ASSIGNMENT OF MASTER’S THESIS

Title:	Finite tree automaton to tree regular expression conversion
Student:	Bc. Jakub Doupal
Supervisor:	Ing. Jan Trávníček
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2019/20

Instructions

Study the definition of finite tree automata and tree regular expressions.
Study the finite tree automaton to tree regular expression conversion using tree regular equations [1].
Design simplifying axioms on tree regular expressions.
Implement the finite tree automaton to tree regular expression conversion using tree regular equations.
Implement tree regular expression simplification by designed axioms.
Propose suitable tests and test your implementation.

References

[1] Younes Guellouma and Hadda Cherroun. From tree automata to rational tree expressions. International Journal of Foundations of Computer Science, 29(06): 1045--1062, 2018.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 8, 2019

Master's thesis

Finite tree automaton to tree regular expression conversion

Bc. Jakub Doupal

Department of Theoretical Computer Science
Supervisor: Ing. Jan Trávníček, Ph.D.

May 7, 2019

Acknowledgements

I would like to thank my supervisor, Ing. Jan Trávníček, Ph.D., for his suggestions and help with writing my thesis. I would also like to thank my family and friends for their support while writing my thesis and throughout my entire studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 7, 2019 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2019 Jakub Doupal. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Doupal, Jakub. *Finite tree automaton to tree regular expression conversion.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstract

This Master's thesis studies regular tree expressions and a method for conversion from a finite tree automaton to a regular tree expression using equations. The studied method is then implemented in Algorithms Library Toolkit, which is being developed at the Department of Theoretical Computer Science at Faculty of Information Technology, Czech Technical University in Prague. This thesis also studies axioms for regular tree expressions and proposes new ones. The proposed axioms are also implemented in the Algorithms Library Toolkit.

Keywords regular tree expressions, tree automata, regular tree equations, automata library, algorithms library toolkit

Abstrakt

Tato diplomová práce studuje regulární stromové výrazy a metodu převodu ze stromového automatu na regulární stromový výraz pomocí rovnic. Daná metoda je poté implementována v knihovně algoritmů vyvíjené na Katedře teoretické informatiky na Fakultě informačních technologií na ČVUT v Praze. Tato práce dále studuje axiomy pro regulární stromové výrazy a také navrhuje nové. Tyto axiomy pak jsou také implementovány v knihovně algoritmů.

Klíčová slova regulární stromové výrazy, stromové automaty, regulární stromové rovnice, automatová knihovna, knihovna algoritmů

Contents

Introduction **1**

1 Terminology **3**

2 Study of regular tree expressions **9**
2.1 Regular expressions 9
2.2 Regular tree expressions 9
2.3 Study of RTE operators 10
2.4 Tree languages 13

3 FTA to RTE conversion **15**
3.1 RTE equation system 15
3.2 Gaussian-elimination-like method 17
3.3 Non-recursive case 18
3.4 Recursive case 19
3.5 Construction of RTE equation system from FTA 20
3.6 Viability and challenges 23

4 RTE axioms **25**
4.1 Former work on RTE axioms 25
4.2 Alteration of axioms 27
4.3 Unbounded RTE 28
4.4 Proposal of new RTE axioms 29

5 Resolving challenges in the conversion method **33**
5.1 Demonstrating the problem 33
5.2 Solution using RTE axioms 35

6 Design **41**
6.1 FTA to RTE conversion 41

6.2 RTE axioms 45

7 Implementation **47**
7.1 Algorithms Library Toolkit 47
7.2 Preexisting data structures used 50
7.3 New data structures implemented 51
7.4 Implementation of the conversion method 54
7.5 Implementation of axioms 54

8 Testing **57**
8.1 FTA files tests 57
8.2 RTE files tests 58
8.3 Random FTA tests 58

Conclusion **59**

Bibliography **61**

A Acronyms **63**

B Automata library toolkit manual **65**
B.1 Requirements 65
B.2 Installation 65
B.3 Running the toolkit 65
B.4 Examples 66

C Contents of enclosed SD card **67**

List of Figures

1.1 An example of a tree 5
1.2 A finite tree automaton 6

2.1 The language denoted by $f(a, b) + h(c)$ 10
2.2 The language denoted by $f(a + b, c)$ 11
2.3 The language denoted by $f(a, b, a) \cdot_a (x + y)$ 11
2.4 The language denoted by $f(a, b, a) \cdot_b (x + y)$ 12
2.5 The trees denoted by $(f(c, x))^{*c}$ 12
2.6 The language denoted by $f(c, x)^{*c} \cdot_c f(y, z)$ 13

3.1 FTA A 21
3.2 FTA A 23

7.1 The finite tree automaton represented by example 7.1 49

List of Tables

5.1 Transitions of the FTA A 34

Introduction

In computer science, trees are a commonly and widely used structure. Examples of tree-structured values include markup languages such as XML or HTML, or data structures such as trees or heaps. Sets of trees can be described by regular tree expressions. A tree then may be matched onto such expression.

Regular tree expressions (RTE) are mutually convertible with finite tree automata (FTA). Study of such conversions is quite a new field, especially in the direction from RTE to FTA. Example of a conversion from RTE to FTA is presented in [1]. RTE may be also converted to pushdown automata. Such methods are studied in [2] or [3]. A method for conversion of a FTA to a RTE is presented in [4].

There exist analogies between RTEs and regular (string) expressions. In fact, in most of the studied literature, this is also acknowledged and even embraced, as the string regular expressions have been studied far more extensively and serve as a good starting point for many fields of research of RTE.

As is case with every formalism, axioms for RTE also exist. The axioms describe some properties which are non-disputable and always true. Their common usages include resolutions of equations and modifications of RTEs, for example for simplification of a long RTE.

The goal of this work comprises two parts: the first is to study RTEs and the method presented in [4], and implement it in the Algorithm Library Toolkit. The second part is to study RTE axioms and propose new ones. The axioms are then to be implemented as well.

This work consists of eight chapters: first, some basic terms are introduced in chapter 1. Then, in chapter 2, RTEs are studied. In chapter 3, the method of conversion from FTA to RTE is studied and its viability for implementation is discussed. This discussion is further extended in chapter 5, along with proposed solutions. RTE axioms are studied and proposed in chapter 4. Chapter 6 then deals with the design of the implementation, while the technical details

of the implementation are discussed in chapter 7. Lastly, testing is described in chapter 8.

Chapter 1

Terminology

In this chapter, basic notions and definitions and examples thereof needed in the scope of this work are provided.

Ranked symbol

Definition 1.1. A *ranked symbol* x is a pair (x', n), where x' is a symbol and $n \in \mathbb{N}$ is the ranked symbol's *arity*.

Note. The arity of a ranked symbol may also be called its *rank*.

Definition 1.2. Symbols with arity of 0 are called *constants*. [5]

Ranked alphabet

Definition 1.3. A *ranked alphabet* Σ is denoted by $\Sigma = \cup_{n \geq 0} \Sigma_n$, where each Σ_n denotes a set of ranked symbols of arity n. [4]

Note. A ranked alphabet may also be referred to as a *graded alphabet*.

Example 1.4. For $\Sigma = \{(x, 2), (y, 1), (z, 0), (a, 0)\}$, $\Sigma_2 = \{(x, 2)\}$, $\Sigma_1 = \{(y, 1)\}$, and $\Sigma_0 = \{(z, 0), (a, 0)\}$. The arity of symbols x, y, z, a is, respectively, $2, 1, 0, 0$.

Note. For simplicity, a ranked symbol $x = (x', n)$ may be referred to by its symbol x' in unambiguous cases.

Word, language

Definition 1.5. A *word* is a sequence of symbols from a ranked alphabet.

Definition 1.6. The set of all words over a ranked alphabet Σ is denoted Σ^*. A language L is a subset of Σ^*.

Regular expression

Definition 1.7. A *regular expression* E over ranked alphabed Σ is defined inductively as:

- $E = \emptyset$,
- $E = \varepsilon$, an empty word, that is a word of length 0, without any symbols,
- $E = f(E_1, \dots, E_n)$,
- $E = E_1 + E_2$, an alternation of E_1 and E_2,
- $E = E_1 \cdot E_2$, a concatenation of E_1 and E_2,
- $E = E_1^*$, an iteration of E_1,

where $f \in \Sigma_n$ and $E_1, \dots E_n$ are any n RTE over Σ.

Example 1.8. $a^* + b(a + b(a)) \cdot a$ is a regular expression over a ranked alphabet $\Sigma = \{(a, 0), (b, 1)\}$.

Note. Just like the multiplication operator in mathematics, the concatenation operator may be also represented by no symbol. For instance, $E_1 E_2$ is and expression equivalent to $E_1 \cdot E_2$, and denotes a concatenation of E_1 and E_2.

Definition 1.9. The language $[E]$ *denoted* by a regular expression E over ranked alphabet Σ is defined inductively by:

- $[\emptyset] = \emptyset$,
- $[\varepsilon] = \varepsilon$, an empty word, that is a word of length 0, without any symbols,
- $[f(E_1, \dots, E_n)] = f([E_1], \dots, [E_n])$,
- $[E_1 + E_2] = [E_1] \cup [E_2]$,
- $[E_1 \cdot E_2] = [E_1] \cdot [E_2]$,
- $[E_1^*] = \varepsilon \cup [E_1] \cup [E_1] \cdot [E_1] \cup [E_1] \cdot [E_1] \cdot [E_1] \cup \dots$,

where a concatenation $[E_1] \cdot [E_2]$ of languages follows: for each word from $[E_1]$ and for each word $[E_2]$, a word is formed by appending the word from $[E_2]$ at the end of the word from $[E_1]$.

Example 1.10. The concatenation of the words $ab \cdot cd$ is the word $abcd$.
The concatenation of languages $L_1 = \{a, b\}$ and $L_2 = \{c, d\}$ is a language $L_1 \cdot L_2 = \{ac, ad, bc, bd\}$.

Example 1.11. The iteration of the word ax, denoted $(ax)^*$ is a language $\{\varepsilon, ax, axax, axaxax, \dots\}$.

Example 1.12. The language denoted by the regular expression 1.8 is
$\{b(a)a, b(b(a))a, \varepsilon, a, aa, aaa, aaaa, \dots\}$

Tree

Definition 1.13. A *tree* t over a ranked alphabet Σ is defined inductively as $t = f(t_1, \ldots, t_n)$ with $f \in \Sigma_n$ and $t_1, \ldots, t_n$ any n trees over Σ.

The set of all such trees is denoted by $T(\Sigma)$. A *tree language* is a subset of $T(\Sigma)$. [4]

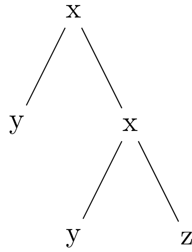

Figure 1.1: An example of a tree

Definition 1.14. In a tree $t = f(t_1, \ldots, t_n)$, the trees $t_1, \ldots, t_n$ are called *children* of f and f is called a *parent* of trees $t_1, \ldots, t_n$.

Let $L_1, \ldots, L_n \subseteq T(\Sigma)$ and $f \in \Sigma_n$. Then $f(L_1, \ldots L_n) = \{f(t_1, \ldots t_n) \mid t_1 \in L_1, \ldots, t_n \in L_n\}$.

Finite tree automaton

Definition 1.15. A *finite tree automaton* (FTA) over a ranked alphabet Σ is defined as a quadruple $A = (\Sigma, Q, Q_f, \Delta)$, where Q is a finite set of states, $Q_f \subseteq Q$ is the set of final states and $\Delta = \cup_{n \geq 0} \Sigma_n \times Q^{n+1}$ is a finite set of transitions.

Note. For a ranked symbol of arity $n \geq 2$, the order of its children is important and not commutable. If an automaton is being described in text, the order is clear from the written order, for example $f(a, b)$. When the automaton is being depicted in a figure, the order of the children is denoted next to the transitions, with numbers $1, \ldots, n$.

Example 1.16. The finite tree automaton in the figure 1.2 is an automaton $A = (\{(a, 2), (b, 1), (c, 0)\}, \{1, 2, 3\}, \{3\}, \Delta)$, where Δ is the set of transitions depicted in the figure.

Definition 1.17. A FTA is considered *deterministic* if for $x = (x', n) \in \Sigma_n$ and n states $q_1, \ldots, q_n \in Q$, the number of transitions from these states using x in Δ is at most 1. If a FTA is not deterministic, it is called *non-deterministic.*

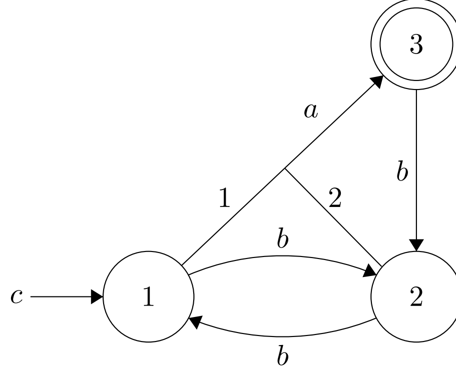

Figure 1.2: A finite tree automaton

Definition 1.18. The *output* of a FTA A is denoted δ and is a function from $T(\Sigma)$ to 2^Q defined inductively for any tree $t = f(t_1, \ldots, t_n)$ by $\delta(t) = \{q \in Q \mid \exists(f, q_1, \ldots, q_n, q) \in \Delta, (\forall 1 \leq i \leq n, q_i \in \delta(t_i))\}$. [4]

Definition 1.19. The *accepted language* of a FTA A is $L(A) = \{t \in T(\Sigma) \mid \delta(t) \subseteq Q_f\}$. The *state language* $L(q)$ of a state $q \in Q$ is defined by $L(q) = \{t \in T(\Sigma) \mid q \in \delta(t)\}$. [4]
A tree t is *accepted* by the FTA A if $t \in L(A)$.

Definition 1.20. For a ranked symbol $c \in \Sigma_0$, the *c-product* is the operation $\cdot_c$ defined for any tree $t \in T(\Sigma)$ and for any tree language L by:

$$t \cdot L = \begin{cases} L & \text{if } t = c, \\ \{d\} & \text{if } t = d \in \Sigma_0 \setminus \{c\}, \\ f(t_1 \cdot_c L, \ldots, t_n \cdot_c L) & \text{otherwise if } t = f(t_1, \ldots t_n). \end{cases} \tag{1.1}$$

[4]

Definition 1.21. The *iterated c-product* is the operation ^{n,c} recursively defined for any integer n by:

$$\begin{aligned} L^{0,c} &= \{c\} \\ L^{n+1,c} &= L^{n,c} \cup L \cdot_c L^{n,c} \end{aligned} \tag{1.2}$$

[4]

Definition 1.22. The *c-closure* is the operation ${}^*c$ defined by $L^*c = \cup_{n \geq 0} L^{n,c}$. [4]

Note. The c-closure operation may also be called *iteration over* c, or iteration.

