Finite state automaton construction through regular expression hashing

R.J.L. Coetser
Supervisor: Prof. Dr. B.W. Watson
Co-supervisor: Prof. Dr. D.G. Kourie

August 9, 2009

Acknowledgements

I would like to thank my parents and sister for their support while writing this dissertation. I would also like to thank my supervisors for their valuable input, and Loek Cleophas, who read my dissertation and article.

Abstract

In this study, the regular expressions forming abstract states in Brzozowski's algorithm are not remapped to sequential state transition table addresses as would be the case in the classical approach, but are hashed to integers. Two regular expressions that are hashed to the same hash code are assigned the same integer address in the state transition table, reducing the number of states in the automaton.

This reduction does not necessarily lead to the construction of a minimal automaton: no restrictions are placed on the hash function hashing two regular expressions to the same code. Depending on the quality of the hash function, a *super-automaton*, previously referred to as an *approximate automaton*, or an exact automaton can be constructed. When two regular expressions are hashed to the same state, and they do not represent the same regular language, a super-automaton is constructed.

A super-automaton accepts the regular language of the input regular expression, in addition to some extra strings. If the hash function is bad, many regular expressions that do not represent the same regular language will be hashed together, resulting in a smaller automaton that accepts extra strings. In the ideal case, two regular expressions will only be hashed together when they represent the same regular language. In this case, an exact minimal automaton will be constructed. It is shown that, using the hashing approach, an exact or super-automaton is always constructed.

Another outcome of the hashing approach is that a non-deterministic automaton may be constructed. A new version of the hashing version of Brzozowski's algorithm is put forward which constructs a deterministic automaton.

A method is also put forward for measuring the difference between an exact and a super-automaton: this takes the form of the k-equivalence measure: the k-equivalence measure measures the number of characters up to which the strings of two regular expressions are equal. The better the hash function, the higher the value of k, up to the point where the hash function results in regular expressions being hashed together if and only if they have the same regular language.

Using the k-equivalence measure, eight generated hash functions and one hand coded hash function are evaluated for a large number of short regular expressions, which are generated using Gödel numbers. The k-equivalence concept is extended to the *average k-equivalence* value in order to evaluate the hash functions for longer regular expressions. The hand coded hash function is found to produce good results.

Contents

1 Introduction **6**
1.1 Problem statement 6
1.2 Related work 7
1.3 Dissertation structure 8
1.4 Presentation and publication 9

2 Basic definitions **10**

3 Strings and regular languages **12**
3.1 Strings 12
3.2 Finite state automata 13
3.3 Regular expressions 14
3.4 Super-automata and exact automata 16
3.5 Properties of regular expressions 16
3.6 Summary 18

4 Brzozowski's algorithm **19**
4.1 Basic definitions for Brzozowski's algorithm 19
4.2 General algorithm overview 20
4.3 Rewrite rules for algorithm termination 21
4.4 Remapping version of the algorithm 21
4.5 Example of algorithm execution 23
4.6 Example of non-termination 26
4.7 Summary 27

5 Brzozowski's algorithm with state merging **28**
5.1 General algorithm overview 28
5.2 Example of the algorithm execution 30
5.3 Algorithm 5.1 constructs a super-automaton 33
5.4 Summary 35

6 Brzozowski's algorithm with state merging — the DFA version 36
6.1 The source of non-determinism . . . 36
6.2 General algorithm overview . . . 38
6.3 Example of the algorithm execution . . . 40
6.4 Summary . . . 43

7 Equivalence classes of regular languages 45
7.1 k-equivalence classes . . . 45
7.2 Exact *versus* super-automaton minimization . . . 47
7.3 Categories of hash functions . . . 48
7.4 Ideal hash functions . . . 48
7.5 There is no ideal hash function *modulo* n . . . 49
7.6 Summary . . . 50

8 Implementation 51
8.1 General implementation outline . . . 51
8.2 Gödel numbering and regular expression generation . . . 53
8.3 Short *versus* long regular expressions . . . 54
8.4 Choosing hash functions operators . . . 55
8.5 Applying operator mappings . . . 56
8.6 An alternative hash function . . . 57
8.7 Summary . . . 59

9 Empirical results 62
9.1 Measured results for short regular expressions . . . 62
9.2 Measured results for long regular expressions . . . 64
9.3 Summary . . . 67

10 Conclusion and future directions 68
10.1 Results . . . 68
10.2 Future directions . . . 69

A Code overview 73
A.1 High level design overview . . . 73
A.2 Rewriting regular expressions . . . 74
A.3 Equality and hashing . . . 75
A.4 Rewrite rules for the empty set and the empty string . . . 76
A.5 Brzozowski's algorithm with remapping . . . 77
A.6 Brzozowski's algorithm with hashing . . . 78
A.7 Brzozowski's algorithm with hashing, the DFA version . . . 79

