma' \in \Delta$), denoted $\partial_{(\sigma,\varepsilon)}(g)$ ($\partial_{(\varepsilon,\sigma')}(g)$), is defined recursively on the structure of $g$ as follows:

$$\begin{aligned}
\partial_{(\sigma,\varepsilon)}(\emptyset) &= \partial_{(\varepsilon,\sigma')}(\emptyset) = \emptyset,\\
\partial_{(\sigma,\varepsilon)}((r_1, r_2)) &= \partial_\sigma(r_1) \times \{r_2\},\\
\partial_{(\varepsilon,\sigma')}((r_1, r_2)) &= \{r_1\} \times \partial_{\sigma'}(r_2),\\
\partial_{(\sigma,\varepsilon)}(g_1 + g_2) &= \partial_{(\sigma,\varepsilon)}(g_1) \cup \partial_{(\sigma,\varepsilon)}(g_2),\\
\partial_{(\varepsilon,\sigma')}(g_1 + g_2) &= \partial_{(\varepsilon,\sigma')}(g_1) \cup \partial_{(\varepsilon,\sigma')}(g_2),\\
\partial_{(\sigma,\varepsilon)}(g_1 g_2) &= \partial_{(\sigma,\varepsilon)}(g_1) g_2 \cup \epsilon_{\text{in}}(g_1)\partial_{(\sigma,\varepsilon)}(g_2),\\
\partial_{(\varepsilon,\sigma')}(g_1 g_2) &= \partial_{(\varepsilon,\sigma')}(g_1) g_2 \cup \epsilon_{\text{out}}(g_1)\partial_{(\varepsilon,\sigma')}(g_2),\\
\partial_{(\sigma,\varepsilon)}(g_1^*) &= \partial_{(\sigma,\varepsilon)}(g_1) g_1^*,\\
\partial_{(\varepsilon,\sigma')}(g_1^*) &= \partial_{(\varepsilon,\sigma')}(g_1) g_1^*,
\end{aligned}$$

where $r_1$, $r_2$ are REs over $\Sigma$ and $\Delta$, respectively, and $g_1$ and $g_2$ are also G2D-REs over $\Sigma \times \Delta$.

We can trivially extend Lemma 4.22 to G2D-REs.

**Lemma 4.32.** *Let $g$ be a G2D-RE over $\Sigma \times \Delta$. The relations defined by $\partial_{(\sigma,\varepsilon)}(g)$ and $\partial_{(\varepsilon,\sigma')}(g)$, denoted by $T\left(\partial_{(\sigma,\varepsilon)}(g)\right)$ and $T\left(\partial_{(\varepsilon,\sigma')}(g)\right)$, respectively, verify the following equalities*

$$\begin{aligned}
T\left(\partial_{(\sigma,\varepsilon)}(g)\right) &= \{(w, w') \mid (\sigma w, w') \in T(g)\},\\
T\left(\partial_{(\varepsilon,\sigma')}(g)\right) &= \{(w, w') \mid (w, \sigma' w') \in T(g)\}.
\end{aligned}$$

*Proof.* The proof follows the same structure as the proof for Lemma 4.22, for the $\emptyset$, concatenation, union and Kleene star.

For the base case $(r_1, r_2)$, where $r_1$, $r_2$ are REs over $\Sigma$ and $\Delta$, respectively, we have that:

$$\begin{aligned}
T\left(\partial_{(\sigma,\varepsilon)}((r_1, r_2))\right) &= \left\{(w, w') \mid (w, w') \in \partial_{(\sigma,\varepsilon)}((r_1, r_2))\right\}\\
&= \{(w, w') \mid (w, w') \in \partial_\sigma(r_1) \times \{r_2\}\}\\
&= \{(w, w') \mid w \in \partial_\sigma(r_1) \wedge w' \in \mathcal{L}(r_2)\}\\
&= \{(w, w') \mid \sigma w \in \mathcal{L}(r_1) \wedge w' \in \mathcal{L}(r_2)\}\\
&= \{(w, w') \mid (\sigma w, w') \in \mathcal{L}(r_1) \times \mathcal{L}(r_2)\}\\
&= \{(w, w') \mid (\sigma w, w') \in T((r_1, r_2))\}.
\end{aligned}$$

Analogously, one can prove that $T\left(\partial_{(\varepsilon,\sigma')}((r_1, r_2))\right) = \{(w, w') \mid (w, \sigma' w') \in T((r_1, r_2))\}$. □

The set of partial derivatives of a G2D-RE over $\Sigma \times \Delta$ w.r.t. a pair $(\sigma_1, \varepsilon)$ extends to the partial derivatives of a set of G2D-REs by

$$\partial_{(\sigma,\varepsilon)}(G) = \bigcup_{g \in G} \partial_{(\sigma,\varepsilon)}(g),$$

where $G \subseteq$ *G2D-RE*.

**Lemma 4.33.** *Let $g$ be a G2D-RE over $\Sigma \times \Delta$. The following equalities hold*

$$\begin{aligned}\epsilon_{in}(\epsilon_{out}(g)) &= \epsilon_{out}(\epsilon_{in}(g)) = \epsilon(g),\\ \epsilon_{in}\Big(\partial_{(\varepsilon,\sigma')}(g)\Big) &= \partial_{(\varepsilon,\sigma')}(\epsilon_{in}(g)),\\ \epsilon_{out}\Big(\partial_{(\sigma,\varepsilon)}(g)\Big) &= \partial_{(\sigma,\varepsilon)}(\epsilon_{out}(g)).\end{aligned}$$

*Proof.* Analogous to proof of Lemma 4.23. □

**Proposition 4.34.** *For any G2D-RE $g$ over $\Sigma \times \Delta$, the following equalities hold*

$$\begin{aligned}T\Big(\partial_{(w,\varepsilon)}(g)\Big) &= \{(w_1, w') \mid (ww_1, w') \in T(g)\},\\ T\Big(\partial_{(\varepsilon,w')}(g)\Big) &= \{(w, w'_1) \mid (w, w'w'_1) \in T(g)\}.\end{aligned}$$

*Proof.* Analogous to proof of Proposition 4.24. □

We can also extend the set of partial derivatives of a G2D-RE $g$ over $\Sigma \times \Delta$ w.r.t. a pair $(w, w') \in \Sigma^* \times \Delta^*$ by

$$\partial_{(w,w')}(g) = \partial_{(w,\varepsilon)}\Big(\partial_{(\varepsilon,w')}(g)\Big).$$

**Proposition 4.35.** *For any G2D-RE $g$ over $\Sigma \times \Delta$, we have that*

$$T\Big(\partial_{(w,\varepsilon)}\Big(\partial_{(\varepsilon,w')}(g)\Big)\Big) = \{(w_1, w'_1) \mid (ww_1, w'w'_1) \in T(g)\}.$$

*Proof.* Follows directly from Proposition 4.34. □

**Corollary 4.36.** *For any G2D-RE $g$ over $\Sigma \times \Delta$ and $(w, w') \in \Sigma^* \times \Delta^*$, we have that $(w, w') \in T(g)$ if and only if $\epsilon\Big(\partial_{(w,w')}(g)\Big) = (\varepsilon, \varepsilon)$.*

*Proof.* Follows directly from Proposition 4.35. □

### 4.2.2 Linear Form

We will now define the linear form of a G2D-RE $g$. Later in this chapter we will present a method for obtaining a transducer that realizes the same relation as $g$ that uses the linear form of $g$.

**Definition 4.37.** Let $r_1$ and $r_2$ be REs over $\Sigma$ and $\Delta$, respectively. Then the linear form of $(r_1, r_2)$ is defined by

$$\mathrm{lf}((r_1, r_2)) = \mathrm{lf}(r_1) \times \mathrm{lf}(r_2) \cup \epsilon((\varepsilon, r_2))\,(\mathrm{lf}(r_1) \times \{(\varepsilon, \varepsilon)\}) \cup \epsilon((r_1, \varepsilon))\,(\{(\varepsilon, \varepsilon)\} \times \mathrm{lf}(r_2)),$$

where $A \times B = \{((\sigma, \sigma'), (r, r')) \mid (\sigma, r) \in A \wedge (\sigma', r') \in B\}$.

The subexpression $\epsilon((\varepsilon, r_2))\,(\mathrm{lf}(r_1) \times \{(\varepsilon, \varepsilon)\})$ appears from the fact that, regardless of the value of $\epsilon(r_1)$, if $\epsilon(r_2) = \varepsilon$, then we need to include $\mathrm{lf}(r_1) \times \{(\varepsilon, \varepsilon)\}$.

**Proposition 4.38.** *For any G2D-RE $g$ over $\Sigma \times \Delta$, $lf(g)$ is such that*

$$g \sim \bigcup_{((\tau,\tau'),g') \in lf(g)} (\tau, \tau')\, g' \cup \epsilon(g).$$

*Proof.* The proof follows the same structure as the proof for Proposition 4.28, therefore we will only explicitly prove the base case $(r_1, r_2)$. But since $r_1 \sim \bigcup_{(\sigma,r_1') \in \mathrm{lf}(r_1)} \sigma r_1' \cup \epsilon(r_1)$ and $r_2 \sim \bigcup_{(\sigma,r_2') \in \mathrm{lf}(r_2)} \sigma r_2' \cup \epsilon(r_2)$, we have that

$$\begin{aligned}
(r_1, r_2) &\sim \left( \bigcup_{(\sigma,r_1') \in \mathrm{lf}(r_1)} \sigma r_1' \cup \epsilon(r_1) \right) \times \left( \bigcup_{(\sigma,r_2') \in \mathrm{lf}(r_2)} \sigma r_2' \cup \epsilon(r_2) \right) \\
&\sim \left( \bigcup_{(\sigma,r_1') \in \mathrm{lf}(r_1)} \sigma r_1' \right) \times \left( \bigcup_{(\sigma,r_2') \in \mathrm{lf}(r_2)} \sigma r_2' \right) \cup \left( \bigcup_{(\sigma,r_1') \in \mathrm{lf}(r_1)} \sigma r_1' \right) \times \epsilon(r_2) \\
&\quad \cup \epsilon(r_1) \times \left( \bigcup_{(\sigma,r_2') \in \mathrm{lf}(r_2)} \sigma r_2' \right) \cup \epsilon(r_1) \times \epsilon(r_2),
\end{aligned}$$

we have that $(r_1, r_2) \sim \bigcup_{((\tau,\tau'),(r_1',r_2')) \in \mathrm{lf}((r_1,r_2))} (\tau, \tau')\, (r_1', r_2') \cup \epsilon((r_1, r_2))$. □

### 4.2.3 Converting a G2D-RE into a S2D-RE

**Lemma 4.39.** *Let $r_1, r_2$ be regular expressions over $\Sigma$ and $r_3, r_4$ regular expressions over $\Delta$. Then, the following equivalences hold*

$$(r_1 r_2, r_3 r_4) \sim (r_1, r_3)\,(r_2, r_4)\,, \tag{4.5}$$
$$(r_1, r_3) \sim (r_1, \varepsilon)\,(\varepsilon, r_3)\,, \tag{4.6}$$
$$(r_1^*, \varepsilon) \sim (r_1, \varepsilon)^*\,, \tag{4.7}$$
$$(\varepsilon, r_3^*) \sim (\varepsilon, r_3)^*\,, \tag{4.8}$$
$$(r_1 + r_2, r_3) \sim (r_1, r_3) + (r_2, r_3)\,, \tag{4.9}$$
$$(r_1, r_3 + r_4) \sim (r_1, r_3) + (r_1, r_4)\,. \tag{4.10}$$

*Proof.* (4.5) Follows directly from the concatenation case of Definition 4.5.

(4.6) Particular case of equivalence (4.5).

(4.7) Observe that $T((r_1, \varepsilon)^*) = \bigcup_n T((r_1, \varepsilon)^n)$. Now note that, from equivalence (4.5), we have $(r_1, \varepsilon)^2 = (r_1, \varepsilon)\,(r_1, \varepsilon) = (r_1 r_1, \varepsilon\varepsilon) = (r_1^2, \varepsilon)$. In fact, we can prove by induction that $(r_1, \varepsilon)^n \sim (r_1^n, \varepsilon)$, for every $n \geq 0$. Thus, $(r_1^*, \varepsilon) \sim (r_1, \varepsilon)^*$.