Regular tree expression

Regular tree expressions over Σ describe a set of trees over Σ.

Definition 1.23. A *regular tree expression* (RTE) E over ranked alphabet Σ is inductively defined by:

- $E = \emptyset$,
- $E = f(E_1, \dots, E_n)$,
- $E = E_1 + E_2$, an alternation of E_1 and E_2,
- $E = E_1 \cdot_c E_2$, a concatenation of E_1 and E_2 through c,
- $E = E_1^{*c}$, an iteration of E_1 over c,

where $f \in \Sigma_n$, $c \in \Sigma_0$ and $E_1, \dots E_n$ are any n RTE over Σ. [4]

Definition 1.24. The language denoted by E is the tree language $[E]$, defined inductively by:

- $[\emptyset] = \emptyset$,
- $[f(E_1, \dots, E_n)] = f([E_1], \dots, [E_n])$,
- $[E_1 + E_2] = [E_1] \cup [E_2]$,
- $[E_1 \cdot_c E_2] = [E_1] \cdot_c [E_2]$,
- $[E_1^{*c}] = [E_1]^{*c}$,

where $f \in \Sigma_n$, $c \in \Sigma_0$ and $E_1, \dots E_n$ are any n RTE over Σ. [4]

Definition 1.25. Two RTEs E and F are *equivalent* $\Leftrightarrow [E] = [F]$. [4]

Auxiliary terms and definitions

Definition 1.26. For a non-negative integer $n \in \mathbb{N}$, $\hat{n}$ is defined as a set of integers $\{1, \dots, n\}$.

CHAPTER 2

Study of regular tree expressions

This chapter studies regular tree expressions, their operators, and the languages denoted by them. It also reviews regular expressions and goes through similarities and differences between them and RTEs.

A tree language denoted by a RTE E over Σ is $[E]$. What that means is the tree language is described by a set of rules given by the RTE, and all trees over Σ which satisfy these rules belong to $[E]$ and all the other trees do not. A tree which is denoted by a RTE E is also accepted by a FTA A and *generated by a regular tree grammar* G. Regular tree expressions, finite tree automata, and regular tree grammars are all mutually convertible. Regular tree grammars are not a subject of this work. In case the reader is interested in them, they may follow [6].

2.1 Regular expressions

For regular expressions, all of the operators are very straightforward and should be easy to comprehend. The alternation operator says "either the first or the second operand". Informally, iteration of x means "zero or more occurrences of x", and concatenation of x and y is "appending y at the end of x".

2.2 Regular tree expressions

Regular tree expressions are, in contrast with regular expressions, which describe languages over words and strings, used to describe sets of tree-structured values. [7]

Tree-structured values are commonly used in programming and computer science in general. As an example, XML or HTML files are tree-structured. A regular tree expression then may be used to describe such file which is valid, satisfies some property, belongs to a given category, etc. A finite tree

automaton then may be used to accept or reject a given RTE. Given the FTAs and RTEs are mutually convertible, there exist methods in both directions – for example, a method of conversion of a FTA into a RTE is a subject of this thesis. A method of construction of a FTA from a RTE is described in [1]. A method of conversion from a RTE to a push-down automaton is described in [2] and [3].

As tree-structured values differ from words or strings, the description of regular expressions built over them must differ as well. Therefore, regular tree expressions are in its core built over the same principle as regular (string) expressions, but there are differences which need to be addressed.

2.3 Study of RTE operators

The regular tree expression operators were formally described in definitions 1.20, 1.22, and 1.23. The languages denoted by them were described in definition 1.24. In this section, the RTE operators are studied and explained further, along with examples.

2.3.1 Alternation

The alternation operator is arguably the easiest one to comprehend of all. As can be deduced from the equation $[E_1 + E_2] = [E_1] \cup [E_2]$ mentioned in definition 1.24, use of this operator simply means a union of the language denoted by E_1 and the language denoted by E_2 is denoted by $E_1 + E_2$. In this regard, it is identical to the alternation operator in regular expressions.

Example 2.1. For the RTE $f(a, b) + h(c)$, the language denoted contains two trees depicted in figure 2.1.

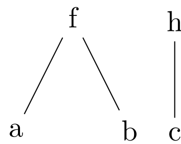

Figure 2.1: The language denoted by $f(a, b) + h(c)$

2.3.2 Ranked symbol

A language denoted by a regular tree expression $f(t_1, \ldots, t_n)$ with f being a ranked symbol of rank n can be visualized as a set of trees with f as the root and all possible $t_1, \ldots, t_n$ as its children. Again, there is no difference from regular expressions here.

Example 2.2. For the RTE $f(a+b,c)$, the language denoted contains two trees depicted in figure 2.2.

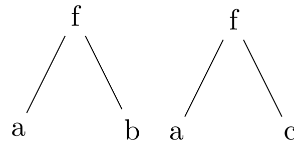

Figure 2.2: The language denoted by $f(a+b,c)$

2.3.3 Concatenation

The concatenation operator $\cdot c$ for RTEs is different from the concatenation operator $\cdot$ for regular expression in one crucial matter. While for regular expressions the right operand is always appended at the end of the left operand and there is no symbol in the concatenation operator, as in the following example:

Example 2.3. $f(a,b,a) \cdot (x+y)$ yields $f(a,b,a)x$ and $f(a,b,a)y$,

for RTEs the resultant language is obtained by replacing the constant c (a part of the concatenation operator) in all trees denoted by the left operand by all trees denoted by the right operand, which results in different outcomes, as the following example demonstrates:

Example 2.4. $f(a,b,a) \cdot_a (x+y)$ yields $f(x,b,x)$, $f(x,b,y)$, $f(y,b,x)$, $f(y,b,y)$. If the concatenation symbol is changed from a to b, then $f(a,b,a) \cdot_b (x+y)$ yields $f(a,x,a)$ and $f(a,y,a)$.

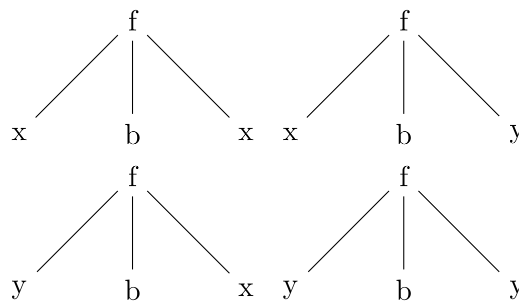

Figure 2.3: The language denoted by $f(a,b,a) \cdot_a (x+y)$

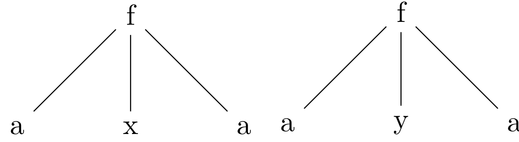

Figure 2.4: The language denoted by $f(a, b, a) \cdot_b (x + y)$

2.3.4 Iteration

As is the case with the concatenation operator, the iteration operator also differs for regular expressions and RTEs. While for regular expressions, the operator * simply means "zero or more occurrences of the operand", and no symbol is a part of the operator, as is the case in the following example:

Example 2.5. $f(c, x)^*$ yields $\varepsilon, f(c, x), f(c, x)f(c, x), f(c, x)f(c, x)f(c, x), \ldots,$

for RTEs the operator ^{*c} yields either the constant c (which is evidently a part of the iteration operator) or a set of trees where all occurrences of c are replaced by the operand. If the operand contains c, then the replacement may be performed again in successive iterations.

Example 2.6. For the RTE $(f(x, c))^{*c}$, the denoted language is the set of trees depicted in figure 2.5.

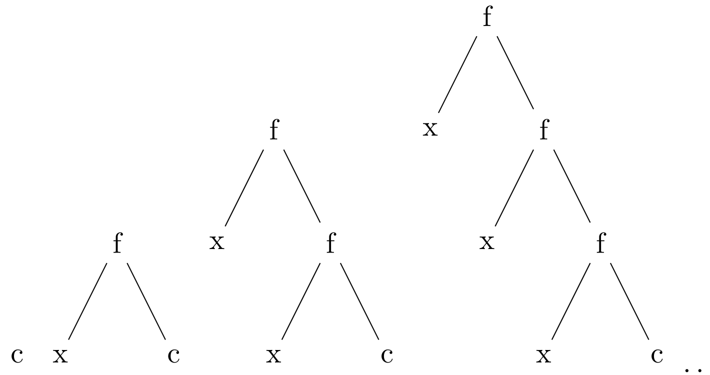

Figure 2.5: The trees denoted by $(f(c, x))^{*c}$

Note. Note that in each successive iteration, the tree with root f and children x, c takes place of the constant c.

2.4 Tree languages

In tree languages, the concatenation and iteration operators are often used together. Informally said, while the iteration denotes an infinite set of trees with a symbol which will be replaced in the next iteration, the concatenation then denotes the final result by replacing the placeholder symbol with its right operand.

Example 2.7. For the RTE $f(x,c)^{*c} \cdot_c f(y,z)$, the denoted language is the set of trees depicted in figure 2.6.

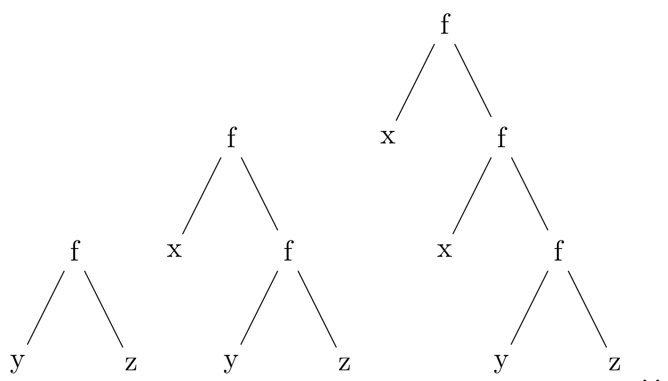

Figure 2.6: The language denoted by $f(c,x)^{*c} \cdot_c f(y,z)$

Note. Note that unlike in the previous example 2.6 without concatenation, c, the symbol over which $f(c,x)$ is iterated, is not present and is instead replaced by $f(y,z)$.

Chapter **3**

FTA to RTE conversion

In this chapter, the proposal of construction of a RTE from a FTA via a rational equation system presented in [4] is studied. It is not presented in such detail as the source, omitting some lemmas and proofs, which the reader may find in the source if they are interested. Then, it is discussed as to how the proposal is viable for implementation and what are its challenges.

The method proposal in question generalizes the method for construction of a regular (string) expression from a finite automaton. However, there are some crucial differences which need to be addressed.

3.1 RTE equation system

Equation systems have been studied before, for example in [7]. To be consistent throughout the chapter, definitions from [4] will be used, and other literature is only used to confirm consistency.

To allow for a resolution of an equation system, the RTE definition 1.23 needs to be extended. The extension is to include *variables.*

Definition 3.1. Let $X = \mathbb{X}_1, \ldots, \mathbb{X}_k$ be a set of k variables. A RTE E over (Σ, X) is then defined inductively by:

- $E = \emptyset$,
- $E = \mathbb{X}_j$,
- $E = f(E_1, \ldots, E_n)$,
- $E = E_1 + E_2$, an alternation of E_1 and E_2,
- $E = E_1 \cdot_c E_2$, a concatenation of E_1 and E_2 through c,
- $E = E_1^{*c}$, an iteration of E_1 over c,

where $f \in \Sigma_n$, $c \in \Sigma_0$, $E_1, \ldots E_n$ are any n RTE over Σ, and $j \in \hat{k}$. [4]

As the definition of a RTE has been extended, the definition of language denoted by the extended RTE also needs to be updated. For the extended RTE, the language denoted needs context to be computed, which means any variable has to be evaluated according to a tree language.

Definition 3.2. Let $\mathcal{L} = (L_1, \ldots, L_k)$ be a k-tuple of tree languages over Σ. The $\mathcal{L}$-language denoted by E is the tree language $[E]_{\mathcal{L}}$, defined inductively by:

- $[\emptyset]_{\mathcal{L}} = \emptyset$,
- $[\mathcal{X}_j]_{\mathcal{L}} = L_j$
- $[f(E_1, \ldots, E_n)]_{\mathcal{L}} = f([E_1]_{\mathcal{L}}, \ldots, [E_n]_{\mathcal{L}})$,
- $[E_1 + E_2] = [E_1]_{\mathcal{L}} \cup [E_2]_{\mathcal{L}}$,
- $[E_1 \cdot_c E_2] = [E_1]_{\mathcal{L}} \cdot_c [E_2]_{\mathcal{L}}$,
- $[E_1^{*c}] = [E_1]_{\mathcal{L}}^{*c}$,

where $f \in \Sigma_n$, $c \in \Sigma_0$, $E_1, \ldots E_n$ are any n RTE, and $j \in \hat{k}$. [4]

What the definition above says can be explained as every variable having its own associated language and right-hand side. An evaluation of a variable then yields its associated language.

Definition 3.3. For ranked alphabet Σ and a set of n variables $X = \{\mathbb{X}_1, \ldots, \mathbb{X}_n\}$, an *equation* over (Σ, X) is an expression $\mathbb{X}_j = F_j$, where $j \in \hat{k}$ and F_j is a RTE over (Σ, X).

Definition 3.4. An equation $\mathbb{X}_j = F_j$ is *variable-free* if F_j is a RTE over Σ. If a RTE E is variable-free, then $[E]_{\mathcal{L}} = [E]$.

Definition 3.5. An *equation system* over (Σ, X) is a set $\mathcal{X} = \{\mathbb{X}_j = F_j \mid j \in \hat{n}\}$ of n equations.

Definition 3.6. An occurrence of a constant is said to be *bounded* if it appears as either the operand of iteration, the left operand of concatenation, or if it is a symbol of the iteration or concatenation operator. A symbol is said to be bounded if it has at least one bounded occurrence in the equation system. Symbols which are not bounded are called *free*. [7] [4]

Definition 3.7. Let $\mathcal{L} = (L_1, \ldots, L_n)$ be a n-tuple of tree languages. $\mathcal{L}$ is a *solution* for an equation $\mathbb{X}_j = F_j$ if $L_j = [F_j]_{\mathcal{L}}$. [4] [7]
$\mathcal{L}$ is a solution for an equation system $\mathcal{X}$ if for any equation $\mathbb{X}_j = F_j$ in $\mathcal{X}$, $\mathcal{L}$ is a solution of the equation.