B Long regular expression test cases **81**

List of Figures

4.1 Brzozowski's algorithm without state merging 20
4.2 Partially constructed automaton for $(a \cdot b \cup b \cdot c)^*$ 25
4.3 Complete automaton for $(a \cdot b \cup b \cdot c)^*$ 26

5.1 Partial automaton constructed using Algorithm 5.1 32
5.2 Complete automaton constructed using Algorithm 5.1 33
5.3 Before state merging 33
5.4 After state merging 34

6.1 Non-determinism resulting from Algorithm 5.1 39
6.2 Determinised results from Algorithm 6.1 40
6.3 Partially constructed automaton constructed using Algorithm 6.1 . . 41
6.4 Determinised automaton for example run of Algorithm 6.1 43

7.1 k-equivalence and finite state automata 46
7.2 k-equivalence and regular expressions 47

8.1 The procedure for testing hash functions and *-equivalence 52

A.1 Class diagram of the regular expression classes 74

List of Tables

4.1 Remapped values for regular expressions in Algorithm 4.1 24
4.2 Nullable values for regular expressions in Algorithm 4.1 25
4.3 The state transition table constructed for $(a \cdot b \cup b \cdot c)^*$ 26

5.1 Hash codes for example run of Algorithm 5.1 31
5.2 State transition table for example run of Algorithm 5.1 32

6.1 Partially constructed transition table for Algorithm 6.1 41
6.2 Hash codes for example run of Algorithm 6.1 43

8.1 Bit string operator properties 55
8.2 Regular expression operator properties 56
8.3 Hash function operator mappings 60
8.4 Hash function operator mappings, continued 61

9.1 Statistics for short regular expressions 64
9.2 Average k-equivalence for long regular expressions 66

B.1 Prefix notation operators 81

Chapter 1

Introduction

This chapter provides a broad overview of the research presented in this dissertation. First the problem statement is given, and then informally discussed. This is followed by a related work section which places the work in context. Finally, the structure of the dissertation is given.

1.1 Problem statement

The aim of this dissertation is to reduce the memory requirements of finite state automata, also referred to as *exact automata*, by turning them into smaller non-deterministic finite state automata, possibly accepting additional strings. This automaton accepting additional strings is called a *super-automaton*, previously referred to as an *approximate automaton* in [23], and is constructed from a regular expression denoting the language of an exact automaton. Based on this basic problem statement, the following research questions can be formulated:

- How can the quality of a super-automaton be measured, relative to the exact automaton?
- By what theoretical framework can the process of constructing a super-automaton be interpreted?

These questions are investigated based on a modified version of Brzozowski's algorithm for constructing a finite state automaton from a regular expression, given in [23][1].

One of the main problems addressed in this dissertation is how to measure the relative quality of an exact and a super-automaton. The contribution of the research

[1]Brzozowski's original algorithm can be found in [1].

is mainly based around this, since no such measure could be found in literature, though related work provided a basis for this.

The problem of measuring the relative quality of a super-automaton *versus* an exact automaton is addressed in Chapter 7, starting from k-equivalence classes.

Having identified a performance measure, the process of reducing the automaton size is addressed. The reduction process proceeds by assigning integers to abstract states in a finite state automaton, which is represented by regular expressions. This process takes the form of a hash function: when two states are assigned the same integer they become merged, and in the process the automaton is reduced. Therefore, the quality of the reduced automaton is solely dependent on the hash function.

1.2 Related work

The work presented here is based on an article that appeared recently in [23]. In this article, the process of finite state automaton minimization through regular expression hashing is put forward. It is pointed out that a super-automaton would be constructed through the hashing process. Even though no literature has been found that is directly related to this work, there are other processes that result in super-automata. Work related to the research presented in this dissertation includes:

- An approach is given in [14] for creating a super-automaton from an automaton that accepts the same language, with k errors or mismatches, based on the Levenshtein or Hamming distance. It duplicates the given automaton, and then updates transitions between the copies of the automaton to take the mismatches into account. This work differs from the approach in this dissertation in the sense that the resulting automaton is bigger than the original. This work is similar to the work presented in this dissertation in the sense that the resulting automaton accepts more strings than the original automaton.