(4.8) Analogous to equivalence (4.7).

(4.9) Note that, by definition, $T((r_1, r_3) + (r_2, r_3)) = T((r_1, r_3)) \cup T((r_2, r_3))$, which is, by definition again, $\mathcal{L}(r_1) \times \mathcal{L}(r_3) \cup \mathcal{L}(r_2) \times \mathcal{L}(r_3)$. By distributivity property, we get

$$\mathcal{L}(r_1) \times \mathcal{L}(r_3) \cup \mathcal{L}(r_2) \times \mathcal{L}(r_3) = (\mathcal{L}(r_1) \cup \mathcal{L}(r_2)) \times \mathcal{L}(r_3),$$

which is, by definition, $T((r_1 + r_2, r_3))$.

(4.10) Analogous to equivalence (4.9). □

Lemma 4.39 gives the tools we need to convert a G2D-RE in a S2D-RE that realizes the same relation, as we will see later in this subsection.

Before defining this, let us define an application $f$ that allows us to obtain a S2D-RE equivalent to a given G2D-RE $g$ over $\Sigma \times \Delta$, which is presented as follows

$$\begin{aligned}
f((\varepsilon, \varepsilon)) &= (\varepsilon, \varepsilon)\,, \\
f((\sigma, \varepsilon)) &= (\sigma, \varepsilon)\,, \\
f((\varepsilon, \sigma')) &= (\varepsilon, \sigma')\,, \\
f((\sigma, \sigma')) &= (\sigma, \sigma')\,, \\
f((r_1^*, \varepsilon)) &= f((r_1, \varepsilon))^{\circledast}, \\
f((\varepsilon, r_2^*)) &= f((\varepsilon, r_2))^{\circledast}, \\
f((r_1, r_2)) &= f((r_1, \varepsilon)) \odot f((\varepsilon, r_2)), \\
f((r_1 r_1', r_2 r_2')) &= f((r_1, r_2)) \odot f((r_1', r_2')), \\
f((r_1 + r_1', r_2)) &= f((r_1, r_2)) \oplus f((r_1', r_2)), \\
f((r_1, r_2 + r_2')) &= f((r_1, r_2)) \oplus f((r_1, r_2')), \\
f(g \odot g') &= f(g) \odot f(g'), \\
f(g \oplus g') &= f(g) \oplus f(g'), \\
f(g^{\circledast}) &= f(g)^{\circledast},
\end{aligned}$$

for every $\sigma \in \Sigma$, $\sigma' \in \Delta$, $r_1$ and $r_2$ REs over $\Sigma$, $r_2$ and $r_2'$ REs over $\Delta$, and where $g'$ is also a G2D-RE over $\Sigma \times \Delta$. The rule $f((r_1, r_2)) = f((r_1, \varepsilon)) \odot f((\varepsilon, r_2))$ is only needed if either $r_1 \equiv r^*$ or $r_2 \equiv r^*$.

We now present some examples. Note that the order in which we present the rules gives a deterministic method for converting a G2D-RE into a S2D-RE.

**Example 4.40.** Let us convert $((ab)^* + c, (a + b)^*)$ into an equivalent S2D-RE.

Applying the application described, we have

$$\begin{aligned}((ab)^* + c, (a+b)^*) &\sim ((ab)^*, (a+b)^*) + (c, (a+b)^*) \\ &\sim ((ab)^*, \varepsilon)\,(\varepsilon, (a+b)^*) + (c, (a+b)^*) \\ &\sim (ab, \varepsilon)^*\,(\varepsilon, (a+b)^*) + (c, (a+b)^*) \\ &\sim ((a,\varepsilon)\,(b,\varepsilon))^*\,(\varepsilon, (a+b)^*) + (c, (a+b)^*) \\ &\sim ((a,\varepsilon)\,(b,\varepsilon))^*\,(\varepsilon, a+b)^* + (c, (a+b)^*) \\ &\sim ((a,\varepsilon)\,(b,\varepsilon))^*\,((\varepsilon, a) + (\varepsilon, b))^* + (c, (a+b)^*) \\ &\sim ((a,\varepsilon)\,(b,\varepsilon))^*\,((\varepsilon, a) + (\varepsilon, b))^* + (c, \varepsilon)\,(\varepsilon, (a+b)^*) \\ &\sim ((a,\varepsilon)\,(b,\varepsilon))^*\,((\varepsilon, a) + (\varepsilon, b))^* + (c, \varepsilon)\,(\varepsilon, a+b)^* \\ &\sim ((a,\varepsilon)\,(b,\varepsilon))^*\,((\varepsilon, a) + (\varepsilon, b))^* + (c, \varepsilon)\,((\varepsilon, a) + (\varepsilon, b))^*\end{aligned}$$

which is a S2D-RE.

**Example 4.41.** Let us convert $(a + b + c^*, (a+b)\,c)$ into an equivalent S2D-RE.

First, observe that $(a+b, a+b) \sim (a,a) + (a,b) + (b,a) + (b,b)$. Now, we have

$$\begin{aligned}&(a + b + c^*, (a+b)\,c) \\ \sim\ &(a + b + c^*, a+b)\,(\varepsilon, c) \\ \sim\ &((a+b, a+b) + (c^*, a+b))\,(\varepsilon, c) \\ \sim\ &(((a,a) + (a,b) + (b,a) + (b,b)) + (c^*, a+b))\,(\varepsilon, c) \\ \sim\ &((a,a) + (a,b) + (b,a) + (b,b) + (c^*, a+b))\,(\varepsilon, c) \\ \sim\ &((a,a) + (a,b) + (b,a) + (b,b) + ((c^*, a) + (c^*, b)))\,(\varepsilon, c) \\ \sim\ &((a,a) + (a,b) + (b,a) + (b,b) + (c^*, a) + (c^*, b))\,(\varepsilon, c) \\ \sim\ &((a,a) + (a,b) + (b,a) + (b,b) + (c^*, \varepsilon)\,(\varepsilon, a) + (c^*, b))\,(\varepsilon, c) \\ \sim\ &((a,a) + (a,b) + (b,a) + (b,b) + (c, \varepsilon)^*\,(\varepsilon, a) + (c^*, b))\,(\varepsilon, c) \\ \sim\ &((a,a) + (a,b) + (b,a) + (b,b) + (c, \varepsilon)^*\,(\varepsilon, a) + (c^*, \varepsilon)\,(\varepsilon, b))\,(\varepsilon, c) \\ \sim\ &((a,a) + (a,b) + (b,a) + (b,b) + (c, \varepsilon)^*\,(\varepsilon, a) + (c, \varepsilon)^*\,(\varepsilon, b))\,(\varepsilon, c)\end{aligned}$$

which is a S2D-RE.

**Example 4.42.** Let us convert $((ab)^* + c, (a+b)^*)\,(a + b + c^*, (a+b)\,c)$ in an equivalent S2D-RE.

Since

$$((ab)^* + c, (a+b)^*) \sim ((a,\varepsilon)\,(b,\varepsilon))^*\,((\varepsilon, a) + (\varepsilon, b))^* + (c, \varepsilon)\,((\varepsilon, a)\,(\varepsilon, b))^*$$

and

$$(a + b + c^*, (a+b)\,c) \sim ((a,a) + (a,b) + (b,a) + (b,b) + (c, \varepsilon)^*\,(\varepsilon, a) + (c, \varepsilon)^*\,(\varepsilon, b))\,(\varepsilon, c)$$

we conclude that

$$\begin{aligned}&((ab)^* + c, (a+b)^*)\,(a+b+c^*, (a+b)\,c)\\ \sim\,&((a,\varepsilon)\,(b,\varepsilon))^*\,((\varepsilon,a)+(\varepsilon,b))^* + (c,\varepsilon)\,((\varepsilon,a)\,(\varepsilon,b))^*\\ &((a,a)+(a,b)+(b,a)+(b,b)+(c,\varepsilon)^*\,(\varepsilon,a)+(c,\varepsilon)^*\,(\varepsilon,b))\,(\varepsilon,c)\end{aligned}$$

which is a S2D-RE.

**Example 4.43.** Let us convert $((ab)^* + c, (a+b)^*)^*$ in an equivalent S2D-RE. Since

$$((ab)^* + c, (a+b)^*) \sim ((a,\varepsilon)\,(b,\varepsilon))^*\,((\varepsilon,a)+(\varepsilon,b))^* + (c,\varepsilon)\,((\varepsilon,a)\,(\varepsilon,b))^*$$

we conclude that

$$((ab)^* + c, (a+b)^*)^* \sim (((a,\varepsilon)\,(b,\varepsilon))^*\,((\varepsilon,a)+(\varepsilon,b))^* + (c,\varepsilon)\,((\varepsilon,a)\,(\varepsilon,b))^*)^*$$

which is a S2D-RE.

The method that we will present now always produces a normalised S2D-RE. The above method, on the other hand, can produce a S2D-RE that is not necessarily normalised.

**Theorem 4.44.** *For every G2D-RE, there is a normalised S2D-RE equivalent to it.*

*Proof.* Let $r_1$, $r_2$ be two REs over $\Sigma$ and $r_3$, $r_4$ be two REs over $\Delta$. Let us consider the following equivalences

$$(r_1, r_3) \sim (r_1,\varepsilon)\,(\varepsilon, r_3) \tag{4.11}$$
$$(r_1 + r_2, \varepsilon) \sim (r_1,\varepsilon) + (r_2,\varepsilon) \tag{4.12}$$
$$(\varepsilon, r_3 + r_4) \sim (\varepsilon, r_3) + (\varepsilon, r_4) \tag{4.13}$$
$$(r_1 r_2, \varepsilon) \sim (r_1,\varepsilon)\,(r_2,\varepsilon) \tag{4.14}$$
$$(\varepsilon, r_3 r_4) \sim (\varepsilon, r_3)\,(\varepsilon, r_4) \tag{4.15}$$
$$(r_1^*, \varepsilon) \sim (r_1,\varepsilon)^* \tag{4.16}$$
$$(\varepsilon, r_3^*) \sim (\varepsilon, r_3)^* \tag{4.17}$$

Let $g$ be a G2D-RE over $\Sigma \times \Delta$. Let us prove by induction that $g$ has a normalised S2D-RE equivalent to it.

The base cases $(\varepsilon,\varepsilon)$, $(\sigma,\varepsilon)$ and $(\varepsilon,\sigma')$ are trivially S2D-REs. For $(\sigma,\sigma')$ we only need to apply (4.11).

For the case $(r_1, r_3)$, it is trivial to prove by induction that the equivalences are sufficient to produce a normalised S2D-RE equivalent to $g$. The idea is to use (4.11) and then use (4.12)–(4.17) according to the symbol appearing in the expression.

Now suppose that we have two G2D-RE $g_1$, $g_2$ over $\Sigma \times \Delta$. By induction hypothesis, $g_1$ and $g_2$ both have normalised S2D-REs equivalent to them, let them be $g_1'$ and $g_2'$. Therefore, $g_1 g_2$, $g_1 + g_2$ and $g_1^*$ have normalised S2D-REs $g_1' g_2'$, $g_1' + g_2'$ and ${g_1'}^*$, respectively.

Note that except for (4.11), which is only applied once for each atom, all the equivalences produce an expression which is obtained from the one on the left side with less operators on the RE-level than the ones on the right side. Therefore, we know that this method terminates.

This concludes the proof. □

**Theorem 4.45.** *Let $g$ be a G2D-RE. The normalised S2D-RE $s$ equivalent to $g$, obtained from Theorem 4.44, verifies the following equalities:*

1. $|s|_{\odot} = |g|_{\odot} + |g|_{.} + |g|_{\Sigma\times\Delta}$.
2. $|s|_{\oplus} = |g|_{\oplus} + |g|_{+}$.
3. $|s|_{\circledast} = |g|_{\circledast} + |g|_{*}$.
4. $|s|_{\Sigma\times\Delta} = |g|_{\Sigma} + |g|_{\Delta}$.