Definition 3.8. Two systems over the same set of variables are *equivalent* if they admit the same set of languages as solutions. [4] [7]

Example 3.9. An example of a RTE equation system provided in [4] is the following:

$$\mathcal{X} = \begin{cases} \mathbb{X}_1 = f(\mathbb{X}_1, \mathbb{X}_1) + f(\mathbb{X}_2, \mathbb{X}_4) \\ \mathbb{X}_2 = b + f(\mathbb{X}_2, \mathbb{X}_4) \\ \mathbb{X}_3 = a + h(\mathbb{X}_4) \\ \mathbb{X}_4 = a + h(\mathbb{X}_3) \end{cases}$$

The six definitions above have established a RTE equation system, which is used in a *Gaussian-elimination-like* (GEL) method described further in the following section.

3.2 Gaussian-elimination-like method

The main idea behind the method proposal is to transform a RTE equation system $\mathcal{X} = \{\mathbb{X}_j = F_j \mid j \in \hat{n}\}$ into an equivalent RTE equation system $\mathcal{X}' = \{\mathbb{X}'_j = F'_j \mid j \in \hat{n}\}$, which is without variables. [4] As the resultant equation system is without variables, it is a RTE over a ranked alphabet Σ and can therefore denote a language over Σ.

To describe the GEL method, *substitution* over RTE needs to be defined.

Definition 3.10. For a RTE E, $E_{\mathbb{X} \leftarrow E'}$ denotes the expression obtained by substituting any occurrence of the variable $\mathbb{X}$ by the RTE E' in the RTE E. Such transformation is then defined inductively as:

- $a_{\mathbb{X} \leftarrow E'} = a$,
- $\emptyset_{\mathbb{X} \leftarrow E'} = \emptyset$,
- $\mathbb{Y}_{\mathbb{X} \leftarrow E'} = \mathbb{Y}$,
- $\mathbb{X}_{\mathbb{X} \leftarrow E'} = E'$,
- $(f(E_1, \ldots, E_n))_{\mathbb{X} \leftarrow E'} = f((E_1)_{\mathbb{X} \leftarrow E'}, \ldots, (E_n)_{\mathbb{X} \leftarrow E'})$,
- $(E_1 + E_2)_{\mathbb{X} \leftarrow E'} = (E_1)_{\mathbb{X} \leftarrow E'} + (E_2)_{\mathbb{X} \leftarrow E'}$,
- $(E_1 \cdot_c E_2)_{\mathbb{X} \leftarrow E'} = (E_1)_{\mathbb{X} \leftarrow E'} \cdot_c (E_2)_{\mathbb{X} \leftarrow E'}$,
- $(E_1^{*c})_{\mathbb{X} \leftarrow E'} = ((E_1)_{\mathbb{X} \leftarrow E'})^{*c}$,

where $a, c \in \Sigma_0$, $f \in \Sigma$, $\mathbb{X} \neq \mathbb{Y}$ are two variables in X, and $E_1, \ldots, E_n$ are any n RTEs over (Σ, X). [4]

The substitution of $\mathbb{X}_k = F_k$, $k \in \hat{n}$ in a RTE equation system $\mathcal{X} = \{\mathbb{X}_j = F_j \mid j \in n\}$ over n variables then gives the system $\mathcal{X}^k = \{\mathbb{X}_k = F_k\} \cup \{\mathbb{X}_j = (F_j)_{\mathbb{X}_k \leftarrow F_k} \mid j \neq k \wedge j \in \hat{n}\}$. The systems $\mathcal{X}$ and $\mathcal{X}^k$ are then equivalent. [4]

Given $\mathcal{X}$ is an equation system with n equations over (Σ, X), φ is a permutation on $\hat{n}$, then GEL equation system $\mathcal{X}^\varphi$ is the n-th term of the sequence

$$\begin{cases} \mathcal{X}_1 = X^{\varphi(1)} \\ \mathcal{X}_i = X_{i-1}^{\varphi(i)}. \end{cases}$$

Then, $\mathcal{X}$ and $\mathcal{X}^\varphi$ are equivalent.

Definition 3.11. The relation $<_\mathcal{X}$ for any two variables $\mathbb{X}_j, \mathbb{X}_k$ from a RTE equation system over n variables $\mathcal{X} = \{\mathbb{X}_j = F_j \mid j \in \hat{n}\}$ is defined by:

$$\mathbb{X}_j <_\mathcal{X} \mathbb{X}_k \Leftrightarrow \mathbb{X}_j \text{ appears in } F_k.$$

The relation $\preceq_\mathcal{X}$ is defined as the transitive closure of $<_\mathcal{X}$. If $\mathbb{X}_k <_\mathcal{X} \mathbb{X}_k$, the equation $\mathbb{X}_k = F_k$ is called *recursive*. A system is called recursive if a symbol $\mathbb{X}_k$ such that $\mathbb{X}_k \preceq_\mathcal{X} \mathbb{X}_k$ exist. [4]

3.3 Non-recursive case

If $\mathcal{X}$ is a non-recursive RTE equation system, then there exists a permutation φ on $\hat{n}$ for which $\mathcal{X}^\varphi$ is a system without variables. [4]

This means that for a non-recursive case, it is sufficient to find a sequence of variables to replace to find an equivalent equation system without variables, which yields a solution.

Example 3.12. Consider the following RTE equation system:

$$\mathcal{X} = \begin{cases} \mathbb{X}_1 = f(\mathbb{X}_2) \\ \mathbb{X}_2 = f(\mathbb{X}_3) + c \\ \mathbb{X}_3 = a + b \end{cases}$$

Then, substituting $\mathbb{X}_3$ by its associated RTE $a + b$ results in:

$$\mathcal{X} = \begin{cases} \mathbb{X}_1 = f(\mathbb{X}_2) \\ \mathbb{X}_2 = f(a + b) + c \\ \mathbb{X}_3 = a + b \end{cases}$$

And substituting $\mathbb{X}_2$ by its associated RTE $f(a+b)+c$ results in an equation system without variables.

$$\mathcal{X} = \begin{cases} \mathbb{X}_1 = f(f(a + b) + c) \\ \mathbb{X}_2 = f(a + b) + c \\ \mathbb{X}_3 = a + b \end{cases}$$

3.4 Recursive case

If a RTE equation system is recursive, using substitutions is not sufficient to obtain a solution for such system.

Arden's lemma states that for a regular expression equation $X = A \cdot X \cup B$, $A^* \cdot B$ is the smallest language that is a solution for the equation X. [8] It is then used for the following lemma:

Lemma 3.13. Let A and B be two tree languages over Σ and E_A, E_B be two RTEs over Σ such that $[E_A] = A$ and $[E_B] = B$. Then the RTE $E_A^{*c} \cdot_c E_B$ denotes the smallest language over Σ satisfying $L = A \cdot_c L \cup B$. [4]

In lemma 3.13, the bounded language, which is the language containing the variable causing recursion, is treated as A and the free language (not containing the recursion causing variable) is treated as B.

Any recursive equation system can be transformed into an equivalent equation system for which there exists a symbol $\mathbb{X}_j$ satisfying $\mathbb{X}_j <_{\mathcal{X}} \mathbb{X}_j$.

Any RTE E can be transformed into an equivalent RTE $E' + E''$ such that a variable $\mathbb{X}$ figures in E' and not in E''. [4]

Example 3.14. For the RTE $E = f(a, \mathbb{X}_1) + h(b)$, the split over the variable $\mathbb{X}_1$ is $E' = f(a, \mathbb{X}_1)$, and $E'' = h(b)$.

To allow for such transformation of any RTE E, first the *decomposition* of E needs to be defined.

Definition 3.15. The decomposition of E is defined inductively as:

$$dec(E) = \begin{cases} \{E\} & \text{if } E = c, E = \mathbb{X}, E = f(E_1, \ldots, E_n), \\ & \; E = E_1 \cdot_c E_2, \text{ or } E = E_1^{*c} \\ dec(E_1) \cup dec(E_2) & \text{if } E = E_1 + E_2 \end{cases}$$

Then, for a given variable $\mathbb{X}$, $dec_{\mathbb{X}}(E) = \{F \in dec(E) \mid \mathbb{X} \text{ appears in } f\}$, and $dec_{\bar{\mathbb{X}}}(E) = dec(E) \setminus dec_{\mathbb{X}}(E)$.

Definition 3.16. For an integer j, the *j-split* of an expression E over $(\Sigma, \{\mathbb{X}_1, \ldots, \mathbb{X}_2\})$ is the pair defined by:

$$j\text{-split}(E) = (\textstyle\sum_{F \in dec_{\mathbb{X}_j}(E)} F, \sum_{F \in dec_{\bar{\mathbb{X}}_j}(E)} F)$$

Then, through proposing that $[E]_{\mathcal{L}} = [(E)_{\mathbb{X} \leftarrow a} \cdot_a \mathbb{X}]_{\mathcal{L}}$, and $\mathcal{L}$ is a language over an alphabet which does not contain constants appearing in either the concatenation or iteration operators in F_j, the following two conditions are equivalent:

(1) $\mathcal{L}$ is a solution for $\mathbb{X}_k = F_j$,

(2) $\mathcal{L}$ is a solution for $(F'_j)_{\mathbb{X}_j \leftarrow a} \cdot_a \mathbb{X}_j + F''_j$.

Then, *contraction* of $\mathbb{X}_k = F_k$ is defined as $\mathbb{X}_k = (F'_k)^{*c} \cdot_c F''_k$. Such an operation is used together with the j-split of a RTE. [4]

Example 3.17. Given the equation system from example 3.9, the 2-split of $\mathbb{X}_2 = b + f(\mathbb{X}_2, \mathbb{X}_4)$ is $(f(\mathbb{X}_2, \mathbb{X}_4), b)$. Then, using a concatenation symbol x_2, the next step yields $f(x_2, \mathbb{X}_4) \cdot_{x_2} \mathbb{X}_2 + b$. Using contraction, the final result is $f(x_2, \mathbb{X}_4)^{*x_2} \cdot_{x_2} b$.

The problem of such approach is that the expression produced is not guaranteed to be equivalent. The cause of this problem is the wording of the proposal which uses language $\mathcal{L}$ over an alphabet not containing constants appearing in either the concatenation or iteration operators. However, it can be remedied by defining a property in order to detect which systems are solvable.

The property in question is called *closedness*. An expression is said to be closed if all occurrences of a bounded symbol are bounded. [4]

Example 3.18. In the RTE $E = f(x_1, a) \cdot_{x_1} b$, x_1 is a bounded symbol, because it appears as the symbol of the concatenation operator. E is also closed, because all occurrences of x_1 are bounded.
On the other hand, the RTE $E' = f(x_1, a) \cdot_{x_1} x_1$ is not closed, because not every occurrence of x_1 is bounded.

Then, using the lemma that for two closed expressions F, F' over $(\Sigma, \mathbb{X})$, $F_{\mathbb{X}_k \leftarrow F'}$ is also closed, it is possible to get to the result that for a closed RTE $E = F \cdot_c F' + F''$, $F^{*c} \cdot_c F''$ is also closed. [4] The proofs and corollaries are omitted here and can be found in the source.

The closedness is a satisfactory condition for an equation system being solvable. However, it is not a necessary condition. A system which is not closed may still be solvable.

Then, an equation system $\mathcal{X}$ admits a solution over a set of symbols which are not bounded. It is proven recurrently that since a closed systems of one equation, and closed systems which are not recursive, admit such solution, and it is possible to transform a recursive equation into a non-recursive one using the j-split. [4]

3.5 Construction of RTE equation system from FTA

Now that it is established how to solve a RTE equation system, it is possible to move to constructing the equation system from a given FTA. The proposed method is the following: given a FTA $A = (\Sigma, Q, Q_f, \Delta)$, its *associated equation system* is the set of equations $\mathcal{X}_A$ over the variables $\mathbb{X}_1, \ldots, \mathbb{X}_n$ defined by $\mathcal{X}_A = \{\mathbb{X}_q = F_q \mid q \in Q\}$, where for any state $q \in Q$, $F_q = \Sigma_{(f, q_1, \ldots, q_n, q) \in \Delta} f(\mathbb{X}_{q_1}, \ldots, \mathbb{X}_{q_n})$.

Definition 3.19. Then, for such a FTA, and $(E_1, \ldots, E_n)$ being a solution of its associated equation system, $L(A)$ is denoted by the RTE $\Sigma_{q_j \in Q_f} E_j$. [4]

In the source, it is noted that the equation system can be solved when it is closed. To ensure closedness, to each equation is associated an extra constant not in Σ_0, to be used in the iteration and concatenation operators while solving the equation system.

After such an equation system is obtained, it can be solved by successive substitutions and using the j-split.

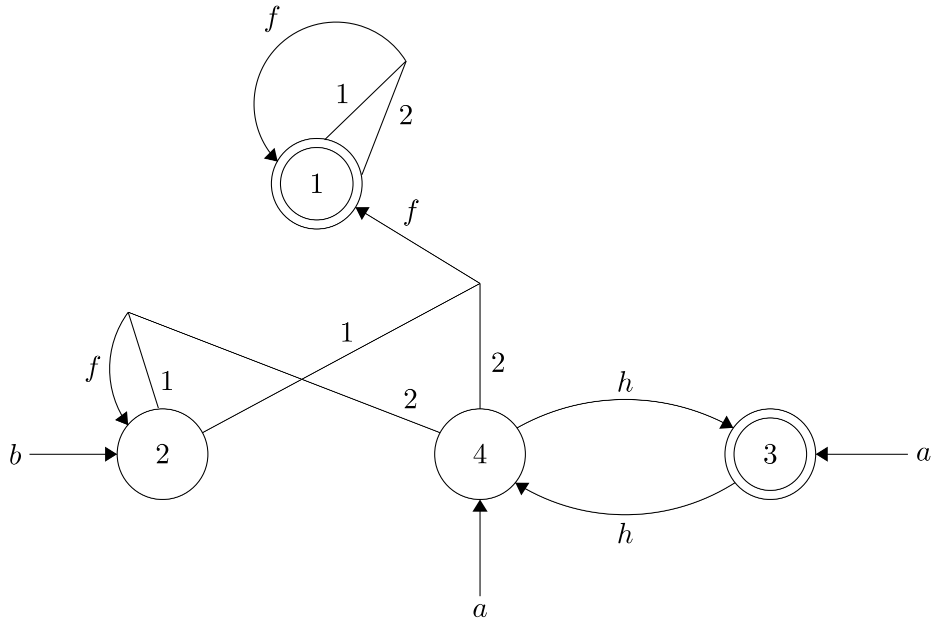

Figure 3.1: FTA A

Example 3.20. The RTE equation system associated with the FTA A from figure 3.1 is the following:

$$\mathcal{X} = \begin{cases} \mathbb{X}_1 = f(\mathbb{X}_1, \mathbb{X}_1) + f(\mathbb{X}_2, \mathbb{X}_4) \\ \mathbb{X}_2 = b + f(\mathbb{X}_2, \mathbb{X}_4) \\ \mathbb{X}_3 = a + h(\mathbb{X}_4) \\ \mathbb{X}_4 = a + h(\mathbb{X}_3) \end{cases}$$