- In [2], a process is given to construct a smaller *cover automaton* from an exact automaton. A cover automaton is an automaton that accepts at least the language of the exact automaton, in addition to strings longer than the longest string. This work is similar in the sense that the resulting automaton is smaller. It differs in the sense that the additional strings in the language are longer. The process in this dissertation results in an automaton that may accept additional shorter or longer strings.

- A factor oracle is a super-automaton, constructed from an input string, that accepts at least the factors of that string. The factors of a string are all

substrings of the string. Factor oracles have been introduced in [5]. The properties of factor oracles have been explored in [3].

- This dissertation focuses on the construction of a finite state automaton, using Brzozowski's algorithm. The reader is referred to [22] for a taxonomy of finite state automaton construction and minimization algorithms.
- Brzozowski's algorithm is based on the derivatives of regular expressions that were presented in [1]. The derivative of a regular expression, with respect to an alphabet symbol, represents all the strings in the regular language represented by that regular expression, with the first symbols removed. Brzozowski's algorithm is presented in Chapter 4.
- A concurrent version of Brzozowski's algorithm can be found in [19] which uses the CSP (Concurrent Sequential Processes) notation.
- An axiomatization of the regular languages can be found in [17].

1.3 Dissertation structure

The dissertation consists of five parts:

- ***Definitions and notation***: Chapters 2 and 3 present the basic definitions of this dissertation, building definitions from set theory up to the properties of strings and regular expressions.
- ***Theory***: Chapters 4 to 7 present the underlying theories of this dissertation. This starts with Brzozowski's algorithm. In Chapter 7 a theory is developed for measuring the quality of a super-automaton, relative to an exact automaton.
- ***Implementation and Results***: In Chapter 8 and 9, nine hash functions are used to illustrate how the concepts in this dissertation are brought together.
- ***Code Overview of the implementation***: This dissertation comes with a compact disk containing the implementation. A short overview of the implementation is given in Appendix A.
- ***Test cases***: Appendix B gives the long regular expressions used to generate statistics.

The convention followed is to present definitions and notation as they are needed, with only the basic definitions being placed in the second and third chapters.

1.4 Presentation and publication

All new work in the form of definitions and theorems not taken from literature is marked with an asterisk "*" in order to simplify the referencing process.

Some aspects of this work were also presented at the Prague Stringology Conference, 2008, and can be found in the proceedings [4]. Research conducted after the conference will be published in a special issue of the International Journal of the Foundations of Computer Science.

Chapter 2

Basic definitions

This chapter contains some basic definitions and notation used in this dissertation. The assumption is made that the reader is familiar with first order logic and set theory. Most of the definitions and notations can be found in [16] and [6].

The notation of a power set is used in Chapter 3 to distinguish between a deterministic and a non-deterministic finite state automaton.

Notation 2.1 (Power set)
The power set of a set A is written as $\mathcal{P}(A)$. □

The notation for a partial function is used in the definition of an automaton:

Notation 2.2 (Partial and Total Functions)
A partial function uses the $\nrightarrow$ symbol instead of the $\rightarrow$ symbol, which is used for total functions. $f : X \nrightarrow Y$ denotes a partial function from X to Y. Functions can also be written as $f(x) = y$, with $x \in X$ and $y \in Y$. □

Functions are also explicitly written as a set of mappings from X to Y with the $\mapsto$ symbol. For example, if $X = \{a, b\}$ and $Y = \{1, 2\}$ then the function f, for which $f(a) = 1$ and $f(b) = 2$, can be written as $f = \{a \mapsto 1, b \mapsto 2\}$.

Definition 2.3 (Onto function)
A function f is *onto* if and only if $\forall y : Y, \exists x : X \bullet f(x) = y$. This is also referred to as a surjection. □

Definition 2.4 (One-to-one function)
A function f is *one-to-one* if and only if $\forall x_1, x_2 : X \bullet [f(x_1) = f(x_2)] \Rightarrow [x_1 = x_2]$ with $f(x_1), f(x_2) \in Y$. This is also referred to as an *injection*. □

Definition 2.5 (Bijection)
A bijection is *one-to-one* and *onto*, and is also referred to as a *one-to-one* correspondence. □

The following notation is used in connection with equivalence relations. Equivalence relations are important to this dissertation because they form the bases of the k-equivalence classes that are used in later chapters in order to measure the difference between an exact and a super-automaton.