*Proof.* Let us prove this by induction.

The base cases $(\varepsilon,\varepsilon)$, $(\sigma,\varepsilon)$, $(\varepsilon,\sigma')$ and $(\sigma,\sigma')$ are left to the reader.

For the case $(r,r')$, where $r$ and $r'$ are REs over $\Sigma$ and $\Delta$, the equalities follow directly from Theorem 4.44, since the number of operations $\oplus$, $\odot$ and $\circledast$ is 0. Remember that for $|(r,r')|_{\Sigma\times\Delta}$ to be non 0 both $r$ and $r'$ need to be different from $\varepsilon$.

Now suppose that $g$ and $g'$, with equivalent normalised S2D-REs $s$ and $s'$, respectively, satisfy the equalities. We now need to consider three cases. We will only prove one of them since the other two are analogous.

Let us consider the case $gg'$.

$$\begin{aligned}
|ss'|_{\odot} &= |s|_{\odot} + |s'|_{\odot} + 1 \\
&\underset{\text{I.H.}}{=} \left(|g|_{\odot} + |g|_{.} + |g|_{\Sigma\times\Delta}\right) + \left(|g'|_{\odot} + |g'|_{.} + |g'|_{\Sigma\times\Delta}\right) + 1 \\
&= \left(|g|_{\odot} + |g'|_{\odot} + 1\right) + \left(|g|_{.} + |g'|_{.}\right) + \left(|g|_{\Sigma\times\Delta} + |g'|_{\Sigma\times\Delta}\right) \\
&= |gg'|_{\odot} + |gg'|_{.} + |gg'|_{\Sigma\times\Delta}
\end{aligned}$$

which proves 1.

$$\begin{aligned}
|ss'|_{\oplus} &= |s|_{\oplus} + |s'|_{\oplus} \\
&\underset{\text{I.H.}}{=} \left(|g|_{\oplus} + |g|_{+}\right) + \left(|g'|_{\oplus} + |g'|_{+}\right) \\
&= \left(|g|_{\oplus} + |g'|_{\oplus}\right) + \left(|g|_{+} + |g'|_{+}\right) \\
&= |gg'|_{\oplus} + |gg'|_{+}
\end{aligned}$$

which proves 2.

$$\begin{aligned}|ss'|_{\circledast} &= |s|_{\circledast} + |s'|_{\circledast} \\ &\underset{\text{I.H.}}{=} (|g|_{\circledast} + |g|_{*}) + \left(|g'|_{\circledast} + |g'|_{*}\right) \\ &= \left(|g|_{\circledast} + |g'|_{\circledast}\right) + (|g|_{*} + |g'|_{*}) \\ &= |gg'|_{\circledast} + |gg'|_{*}\end{aligned}$$

which proves 3.

$$\begin{aligned}|ss'|_{\Sigma\times\Delta} &= |s|_{\Sigma\times\Delta} + |s'|_{\Sigma\times\Delta} \\ &\underset{\text{I.H.}}{=} (|g|_{\Sigma} + |g|_{\Delta}) + (|g'|_{\Sigma} + |g'|_{\Delta}) \\ &= (|g|_{\Sigma} + |g'|_{\Sigma}) + (|g|_{\Delta} + |g'|_{\Delta}) \\ &= |gg'|_{\Sigma} + |gg'|_{\Delta}\end{aligned}$$

which proves 4, thus concluding the proof. □

## 4.3 Equivalence Between S2D-REs and Transducers

In this section we will discuss some methods for obtaining a S2D-RE that realizes the same relation as a given transducer and vice-versa. These methods are, all in all, extensions or modifications of methods for automata.

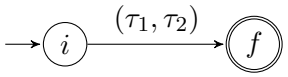
### 4.3.1 Extension of Thompson's Method to Transducers

In this subsection, we will see that the Thompson's method for converting a regular expression into an automaton can be extended to transducers, that is, we can adapt this method for converting a S2D-RE into a transducer that realizes the exact same relation.

**Theorem 4.46.** *Given a S2D-RE, there is a standard form transducer that realizes the same relation.*

*Proof.* The proof follows by induction.

Let us consider the base case, when the S2D-RE is simply $(\tau_1, \tau_2)$, where $\tau_1 \in (\Sigma \cup \{\varepsilon\})$ and $\tau_2 \in (\Delta \cup \{\varepsilon\})$. Consider, then, the following transducer $\dot{t}_{(\tau_1,\tau_2)}$

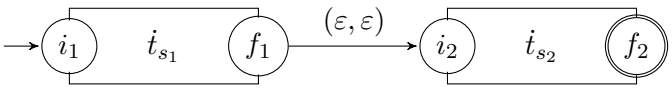



We can see trivially that $T((\tau_1, \tau_2)) = R\left(\dot{t}_{(\tau_1,\tau_2)}\right)$.

Suppose that we have, for the following cases, two S2D-RE $s_1$ and $s_2$, and let $\dot{t}_{s_1}$ and $\dot{t}_{s_2}$ be the transducers for the expressions $s_1$ and $s_2$, respectively. We know that, by construction, $\dot{t}_{s_1}$ and $\dot{t}_{s_2}$ have only one initial and one final state.

Let us also consider that we have that $T(s_1) = R(\dot{t}_{s_1})$ and $T(s_2) = R(\dot{t}_{s_2})$. The transducers presented follow the convention that the initial states for $\dot{t}_{s_1}$ and $\dot{t}_{s_2}$ are $i_1$ and $i_2$, respectively, and that $f_1$ and $f_2$ are the final states, respectively. The states $i$ and $f$ are new initial and final states that we need to add, respectively.

Consider the following transducer, $\dot{t}_{s_1 s_2}$




Let $(w, w') \in T(s_1 s_2)$. Then, by definition, there are $x, x', y, y'$ such that $w = xx'$ and $w' = yy'$, and $(x, y) \in T(s_1)$ and $(x', y') \in T(s_2)$. By hypothesis, $(x, y) \in R(\dot{t}_{s_1})$ and $(x', y') \in R(\dot{t}_{s_2})$. Then, $(x, y)\,(\varepsilon, \varepsilon)\,(x', y') \in R(\dot{t}_{s_1 s_2})$.

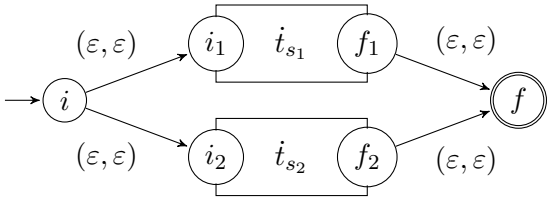
But $(x, y)\,(\varepsilon, \varepsilon)\,(x', y') = (x, y)\,(x', y') = (xx', yy') = (w, w')$. Hence, $(w, w') \in R(\dot{t}_{s_1 s_2})$.

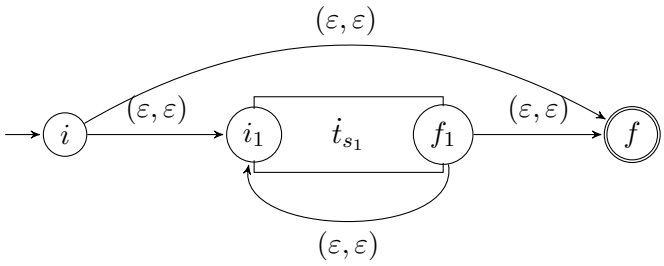
Conversely, let us consider that $(w, w') \in R(\dot{t}_{s_1 s_2})$. Then, by definition of path, there are $x, x', y, y'$ such that $w = xx'$ and $w' = yy'$, and $(x, y) \in R(\dot{t}_{s_1})$ and $(x', y') \in R(\dot{t}_{s_2})$. By induction hypothesis, $(x, y) \in T(s_1)$ and $(x', y') \in T(s_2)$. Then $(x, y)\,(\varepsilon, \varepsilon)\,(x', y') \in T(s_1 s_2)$.

But $(x, y)\,(\varepsilon, \varepsilon)\,(x', y') = (x, y)\,(x', y') = (xx', yy') = (w, w')$. Hence, $(w, w') \in T(s_1 s_2)$.

This proves that $T(s_1 s_2) = R(\dot{t}_{s_1 s_2})$.

Consider the following two transducer, $\dot{t}_{s_1 + s_2}$ and $\dot{t}_{s_1^*}$, respectively




It can be proved in a similar way that $T(s_1 + s_2) = R(\dot{t}_{s_1 + s_2})$ and $T(s_1^*) = R\left(\dot{t}_{s_1^*}\right)$. □

*Remark* 4.47. For any S2D-RE, the transducer obtained by applying the Thompson's method is a SFT. If the S2D-RE is normalised, then the transducer obtained is a NFT.

### 4.3.2 Extension of the State Elimination Method to Transducers

The state elimination method for obtaining a RE from a given automaton can be extended to transducers in order to obtain a S2D-RE from a given SFT.

**Definition 4.48.** A *generalized transducer* (or Regular Expression Labeled transducer) is a sextuple

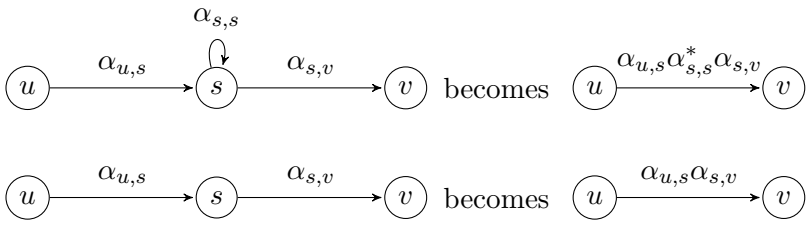
$$\dot{t} = (Q, \Sigma, \Delta, \delta, I, F)$$

where $Q$, $I$, $F$, $\Sigma$, $\Delta$ are exactly the same as those defined for SFTs, and $\delta \subseteq Q \times \textit{S2D-RE} \to Q$ is the finite set of transitions.

**Theorem 4.49.** *Given a SFT, there is a S2D-RE that realizes the same relation.*

*Proof.* The state elimination method works as follows

- If the transducer has more than one initial state, $i_1, \ldots, i_n$, add a new initial state $i$ and $(\varepsilon, \varepsilon)$ transitions from $i$ to $i_1, \ldots, i_n$;
- Remove all the states that are not accessible or coaccessible;
- If there are no final states left, return as result $\emptyset$ and terminate;
- If the initial state $s_0$ has in-degree not 0, add a new initial state $i$ and an $(\varepsilon, \varepsilon)$ transition from $i$ to $s_0$;
- If the transducer has several final states, $f_1, \ldots, f_n$, add a new final state $f$ which will be the only final state, with $(\varepsilon, \varepsilon)$ transitions from $f_1, \ldots, f_n$ to $f$;
- If the final state $f$ has out-degree not 0, add a new final state $f'$, which will be the only final state, and an $(\varepsilon, \varepsilon)$ transition from $f$ to $f'$;
- Convert the transducer into a generalized one, that is, replace the labels of the transitions with S2D-RE. This means that when a transition from a state to another has more than one atom as its label, namely $(\sigma_1, \sigma_1'), \ldots, (\sigma_n, \sigma_n')$, the label of the transition becomes $\sum_{i=1}^{n} (\sigma_i, \sigma_i')$;
- Delete the states, one by one, with exception to the initial and final states. To delete a state $s$, replace each transitions $(u, \alpha_{u,s}, s)$ and $(s, \alpha_{s,v}, v)$, with $u \neq s$ and $s \neq v$ by a new transiton from $u$ to $v$ which label is as follows




If a transition from $u$ to $v$ already exists, with label $\alpha_{u,v}$, replace it with $\left(\alpha_{u,v} + \alpha_{u,s}\alpha^*_{s,s}\alpha_{s,v}\right)$ or $(\alpha_{u,v} + \alpha_{u,s}\alpha_{s,v})$, respectively.

This method terminates when the transducer has only two states, the initial and final states. The S2D-RE equivalent to the SFT is the expression that labels the only transition from the initial state to the final state.