The equation system $\mathcal{X}^4$ is obtained through replacing all occurrences of $\mathbb{X}_4$ by its right-hand side in equations for other variables.

$$\mathcal{X}^4 = \begin{cases} \mathbb{X}_1 = f(\mathbb{X}_1, \mathbb{X}_1) + f(\mathbb{X}_2, a + h(\mathbb{X}_3)) \\ \mathbb{X}_2 = b + f(\mathbb{X}_2, a + h(\mathbb{X}_3)) \\ \mathbb{X}_3 = a + h(a + h(\mathbb{X}_3)) \\ \mathbb{X}_4 = a + h(\mathbb{X}_3) \end{cases}$$

Then, the following closed subsystem can be solved

$$\begin{cases} \mathbb{X}_1 = f(\mathbb{X}_1, \mathbb{X}_1) + f(\mathbb{X}_2, a + h(\mathbb{X}_3)) \\ \mathbb{X}_2 = b + f(\mathbb{X}_2, a + h(\mathbb{X}_3)) \\ \mathbb{X}_3 = a + h(a + h(\mathbb{X}_3)) \end{cases}.$$

As no equation is non-recursive now, a j-split is performed. With j being chosen as 3, the 3-split of $a + h(a + h(\mathbb{X}_3))$ leads, through factorization and contraction, to $(h(a+h(x_3)))^{*x_3} \cdot_{x_3} a$. By substitution of $\mathbb{X}_3$, whose recursion has been removed now, in the remaining equations, the new subsystem to solve is obtained.

$$\begin{cases} \mathbb{X}_1 = f(\mathbb{X}_1, \mathbb{X}_1) + f(\mathbb{X}_2, a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a))) \\ \mathbb{X}_2 = b + f(\mathbb{X}_2, a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a))) \end{cases}$$

Again, both equations being recursive, a j-split with $j = 2$ is performed. The 2-split of $b+f(\mathbb{X}_2, a+h((h(a+h(x_3)))^{*x_3} \cdot_{x_3} a))$ leads, through factorization and contraction, to $(f(x_2, a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a)))^{*x_2} \cdot_{x_2} b$. By substituting $\mathbb{X}_2$ in the equation of $\mathbb{X}_1$, it remains to solve

$$\mathbb{X}_1 = f(\mathbb{X}_1, \mathbb{X}_1) + f((f(x_2, a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a)))^{*x_2} \cdot_{x_2} b, a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a))).$$

The 1-split of the above right hand side yields:

$$(f(x_1, x_1))^{*x_1} \cdot_{x_1} f((f(x_2, a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a)))^{*x_2} \cdot_{x_2} b, a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a))).$$

As all states are recursion free now, the solution is obtained by replacing the variable occurrences by the resultant right hand sides.

$$\begin{cases} \mathbb{X}_1 = & (f(x_1, x_1))^{*x_1} \cdot_{x_1} f((f(x_2, a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a)))^{*x_2} \cdot_{x_2} b, \\ & a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a))) \\ \mathbb{X}_2 = & (f(x_2, a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a)))^{*x_2} \cdot_{x_2} b \\ \mathbb{X}_3 = & (h(a + h(x_3)))^{*x_3} \cdot_{x_3} a \\ \mathbb{X}_4 = & a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a) \end{cases}$$

Since the final states are 1 and 3, as per definition 3.19, $L(A)$ is denoted by:

$$(f(x_1, x_1))^{*x_1} \cdot_{x_1} f((f(x_2, a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a)))^{*x_2} \cdot_{x_2} b, a + h((h(a + h(x_3)))^{*x_3} \cdot_{x_3} a))) + (h(a + h(x_3)))^{*x_3} \cdot_{x_3} a.$$

3.6 Viability and challenges

The method presented in this chapter is quite straightforward and easy to understand. However, there are two challenges that need to be addressed before the implementation design can be proposed. They will be presented here and solutions will be proposed in further chapters.

3.6.1 Number of alternation elements

As per definition 1.23, an alternation is of the form $E_1 + E_2$. However, in the previous sections, the construction of a RTE equation for a state of a FTA is presented as a sum of transitions into the state. While this can be achieved by nesting alternations into each other, for implementation purposes, this would be very impractical.

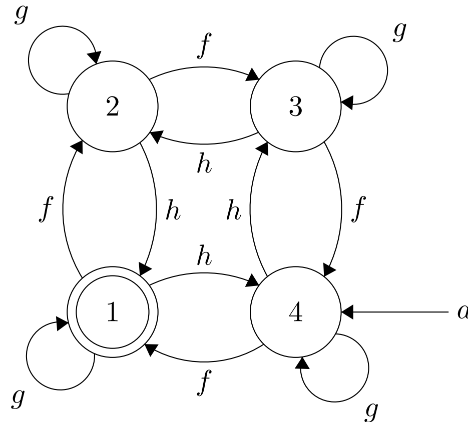

Figure 3.2: FTA A

Example 3.21. Consider the FTA from figure 3.2. Each of the states has three transitions going into it, with state 4 having four. For such state, the equation constructed could look like $\mathbb{X}_4 = (f(\mathbb{X}_3) + g(\mathbb{X}_4)) + (a + h(\mathbb{X}_1))$. Then, while performing a 4-split, $f(\mathbb{X}_3)$ would get "stuck" in $\mathbb{X}_4'$, and be in the operand for iteration and concatenation, while not containing the symbol $\mathbb{X}_4$. While the result would be valid, readability would be improved by allowing more elements in alternations.

3.6.2 *j*-split

The challenge for a j-split is that for some RTE equation denoted by right hand side E, the second set, E'', could be empty. By applying the RTE equation method on E, according to the definition of j-split 3.16, and through

substitution and contraction, E would be $(E')^{*c} \cdot_c \emptyset$. However, the equation system may still admit solutions and such challenge only be a result of unfortunately chosen sequence of variables to split.

Chapter **4**

RTE axioms

The RTE axioms are an important part of this work for several reasons. Firstly, it is not difficult imagining a regular tree expression which could be easily simplified while remaining equivalent. However, such changes need to be formalized, which the axioms achieve. Secondly, as implementation of an algorithm whose output is a RTE is a part of this work, simplifying the final result to be more easily readable – both by humans and by computers – is definitely a desirable outcome. Thirdly, they will show to help resolve the challenges of the conversion method presented in the previous chapter. And lastly, while it may not show immediately, a further study of RTEs may show the proposed axioms find their use elsewhere.

4.1 Former work on RTE axioms

Regular tree expression axioms have been studied, along with other properties of RTEs, in [2] and [7]. In [2], numerous axioms were proposed, mainly based on established axioms for regular (string) expressions. For the purposes of his work, the author augmented the RTEs to be over alphabet $\Sigma \cup K$, with K being a set of constants, $c \in K$, with $\Sigma \cap K = \emptyset$. Note that this is equivalent with extending the alphabet Σ_0 with concatenation and iteration symbols to ensure closedness in the FTA to RTE conversion method. The constants from K are used as parts of the concatenation and iteration operators, that is $E_1 \cdot_c E_2$ and E^*c. This is consistent with the definitions used in previous chapters, where constants such as $x_1, \ldots, x_n$ were used as the placeholder symbols. Then, given $x, y, z \in T(\Sigma \cup K)$, $a \in T(\Sigma \cup K \setminus c)$ and $k, l \in T(\Sigma \cup K) \setminus T(\Sigma \cup K \setminus c)$, and ε being a special constant denoting an empty word, the following axioms were proposed.

Alternation

Given definition 1.24 stating that $[E_1 + E_2] = [E_1] \cup [E_2]$, it is concluded that the following axioms apply.

The following axioms are in line with known axioms for regular expressions. [9] [7]

RTE Axiom 1. *(A1).* $x + (y + z) = (x + y) + z$

RTE Axiom 2. *(A2).* $x + y = y + x$

RTE Axiom 3. *(A3).* $x + \emptyset = x$

RTE Axiom 4. *(A4).* $x + x = x$

Concatenation

For concatenation, not every regular string axiom was possible to transform into a RTE one. The reason for this is the difference between the concatenation operators in regular expressions and RTEs.

RTE Axiom 5. *(A5).* $x \cdot_c (y \cdot_c z) = (x \cdot_c y) \cdot_c z$

RTE Axiom 6. *(B1).* $\varepsilon \cdot_c x = \varepsilon$

RTE Axiom 7. *(B2).* $\emptyset \cdot_c x = \emptyset$

RTE Axiom 8. *(B3).* $c \cdot_c x = x$

RTE Axiom 9. *(B4).* $a \cdot_c x = a$

RTE Axiom 10. *(B5)* $k \cdot_c \varepsilon \in T(\Sigma \cup K \setminus c)$

RTE Axiom 11. *(B6).* $k \cdot_c \emptyset = \emptyset$

RTE Axiom 12. *(B7).* $k \cdot_c c = k$

RTE Axiom 13. *(B8).* $k \cdot_c a \in T(\Sigma \cup K \setminus c)$

RTE Axiom 14. *(B9).* $k \cdot_c l \in T(\Sigma \cup K) \setminus T(\Sigma \cup K \setminus c)$

RTE Axiom 15. *(A8).* $x \cdot_c (y + z) = x \cdot_c y + x \cdot_c z$ if x does not contain more than one occurrence of the concatenation symbol c

RTE Axiom 16. *(A9).* $(x + y) \cdot_c z = x \cdot_c y + y \cdot_c z$

Iteration

For iteration, the axioms are derived from existing regular expression axioms. [2] [9]

RTE Axiom 17. *(A10).* $x^{*c} = \varepsilon + x^{*c} \cdot_c x$

RTE Axiom 18. *(A11).* $x^{*c} = (\varepsilon + x)^{*c}$

Regular equations

The axioms for regular equations are also derived from existing regular expression axioms. Note that the axiom *(A13).* found its use in the conversion method presented in previous chapters.

RTE Axiom 19. *(A12).* $x = x \cdot_c \alpha + \beta \Rightarrow x = \beta \cdot_c \alpha^{*c}$

RTE Axiom 20. *(A13).* $x = \alpha \cdot_c x + \beta \Rightarrow x = \alpha^{*c} \cdot_c \beta$

4.2 Alteration of axioms

In this section, axioms from section 4.1 which need to be altered are discussed and the changes are proposed.

4.2.1 Axiom (B6)

In axiom *(B6)*, it is proposed that for $k \in T(\Sigma \cup K) \setminus T(\Sigma \cup K \setminus c)$, the following applies: $k \cdot_c \emptyset = \emptyset$. However, upon closer inspection it was found out that this is not always the case and the axiom needs to be altered.

Example 4.1. Consider the following RTE: $(f(c) + f(a))^{*c} \cdot_c \emptyset$. Clearly, $(f(c)+f(a))^{*c} \in T(\Sigma \cup K) \setminus T(\Sigma \cup K \setminus c)$. However, one of the trees described by such RTE is $f(a)$. Then, in accordance with axiom *(B4)*, $f(a) \cdot_c x = f(a)$. Therefore, the described language is not empty and the axiom *(B6)* needs to be changed.

As can be seen from the example provided above, the condition that $k \in T(\Sigma \cup K) \setminus T(\Sigma \cup K \setminus c)$ is not strong enough, as even such trees can describe languages which contain RTEs not containing c. Clearly, the language that k is describing must not contain any RTE from $T(\Sigma \cup K \setminus c)$ at all. With that in consideration, the following is proposed:

RTE Axiom 21. *(B6-1).* $k \cdot_c \emptyset = \emptyset$ if $[k] \cap T(\Sigma \cup K \setminus c) = \emptyset$

It is clear that for k which does not match the above condition, even a concatenation with $\emptyset$ as the right operand still describes a non-empty language. It is now a question of finding out what language it describes. However, one thing is certain: it does not contain any occurrences of c.

RTE Axiom 22. *(B6-2).* $k \cdot_c \emptyset \in T(\Sigma \cup K \setminus c)$

4.3 Unbounded RTE

As was mentioned in subsection 3.6.1, to make implementation and readability easier, it would be helpful to allow more than two elements in an alternation. Therefore, *unbounded* regular tree expressions were defined.

Definition 4.2. An *unbounded* regular tree expression E over ranked alphabet Σ is inductively defined by:

- $E = 0$,
- $E = f(E_1, \ldots, E_n)$,
- $E = E_1 + \ldots + E_m$,
- $E = E_1 \cdot_c E_2$,
- $E = E_1^* c$,

where $f \in \Sigma_n$, $c \in \Sigma_0$, $E_1, \ldots E_n$ are any n unbounded RTE over Σ, and $E_1, \ldots E_m$ are any $m > 0$ unbounded RTE over Σ.

As one of the operators has been re-defined, the language denoted by unbounded regular tree expressions also needs to be re-defined.

Definition 4.3. The language denoted by an unbounded RTE E is the tree language $[E]$, defined inductively by:

- $[\emptyset] = \emptyset$,
- $[f(E_1, \ldots, E_n)] = f([E_1], \ldots, [E_n])$,
- $[E_1 + \ldots + E_m] = [E_1] \cup \ldots \cup [E_m]$,
- $[E_1 \cdot_c E_2] = [E_1] \cdot_c [E_2]$,
- $[E_1^* c] = [E_1]^* c$,

where $f \in \Sigma_n$, $c \in \Sigma_0$, $E_1, \ldots E_n$ are any n RTE, and $E_1, \ldots E_m$ are any $m > 0$ unbounded RTE over Σ.

4.3.1 Impact on formerly proposed axioms

Obviously, the only axioms the new definition could have affected are the ones with an alternation operator in them. They will be listed below, with U after their respective numbers to reflect the fact they are used for unbounded RTEs.

Alternation

The only affected axiom is *(A1)*, which allows for removing the parentheses in unbounded RTEs.

RTE Axiom 23. *(A1U).* $x + (y + z) = (x + y) + z = x + y + z$

The other axioms from this group do not need to be modified, as for larger number of elements they can be deduced from the not-unbounded ones.

Concatenation

The axioms for concatenation distributivity *(A8)*, *(A9)* need to be modified to allow for an unlimited number of elements in alternation.

RTE Axiom 24. *(A8U).* $x \cdot_c (y_1 + \ldots + y_n) = x \cdot_c y_1 + \ldots + x \cdot_c y_n$ if x does not contain more than one occurrence of the concatenation symbol c

RTE Axiom 25. *(A9U).* $(x_1 + \ldots + x_n) \cdot_c y = x_1 \cdot_c y + \ldots + x_n \cdot_c y$

The other axioms from this group do not need to be modified.

Iteration and regular equations

No axioms from these groups need to be modified.

4.4 Proposal of new RTE axioms

For the proposal of new RTE axioms, the inspiration came from different sources: preexisting regular expression axioms [9], some of which were also implemented or described in [10], and study of languages denoted by the RTEs, as presented in definitions 1.24 and 4.3.

Note. The newly proposed axioms' names start with the letter *N*, as for *new*. In order to keep consistency, the prerequisites and definitions from section 4.1 are kept.

Ranked symbol

These axioms describe how a RTE describing a ranked symbol of arity n can be transformed.

RTE Axiom 26. *(N1).* $f(x_1 \cdot_c z, \ldots, x_n \cdot_c z) = f(x_1, \ldots, x_n) \cdot_c z$

The above axiom *(N1)* is analogous to the axiom *(A9)*. If all children of the ranked symbol f are a concatenation through $c \in K$, then the concatenation can be placed outside f and f's children modified by removing the concatenation, and vice versa.

RTE Axiom 27. *(N2).* $f(y_1, \ldots, y_j, x_1 + x_2, z_1, \ldots, z_k) = f(y_1, \ldots, y_j, x_1, z_1, \ldots, z_k) + f(y_1, \ldots, y_j, x_2, z_1, \ldots, z_k)$

The axiom *(N2)* needs to be modified for unbounded RTEs.

RTE Axiom 28. *(N2U).* $f(y_1, \ldots, y_j, x_1 + \ldots + x_n, z_1, \ldots, z_k) = f(y_1, \ldots, y_j, x_1, z_1, \ldots, z_k) + \ldots + f(y_1, \ldots, y_j, x_n, z_1, \ldots, z_k)$

RTE Axiom 29. *(N3).* $f(x_1, \ldots, \emptyset, \ldots, x_n) = \emptyset$

Iteration

This section proposes axioms for RTEs with an iteration operator. Most of them are analogous to regular expression axioms. [9]

RTE Axiom 30. *(N4).* $x^{*c} = (x^{*c})^{*c}$

RTE Axiom 31. *(N5).* $(x + y)^{*c} = (x^{*c} + y^{*c})^{*c}$

The axiom *(N5)* needs to be modified for unbounded RTEs.

RTE Axiom 32. *(N5U).* $(x_1 + \ldots + x_n)^{*c} = (\sum_{i \in N_1} x_i^{*c} + \sum_{j \in N_2} x_j)^{*c}$, $N_1 \cup N_2 = \hat{n}$, $N_1 \cap N_2 = \emptyset$

Axiom *(N5U)* is a generalization of axiom *(N5)*. It says that if the operand of an iteration over c is an alternation, any of the elements of the alternation may be written as either themselves in unchanged form or as an iteration of themselves over c, and vice versa.

RTE Axiom 33. *(N6).* $x^{*c} = x^{*c} + c$

RTE Axiom 34. *(N7).* $(x + y)^{*c} = (x^{*c} \cdot_c y^{*c})^{*c} = (y^{*c} \cdot_c x^{*c})^{*c}$

RTE Axiom 35. *(N8).* $x^{*c} = x^{*c} \cdot_c x + c$

RTE Axiom 36. *(N9).* $x^{*c} = (x + c)^{*c}$

RTE Axiom 37. *(N10).* $x^{*c} = x^{*c} + x$

RTE Axiom 38. *(N11).* $a^{*c} = a + c$

RTE Axiom 39. *(N12).* $\emptyset^{*c} = c$

The following axiom describes a subset property of the iteration operator.

RTE Axiom 40. *(N13).* $x^{*c} \subseteq (x + y)^{*c}$

Concatenation

The following two axioms describe subset properties of the concatenation operator.

RTE Axiom 41. *(N14).* $x \cdot_c y \subseteq x \cdot_c (y + z)$

RTE Axiom 42. *(N15).* $x \cdot_c z \subseteq (x + y) \cdot_c z$

Chapter **5**

Resolving challenges in the conversion method

The challenges of the method presented in chapter 3 were shown in subsections 3.6.1 and 3.6.2. By proposing an unbounded RTE in definition 4.2, the first challenge has been resolved. This chapter will then focus on the second challenge, regarding j-split.

5.1 Demonstrating the problem

Throughout this chapter, an example tree automaton from the Algorithm Library Toolkit [10] will be used. The tree automaton is defined as follows: $A = (\{(a, 2), (b, 1), (c, 0)\}, \{1, 2, 3, 4, 5\}, \{3, 4, 5\}, \Delta)$, where the set of transitions Δ is described in table 5.1.

Then, a RTE equation system can be constructed from such automaton.

Example 5.1. The RTE equation system for automaton A:

$$\mathcal{X} = \begin{cases} \mathbb{X}_1 = & b(\mathbb{X}_2) + b(\mathbb{X}_5) \\ \mathbb{X}_2 = & a(\mathbb{X}_3, \mathbb{X}_3) + a(\mathbb{X}_3, \mathbb{X}_4) + a(\mathbb{X}_3, \mathbb{X}_5) \\ \mathbb{X}_3 = & c \\ \mathbb{X}_4 = & a(\mathbb{X}_1, \mathbb{X}_3) + a(\mathbb{X}_1, \mathbb{X}_4) + a(\mathbb{X}_1, \mathbb{X}_5) \\ \mathbb{X}_5 = & a(\mathbb{X}_4, \mathbb{X}_3) + a(\mathbb{X}_4, \mathbb{X}_4) + a(\mathbb{X}_4, \mathbb{X}_5) \\ & +a(\mathbb{X}_5, \mathbb{X}_3) + a(\mathbb{X}_5, \mathbb{X}_4) + a(\mathbb{X}_5, \mathbb{X}_5) \end{cases}$$

The equations for $\mathbb{X}_1$, $\mathbb{X}_2$, and $\mathbb{X}_3$ are non-recursive, so the first three steps of the conversion method will be to substitute them (in any order) and solve a subsystem without them.

Symbol	From	To
c		3
b	2	1
b	5	1
a	3, 3	2
a	3, 4	2
a	3, 5	2
a	1, 3	4
a	1, 4	4
a	1, 5	4
a	4, 3	5
a	4, 4	5
a	4, 5	5
a	5, 3	5
a	5, 4	5
a	5, 5	5

Table 5.1: Transitions of the FTA A

$$\begin{cases} \mathbb{X}_2 = & a(\mathbb{X}_3, \mathbb{X}_3) + a(\mathbb{X}_3, \mathbb{X}_4) + a(\mathbb{X}_3, \mathbb{X}_5) \\ \mathbb{X}_3 = & c \\ \mathbb{X}_4 = & a(b(\mathbb{X}_2) + b(\mathbb{X}_5), \mathbb{X}_3) + a(b(\mathbb{X}_2) + b(\mathbb{X}_5), \mathbb{X}_4) + a(b(\mathbb{X}_2) + b(\mathbb{X}_5), \mathbb{X}_5) \\ \mathbb{X}_5 = & a(\mathbb{X}_4, \mathbb{X}_3) + a(\mathbb{X}_4, \mathbb{X}_4) + a(\mathbb{X}_4, \mathbb{X}_5) \\ & +a(\mathbb{X}_5, \mathbb{X}_3) + a(\mathbb{X}_5, \mathbb{X}_4) + a(\mathbb{X}_5, \mathbb{X}_5) \end{cases}$$

$$\begin{cases} \mathbb{X}_3 = & c \\ \mathbb{X}_4 = & a(b(a(\mathbb{X}_3, \mathbb{X}_3) + a(\mathbb{X}_3, \mathbb{X}_4) + a(\mathbb{X}_3, \mathbb{X}_5)) + b(\mathbb{X}_5), \mathbb{X}_3) \\ & +a(b(a(\mathbb{X}_3, \mathbb{X}_3) + a(\mathbb{X}_3, \mathbb{X}_4) + a(\mathbb{X}_3, \mathbb{X}_5)) + b(\mathbb{X}_5), \mathbb{X}_4) \\ & +a(b(a(\mathbb{X}_3, \mathbb{X}_3) + a(\mathbb{X}_3, \mathbb{X}_4) + a(\mathbb{X}_3, \mathbb{X}_5)) + b(\mathbb{X}_5), \mathbb{X}_5) \\ \mathbb{X}_5 = & a(\mathbb{X}_4, \mathbb{X}_3) + a(\mathbb{X}_4, \mathbb{X}_4) + a(\mathbb{X}_4, \mathbb{X}_5) \\ & +a(\mathbb{X}_5, \mathbb{X}_3) + a(\mathbb{X}_5, \mathbb{X}_4) + a(\mathbb{X}_5, \mathbb{X}_5) \end{cases}$$

$$\begin{cases} \mathbb{X}_4 = & a(b(a(c, c) + a(c, \mathbb{X}_4) + a(c, \mathbb{X}_5)) + b(\mathbb{X}_5), c) \\ & +a(b(a(c, c) + a(c, \mathbb{X}_4) + a(c, \mathbb{X}_5)) + b(\mathbb{X}_5), \mathbb{X}_4) \\ & +a(b(a(c, c) + a(c, \mathbb{X}_4) + a(c, \mathbb{X}_5)) + b(\mathbb{X}_5), \mathbb{X}_5) \\ \mathbb{X}_5 = & a(\mathbb{X}_4, c) + a(\mathbb{X}_4, \mathbb{X}_4) + a(\mathbb{X}_4, \mathbb{X}_5) \\ & +a(\mathbb{X}_5, c) + a(\mathbb{X}_5, \mathbb{X}_4) + a(\mathbb{X}_5, \mathbb{X}_5) \end{cases}$$

Where the challenge comes in is now. Both equations for $\mathbb{X}_4$ and $\mathbb{X}_5$ are recursive, so a j-split needs to be performed. Assume the 4-split is performed

first. Then, the right hand side of $\mathbb{X}_4$ is

$$\begin{aligned} & a(b(a(c,c)+a(c,\mathbb{X}_4)+a(c,\mathbb{X}_5))+b(\mathbb{X}_5),c) \\ +& a(b(a(c,c)+a(c,\mathbb{X}_4)+a(c,\mathbb{X}_5))+b(\mathbb{X}_5),\mathbb{X}_4) \\ +& a(b(a(c,c)+a(c,\mathbb{X}_4)+a(c,\mathbb{X}_5))+b(\mathbb{X}_5),\mathbb{X}_5) \end{aligned} \tag{5.1}$$

which is an alternation of three occurrences of the ranked symbol $(a, 2)$. All of its occurrences contain the symbol $\mathbb{X}_4$, so for the 4-split of such RTE, $\mathbb{X}_4''$ would be empty.

5.2 Solution using RTE axioms

With the alteration of axiom *(B6)*, it is clear that even a concatenation $\mathbb{X}_4' \cdot_{x_4} \emptyset$ may be describing a non-empty language. However, it is not defined what exactly such concatenation is describing and the problem can be remedied in another way – using axioms *(A4)*, *(N2U)*, *(N13)*, *(N14)*, *(N15)*, *(A3)*, *(N3)*, *(N12)*, *(B2)*, *(B6-1)*, and *(B6-2)*.

The proposed idea is the following:

1. Since $x + x = x$ *(A4)*, then $x + (x + y) = (x + x) + y = x + y$.
2. This means that if a right hand side of an equation is taken and its subset appended to it in an alternation, the RTE remains equivalent.
3. It is known how to take a subset from an alternation. Since the denoted language of E_1+E_2 is $[E_1]\cup[E_2]$, if only some of the operands are picked, such action results in a subset.
4. The axiom *(N13)* shows how to get a subset from an iteration.
5. Thanks to axioms *(N14)* and *(N15)*, it is known how to get a subset from concatenation.
6. The axiom *(N2U)* shows how to get a subset from a ranked symbol, if it contains an alternation within.

With such proposal in mind, a function *subset-without* can be defined for RTE E over $(\Sigma, \mathbb{X})$, and a variable $\mathbb{X}_j \in \mathbb{X}$ or a symbol $d \in \Sigma_0$:

Definition 5.2. The function *subset-without*: $RTE(\Sigma \cup \mathbb{X}) \times \{\Sigma \cup \mathbb{X}\} \to RTE(\Sigma \cup \mathbb{X})$, where $RTE(\Sigma \cup \mathbb{X})$ is the set of all RTEs over $\Sigma \cup \mathbb{X}$, is defined inductively as follows:

- $\textit{subset-without}(c, \mathbb{X}_j) = c$ for any $c \in \Sigma_0$,
- $\textit{subset-without}(c, d) = c$ for any $c \in \Sigma_0$, $c \neq d$,
- $\textit{subset-without}(d, d) = \emptyset$,

- *subset-without*$(\mathbb{X}_i, \mathbb{X}_j) = \mathbb{X}_i$, for $i \neq j$,
- *subset-without*$(\mathbb{X}_i, d) = \mathbb{X}_i$,
- *subset-without*$(\mathbb{X}_j, \mathbb{X}_j) = \emptyset$,
- *subset-without*$(\emptyset, \mathbb{X}_j) = \emptyset$,
- *subset-without*$(\emptyset, d) = \emptyset$,
- *subset-without*$(f(E_1, \ldots, E_n), \mathbb{X}_j) = f($*subset-without*$(E_1, \mathbb{X}_j), \ldots,$ *subset-without*$(E_n, \mathbb{X}_j))$,
- *subset-without*$(f(E_1, \ldots, E_n), d) = f($*subset-without*$(E_1, d), \ldots,$ *subset-without*$(E_n, d))$,
- *subset-without*$(E_1 + \ldots + E_n, \mathbb{X}_j) =$*subset-without*$(E_1, \mathbb{X}_j) + \ldots +$ *subset-without*$(E_n, \mathbb{X}_j)$,
- *subset-without*$(E_1 + \ldots + E_n, d) =$*subset-without*$(E_1, d) + \ldots +$ *subset-without*(E_n, d),
- *subset-without*$(E^{*c}, \mathbb{X}_j) = ($*subset-without* $(E, \mathbb{X}_j))^{*c}$,
- *subset-without*$(E^{*c}, d) = ($*subset-without* $(E, d))^{*c}$,
- *subset-without*$(E^{*d}, d) =$*subset-without* (E, d),
- *subset-without*$(E_1 \cdot_c E_2, \mathbb{X}_j) =$*subset-without*$(E_1, \mathbb{X}_j) \cdot_c$ *subset-without*$(E_2, \mathbb{X}_j)$,
- *subset-without*$(E_1 \cdot_c E_2, d) =$*subset-without*$(E_1, d) \cdot_c$ *subset-without*(E_2, d)

Now axioms which simplify RTEs containing the empty set $\emptyset$ will be utilized.