The correction of this algorithm trivially follows from definition of path. □

## 4.4 Applications of Linear Form

In this section, we discuss some methods to obtain a transducer from a given 2D-RE, either standard or general. The methods are applications of the notion of linear form.

### 4.4.1 Conversion from S2D-REs fo SFTs

**Definition 4.50.** Let $s$ be a S2D-RE over $\Sigma_1 \times \Delta_1$. The *linear-form-transducer* of $s$, denoted by $\dot{t}_{\mathrm{lf}}(s)$, is a transducer such that $\dot{t}_{\mathrm{lf}}(s) = (Q, \Sigma, \Delta, \delta, I, F)$ and $Q = \pi(s) \cup \{s\}$, $I = \{s\}$, $F = \{s_1 \mid s_1 \in Q \wedge \epsilon(s_1)\} = \{(\varepsilon, \varepsilon)\}$, $\delta = \{(s_1, (\sigma_1, \sigma_2), s_2) \mid ((\sigma_1, \sigma_2), s_2) \in \mathrm{lf}(s_1) \wedge s_1 \in Q\}$, $\Sigma = \Sigma_1$ and $\Delta = \Delta_1$.

The set $\pi(s)$ is a set that satisfies the following proposition, which is proved alongside Proposition 4.54.

**Proposition 4.51.** *Let $s$ be a S2D-RE over $\Sigma \times \Delta$. Then the set $\pi(s)$ satisfies*

$$\begin{aligned}
\pi(\emptyset) &= \emptyset, \\
\pi((\varepsilon, \varepsilon)) &= \emptyset, \\
\pi((\sigma, \varepsilon)) &= \{(\varepsilon, \varepsilon)\}, \\
\pi((\varepsilon, \sigma')) &= \{(\varepsilon, \varepsilon)\}, \\
\pi((\sigma, \sigma')) &= \{(\varepsilon, \varepsilon)\}, \\
\pi(s_1 + s_2) &= \pi(s_1) \cup \pi(s_2), \\
\pi(s_1 s_2) &= \pi(s_1) s_2 \cup \pi(s_2), \\
\pi(s_1^*) &= \pi(s_1) s_1^*.
\end{aligned}$$

**Lemma 4.52.** *Let $s$ be a S2D-RE over $\Sigma \times \Delta$. The set $Tr(s)$, inductively defined by*

$$\begin{aligned} Tr(\emptyset) &= Tr((\varepsilon,\varepsilon)) = \emptyset, \\ Tr((\sigma,\varepsilon)) &= Tr((\varepsilon,\sigma')) = Tr((\sigma,\sigma')) = \emptyset, \\ Tr(s_1 + s_2) &= Tr(s_1) \cup Tr(s_2), \\ Tr(s_1 s_2) &= Tr(s_1)s_2 \cup Tr(s_2) \cup F(s_1)s_2 \times \mathit{lf}(s_2), \\ Tr(s_1^*) &= Tr(s_1)s_1^* \cup F(s_1)s_1^* \times \mathit{lf}(s_1^*), \end{aligned}$$

*is such that $Tr(s) = \{(s_1, (\tau,\tau'), s_2) \mid ((\tau,\tau'), s_2) \in \mathit{lf}(s_1) \wedge s_1 \in \pi(s)\}$. The set $F(s)$ is the set of final states in the linear-form-transducer of $s$, that is, $F(s) = \{s' \mid s' \in \pi(s) \wedge \epsilon(s') = (\varepsilon,\varepsilon)\}$.*

*Proof.* We follow by induction on the structure of $s$. The base cases $\emptyset$ and $(\varepsilon,\varepsilon)$ are trivial. The cases $(\sigma,\varepsilon)$, $(\varepsilon,\sigma')$ and $(\sigma,\sigma')$ are analogous to each other, so we will only explicitly present the last one.

We have that $\pi((\sigma,\sigma')) = \{(\varepsilon,\varepsilon)\}$ and $\mathrm{lf}((\varepsilon,\varepsilon)) = \emptyset$.

Therefore, $\{(s_1, (\sigma,\sigma'), s_2) \mid ((\sigma,\sigma'), s_2) \in \mathrm{lf}(s_1) \wedge s_1 \in \{(\varepsilon,\varepsilon)\}\} = \emptyset$.

Now let us suppose the claim is true for some 2D-REs $s_1$ and $s_2$.

- **Case $s_1 + s_2$.**

  We have that

$$\begin{aligned} Tr(s_1 + s_2) =& Tr(s_1) \cup Tr(s_2) \\ \underset{\text{I.H.}}{=}& \{(s_1', (\tau,\tau'), s_2') \mid ((\tau,\tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_1)\} \\ & \cup \{(s_1', (\tau,\tau'), s_2') \mid ((\tau,\tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_2)\} \\ =& \{(s_1', (\tau,\tau'), s_2') \mid ((\tau,\tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_1) \cup \pi(s_2)\} \\ =& \{(s_1', (\tau,\tau'), s_2') \mid ((\tau,\tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_1 + s_2)\}. \end{aligned}$$

- **Case $s_1 s_2$.**

  We have that

$$\begin{aligned} Tr(s_1 s_2) =& Tr(s_1)s_2 \cup Tr(s_2) \cup F(s_1)s_2 \times \mathrm{lf}(s_2) \\ \underset{\text{I.H.}}{=}& \{(s_1', (\tau,\tau'), s_2') \mid ((\tau,\tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_1)\}\{s_2\} \\ & \cup \{(s_1', (\tau,\tau'), s_2') \mid ((\tau,\tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_2)\} \cup F(s_1)s_2 \times \mathrm{lf}(s_2) \\ =& \{(s_1's_2, (\tau,\tau'), s_2's_2) \mid ((\tau,\tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_1)\} \\ & \cup F(s_1)s_2 \times \mathrm{lf}(s_2) \cup \{(s_1', (\tau,\tau'), s_2') \mid ((\tau,\tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_2)\}. \end{aligned}$$

  Since $F(s_1)s_2 \times \mathrm{lf}(s_2) = \{(s_1's_2, (\tau,\tau'), s_2') \mid s_1' \in F(s_1) \wedge ((\tau,\tau'), s_2') \in \mathrm{lf}(s_2)\}$ and $\mathrm{lf}(s_1 s_2) = \mathrm{lf}(s_1)s_2 \cup \epsilon(s_1)\mathrm{lf}(s_2)$, we have that

$$\begin{aligned} & \{(s_1's_2, (\tau,\tau'), s_2's_2) \mid ((\tau,\tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_1)\} \cup F(s_1)s_2 \times \mathrm{lf}(s_2) \\ =& \{(s_1'', (\tau,\tau'), s_2'') \mid ((\tau,\tau'), s_2'') \in \mathrm{lf}(s_1'') \wedge s_1'' \in \pi(s_1)s_2\}. \end{aligned}$$

Thus,

$$\begin{aligned} Tr(s_1 s_2) &= \{(s_1'', (\tau, \tau'), s_2'') \mid ((\tau, \tau'), s_2'') \in \mathrm{lf}(s_1'') \wedge s_1'' \in \pi(s_1)s_2\} \\ &\quad \cup \{(s_1'', (\tau, \tau'), s_2'') \mid ((\tau, \tau'), s_2'') \in \mathrm{lf}(s_1'') \wedge s_1'' \in \pi(s_2)\} \\ &= \{(s_1'', (\tau, \tau'), s_2'') \mid ((\tau, \tau'), s_2'') \in \mathrm{lf}(s_1'') \wedge s_1'' \in \pi(s_1)s_2 \cup \pi(s_2)\} \\ &= \{(s_1'', (\tau, \tau'), s_2'') \mid ((\tau, \tau'), s_2'') \in \mathrm{lf}(s_1'') \wedge s_1'' \in \pi(s_1 s_2)\}. \end{aligned}$$

- **Case $s_1^*$.**

  We have that

$$\begin{aligned} Tr(s_1^*) &= Tr(s_1)s_1^* \cup F(s_1)s_1^* \times \mathrm{lf}(s_1^*) \\ &\underset{\text{I.H.}}{=} \{(s_1', (\tau, \tau'), s_2') \mid ((\tau, \tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_1)\} s_1^* \cup F(s_1)s_1^* \times \mathrm{lf}(s_1^*) \\ &= \{(s_1' s_1^*, (\tau, \tau'), s_2' s_1^*) \mid ((\tau, \tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_1)\} \cup F(s_1)s_1^* \times \mathrm{lf}(s_1^*). \end{aligned}$$

  Since $F(s_1)s_1^* \times \mathrm{lf}(s_1^*) = \{(s_1' s_1^*, (\tau, \tau'), s_2') \mid s_1' \in F(s_1) \wedge ((\tau, \tau'), s_2') \in \mathrm{lf}(s_1^*)\}$ and $\mathrm{lf}(s_1^*) = \mathrm{lf}(s_1)s_1^*$, we have that

$$\begin{aligned} &\{(s_1' s_1^*, (\tau, \tau'), s_2' s_1^*) \mid ((\tau, \tau'), s_2') \in \mathrm{lf}(s_1') \wedge s_1' \in \pi(s_1)\} \cup F(s_1)s_1^* \times \mathrm{lf}(s_1^*) \\ &= \{(s_1'', (\tau, \tau'), s_2'') \mid ((\tau, \tau'), s_2'') \in \mathrm{lf}(s_1'') \wedge s_1'' \in \pi(s_1)s_1^*\}. \end{aligned}$$

  Thus,

$$\begin{aligned} Tr(s_1^*) &= \{(s_1'', (\tau, \tau'), s_2'') \mid ((\tau, \tau'), s_2'') \in \mathrm{lf}(s_1'') \wedge s_1'' \in \pi(s_1)s_1^*\} \\ &= \{(s_1'', (\tau, \tau'), s_2'') \mid ((\tau, \tau'), s_2'') \in \mathrm{lf}(s_1'') \wedge s_1'' \in \pi(s_1^*)\}. \end{aligned}$$

This concludes the proof. □

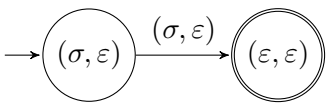
**Corollary 4.53.** *For any S2D-RE $s$,*

$$Tr(s) \cup s \times \mathit{lf}(s) = \{(s_1, (\tau, \tau'), s_2) \mid ((\tau, \tau'), s_2) \in \mathit{lf}(s_1) \wedge s_1 \in \pi(s) \cup \{s\}\}.$$

*Proof.* Trivial from Lemma 4.52. □

**Proposition 4.54.** *Let $s$ be a S2D-RE over $\Sigma \times \Delta$. Then $T(s) = R(\dot{t}_{\mathit{lf}}(s))$.*

*Proof.* The proof follows by induction on the structure of $s$. The case $\emptyset$ is trivial. The cases $(\sigma, \varepsilon)$, $(\varepsilon, \sigma')$ and $(\sigma, \sigma')$ are analogous and therefore we will only explicitly show the first one. It is obvious that $T((\sigma, \varepsilon)) = \{(\sigma, \varepsilon)\}$. On the other hand, the linear-form-transducer of $(\sigma, \varepsilon)$ is such that, by definition, $Q = \pi((\sigma, \varepsilon)) \cup \{(\sigma, \varepsilon)\} = \{(\varepsilon, \varepsilon)\} \cup \{(\sigma, \varepsilon)\} = \{(\sigma, \varepsilon), (\varepsilon, \varepsilon)\}$, $I = \{(\sigma, \varepsilon)\}$, $F = \{(\varepsilon, \varepsilon)\}$ and $\delta = Tr((\sigma, \varepsilon)) \cup (\sigma, \varepsilon) \times \mathrm{lf}((\sigma, \varepsilon)) = \{((\sigma, \varepsilon), (\sigma, \varepsilon), (\varepsilon, \varepsilon))\}$, and has the following representation.



It is obvious that $R(\dot{t}_{\text{lf}}((\sigma,\varepsilon))) = \{(\sigma,\varepsilon)\}$.

Now let us suppose that we have $s_1$ and $s_2$ such that $T(s_1) = R(\dot{t}_{\text{lf}}(s_1))$ and $T(s_2) = R(\dot{t}_{\text{lf}}(s_2))$, and let $Q_i$, $I_i$, $F_i$ and $\delta_i$, be the set of states, the set of initial states, the set of final states, and the set of transitions of $\dot{t}_{\text{lf}}(s_i)$, for $i = 0, 1$.

- **Case** $s_1 + s_2$.

  We construct a transducer $\dot{t}$ from $\dot{t}_{\text{lf}}(s_1)$ and $\dot{t}_{\text{lf}}(s_2)$ such that $Q = \pi(s_1) \cup \pi(s_2) \cup \{s_1 + s_2\} = \pi(s_1 + s_2) \cup \{s_1 + s_2\}$. On the other hand, $I = \{s_1 + s_2\}$ and $F = F_1' \cup F_2' \cup \epsilon(s_1 + s_2)\{s_1 + s_2\}$, where $F_i' = F_i$ if $s_i \notin F_i$ or if $s_i \in F_i$ and $s_i \in \pi(s_i)$, and $F_i' = F_i \setminus \{s_i\}$, otherwise. Therefore, $F = \{q \in Q \mid \epsilon(q) = \{(\varepsilon,\varepsilon)\}\}$. Finally, $\delta = Tr(s_1 + s_2) \cup (s_1 + s_2) \times \text{lf}(s_1 + s_2)$, and from Corollary 4.53 this is $\{(s_1', (\tau,\tau'), s_2') \mid ((\tau,\tau'), s_2') \in \text{lf}(s_1') \wedge s_1' \in Q\}$. Therefore, $\dot{t}$ is a linear-form-transducer for $s_1 + s_2$.

  We know that $T(s_1 + s_2) = T(s_1) \cup T(s_2)$. Therefore, for a pair of words $(w, w')$ such that $w \in \Sigma^*$ and $w' \in \Delta^*$ we know that if $(w, w') \in T(s_1 + s_2)$ then either $(w, w') \in T(s_1)$ or $(w, w') \in T(s_2)$. W.l.o.g., let us consider that $(w, w') \in T(s_1)$. Thus, by induction hypothesis, we know that there is an accepting path $(s_1, (\tau_1, \tau_1'), s_1', (\tau_2, \tau_2'), \ldots, s_n')$, where $((\tau_1, \tau_1'), s_1') \in \text{lf}(s_1)$, where $w = \tau_1 \cdots \tau_n$, $w' = \tau_1' \cdots \tau_n'$, and by definition we know that $s_1' \in \pi(s_1)$. Therefore, $s_1' \in Q$. On the other hand, $\text{lf}(s_1 + s_2) = \text{lf}(s_1) \cup \text{lf}(s_2)$, thus $((\tau_1, \tau_1'), s_1') \in \text{lf}(s_1 + s_2)$ and by construction $(s_1', (\tau_2, \tau_2'), \ldots, s_n')$ is a valid path from $s_1'$ in $\dot{t}$ and since $s_n' \in F_1'$, we have that $s_n' \in F$, that is, the path where we replace the first state $s_1$ by $s_1 + s_2$ is an accepting path in $\dot{t}$ as well. Therefore, $(w, w') \in R(\dot{t})$.

  Conversely, we can prove that if $(w, w') \in R(\dot{t})$ then $(w, w') \in T(s_1 + s_2)$.

- **Case** $s_1 s_2$.

  We construct a transducer $\dot{t}$ from $\dot{t}_{\text{lf}}(s_1)$ and $\dot{t}_{\text{lf}}(s_2)$ such that $Q = \pi(s_1)s_2 \cup \pi(s_2) \cup \{s_1 s_2\} = \pi(s_1 s_2) \cup \{s_1 s_2\}$. On the other hand, $I = \{s_1 s_2\}$ and $F = \epsilon(s_2)F_1 s_2 \cup F_2 \cup \epsilon(s_1 s_2)\{s_1 s_2\} = \{s \in Q \mid \epsilon(s) = \{(\varepsilon,\varepsilon)\}\}$. Finally, $\delta = Tr(s_1 s_2) \cup (s_1 s_2) \times \text{lf}(s_1 s_2)$, and we know from Corollary 4.53 that this is $\{(s_1', (\tau,\tau'), s_2') \mid ((\tau,\tau'), s_2') \in \text{lf}(s_1') \wedge s_1' \in Q\}$. Therefore, $\dot{t}$ is a linear-form-transducer for $s_1 s_2$.

  We know that $T(s_1 s_2) = T(s_1)T(s_2)$. Therefore, for a pair of words $(w, w')$ such that $w \in \Sigma^*$ and $w' \in \Delta^*$ we know that if $(w, w') \in T(s_1 s_2)$, then there exist $x, x' \in \Sigma^*$ and $y, y' \in \Delta^*$ such that $w = xx'$, $w' = yy'$, $(x, y) \in T(s_1)$ and $(x', y') \in T(s_2)$. Thus, by induction hypothesis, we know that there is one accepting path $(s_1, (\tau_1, \tau_1'), s_1', \ldots, (\tau_n, \tau_n'), s_n')$ in $\dot{t}_{\text{lf}}(s_1)$ such that $x = \tau_1 \cdots \tau_n$, $y = \tau_1' \cdots \tau_n'$, $((\tau_1, \tau_1'), s_1') \in \text{lf}(s_1)$ and $s_n' \in F_1$, and one accepting path $(s_2, (\tau_{n+1}, \tau_{n+1}'), s_{n+1}', \ldots, (\tau_m, \tau_m'), s_m')$ in $\dot{t}_{\text{lf}}(s_2)$ such that $x' = \tau_{n+1} \cdots \tau_m$, $y' = \tau_{n+1}' \cdots \tau_m'$, $((\tau_{n+1}, \tau_{n+1}'), s_{n+1}') \in \text{lf}(s_2)$ and $s_m' \in F_2$, which means that $s_m' \in F$, where $m > n$. By construction, we know that if we take the first path and concatenate $s_2$ to each of the states in such path, we obtain a valid path in $\dot{t}$ from the initial state $s_1 s_2$ to a state $s_n' s_2$. Also by construction, if we take the second path and we concatenate the state $s_2$ to $s_n'$ we obtain a valid path in $\dot{t}$ from $s_n' s_2$ to a final state $s_m''$. Therefore, the path $(s_1 s_2, (\tau_1, \tau_1'), s_1' s_2, \ldots, s_n' s_2, (\tau_{n+1}, \tau_{n+1}'), s_{n+1}', \ldots, (\tau_m, \tau_m'), s_m')$ is a valid path in $\dot{t}$

where $s_1 s_2$ is an initial state and $s'_m$ a final state in $\dot{t}$, thus the path is an accepting one. Therefore, $(xx', yy') = (w, w') \in R(\dot{t})$.

Conversely, we can prove that if $(w, w') \in R(\dot{t})$ then $(w, w') \in T(s_1 s_2)$.

- **Case** $s_1^*$.

  We construct a transducer $\dot{t}$ from $\dot{t}_{\text{lf}}(s_1)$ such that $Q = \pi(s_1)s_1^* \cup \{s_1^*\} = \pi(s_1^*) \cup \{s_1^*\}$. $I = \{s_1^*\}$ and $F = F_1 s_1^* \cup \{s_1^*\} = \{s \in Q \mid \epsilon(s) = \{(\varepsilon, \varepsilon)\}\}$. Finally, $\delta = Tr(s_1^*) \cup s_1^* \times \text{lf}(s_1^*)$, and from Corollary 4.53 we know that this is $\{(s'_1, (\tau, \tau'), s'_2) \mid ((\tau, \tau'), s'_2) \in \text{lf}(s'_1) \wedge s'_1 \in Q\}$. Therefore, $\dot{t}$ is a linear-form-transducer for $s_1^*$.

  We know that $T(s_1^*) = (T(s_1))^*$. Therefore, for a pair of words $(w, w')$ such that $w \in \Sigma^*$ and $w' \in \Delta^*$ we know that if $(w, w') \in T(s^*)$ then there exist $x_1, \ldots, x_n \in \Sigma^*$ and $y_1, \ldots, y_n \in \Delta^*$ such that $w = x_1 \cdots x_n$ and $w' = y_1 \cdots y_n$ and each $(x_i, y_i) \in T(s_1)$. Thus, by induction hypothesis, we know that for each $(x_i, y_i)$ there is an accepting path $p_i$ in $\dot{t}_{\text{lf}}(s_1)$

  $$\left(s_1^*, (\tau_{1i}, \tau'_{1i}), s'_{1i}, \ldots, \left(\tau_{m_i i}, \tau'_{m_i i}\right), s'_{m_i i}\right)$$

  where $x_i = \tau_{1i} \cdots \tau_{m_i i}$ and $y_i = \tau'_{1i} \cdots \tau'_{m_i i}$. Now let $p'_i$ be a path obtained from $p_i$ by concatenating $s_1^*$ to each state, except the first one. On the other hand, given the final state $s'_{m_i} s_1^*$ in $p'_i$, we replace the first state in $p'_{i+1}$ by that final state $s'_{m_i i} s_1^*$. By construction, we know that each $p'_i$ is a valid path in $\dot{t}$.

  We now construct a path $p$ from each $p'_i$ such that we "concatenate" each $p'_i$ to $p'_{i+1}$, that is, since the final state of each $p'_i$ is the first state of $p'_{i+1}$, the "concatenation" of $p'_i$ and $p'_{i+1}$ is

  $$\left(s'_{m_{i-1} i-1} s_1^*, (\tau_{1i}, \tau'_{1i}), \ldots, \left(\tau_{m_i i}, \tau'_{m_i i}\right), s'_{m_i i} s_1^*, \left(\tau_{1(i+1)}, \tau'_{1(i+1)}\right), \ldots, s'_{m_{i+1} i+1}\right).$$

  Thus, $p$ is a valid path in $\dot{t}$. Since $p$ is from $s_1^* \in I$ to $s'_{m_n n} s_1^* \in F_1 s_1^*$, which means that $s'_{m_n n} s_1^* \in F$, we have that $p$ is an accepting path in $\dot{t}$. Therefore, $(x_1 \cdots x_n, y_1 \cdots y_n) = (w, w') \in R(\dot{t})$.

  Conversely, we can prove that if $(w, w') \in R(\dot{t})$ then $(w, w') \in T(s_1^*)$.

This concludes the proof. □

**Theorem 4.55.** *For any S2D-RE $s$ the following inequality hold*

$$|\pi(s) \cup \{s\}| \leq |s|_{\Sigma \cup \Delta} + |s|_{\Sigma \times \Delta} + 1.$$

*Proof.* Follows directly from Proposition 4.54. □

### 4.4.2 Conversion from G2D-REs fo SFTs

**Definition 4.56.** Let $g$ be a G2D-RE over $\Sigma_1 \times \Delta_1$. The *linear-form-transducer* of $g$, denoted by $\dot{t}_{\mathrm{lf}}(g)$, is a transducer such that $\dot{t}_{\mathrm{lf}}(g) = (Q, \Sigma, \Delta, \delta, I, F)$ and $Q = \pi(g) \cup \{g\}$, $I = \{g\}$, $F = \{g_1 \mid g_1 \in Q \wedge \epsilon(g_1)\} = \{(\varepsilon, \varepsilon)\}$, $\delta = \{(g_1, (\tau_1, \tau_2), g_2) \mid ((\tau_1, \tau_2), g_2) \in \mathrm{lf}(g_1) \wedge g_1 \in Q\}$, $\Sigma = \Sigma_1$ and $\Delta = \Delta_1$.

The proof for the following proposition is provided alongside proof for Proposition 4.60.