1. Thanks to axiom *(A3)*, it is known that $x + \emptyset = x$. Thus, *subset-without* of an alternation will be the elements of the alternation which either did not contain the variable being removed, or modified RTEs from the other RTE types. Only if all alternation elements contain the variable or constant and it cannot be removed from a single one, will the resulting RTE be an empty set.

2. Thanks to axiom *(N3)*, it is known that if any of the child elements of a ranked symbol are an empty set, the whole symbol yields an empty set. From this it can be concluded that *subset-without* of a ranked symbol is either the symbol with its child elements modified not to contain the variable or constant being removed, or an empty set in case it is not possible to modify at least one of them.

3. The axiom *(N12)* says that an iteration of an empty set can be simplified to the iteration symbol. Therefore, if the operand of the iteration cannot be modified to remove the variable or constant in question, the iteration can be simplified to the constant, otherwise the modified RTE is an iteration over the same constant, with a modified operand.

4. Axiom *(B2)* says that if the left operand of the concatenation operator is an empty set, the whole concatenation can be simplified to an empty set.

5. Axioms *(B6-1)* and *(B6-2)* say that if the right operand of the concatenation operator is an empty set, the whole concatenation can be simplified to either an empty set, or a subset of the left operand not containing the concatenation symbol. Therefore, for concatenation $E = F_1 \cdot_d F_2$ if *subset-without* of the right operand F_2 is an empty set, the outcome is $\textit{subset-without}(F_1, d)$.

6. Lastly, if both operands of a concatenation are not empty sets, the modified RTE will be a concatenation over the same constants, with modified operands.

With this knowledge, the next step is performing the 4-split. The RTE on right hand side of $\mathbb{X}_4$ is an alternation of three ranked symbols $(a, 2)$. The second occurrence of a contains $\mathbb{X}_4$ as the sole second element, therefore there is no way of modifying the occurrence to be without $\mathbb{X}_4$.

However, the first and third occurrence of a in the alternation have c and $\mathbb{X}_5$, respectively, as the sole second element of a, so the next step is to look at the first element of a, which is

$$b(a(c,c) + a(c,\mathbb{X}_4) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5) \tag{5.2}$$

It is an alternation of two symbols b of arity 1. The second occurrence of b contains only $\mathbb{X}_5$, so it does not need to be modified, and the first occurrence is an alternation of three occurrences of the symbol a, two of which do not contain the symbol $\mathbb{X}_4$.

Therefore, the equation system can be modified as follows:

$$\begin{cases} \mathbb{X}_4 = & a(b(a(c,c) + a(c,\mathbb{X}_4) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), c) \\ & +a(b(a(c,c) + a(c,\mathbb{X}_4) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), \mathbb{X}_4) \\ & +a(b(a(c,c) + a(c,\mathbb{X}_4) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), \mathbb{X}_5) \\ & +a(b(a(c,c) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), c) \\ & +a(b(a(c,c) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), \mathbb{X}_5) \\ \mathbb{X}_5 = & a(\mathbb{X}_4, c) + a(\mathbb{X}_4, \mathbb{X}_4) + a(\mathbb{X}_4, \mathbb{X}_5) \\ & +a(\mathbb{X}_5, c) + a(\mathbb{X}_5, \mathbb{X}_4) + a(\mathbb{X}_5, \mathbb{X}_5) \end{cases}$$

Now, after performing the 4-split with contraction, the equation system will change to the following:

$$\begin{cases} \mathbb{X}_4 = & (a(b(a(c,c) + a(c,x_4) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), c) \\ & +a(b(a(c,c) + a(c,x_4) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), x_4) \\ & +a(b(a(c,c) + a(c,x_4) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), \mathbb{X}_5))^{*x_4} \\ & \cdot_{x_4} (a(b(a(c,c) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), c) \\ & +a(b(a(c,c) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), \mathbb{X}_5)) \\ \mathbb{X}_5 = & a(\mathbb{X}_4, c) + a(\mathbb{X}_4, \mathbb{X}_4) + a(\mathbb{X}_4, \mathbb{X}_5) \\ & +a(\mathbb{X}_5, c) + a(\mathbb{X}_5, \mathbb{X}_4) + a(\mathbb{X}_5, \mathbb{X}_5) \end{cases}$$

Note. The variable $\mathbb{X}_4$ is not substituted in $\mathbb{X}_5$ here because it is repeated multiple times, to allow for brevity.

The last remaining unresolved variable is $\mathbb{X}_5$. It is an alternation of six occurrences of the symbol a of arity 2. Four of its occurrences contain $\mathbb{X}_5$ as the sole first or second element of a and cannot therefore be modified to not contain $\mathbb{X}_5$. The first two occurrences, $a(\mathbb{X}_4, c)$ and $a(\mathbb{X}_4, \mathbb{X}_4)$, contain $\mathbb{X}_5$ in $\mathbb{X}_4$.

$\mathbb{X}_4$ is a concatenation through x_4. Its right operand is

$$\begin{aligned} (a(b(a(c,c) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), c) \\ +a(b(a(c,c) + a(c,\mathbb{X}_5)) + b(\mathbb{X}_5), \mathbb{X}_5)) \end{aligned} \tag{5.3}$$

which is an alternation of two occurrences of the ranked symbol $(a, 2)$. The second occurrence contains $\mathbb{X}_5$ as the sole second element and cannot therefore be modified. The first occurrence contains c as the sole second element, and the first element is an alternation of two occurrences of $(b, 1)$. The second occurrence contains $\mathbb{X}_5$ as its operand and cannot therefore be modified. The first occurrence contains the alternation $a(c,c) + a(c,\mathbb{X}_5)$ and will therefore yield $a(c,c)$. The *subset-without* of the right operand is then $a(b(a(c,c)), c)$.

The left operand is an iteration over x_4. Its operand is an alternation of three occurrences of the ranked symbol $(a, 2)$. The third occurrence contains $\mathbb{X}_5$ as the sole second element and cannot therefore be modified. The first two occurrences contain c and x_4, respectively, as the sole second elements, so the first element can be inspected then. It is an alternation of two occurrences of $(b, 1)$, where the second occurrence contains $\mathbb{X}_5$ as its sole operand. The first occurrence is an alternation $a(c,c) + a(c,x_4) + a(c,\mathbb{X}_5)$ and will therefore yield $a(c,c) + a(c,x_4)$. The *subset-without* of the left operand is then $(a(b(a(c,c) + a(c,x_4)), c) + a(b(a(c,c) + a(c,x_4)), x_4))^{*x_4}$.

Put together, the *subset-without*$(\mathbb{X}_4, \mathbb{X}_5)$ is the following:

$$\begin{aligned} (a(b(a(c,c) + a(c,x_4)), c) + a(b(a(c,c) + a(c,x_4)), x_4))^{*x_4} \\ \cdot_{x_4} (a(b(a(c,c)), c)) \end{aligned} \tag{5.4}$$

Then the equation for $\mathbb{X}_5$ can be modified as follows:

$$\begin{cases} \mathbb{X}_5 = & a(\mathbb{X}_4, c) + a(\mathbb{X}_4, \mathbb{X}_4) + a(\mathbb{X}_4, \mathbb{X}_5) \\ & +a(\mathbb{X}_5, c) + a(\mathbb{X}_5, \mathbb{X}_4) + a(\mathbb{X}_5, \mathbb{X}_5) \\ & +a(\text{subset-without}(\mathbb{X}_4, \mathbb{X}_5), c) \\ & +a(\text{subset-without}(\mathbb{X}_4, \mathbb{X}_5), \text{subset-without}(\mathbb{X}_4, \mathbb{X}_5)) \end{cases}$$

As the last two occurrences of a do not contain $\mathbb{X}_5$, the 5-split may be performed, with the first six occurrences of a being in $\mathbb{X}_5'$ and the last two in $\mathbb{X}_5''$. Through contraction, the following is obtained:

$$\begin{cases} \mathbb{X}_5 = & (a(\mathbb{X}_4, c)_{\mathbb{X}_5 \leftarrow x_5} + a(\mathbb{X}_4, \mathbb{X}_4)_{\mathbb{X}_5 \leftarrow x_5} + a(\mathbb{X}_4, x_5)_{\mathbb{X}_5 \leftarrow x_5} \\ & +a(x_5, c) + a(x_5, \mathbb{X}_4)_{\mathbb{X}_5 \leftarrow x_5} + a(x_5, x_5))^{*x_5} \\ & \cdot_{x_5} (a(\text{subset-without}(\mathbb{X}_4, \mathbb{X}_5), c) \\ & +a(\text{subset-without}(\mathbb{X}_4, \mathbb{X}_5), \text{subset-without}(\mathbb{X}_4, \mathbb{X}_5))) \end{cases}$$

Note. As before, $\mathbb{X}_4$ is not substituted here to allow for brevity. The expression $a(\mathbb{X}_4, c)_{\mathbb{X}_5 \leftarrow x_5}$ denotes that all occurrences of $\mathbb{X}_5$ have been substituted by x_5 in $\mathbb{X}_4$.

The resultant RTE is then obtained by substituting all occurrences of $\mathbb{X}_5$ with the outcome of the 5-split and creating an alternation of right hand sides of $\mathbb{X}_3$, $\mathbb{X}_4$, and $\mathbb{X}_5$.

Chapter 6

Design

In this chapter, the final proposed design for the implementations of the conversion method described in chapters 3 and 5 is presented. The design for implementation of axioms introduced in chapter 4 is also presented.

The design in this chapter is described in high-level approach, in form of algorithms. The technical details of the implementation are described in chapter 7.

6.1 FTA to RTE conversion

The design of the algorithm for conversion from FTA to RTE follows the method described in chapter 3.

For the whole algorithm, the input is a finite tree automaton and the output is a regular tree expression. There are several subroutines which may have different inputs and outputs.

The method for FTA to RTE conversion consists of the following subproblems:

1. Construction of a RTE equation system from the given FTA.
2. Solution of the RTE equation system constructed in the previous step.
   a) Determining which variable to solve next.
   b) j-split, factorization, and contraction of the chosen variable.
   c) Replacing the variable with its associated RTE.
   d) Construction of the resultant RTE.
3. Optimization of the resultant RTE.

6.1.1 Construction of a RTE equation system

The following algorithm describes how to construct a RTE equation system for a given FTA. It follows the method described in section 3.5.

Algorithm 6.1. Construction of a RTE equation system from the given FTA.
Input: FTA $A = (\Sigma, Q, Q_f, \Delta)$, where $|Q| = n$

Output: RTE equation system $\mathcal{X} = \{\mathbb{X}_j = F_j \mid j \in \hat{n}\}$

1. For every state $q \in Q$, create an empty RTE, $\mathbb{X}_q$. This denotes an empty alternation.
2. For every transition $\delta \in \Delta$, $\delta = (f, q_1, \ldots, q_m, q)$, where m is arity of the ranked symbol f, add to the alternation of the RTE associated to the state q, $\mathbb{X}_q$, the following: $f(\mathbb{X}_{q_1}, \ldots, \mathbb{X}_{q_m})$.
3. After all transitions have been visited and added to the corresponding alternations, the RTE equation system $\mathcal{X}$ is ready to be solved.

6.1.2 Solution of the RTE equation system

The following algorithm outlines the algorithm for solution of the RTE equation system. It contains references to several algorithms which solve given subproblems, which are defined later in this chapter.

Algorithm 6.2. Solution of a RTE equation system.
Input: RTE equation system $\mathcal{X} = \{\mathbb{X}_j = F_j \mid j \in \hat{n}\}$

Output: RTE F

1. Initialize the set of unresolved variables to all variables.
2. While the set of unresolved variables is not empty, do the following:
    a) If there are any non-recursive variables, replace all of their occurrences with their associated RTEs using algorithm 6.5 and remove them from the set of unresolved variables.
    b) If there are no more unresolved variables, break from this loop.
    c) Determine which variable to solve next using algorithm 6.3.
    d) Perform the j-split, factorization, and contraction for the chosen variable using algorithm 6.4 and replace the associated RTE of the chosen variable by its outcome.
    e) Replace all occurrences of the chosen variable with its associated RTE using algorithm 6.5 and remove the chosen variable from the set of unresolved variables.

3. Construct the resultant RTE using algorithm 6.6.

The next algorithm describes how to choose the next unresolved variable to solve. While it would be correct to simply choose a random one, it is proposed to sort the associated RTEs of unresolved variables by their *depth*, to allow for shorter resultant RTEs.

Algorithm 6.3. Determining which variable to solve next.
Input: RTE equation system $\mathcal{X} = \{\mathbb{X}_j = F_j \mid j \in \hat{n}\}$

Output: Variable $\mathbb{X}_j$

1. For each variable in the set of unresolved variables, compute the *depth* of the variable $\mathbb{X}_q$'s associated RTE, $F_{\mathbb{X}_q}$. The computation of depth for RTE F is defined recursively as follows:
   a) For an alternation $F = E_1 + \ldots + E_n$, return $1 + max(depth(E_1), \ldots, depth(E_n))$.
   b) For a ranked symbol $F = f(E_1, \ldots, E_n)$ of arity n, return $1 + max(depth(E_1), \ldots, depth(E_n))$.
   c) For an empty set $F = \emptyset$, return 1.
   d) For an iteration $F = E^{*c}$, return $1 + depth(E)$.
   e) For a concatenation $F = E_1 \cdot_c E_2$, return $1 + max(depth(E_1), depth(E_2))$.
2. Sort the set of unresolved variables by depth.
3. Choose the variable with lowest depth.

The following algorithm describes how to modify a variable's associated RTE by performing a j-split with factorization and contraction on it. It follows the method described in chapter 3.

Algorithm 6.4. j-split, factorization, and contraction for a given variable.
Input: Variable $\mathbb{X}_j$ and its associated RTE F_j

Output: Modified RTE F_j^{split}

1. Take F_j as an alternation. If it is not an alternation, it can be represented as an alternation of one element.
2. For elements of F_j as an alternation, create a pair (F_j', F_j'') where F_j is an alternation of elements containing $\mathbb{X}_j$ and F_j'' is an alternation of elements not containing $\mathbb{X}_j$.

3. If F''_j is empty, assign to F''_j *subset-without*$(F'_j, \mathbb{X}_j)$ as defined in definition 5.2.

4. Create a constant x_j not yet in Σ_0 and replace all occurrences of $\mathbb{X}_j$ in F'_j and F''_j by it.

5. The modified RTE F_j^{split} is then $F'_j \cdot_{x_j} F''_j$.

Note. In the above algorithm, unlike F''_j, F'_j cannot be empty. The reason for this is step 2a in algorithm 6.2, which removes all non-recursive variables. F_j must therefore be recursive and contain at least one occurrence of $\mathbb{X}_j$.