**Proposition 4.57.** *Let $g$ be a G2D-RE over $\Sigma \times \Delta$. Then the set $\pi(g)$ satisfies*

$$\begin{aligned}
\pi(\emptyset) &= \emptyset, \\
\pi((r_1, r_2)) &= \pi(r_1) \times \pi(r_2) \cup \pi(r_1) \times \{\varepsilon\} \cup \{\varepsilon\} \times \pi(r_2), \\
\pi(g_1 + g_2) &= \pi(g_1) \cup \pi(g_2), \\
\pi(g_1 g_2) &= \pi(g_1) g_2 \cup \pi(g_2), \\
\pi(g_1^*) &\subseteq \pi(g_1) g_1^*,
\end{aligned}$$

*where, for any $r$ over an arbitrary alphabet $\Sigma_1$, $\pi(r)$ satisfies*

$$\begin{aligned}
\pi(\emptyset) = \pi(\varepsilon) &= \emptyset, \\
\pi(\sigma) &= \{\varepsilon\}, \\
\pi(r_1 + r_2) &= \pi(r_1) \cup \pi(r_2), \\
\pi(r_1 r_2) &= \pi(r_1) r_2 \cup \pi(r_2), \\
\pi(r_1^*) &= \pi(r_1) r_1^*.
\end{aligned}$$

**Lemma 4.58.** *Let $g$ be a G2D-RE over $\Sigma \times \Delta$. The set $Tr(g)$, inductively defined by*

$$\begin{aligned}
Tr(\emptyset) &= \emptyset \\
Tr((r_1, r_2)) &= \{(g_1, (\tau, \tau'), g_2) \mid ((\tau, \tau'), g_2) \in \mathit{lf}(g_1) \wedge g_1 \in \pi((r_1, r_2))\} \\
Tr(g_1 + g_2) &= Tr(g_1) \cup Tr(g_2) \\
Tr(g_1 g_2) &= Tr(g_1) g_2 \cup Tr(g_2) \cup F(g_1) g_2 \times \mathit{lf}(g_2) \\
Tr(g_1^*) &= Tr(g_1) g_1^* \cup F(g_1) g_1^* \times \mathit{lf}(g_1^*)
\end{aligned}$$

*is such that $Tr(g) = \{(g_1, (\tau, \tau'), g_2) \mid ((\tau, \tau'), g_2) \in \mathit{lf}(g_1) \wedge g_1 \in \pi(g)\}$.*

*Proof.* This follows by induction on the structure of $g$. The base cases are trivial and the other cases are proved exactly as in Lemma 4.52. □

**Corollary 4.59.** *For any G2D-RE $g$,*

$$Tr(g) \cup g \times \mathit{lf}(g) = \{(g_1, (\tau, \tau'), g_2) \mid ((\tau, \tau'), g_2) \in \mathit{lf}(g_1) \wedge g_1 \in \pi(g) \cup \{g\}\}.$$

*Proof.* Trivial from Lemma 4.58. □

**Proposition 4.60.** *Let $g$ be a G2D-RE over $\Sigma \times \Delta$. Then $T(g) = R\left(\dot{t}_{lf}(g)\right)$.*

*Proof.* This follows by induction on the structure of $g$. We will only show the base cases since the rest of the proof follows the same structure as the proof for Proposition 4.54.

The case $\emptyset$ is trivial. For the case $(r_1, r_2)$, let us suppose that we have $(w, w') \in T((r_1, r_2))$. By definition, we know that $w \in \mathcal{L}(r_1)$ and $w' \in \mathcal{L}(r_2)$. From Proposition 4.38 and by construction, we know that if $w = \tau_1 \cdots \tau_n$ and $w' = \tau'_1 \cdots \tau'_n$, where each $\tau_i \in \Sigma \cup \{\varepsilon\}$ and each $\tau'_i \in \Delta \cup \{\varepsilon\}$, then there is a path in $\dot{t}_{\text{lf}}((r_1, r_2))$ from the state $(r_1, r_2)$ to the state $(\varepsilon, \varepsilon)$ with label $(\tau_1 \cdots \tau_n, \tau'_1 \cdots \tau'_n)$, and since the state $(\varepsilon, \varepsilon)$ is a final state, the path is accepting. Therefore, $(w, w') \in R(\dot{t}_{\text{lf}}((r_1, r_2)))$.

Conversely, we can prove that if $(w, w') \in R(\dot{t}_{\text{lf}}((r_1, r_2)))$, then $(w, w') \in T((r_1, r_2))$. □

**Theorem 4.61.** *For any G2D-RE $g$ the following inequality holds*

$$|\pi(g) \cup \{g\}| \leq |g|_{\Sigma \cup \Delta} + |g|_{\Sigma} \times |g|_{\Delta} + 1.$$

*Proof.* This follows directly from Proposition 4.60. □

## 4.5 Input and Output Projections of 2D-REs

In this section, we explore different applications of input and output projections of G2D-REs.

**Definition 4.62.** The *input-projection* regular expression of a G2D-RE $g$ over $\Sigma \times \Delta$, denoted by $\pi_{\text{i}}(g)$, is a regular expression defined recursively on the structure of $g$ as follows, for every $r_1$ over $\Sigma$ and $r_2$ over $\Delta$

$$\begin{aligned}
\pi_{\text{i}}(\emptyset) &= \emptyset, \\
\pi_{\text{i}}((r_1, r_2)) &= r_1, \\
\pi_{\text{i}}(g_1 g_2) &= \pi_{\text{i}}(g_1)\pi_{\text{i}}(g_2), \\
\pi_{\text{i}}(g_1 + g_2) &= \pi_{\text{i}}(g_1) + \pi_{\text{i}}(g_2), \\
\pi_{\text{i}}(g_1^*) &= \pi_{\text{i}}(g_1)^*,
\end{aligned}$$

where $g_1, g_2$ are also G2D-REs over $\Sigma \times \Delta$.

**Definition 4.63.** The *output-projection* regular expression of a 2D-RE $g$ over $\Sigma \times \Delta$, denoted by $\pi_{\text{o}}(g)$, is a regular expression defined recursively on the structure of $g$ as follows, for every $r_1$ over $\Sigma$ and $r_2$ over $\Delta$

$$\begin{aligned}
\pi_{\text{o}}(\emptyset) &= \emptyset, \\
\pi_{\text{o}}((r_1, r_2)) &= r_2, \\
\pi_{\text{o}}(g_1 g_2) &= \pi_{\text{o}}(g_1)\pi_{\text{o}}(g_2), \\
\pi_{\text{o}}(g_1 + g_2) &= \pi_{\text{o}}(g_1) + \pi_{\text{o}}(g_2), \\
\pi_{\text{o}}(g_1^*) &= \pi_{\text{o}}(g_1)^*,
\end{aligned}$$

where $g_1, g_2$ are also G2D-REs over $\Sigma \times \Delta$.

We now present a method for getting a transducer that realizes the same relation as a A2D-RE $(r_1, r_2)$ over $\Sigma \times \Delta$.

**Theorem 4.64.** *Let $(r_1, r_2)$ be a A2D-RE over $\Sigma \times \Delta$. Then there is a transducer that realizes the same relation as $g$ obtained from the automata equivalent to $\pi_i((r_1, r_2))$ and $\pi_o((r_1, r_2))$.*

*Proof.* Observe that, from (4.6) in Lemma 4.39, we have that $(r_1, r_2) \sim (r_1, \varepsilon)\,(\varepsilon, r_2)$.

Now let $\dot{a}(r_1)$ and $\dot{a}(r_2)$ be the automata equivalent to $r_1$ and $r_2$, respectively, where by definition, $r_1 = \pi_{\mathrm{i}}((r_1, r_2))$ and $r_2 = \pi_{\mathrm{o}}((r_1, r_2))$. From $\dot{a}(r_1)$ we construct a transducer where the output labels of each transition are $\varepsilon$. It is obvious that the transducer obtained realizes the same relation as $T((r_1, \varepsilon))$.

On the other hand, from $\dot{a}(r_2)$ we construct a transducer where the input labels of each transition are $\varepsilon$. It is obvious that the transducer obtained realizes the same relation as $T((\varepsilon, r_2))$.

Now we can concatenate these two transducers, which produces a transducer that realizes the concatenation of the two relations $T((r_1, \varepsilon))$ and $T((\varepsilon, r_2))$, that is, a transducer that realizes the same relation as $T((r_1, \varepsilon)\,(\varepsilon, r_2)) = T((r_1, r_2))$. □

**Corollary 4.65.** *From Theorem 4.64, we can construct an extension of the Thompson's method for G2D-RE.*

*Proof.* Use the method described in Theorem 4.64 for the base cases $(r_1, r_2)$, $(r_1, \varepsilon)$, $(\varepsilon, r_2)$ and $(\varepsilon, \varepsilon)$. The rest of the proof is the same as the proof for Theorem 4.46. □

## 4.6 Word Problem

One of the most important problems in the theory of computation is the word problem. In our case, this problem translates to the following definition.

**Definition 4.66.** Given a S2D-RE $s$ over $\Sigma \times \Delta$, the *word problem*, for a pair of words $(w, w')$, where $w \in \Sigma^*$ and $w' \in \Delta^*$, is to decide whether $(w, w') \in T(s)$.

This problem is decidable and we will now present an algorithm for deciding this. We will analyse the problem for S2D-REs, but the problem can be seen for G2D-REs, either by extending the methods to G2D-REs or by applying a conversion from G2D-RE to S2D-RE prior to applying the method described.

From Corollary 4.26, we know that $(w, w') \in T(s)$ if and only if $\epsilon\Big(\partial_{(w,w')}(s)\Big) = (\varepsilon, \varepsilon)$. Therefore, we only need to compute $\partial_{(w,w')}(s)$ and verify if $\epsilon\Big(\partial_{(w,w')}(s)\Big) = (\varepsilon, \varepsilon)$.

## 4.7 Equivalence Between 2D-REs

As seen in Berstel [3], the problem of deciding wether two relations are the same is undecidable. Even so, there is a class of transducers for which this problem is decidable. We now revisit this to present a method for deciding if two 2D-RE represent the same relation. Berstel presents a proof for the decidability of the problem of deciding wether two sequential transducers realize the same relation or not. Moreover, a transducer is equivalent to a sequential one if it is functional.

**Definition 4.67.** Let $s$ ($g$) be a S2D-RE (G2D-RE) over $\Sigma \times \Delta$. Then $s$ ($g$) is *functional* if the transducer that realizes the same relation is either functional or sequential.

**Theorem 4.68.** *Let $r$ and $r'$ be two functional 2D-REs (either standard or general) over $\Sigma \times \Delta$. Then the problem of deciding wether $r$ and $r'$ realize the same relation is decidable.*

*Proof.* Let $\dot{t}$ and $\dot{t}'$ be two transducers that realize the same relation as $r$ and $r'$, respectively. Then it is decidable if $\dot{t}$ and $\dot{t}'$ realize the same relation. The method, described in [3], is to see if the following two conditions hold

- $Dom(\dot{t}) = Dom(\dot{t}')$;
- $R(\dot{t}) \cup R(\dot{t}')$ is functional.

Hence, we can see if $\pi_{\mathrm{i}}(r) = \pi_{\mathrm{i}}(r')$ using the methods for automata. The second condition can be verified using a method for testing if a transducer is functional. □

*Remark* 4.69. Note that for any given 2D-RE (either S2D-RE or G2D-RE) we can produce the transducer that realizes the same relation and test if it is functional or not. Then, for two given 2D-REs, if the two transducers that realize the same relations are functional, Theorem 4.68 lets us know if we can determine if the two 2D-REs realize the same relation or not. If either is not functional, we can not apply this method.

# Chapter 5

# Implementation

In this chapter, we will discuss the implementation in FAdo [12, 24], a tool written in Python for finite automata and regular languages manipulation. We will start by introducing some methods already in FAdo prior to this work, in the first two sections, and the newly introduced ones in the last section.

FAdo is organized in modules. Each module is organized in classes. An object in a class can have attributes and methods to manipulate such objects. Some classes can use or produce objects of a class in a different module.

Some objects, like FAs and transducers can be viewed using a graphical tool. In order to obtain the graphical representation, one should use the method `display()`. This method as an optional argument, `strict`, which has the value `False` by default. If this value is `True`, each state will be displayed with its name, instead of the index that represents such state.

## 5.1 Finite Automata and Regular Languages

In this section, we will discuss some of the methods present in FAdo for finite automata and regular languages manipulation, as well as some examples.

### 5.1.1 DFAs

The `fa` module defines the classes representing finite automata. Since different types of automata have similarities between each other, FAdo provides a parent class, `FA`, that defines the basic structure of finite automata. The attributes of a `FA` object are

- `States`: The set of states, represented by a list where each state is represented by an index;
- `Sigma`: The alphabet, represented by a set of symbols;

- `Initial`: The index of the initial state;
- `Final`: The set of final states;
- `delta`: The transition function, represented by a dictionary.

The class `FA` also implements the methods necessary to manipulate one object of this class. Some examples of such methods are `addState`: adds a state with a given name to the set of states; `deleteState`: deletes a state with a given index from the set of states; and `setFinal`: sets the set of final states to a given list of states.

The class `DFA` inherits the attributes and methods present in `OFA`, a subclass of `FA`. In FAdo, `FA` is the abstract class for finite automata, and `OFA` is the base class for one way automata. The DFA present in Figure 2.1 can be built in FAdo using the following code

```
>>> from FAdo.fa import *
>>> a = DFA()
>>> a.addState()
0
>>> a.addState()
1
>>> a.addState()
2
>>> a.setInitial(0)
>>> a.addFinal(2)
>>> a.addTransition(0,'a',2)
>>> a.addTransition(0,'b',1)
>>> a.addTransition(1,'a',2)
>>> a.addTransition(1,'b',2)
>>> a.addTransition(2,'a',2)
>>> a.addTransition(2,'b',1)
```

Listing 5.1: DFA example in FAdo.

### 5.1.2 NFAs

The `NFA` class is the class that implements NFAs. Like the `DFA` class, it is a subclass of OFA. The attributes of a `NFA` object are the same as in `FA`, except that instead of `Initial` being the index of the initial state, it is the set of indexes of the initial states.

The class `OFA` also has a method called `regexpSE`, that converts a finite automaton to an equivalent regular expression, by applying the state elimination method. The NFA present in Figure 2.2 can be built in FAdo using the following code

```
>>> n = DFA()
>>> n.addState()
0
```

```
>>> n.addState()
1
>>> n.addState()
2
>>> n.addInitial(0)
>>> n.addFinal(2)
>>> n.addTransition(0,'a',2)
>>> n.addTransition(0,'b',1)
>>> n.addTransition(0,'a',0)
>>> n.addTransition(1,'a',2)
>>> n.addTransition(1,'b',2)
>>> n.addTransition(1,'a',1)
>>> n.addTransition(2,'a',2)
>>> n.addTransition(2,'b',1)
```

Listing 5.2: NFA example in FAdo.

and the regular expression equivalent to such NFA can be obtained by `n.regexpSE()`.

### 5.1.3 Regular Expressions

The `reex` module is the one that allows the tools to manipulate regular expressions. The main class in `reex` is the class `regexp`, which has as attributes

- `val`: A symbol;
- `Sigma`: The alphabet of symbols;
- `arg`: An argument of an operation.

From Definition 2.8, we know that a regular expression can be either $\emptyset$ (the `emptyset` class) or an expression given by the grammar in such definition, which can be $\varepsilon$ (the `epsilon` class), an atom which consists of a symbol from the alphabet (the `atom` class, where the symbol is given by the `val` attribute), a disjunction of expressions (the `disj` class, where the expressions on the left and on the right are given by the attributes `arg1` and `arg2`, respectively), a concatenation of expressions (the `concat` class, where the expressions on the left and on the right are given by the same attributes as the `disj` class) and the Kleene star of an expression (the `star` class, where the expression argument is given by the attribute `arg`). Since concatenations and disjunctions have characteristics in common (for instance, both have a left and a right argument), they are both subclasses of a `connective` class that implements the characteristics and methods they have in common. In a similar way, neither `emptyset` nor `epsilon` have an argument or a symbol, and like disjunctions and concatenations, they have other characteristics in common. Therefore, they both inherit attributes and methods from a superclass called `specialConstant`.

FAdo provides a function to parse a regular expression from a given word, using the global method `str2regexp`, which takes `str`, the word to be parsed, as an argument. The regular expression in Example 2.9 can be parsed in FAdo using

```
>>> from FAdo.reex import *
>>> r = str2regexp("(a+b(a+b))(a+b(a+b))*")
>>> r
concat(disj(atom(a),concat(atom(b),disj(atom(a),atom(b)))),
       star(disj(atom(a),concat(atom(b),disj(atom(a),atom(b))))))
>>> print r
(a + (b (a + b))) (a + (b (a + b)))*
```

Listing 5.3: Regular expression example in FAdo.

We can use FAdo to calculate the derivative of a regular expression w.r.t. a symbol using the method `derivative`. The partial derivatives of a RE w.r.t. a symbol can be calculated using `partialDerivatives`. We can also calculate the linear form and the set of all partial derivatives. These methods can be used as follows

```
>>> r.derivative('a').reduced()
star(disj(atom(a),concat(atom(b),disj(atom(a),atom(b)))))
>>> r.partialDerivatives('a')
{star(disj(atom(a),concat(atom(b),disj(atom(a),atom(b)))))}
```

Listing 5.4: Derivatives and partial derivatives example in FAdo.

where `reduced()` is a method that applies some simplification methods, using equivalences like $r + \emptyset \sim r$.

Finally, we can compute the NFA equivalent to a given RE with `nfaThompson()`, which uses Thompson's method and `nfaPD()` which uses the partial derivatives. FAdo does not allow one to use Brzozowski's method since there is no support for $\text{ACI}_+$ equivalences.

FAdo also provides a method `toNFA` to produce the NFA equivalent to the given RE, which uses by default the partial derivatives' method. The name of the method to be used can be passed by the argument `nfa_method`. For instance, `r.toNFA(nfa_method = 'nfaThompson')` computes the NFA equivalent to `r` using Thompson's method.

## 5.2 Transducers

The module `transducers` in FAdo provides the tools for manipulation of transducers. The superclass for each type of transducers that FAdo implements is `Transducer`, which inherits from the class `nfa` seen in the previous section. This class has a new attribute, `Output`: the set of output symbols. We consider `Sigma` to be the set of input symbols. SFTs and NFTs are implemented by classes `SFT` and `NFT`, respectively, and both are subclasses of a parent class `GFT`, the *general form transducer*, which is what we called in Section 3.2 a finite transducer. This parent class implements, among others, a new `addTransition` method, which allows to have input and output labels.

The SFT in Example 3.11 and the NFT in Example 3.14 can be constructed in FAdo using

```
>>> from FAdo.transducers import *
>>> sft = SFT()
>>> sft.addState()
0
>>> sft.addState()
1
>>> sft.addInitial(0)
>>> sft.addFinal(0)
>>> sft.addTransition(0, 'a', 'a', 0)
>>> sft.addTransition(0, 'a', 'b', 1)
>>> sft.addTransition(1, 'b', 'b', 1)
>>> sft.addTransition(1, 'b', 'a', 0)
```

Listing 5.5: SFT example in FAdo.

and

```
>>> nft = sft.toNFT()
```

Listing 5.6: NFT example in FAdo.

respectively. The method `toNFT()` converts a given SFT in an equivalent NFT.

We will now present some newly added classes and methods to the `transducers` module.

### 5.2.1 Regular Expression Labeled Finite Transducers

Regular expression labeled finite transducers, as seen in Definition 4.48, are a type of transducers that allow 2D-REs as labels of its transitions. Finite automata also have a similar definition (Definition 2.31). FAdo provides a class to represent generalized finite automata, the `GFA` class. This is useful for the state elimination method.

With that in mind, regular expression labeled finite transducers are implemented in FAdo by the class `RELFT`, inheriting from the `GFA` class. Like we did for the `Transducer` class, we add a new attribute `Output`: the set of output symbols. This class has some methods based on the implementation of the similar methods in `GFA`.

The method `regexpSE` was added to the `SFT` class in `transducers` module, which implements the extension of the state elimination method presented in Chapter 4. One can apply this method to the transducer in Example 3.11. The method `regexpSE` can be used as follows

```
>>> s2dsft = sft.regexpSE()
>>> s2dsft
StarS2D(DisjS2D(AtomS2D(a/a),ConcatS2D(AtomS2D(a/b),
    ConcatS2D(StarS2D(AtomS2D(b/b)),AtomS2D(b/a)))))
>>> print s2dsft
```

```
(a/a + a/b ((b/b)* b/a))*
```

Listing 5.7: State elimination method for transducers example in FAdo.

### 5.2.2 Sequential Transducers and Functionality Test

Sequential transducers were not yet implemented in FAdo, as well as the method described in Section 3.3 for testing the functionality of a transducer. Therefore, a new method was added to the `SFT` class, `functionalIdentity`, which uses the method `identity`. The last method tests if the relation realized by a given transducer is equivalent to the identity relation.

On the other hand, as seen in Section 3.3 as well, an algorithm for converting a SFT into an equivalent sequential transducer, provided the SFT is functional, was also newly added to FAdo. Thus, the `SFT` class now has a method `toSeq`, which sequentializes a given SFT. Since FAdo did not yet support sequential transducers, the class `SeqFT` was added to the `transducers` module.

This new class is a subclass of `SFT`, and its attributes are the same as `SFT` except for a new one, `TerminalFunction`: a dictionary that represents the terminal function defined for the sequential transducer. This class implements a method called `addTerminalFunction`, that given two attributes, `srci` and `w` adds the the representation of $T(srci) = w$ to the `SeqFT` object.

The examples Example 3.25 and Example 3.29 can be seen in FAdo by

```
>>> sft2 = SFT()
>>> sft2.addState()
0
>>> sft2.addState()
1
>>> sft2.addState()
2
>>> sft2.addState()
3
>>> sft2.addState()
4
>>> sft2.addInitial(0)
>>> sft2.addFinal(4)
>>> sft2.addTransition(0, 'a', 'x', 1)
>>> sft2.addTransition(0, 'a', 'x', 2)
>>> sft2.addTransition(1, 'a', 'x', 3)
>>> sft2.addTransition(2, 'a', '@epsilon', 3)
>>> sft2.addTransition(3, 'b', 'y', 4)
>>> sft2.functionalityIdentityP()
False
```

Listing 5.8: Functionality test for a transducer example in FAdo.

and

```
>>> seqsft = sft.toSeq()
```

Listing 5.9: Sequentialization of a transducer example in FAdo.

respectively.

We can obtain a witness of non-functionality using the method described in this work using `nonFunctionalIdW`, which is implemented inside the `SFT` class.

## 5.3 2D-REs

In this section, we will discuss the implementation of the newly added `reex2d` module to FAdo, which implements the notions discussed in Chapter 4.

Since REs and 2D-REs have characteristics in common, we can define a parent class for both REs and 2D-REs, a class called `RegularExpression`, with a single attribute, `Sigma`: an alphabet of symbols. For REs, it will be the alphabet of symbols, and for 2D-REs, it will be the input alphabet of symbols. There are some methods that can also be defined in this parent class. These methods will have different implementations in REs, S2D-REs and G2D-REs, but the name of these methods is the same for all three types.

### 5.3.1 G2D-REs

The implementation of G2D-REs is very similar to the implementation of REs. The basic structure follows the same idea as the basic structure for REs. The main class in module `reex2d` is a class called `G2Dre`, which is a subclass of `RegularExpression`, thus inheriting the `Sigma` attribute. The new attributes of `G2Dre` comparatively to `RegularExpression` are

- `Output`: The output alphabet of symbols;
- `reexi`: The input regular expression (over `Sigma`);
- `reexo`: The output regular expression (over `Output`).

As referred above, the basic structure of G2D-REs follows a structure similar to the classes that represent REs. From Definition 4.2, we know that a regular expression can be either $\emptyset$ (the `EmptysetG2D` class) or an expression given by the grammar in such definition, which can be an atom $(r_1, r_2)$ which is a pair of regular expressions $r_1$ and $r_2$ over $\Sigma$ and $\Delta$, respectively (the `AtomAtom` class, where $r_1$ and $r_2$ are given by `reexi` and `reexo` attributes, respectively), a disjunction of expressions (the `DisjG2D` class, where the expressions on the left and on the right are given by the attributes `arg1` and `arg2`, respectively), a concatenation of expressions

(the `ConcatG2D` class, where the expressions on the left and on the right are given by the same attributes as the `DisjG2D` class) and the Kleene star of an expression (the `StarG2D` class, where the argument is given by the attribute `arg`). Similarly to REs, we have the classes `ConnectiveG2D` and `SpecialConstatG2D`. Additionally, since it eases the implementation, we have the classes `AtomEpsilon` and `EpsilonAtom`, for the special cases of when the output and the input is $\varepsilon$, respectively. Furthermore, when both the input and output expressions are $\varepsilon$, we have the class `EpsilonEpsilon`.