The following algorithm describes how to replace a given variable with a given RTE. It is used with either the variable's associated RTE, or a newly created constant used as a concatenation symbol.

Algorithm 6.5. Replacing the variable with a given RTE.
Input: RTE equation system $\mathcal{X} = \{\mathbb{X}_j = F_j \mid j \in \hat{n}\}$, variable $\mathbb{X}_k$, where $k \in \hat{n}$, and RTE E_r
Output: Modified RTE equation system $\mathcal{X}' = \{\mathbb{X}_j = F'_j \mid j \in \hat{n}\}$

1. The replacement for RTE F is defined recursively.
   a) For an alternation $F = E_1 + \ldots + E_n$, if any of $E_1, \ldots, E_n$ are $\mathbb{X}_k$, replace them by E_r. Perform the replacement method on the remaining elements.
   b) For a ranked symbol $F = f(E_1, \ldots, E_n)$ of arity n, if any of $E_1, \ldots, E_n$ are $\mathbb{X}_k$, replace them by E_r. Perform the replacement method on the remaining elements.
   c) For an empty set $F = \emptyset$, do nothing.
   d) For an iteration $F = E^{*c}$, if E is $\mathbb{X}_k$, replace it by E_r. Perform the replacement method on E otherwise.
   e) For a concatenation $F = E_1 \cdot_c E_2$, if any of E_1, E_2 are $\mathbb{X}_k$, replace them by E_r. Perform the replacement method on the remaining elements.

The final result is, according to chapter 3, an union of RTEs of all RTEs associated to variables of final states.

Algorithm 6.6. Construction of the resultant RTE.
Input: RTE equation system $\mathcal{X} = \{\mathbb{X}_j = F_j \mid j \in \hat{n}\}$
Output: RTE F

1. Create a RTE representing an empty alternation.

2. For all final states $q \in Q_f$, add the RTE associated to their associated variables $\mathbb{X}_q$ to the alternation.

6.1.3 Optimization of the resultant RTE

The optimization of a RTE is performed through applying different axioms on it. In this context, optimization means making the RTE shorter and better readable.

Algorithm 6.7. Optimization of a given RTE.
Input: RTE F

Output: RTE F_{opt}

1. While any of the axioms may be applied on F, apply it.
2. Application of an axiom changes F and a different axiom which may not have been applicable before may have become applicable.
3. F_{opt} is the final RTE when no axioms are applicable anymore.

6.2 RTE axioms

The implementations of RTE axioms vary depending on the axiom, but the common property is that they all are only applicable on a given type of a RTE and they all change the RTE.

Algorithm 6.8. Implementation of a given axiom.
Input: RTE F

Output: RTE F_{opt}

1. Inspect if the axiom is applicable to F in the given direction.
2. An axiom may change the type of some of the RTE elements. For example, axiom *(N12)* changes an iteration to a constant. Depending on the implementation, this may be necessary to deal with.
3. After the axiom is applied on F, F_{opt} is the result.

Chapter 7

Implementation

7.1 Algorithms Library Toolkit

The *Algorithms Library Toolkit* (ALT), formerly known as *Automata Library*, is a set of implementations of structures such as automata, grammars, regular (tree) expressions and others, and algorithms over the aforementioned structures. It is constantly being updated and developed at the Department of Theoretical Computer Science at the Faculty of Information Technology, Czech Technical University in Prague. [10] The library is always growing and evolving, receiving updates from faculty members as well as students as part of their bachelor's or Master's thesis.

The ALT is written in pure C++. C++ is a widely known and one of the most used programming languages in the world [11]. Originally developed as an extension of the C language, being called "C with Classes", C++ has since developed into a general-purpose programming language, combining object-oriented, imperative, and generic paradigms, while providing both C-like access to low-level memory and high-level-language-like features such as containers in its standard library.

As the programmer is able to manage memory themself, C++ is regarded as a very efficient and high-performance language. This makes it ideal for implementations of performance-sensitive algorithms, which the ALT fulfills.

The C++ language is still being actively developed as of May 2019. The latest standard of C++ is C++17 and the next standard planned is C++20. The ALT utilizes new features from C++17.

The ALT is built by `CMake`. `CMake` is an "open-source, cross-platform family of tools designed to build, test and package software," used to control compilation of the project independently on the platform and compiler. [12] However, as of the writing of this thesis, the ALT is only available on UNIX platforms.

The ALT consists of two binaries, a command line interface executable and a GUI environment. This is a recent development, formerly the ALT used

to consist of several specialized executable files, for example `agenerate2` for generation of random structures, and `aconversions2` for conversions between different structures such as RTEs and FTAs. The ALT also consists of several libraries, such as `alib2algo`, which contains the algorithms themselves, and `alib2data`, which contains classes representing the data structures.

Extensions of the standard C++ library can also be found in the `alib2std` library, in the `ext` namespace. Such extensions include pointer vectors, nullary, unary, binary, and varary nodes, and others.

Upon launching the CLI executable, the user may input a series of commands. An input for a command may either be read from the command line, from a variable or from a file. The most common file type in the `examples2` folder, which contains categorized examples of the data structures implemented, is XML. Each structure has its own XML composer and XML parser implemented.

Code 7.1. The following is a XML representation of a deterministic FTA.

```
<DFTA>
  <states>
    <Set><Unsigned>1</Unsigned></Set>
  </states>
  <finalStates>
    <Set><Unsigned>1</Unsigned></Set>
  </finalStates>
  <rankedInputAlphabet>
    <RankedSymbol><Character>97</Character><Unsigned>1</
       ↪ Unsigned></RankedSymbol>
    <RankedSymbol><Character>98</Character><Unsigned>0<
       ↪ Unsigned></RankedSymbol>
  </rankedInputAlphabet>
  <transitions>
    <transition>
      <input><RankedSymbol><Character>97</Character><
         ↪ Unsigned>1</Unsigned></RankedSymbol></input>
      <from><Set><Unsigned>1</Unsigned></Set></from>
      <to><Set><Unsigned>1</Unsigned></Set></to>
    </transition>
    <transition>
      <input><RankedSymbol><Character>98</Character><
         ↪ Unsigned>0</Unsigned></RankedSymbol></input>
      <from/>
      <to><Set><Unsigned>1</Unsigned></Set></to>
    </transition>
  </transitions>
</DFTA>
```

It is a FTA with the ranked alphabet of $\{(a,1),(b,0)\}$, one state 1, which is also a final state, and two transitions depicted in figure 7.1.

Note. The lowercase letter `a` is represented by number 97 and lowercase `b` is represented by number 98 in ASCII.

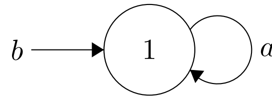

Figure 7.1: The finite tree automaton represented by example 7.1

Code 7.2. The following sets of commands:

```
execute automaton::generate::RandomTreeAutomatonFactory
    ↪ (size_t)5 (size_t)3 (size_t)3 (bool)true (double)
    ↪ "0.5" > $fta
execute $fta | automaton::convert::ToRegTreeExp -
quit
```

will generate a random tree automaton with the given parameters and store it into the variable `fta`. Then, it will pass it as a parameter through the pipeline to the algorithm for conversion to a RTE. With the `quit` command, the CLI executable will quit.

After the name of the algorithm (with the corresponding namespaces) is provided, the exact method to be executed is chosen from a list of overloads of registered methods. The best candidate is chosen based on the number and types of parameters.

Code 7.3. The following is a snippet of code showing registration of algorithms performing conversions from a finite automaton to a regular expression. This code is in the namespace `automaton::convert`.

```
// non-deterministic finite automaton
auto ToRegExpAlgebraicNFA = registration::
    ↪ AbstractRegister < ToRegExpAlgebraic, regexp::
    ↪ UnboundedRegExp < >, const automaton::NFA < > & >
    ↪ ( ToRegExpAlgebraic::convert, "automaton" ).
    ↪ setDocumentation ("...")
// deterministic finite automaton
auto ToRegExpAlgebraicDFA = registration::
    ↪ AbstractRegister < ToRegExpAlgebraic, regexp::
    ↪ UnboundedRegExp < >, const automaton::DFA < > & >
    ↪ ( ToRegExpAlgebraic::convert, "automaton" ).
    ↪ setDocumentation ("...")
```

`UnboundedRegExp` is the returned type, `NFA` or `DFA` is the argument which was passed from the CLI.

The integration tests for the ALT are run during the installation, or explicitly by calling `make test`.

7.2 Preexisting data structures used

The ALT already contained many data structures which were possible to reuse in the algorithms described in previous chapters.

DFTA

One such structure already in the ALT is DFTA - a deterministic finite tree automaton. A snippet highlighting some of the most important properties and methods is shown in code 7.4.

Code 7.4. Snippet of `DFTA` class.

```
template < class SymbolTypeT = DefaultSymbolType , class
   ↪ RankTypeT = DefaultRankType , class StateTypeT =
   ↪ DefaultStateType >
class DFTA final : public ext::CompareOperators < DFTA <
   ↪  SymbolTypeT , RankTypeT , StateTypeT > >, public
   ↪ core::Components < DFTA < SymbolTypeT , RankTypeT ,
   ↪ StateTypeT >, ext::set < common::ranked_symbol <
   ↪ SymbolTypeT , RankTypeT > >, component::Set ,
   ↪ InputAlphabet , ext::set < StateTypeT >, component
   ↪ ::Set , std::tuple < States , FinalStates > > {

ext::map < ext::pair < common::ranked_symbol <
   ↪ SymbolType , RankType >, ext::vector < StateType >
   ↪ >, StateType > transitions;

const ext::set<StateType>& getStates () const & {
  return this->template accessComponent<States> ().get()
     ↪ ;
}

bool addState ( StateType state ) {
  return this->template accessComponent<States> ().add (
     ↪  std::move ( state ) );
}

const ext::map <ext::pair <common::ranked_symbol <
   ↪ SymbolType , RankType>, ext::vector <StateType> >,
   ↪ StateType> & getTransitions () const & {
  return transitions;
}
```

```
  bool addTransition ( common::ranked_symbol < SymbolType ,
     ↪ RankType > current , ext::vector < StateType >
     ↪ children , StateType next );
  }
```

As can be seen, the class is implemented through templates, and access to properties common for all automata, such as states, is implemented through the templates. Transitions differ for each type of automaton and therefore are implemented in this class as a map of pairs of a symbol and a vector of states. For example, a transition (f, q_1, q_2, q) is then represented as $(f, (q_1, q_2, q))$.

FormalRTE and related

The ALT already contained classes for `FormalRTE` and related classes. Types of RTE are implemented as classes, inheriting from `FormalRTEElement`. `FormalRTE` then contains a `FormalRTEStructure` member, which points to an instance of a class extending `FormalRTEElement`.

The implemented algorithm works with unbounded RTEs. However, the final result is, because of compatibility, transformed into a formal RTE. Code snippets showing unbounded RTEs and related are shown in the following section so they are omitted here as they are similar.

7.3 New data structures implemented

UnboundedRTE and related

Unbounded RTE is represented by the `UnboundedRTE` class. It uses templates and inheritance from common classes, which is used to access the alphabet, for example, and contains an instance of a `UnboundedRTEStructure` class, as is shown in the following code snippet. The structure of classes describing unbounded RTEs was derived from classes describing formal RTEs.

Code 7.5. `UnboundedRTE` class.

```
  template < class SymbolType = DefaultSymbolType , class
     ↪ RankType = DefaultRankType >
class UnboundedRTE final : public ext::CompareOperators <
   ↪ UnboundedRTE < SymbolType , RankType > >, public core
   ↪ ::Components < UnboundedRTE < SymbolType , RankType
   ↪ >, ext::set < common::ranked_symbol < SymbolType ,
   ↪ RankType > >, component::Set , std::tuple <
   ↪ GeneralAlphabet , ConstantAlphabet > > {
    UnboundedRTEStructure < SymbolType , RankType > m_rte;
    //constructors , methods ,...
    }
```

The `UnboundedRTEStructure` class describes the structure of the RTE itself.

Code 7.6. `UnboundedRTEStructure` class.

```
template < class SymbolType , class RankType >
class UnboundedRTEStructure final {
ext::smart_ptr < UnboundedRTEElement < SymbolType ,
    ↪ RankType > > m_structure;
const UnboundedRTEElement < SymbolType , RankType > &
    ↪ getStructure ( ) const;
void setStructure ( const UnboundedRTEElement <
    ↪ SymbolType , RankType > & structure );
// other constructors , methods...
}
```

The different types of RTEs are described by the following classes, which all inherit from `UnboundedRTEElement`.

- `UnboundedRTEAlternation`
- `UnboundedRTEIteration`
- `UnboundedRTESubstitution` - this represents concatenation. The name substitution was chosen to remain consistent with the formal implementation.
- `UnboundedRTESymbolAlphabet`
- `UnboundedRTESymbolSubst` for constants used in iterations and concatenations
- `UnboundedRTEEmpty` for the empty set $\emptyset$

Structure of the `UnboundedRTEElement` class is shown below.

Code 7.7. `UnboundedRTEElement` class.

```
template < class SymbolType , class RankType >
class UnboundedRTEElement : public ext::CompareOperators
    ↪  < UnboundedRTEElement < SymbolType , RankType > >,
    ↪  public ext::BaseNode < UnboundedRTEElement <
    ↪ SymbolType , RankType > > {
virtual ext::smart_ptr <UnboundedRTEElement < SymbolType
    ↪ , RankType > > cloneWithoutSymbol ( const common::
    ↪ ranked_symbol < SymbolType , RankType > & symbol )
    ↪ const = 0;
virtual void replaceAllOccurrences ( const
    ↪ UnboundedRTESymbolAlphabet < SymbolType , RankType
    ↪ > & replaced , const UnboundedRTEElement <
    ↪ SymbolType , RankType > & by ) = 0;
```

```
virtual bool testSymbol ( const common::ranked_symbol <
    ↪ SymbolType , RankType > & symbol ) const = 0;
virtual unsigned int getSymbolDepth ( ) const = 0;
virtual ext::smart_ptr <FormalRTEElement < SymbolType ,
    ↪ RankType > > asFormal ( ) const = 0;
// other methods , constructors , ...
}
```

The code snippet above shows some of the common methods for all RTE types. For example, `cloneWithoutSymbol()` returns a clone of the given element without any references to a given symbol, following the function *subset-without* defined in definition 5.2. The `replaceAllOccurrences()` method implements algorithm 6.5, `getSymbolDepth()` implements the *depth* method described in algorithm 6.3, `testSymbol()` returns a Boolean value depending on whether the RTE contains the given symbol.

All types of RTE inherit from `BaseNode`, and depending on the number of children elements the RTE type can have, a concrete implementation of it – `NullaryNode` for zero, as is case for empty set and substitution symbols, `UnaryNode` for one, as is case for iteration, `BinaryNode` for two, as is case for concatenation, and `VararyNode` for any integer, as is case for alternation and ranked symbols.

All RTE elements also contain a method `asFormal()` which returns the element's equivalent as a formal RTE. Its implementation is trivial in all cases but alternation, which is constrained by two elements for formal RTEs, and thus for unbounded alternations containing more than two elements, needs to be transformed through nesting. For example, $a+b+c+d+e$ is transformed into $(a+(b+c))+(d+e)$.

RegularTreeEquationSolver

A new class for a solver of RTE equations was added. The following snippet shows its structure.

Code 7.8. Structure of the `RegularTreeEquationSolver` class.

```
template < class SymbolType , class RankType , class
    ↪ VariableType >
class RegularTreeEquationSolver {
public:
  rte::UnboundedRTE < SymbolType , RankType > solve();
  void addVariableNonFinalSymbol ( const VariableType &
      ↪ state );
  void addVariableFinalSymbol ( const VariableType &
      ↪ state );
  void addTransition ( const VariableType & to , const
      ↪ common::ranked_symbol < SymbolType , RankType > &
```

```
      ↪  rankedSymbol, const ext::vector < VariableType
      ↪ > & fromStates );
  private:
ext::map < rte::UnboundedRTESymbolAlphabet < SymbolType,
    ↪  RankType >, rte::UnboundedRTEAlternation <
    ↪ SymbolType, RankType > > equations;
ext::map < VariableType, rte::UnboundedRTESymbolAlphabet
    ↪  < SymbolType, RankType > > stateToVariable;
ext::map < VariableType, rte::UnboundedRTESymbolSubst <
    ↪ SymbolType, RankType > > stateToSubst;
ext::vector < ext::pair < VariableType, unsigned int > >
    ↪  unresolvedStates;
  void jSplit ( const VariableType & state );
  bool isRecursive ( const VariableType & state );
// and others...
  }
```

The `solve()` method implements algorithm 6.2. It utilizes, among others, methods `jSplit()` and `isRecursive()`. Each state has its associated variable and substitution symbol. Both are represented as constants. The equation is a map from a variable to its associated RTE, which is represented as an alternation. At the end of the `solve()` method, the RTE is optimized and all alternations of one element are transformed into the one element.

7.4 Implementation of the conversion method

In the namespace `automaton::convert`, a class `ToRegTreeExp` was created. This class has one method, `convert()`, which creates an instance of `RegularTreeEquationSolver`, adds the corresponding states and transitions on it, and gets a result through the `solve()` method. The obtained unbounded RTE is then transformed into formal RTE using the `asFormal()` method.

The `solve()` method of the equation solver follows algorithm 6.2. The equations are represented as a map from a symbol, representing the variable, to an alternation. Each state has its associated variable and substitution symbols. After the j-split for a chosen variable, its associated RTE is optimized before replacing its variable in all other equations. In the end, a new `UnboundedRTEStructure` is created and its RTE is set to a newly created alternation of RTEs of variables of the final states. This alternation is then optimized.

7.5 Implementation of axioms

The axioms are implemented in the `alib2algo` library, in the class `rte::simplify::RTEOptimize`. While the work was being performed, a commit to the master branch of the ALT created the class and implementation of

axioms for formal RTEs. This change was then merged with the work being performed, so in the end the result is that only the unbounded part was added.

The optimization through axioms may be performed either on an `UnboundedRTE`, `UnboundedRTEStructure`, or `UnboundedRTEElement`. If performed on a RTE or its structure, it returns a clone of the respective structure, with the inner RTE optimized. If performed on a RTE element, it changes its structure. In the implementation of the conversion method, optimization of `UnboundedRTE` is used.

The structure of the optimization is very simple and is shown in the code snippet below.

Code 7.9. Optimization through axioms.

```
  template < class SymbolType , class RankType >
ext::smart_ptr < rte::UnboundedRTEElement < SymbolType ,
   ↪ RankType > > RTEOptimize::optimizeInner ( const rte
   ↪ ::UnboundedRTEElement < SymbolType , RankType > &
   ↪ node ) {
    UnboundedRTEElement < SymbolType , RankType > * element
        ↪ = node.clone();
    while ( X0 (element) || A1 (element) || A2 (element)
        ↪ || A3 (element) || A4 (element) || A5 (element)
        ↪ || A8 (element) || A9 (element) || B2 (element)
        ↪ || B3 (element) || B4 (element) || B7 (element)
        ↪ || N1 (element) || N2 (element) || N3 (element)
        ↪ || N4 (element) || N5 (element) || N6 (element)
        ↪ || N7 (element) || N8 (element) || N9 (element)
        ↪ || N10 (element) || N12(element) || N11 (element
        ↪ ) || S (element) );

    return ext::smart_ptr < UnboundedRTEElement <
        ↪ SymbolType , RankType > > ( element );
}
```

Each axiom has its own associated method, where it accepts a pointer to the element as an argument and either modifies the element it points to, or creates a new element and changes the pointer. Axioms *(A1)* through *(A9)*, *(B2)* through *(B7)*, and *(N1)* through *(N11)* are named accordingly. Note that even though the axioms are implemented for unbounded RTEs, they are not named with the *U* modifier at the end. Method *(X0)* implements an optimization where an alternation of one element is transformed into the element, and method *(S)* implements optimization of inner elements of the RTE - that is, if the RTE itself cannot be optimized anymore, its children are optimized recursively. The order in which the optimizations are performed is important – some axioms rely on the children of an element being sorted, for example $x = x + x$. A signature of an axiom method is shown below.

Code 7.10. Signature of an axiom method.

```
template < class SymbolType , class RankType >
static bool axiom_name( rte::FormalRTEElement <
    ↪ SymbolType , RankType > * & node );
}
```

Some of the axioms were very easy to implement, some were more difficult and also complex. For example, the axiom *(N2U)* needed to compare all occurrences of a ranked symbol to each other.

Originally, axiom *(B6)* was implemented. As was later found out, however, that it needed to be modified, it showed that with such modifications its implementation would be very demanding and requires more time to study thoroughly. Therefore, axioms *(B6-1)* and *(B6-2)* were not implemented.

Chapter **8**

Testing

The algorithms library toolkit already contains many integration tests in the `alib2integrationtest` library. Therefore, the tests for implementation of this work were made accordingly to be consistent.

The tests utilize the following pre-existing algorithms:

- `automaton::generate::RandomTreeAutomatonFactory`
- `automaton::determinize::Determinize`
- `Trim`, `Minimize`, `Normalize` from `automaton::simplify`
- `rte::convert::ToFTAGlushkov`
- `compare::AutomatonCompare`

Additionally, the implemented method, `automaton::convert::ToRegTreeExp`, is of course used. The simplifications using axioms are a part of this algorithm.

Unfortunately, there is no random RTE generator in the ALT. However, with both random tests and tests from files, it was determined that the coverage was sufficient.

All tests may be run using the `make test` command.

8.1 FTA files tests

The first part are integration tests for files describing FTAs in the `examples2` folder. There are multiple deterministic and non-deterministic FTAs which describe automata with various properties. The tests follow the pattern described in the algorithm below.

Algorithm 8.1. FTA files tests.

1. From all files `(N|D)FTA*.xml`, read the automaton and determinize it.

2. Convert the automaton to RTE using the newly implemented method and convert it back to an automaton using `ToFTAGlushkov`.
3. Trim, minimize, and normalize the automaton from file and the automaton obtained through conversions.
4. The automata should be equivalent.

8.2 RTE files tests

The second part are integration tests for files describing RTEs in the `examples2` folder. The tests follow the pattern described in the algorithm below.

Algorithm 8.2. RTE files tests.

1. From all files `rte[0-9].xml`, read the RTE, convert it to an automaton using `ToFTAGlushkov` and determinize it.
2. Convert the automaton to RTE using the newly implemented method and convert it back to an automaton using `ToFTAGlushkov`.
3. Trim, minimize, and normalize the automaton from `ToFTAGlushkov` and the automaton obtained through conversions.
4. The automata should be equivalent.

8.3 Random FTA tests

The last part are integration tests for randomly generated FTAs. The tests follow the pattern described in the algorithm below.

Algorithm 8.3. Random FTA tests.

1. Generate a random tree automaton and determinize, trim, minimize, and normalize it.
2. Convert the automaton to RTE using the newly implemented method and convert it back to an automaton using `ToFTAGlushkov`.
3. Trim, minimize, and normalize the automaton obtained through conversions.
4. The automata should be equivalent.

Conclusion

The goal of this work consisted of two parts.

The first part was to study regular tree expressions and a method of conversion from finite tree automata to regular tree expressions, and then implement and test this method. This goal was met and the method was implemented in the Algorithms Library Toolkit.

The second part was to study axioms for regular tree expressions, discuss them, propose new ones, and implement them. This goal was also met – fifteen (plus two unbounded) new axioms were proposed, three axioms were extended for unbounded regular tree expressions, and one axiom was found not to always hold and two new axioms were proposed in its stead. The implementation was also completed in the Algorithms Library Toolkit.

For future work, one thing that stands out is to try and propose an axiom similar to *(B6)*, which will hold. Determining whether axiom *(N7)* can be modified for unbounded RTEs is also worth considering.

Other ideas for future work include extending unboundedness to the concatenation operator and research in other methods for conversion from FTA to RTE.

For the ALT, the newly proposed axioms may be implemented for formal RTEs. Other methods for conversion may also be implemented after their proposals. Extending the current method to work on non-deterministic FTAs is also a possibility. For now, such FTA needs to be determinized first.

Bibliography

[1] Belabbaci, A.; Cherroun, H.; et al. Tree pattern matching from regular tree expressions. *Kybernetika*, volume 54, 04 2018: pp. 221–242, doi: 10.14736/kyb-2018-2-0221.

[2] Polách, R. *Tree Pattern Matching and Tree Expressions*. Master's thesis, Czech Technical University in Prague, 2011.

[3] Pecka, T.; Trávníček, J.; et al. Construction of a Pushdown Automaton Accepting a Postfix Notation of a Tree Language Given by a Regular Tree Expression. In *7th Symposium on Languages, Applications and Technologies (SLATE 2018)*, *OpenAccess Series in Informatics (OASIcs)*, volume 62, edited by P. R. Henriques; J. P. Leal; A. M. Leitão; X. G. Guinovart, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, ISBN 978-3-95977-072-9, ISSN 2190-6807, pp. 6:1–6:12, doi:10.4230/OASIcs.SLATE.2018.6. Available from: `http://drops.dagstuhl.de/opus/volltexte/2018/9264`

[4] Guellouma, Y.; Cherroun, H. From Tree Automata to Rational Tree Expressions. *International Journal of Foundations of Computer Science*, volume 29, no. 06, 2018: pp. 1045–1062, doi:10.1142/S012905411850020X.

[5] Schimpf, K.; Gallier, J. Tree Pushdown Automata. *Journal of Computer and System Sciences*, volume 30, no. 1, 1985: pp. 25–40, doi:10.1016/0022-0000(85)90002-9.

[6] Koller, A.; Regneri, M.; et al. Regular tree grammars as a formalism for scope underspecification. 2008, doi:10.1.1.164.5484.

[7] Aiken, A.; Murphy, B. R. Implementing regular tree expressions. In *Functional Programming Languages and Computer Architecture*, edited by J. Hughes, Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 427–447.

[8] Encyclopedia.com. Ardens Rule. Available from: https://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/ardens-rule

[9] Holub, J. BIE-AAG Lecture 4. Regular expressions. 2018. Available from: https://courses.fit.cvut.cz/BIE-AAG/lectures/bie-aag-04-regularni_vyrazy.pdf

[10] FIT-GitLab. Algorithms library toolkit. 2019. Available from: https://gitlab.fit.cvut.cz/algorithms-library-toolkit/automata-library

[11] GitHub. Octoverse 2018. 2018. Available from: https://octoverse.github.com/projects

[12] CMake. CMake. Available from: https://cmake.org

Appendix A

Acronyms

ALT Algorithms Library Toolkit

CLI Command line interface

FTA Finite tree automaton

GUI Graphical user interface

HTML Hypertext markup language

RTE Regular tree expression

XML Extensible markup language

Appendix B

Automata library toolkit manual

B.1 Requirements

As of the date of this thesis, the ALT can be only compiled and run on Unix operating systems. Running a Unix operating system virtually should be sufficient with enough RAM, which all modern computers should have enough of.

The project is written in pure C++ and is built by `cmake`. The requirements are:

- `cmake` $\geq$ `3.7`
- `make` $\geq$ `3.9`
- `g++` $\geq$ `6.3` or `clang` $\geq$ `5.0`
- `tclap`
- `libxml2`
- `readline`

For GUI, the following are also required:

- `Qt5-qtbase` $\geq$ `5.7`
- `jsoncpp`
- `graphviz`

B.2 Installation

Extract the `.zip` file and in the folder with the project, run `all-cmake-release.sh`.

B.3 Running the toolkit

The binary for CLI is `aql2`. The binary for GUI is `agui2`.

B.4 Examples

The examples below are provided for the CLI.

```
execute < sandbox/rte.xml
```

Displays the RTE represented by the XML file.

```
execute < sandbox/rte.xml > $rte
```

Saves the RTE represented by the XML file to a variable `rte`.

```
execute < $rte
```

Displays the RTE saved in the variable `rte`.

```
execute automaton::generate::RandomTreeAutomatonFactory
(size_t)5 (size_t)3 (size_t)3 (bool)true (double)0.5 > $nfta
```

Generates a random non-deterministic tree automaton with the given parameters.

```
execute $nfta | automaton::determinize::Determinize - |
automaton::simplify::Trim - | automaton::simplify::Minimize
- | automaton::simplify::Normalize - > $dfta
```

determinizes, trims, minimizes, and normalizes the automaton saved in `$nfta` and saves it to a variable `dfta`.

```
execute $dfta | automaton::convert::ToRegTreeExp - > $genrte
```

Converts the automaton from `dfta` and converts it to RTE using the method studied in this thesis. The result is saved in variable `genrte`.

```
execute < sandbox/rte.xml | rte::simplify::RTEOptimize -
```

Displays the RTE represented by the XML file after optimizations.

```
quit
```

Quits the CLI.

Example data structures, such as automata, RTEs, regular expressions, and others, can be found in the `examples2` folder.

Appendix C

Contents of enclosed SD card

```
readme.txt ................... the file with SD card contents description
alt.zip ........................... archive with source codes for ALT
text ......................................... the thesis text directory
  DP_Doupal_Jakub_2019.pdf ........... the thesis text in PDF format
  src .................................... source codes of this thesis
```