The `reex2d` module provides a function to parse a G2D-RE from a given word, using the global method `str2G2D`, which takes `str`, the word to be parsed, as an argument. The parser used requires atoms to be encapsulated under square brackets. If the word represents a S2D-RE, the method `standardP` returns `True` after the parsing is done and the expression given is an object of the `S2Dre` class, which is discussed in the next subsection. For instance, the G2D-RE $((a+b)^*a, ba)\,(\varepsilon, a^*)$, can be parsed in FAdo using

```
>>> from FAdo.reex2d import *
>>> g = str2G2D("[(a+b)*a/ba][@epsilon/a*]")
>>> g
ConcatG2D(AtomAtom((a + b)* a/b a),EpsilonAtom(a*))
>>> print g
[[(a + b)* a/b a] [@epsilon/a*]]
```

Listing 5.10: Parsing a G2D-RE example in FAdo.

Furthermore, if we want to convert this G2D-RE into an equivalent S2D-RE, we can use `g.toStandard()`. This conversion follows the application provided in the beginning of Subsection 4.2.3. This method has an argument which is by default `False` (`normalised`), which handles if one wants to convert the G2D-RE into an equivalent normalised one, which uses Theorem 4.44.

We can convert an A2D-RE to an equivalent SFT using the Thompson's method described in Theorem 4.64 using `g.sftThompson()`. This method takes as argument `nfa_method`, the method to use for converting the RE into the equivalent NFA, which is by default `nfaThompson`. Note that since an A2D-RE is a particular case of G2D-REs (that is, since it is generated by the rule for $(r_1, r_2)$), A2D-REs are represented by the `AtomAtom`, `AtomEpsilon` and `EpsilonAtom` classes.

The methods related to partial derivatives and linear form are discussed in the next subsection, since the names of these methods are the same, thus having the same usage. The implementation, though, differs from G2D-REs to S2D-REs.

### 5.3.2 S2D-REs

Since S2D-REs are a particular case of G2D-REs, the main class we use to represent S2D-REs, `S2Dre`, is a subclass of `G2Dre`. The implementation structure is the same as G2D-REs. Namely,

we have the classes `EmptysetS2D`, `EpsilonS2D`, `LeftEpsilonS2D`, `RightEpsilonS2D`, `AtomS2D`, `DisjS2D`, `ConcatS2D` and `StarS2D`, as well as the classes `ConnectiveS2D` and `SpecialConstantS2D`.

We can parse the S2D-RE in Example 4.10, calculate the linear form and equivalent SFT using the linear form method and Thompson's method as follows

```
>>> s = str2S2D("(a/b+@epsilon/b*)a/@epsilon+@epsilon/b")
>>> s
DisjS2D(ConcatS2D(DisjS2D(AtomS2D(a/b),
    StarS2D(LeftEpsilonS2D(b))),RightEpsilonS2D(a)),LeftEpsilonS2D(b))
>>> print s
(a/b + (@epsilon/b)*) a/@epsilon + @epsilon/b
>>> s.linearForm()
{('@epsilon', 'b'):
    {ConcatS2D(StarS2D(LeftEpsilonS2D(b)),RightEpsilonS2D(a)),
    EpsilonS2D()}, ('a', '@epsilon'): {EpsilonS2D()}, ('a', 'b'):
    {RightEpsilonS2D(a)}}
>>> s.linearForm(extended=True)
{('@epsilon', 'b'): {ConcatS2D(RightEpsilonS2D(a),RightEpsilonS2D(a)),
    ConcatS2D(StarS2D(LeftEpsilonS2D(b)),RightEpsilonS2D(a)), EpsilonS2D()},
    ('a', '@epsilon'): {ConcatS2D(LeftEpsilonS2D(b),RightEpsilonS2D(a)),
    EpsilonS2D()}, ('a', 'b'): {RightEpsilonS2D(a)}}
>>> sftpd1 = s.toSFT()
>>> sftpd2 = s.toSFT(sft_method="sftThompson")
```

Listing 5.11: Manipulation of S2D-REs example in FAdo.

Furthermore, if we want to check if a pair of words is in the relation realized by the S2D-RE or if we want to check if two S2D-REs, we can use

```
>>> s.evalWordP(("aa","bbbbb"))
True
>>> s1 = str2S2D("a/b")
>>> s2 = str2S2D("b/a")
>>> s3 = str2S2D("a/@epsilon@epsilon/b")
>>> s1.equivalentP(s2)
False
>>> s1.equivalentP(s3)
True
```

Listing 5.12: Membership and equivalence between S2D-REs example in FAdo.

# Chapter 6

# Conclusion and Future Work

In this work, we presented a new way to represent binary relations over words by means of some type of regular expressions.

We started by introducing some of the work known for automata and regular expressions and on the following chapters we extended some of these definitions, theorems and algorithms. Unfortunately, not everything that can be applied to automata can be applied to transducers. For instance, we know that one can determine if two FA $\dot{a}_1$ and $\dot{a}_2$ are equivalent, that is, if $\mathcal{L}(\dot{a}_1) = \mathcal{L}(\dot{a}_2)$ or not, but from [3], since it is undecidable to know if two rational relations are equivalent, this is undecidable for transducers as well, since the relation realized by a transducer is a special type of relation. Even so, as we presented in this work, some popular methods can be extended to transducers, for instance, we have the state elimination method and the Thompson's method, two of the methods usually taught on an introductory computation models' class.

In Section 3.3, we introduced sequential transducers. In [20], the author presents a method for minimizing a sequential transducer. This method is not yet implemented in FAdo, since it uses a different representation of transducers. It might be interesting to implement this method in FAdo by either adapting the method to the representation in FAdo, or by implementing this new representation alongside with the already present.

In Section 4.4, we presented efficient algorithms for obtaining a transducer from a given 2D-RE. This is an important fact, since these methods are efficient both in time and space, so there is no trade off of one for the other.

Since this work is just an introduction to 2D-regular expressions, there is a lot of work that can follow. For instance, one can study the applications of this to code theory. For this, we revert to Konstantinidis, Meijer, Moreira and Reis [18] and for symbolic transducers, we revert to Veanes [29].

One theme that is very interesting to study is to find a way to characterise 2D-REs where the linear-form transducer is functional. This is studied for deterministic regular languages in [4].

It might be interesting as well to find out if $(\mathit{G2D\text{-}RE}, \oplus, \odot, \circledast, \emptyset, (\varepsilon, \varepsilon))$ forms a Kleene

algebra. For that, we revert to Kozen [19].

Finally, one can study the number of states on average produced by the linear-form method w.r.t. the size of the 2D-RE. This study would make sense for both S2D-REs and G2D-REs.

Better characterisation of different 2D-regular expressions might lead to finding an algorithm for deciding if two functional 2D-regular expressions are equivalent or not, without the need to convert each to an equivalent transducer.

# Bibliography


[1] Cyril Allauzen and Mehryar Mohri. Efficient algorithms for testing the twins property. *J. Autom. Lang. Comb.*, 8(2):117–144, April 2003. ISSN: 1430-189X.

[2] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computational Science*, 155(2):291–319, 1996.

[3] Jean Berstel. Transductions and context-free languages. *Ed. Teubner*, pages 1–278, 1979.

[4] Anne Brüggemann-Klein and Derick Wood. Deterministic regular languages. In Alain Finkel and Matthias Jantzen, editors, *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*, volume 577 of *Lecture Notes in Computer Science*, pages 173–184. Springer, 1992. doi:10.1007/3-540-55210-3_182.

[5] Janusz A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical theory of Automata*, Volume 12 of MRI Symposia Series, pages 529–561. Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1962.

[6] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964. ISSN: 0004-5411. doi:10.1145/321239.321249.

[7] Marie-Pierre Béal, Olivier Carton, Christophe Prieur, and Jacques Sakarovitch. Squaring transducers: an efficient procedure for deciding functionality and sequentiality. *Theoretical Computer Science*, 292(1):45 – 63, 2003. ISSN: 0304-3975. Selected Papers in honor of Jean Berstel. doi:https://doi.org/10.1016/S0304-3975(01)00214-6.

[8] Jean-Marc Champarnaud and Djelloul Ziadi. From mirkin's prebases to antimirov's word partial derivatives. *Fundam. Inf.*, 45(3):195–205, January 2001. ISSN: 0169-2968.

[9] Christian Choffrut. Une caracterisation des fonctions sequentielles et des fonctions sous-sequentielles en tant que relations rationnelles. *Theoretical Computer Science*, 5(3):325 – 337, 1977. ISSN: 0304-3975. doi:https://doi.org/10.1016/0304-3975(77)90049-4.

[10] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN: 0070131511.

[11] C. C. Elgot and J. E. Mezei. On relations defined by generalized finite automata. *IBM J. Res. Dev.*, 9(1):47–68, January 1965. ISSN: 0018-8646. doi:10.1147/rd.91.0047.

[12] Tools for Formal Languages manipulation FAdo. http://fado.dcc.fc.up.pt, 2018.

[13] Yuan Gao, Nelma Moreira, Rogério Reis, and Sheng Yu. A survey on operational state complexity. *J. Autom. Lang. Comb.*, 21(4):251–310, June 2016. ISSN: 1430-189X.

[14] Seymour Ginsburg. *Algebraic and Automata-Theoretic Properties of Formal Languages*. Elsevier Science Inc., New York, NY, USA, 1975. ISBN: 0444105867.

[15] Western University. Grail+. http://www.csit.upei.ca/ ccampeanu/grail/, 2018.

[16] John E. Hopcroft. An $n$ log $n$ algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[17] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990. ISBN: 020102988X.

[18] Stavros Konstantinidis, Casey Meijer, Nelma Moreira, and Rogério Reis. Symbolic manipulation of code properties. *Journal of Automata, Languages and Combinatorics*, 23(1–3):243–269, 2018. doi:10.25596/jalc-2018-243.

[19] D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Inf. Comput.*, 110(2):366–390, May 1994. ISSN: 0890-5401. doi:10.1006/inco.1994.1037.

[20] M. Lothaire. *Applied Combinatorics on Words*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2005. doi:10.1017/CBO9781107341005.

[21] Boris G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:51–57, 1966.

[22] Edward F. Moore. Gedanken-experiments on sequential machines. *The Journal of Symbolic Logic*, 23(1), 1958.

[23] Darrell Raymond and Derick Wood. Grail: A c++ library for automata and expressions. *J. Symb. Comput.*, 17(4):341–350, April 1994. ISSN: 0747-7171. doi:10.1006/jsco.1994.1023.

[24] Rogério Reis and Nelma Moreira. Fado: tools for finite automata and regular expressions manipulation. 2002.

[25] OpenFST Library. Google Research and NYU's Courant Institute. http://www.openfst.org/twiki/bin/view/fst/webhome, 2018.

[26] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, New York, NY, USA, 2009. ISBN: 0521844258, 9780521844253.

[27] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. ISSN: 0001-0782. doi:10.1145/363347.363387.

[28] Vaucanson. The vaucanson project. http://vaucanson-project.org, 2018.

[29] Margus Veanes. Applications of symbolic finite automata. In *Proceedings of the 18th International Conference on Implementation and Application of Automata*, CIAA'13, pages 16–23, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN: 978-3-642-39273-3. doi:10.1007/978-3-642-39274-0_3.

[30] JFLAP. Experimenting with formal languages. http://www.jflap.org, 2018.

[31] Sheng Yu. Handbook of formal languages, vol. 1. chapter Regular Languages, pages 41–110. Springer-Verlag, Berlin, Heidelberg, 1997. ISBN: 3-540-60420-0.