Definition 2.6 (Equivalence classes of a relation)
If $R \subseteq (Q \times Q)$ is a relation such that

- R is *symmetric*: $\forall x, y : Q \bullet x \; R \; y \Rightarrow y \; R \; x$;
- R is *transitive*: $\forall x, y, z : Q \bullet x \; R \; y \wedge y \; R \; z \Rightarrow x \; R \; z$; and
- R is *reflexive*: $\forall x : Q \bullet x \; R \; x$.

then R is an *equivalence relation* on Q. An equivalence relation on Q partitions Q into mutually disjoint subsets. The set of equivalence classes for the relation R on the set Q is written as $[Q]_R$. □

Notation 2.7 (The number of equivalence classes in an equivalence relation)
The number of classes in an equivalence relation R on a set Q is written $|[Q]_R|$. □

The following definitions are used in connection with hash functions and axiomatic semantics. They are used in a later chapter to construct "good" hash functions which are used in the implementation and results chapters.

Definition 2.8 (Commutativity)
A function is commutative if $\forall x, y \bullet f(x, y) = f(y, x)$. □

Definition 2.9 (Associativity)
A function is associative if $\forall x, y, z \bullet f(f(x, y), z) = f(x, f(y, z))$. □

Definition 2.10 (Idempotence)
A function is idempotent if $\forall x \bullet f(x, x) = x$. □

Chapter 3

Strings and regular languages

This chapter contain definitions and notation for strings, finite state automata and regular expressions. Properties of regular expressions relevant to Brzozowski's algorithm are also given.

3.1 Strings

Notation 3.1 (Alphabet)
The alphabet of a string is a non-empty finite set denoted by Σ. □

Notation 3.2 (String)
A string s is written as a sequence of characters $c_0 \dots c_{n-1}$. The empty string is written ε. Therefore $c_i \in \Sigma$. □

Notation 3.3 (String length)
The length of string s is written as $|s|$. The length of ε is 0. □

Notation 3.4 (All strings over an alphabet)
The set of all strings from an alphabet Σ is written as Σ^*. Note that $\varepsilon \in \Sigma^*$ and that $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$. □

Notation 3.5 (Language)
d can be an automaton, a regular expression or an automaton state, defined in the following sections. $L(d)$ denotes a language whose description is d. The language $L(d)$ is always a subset of all possible strings over the alphabet of d, i.e. $L(d) \subseteq \Sigma^*$. □

3.2 Finite state automata

The definition for a deterministic finite state automaton is based on the one in [10]:

Definition 3.6 (Deterministic finite state automaton)
A deterministic finite state automaton (abbreviated DFA) is a 5-tuple $\langle Q, \Sigma, \delta, s, F \rangle$ where

- Q is the set of states;
- Σ is the alphabet;
- $\delta : Q \times \Sigma \nrightarrow Q$ is the state transition function;
- $s \in Q$ is the start state; and
- $F \subseteq Q$ is the set of final states.

□

Definition 3.7 (Non-deterministic finite state automaton)
A nondeterministic finite state automaton (abbreviated NFA) is a 5-tuple $\langle Q, \Sigma, \delta, s, F \rangle$ where

- Q is the set of states;
- Σ is the alphabet;
- $\delta : Q \times \Sigma \nrightarrow \mathcal{P}(Q)$ is the state transition function;
- $s \in Q$ is the start state; and
- $F \subseteq Q$ is the set of final states.

□

Note that the state transition function in Definition 3.6 is changed in Definition 3.7: it now goes to a power set of Q. Based on this change, a combination of a state and a transition symbol can have multiple next states. In the modified version of Brzozowski's algorithm, which is presented in a later chapter, a non-deterministic finite state automaton is constructed: its non-determinism does not follow from empty transition symbols, but from multiple next states. Therefore, the empty symbol has been omitted from Definition 3.7: it is not applicable to this dissertation.

In order to define the language of an automaton, an extended transition function is used as was done in [10]:

Definition 3.8 (Extended transition function)
The extended transition function is written as $\delta^*(q, w)$ where q is the current state, and w is an input string. $\delta^*(q, w)$ is defined recursively in [10]: if $w = ab$ with $a \in \Sigma$ and $b \in \Sigma^*$ then

- $\delta^*(q, \varepsilon) = q$
- $\delta^*(q, ab) = \delta^*(\delta(q, a), b)$

□

When $\delta^*(q, w)$ is undefined for input string w, w is not in the language of the automaton. Formally:

Definition 3.9 (Language of an automaton)
The set of strings representing the language of an automaton $FA = \langle Q, \Sigma, \delta, s, F \rangle$ is denoted $L(FA)$, and is given by $L(FA) = \{w | \delta^*(s, w) \in F \wedge w \in \Sigma^*\}$ □

In later chapters, the concepts of the left and the right languages of a state in an automaton are used. Informally, the left language of a state q is the set of all strings accepted by the automaton, where the last state reached in traversing the automaton is q. The set of strings accepted in this manner is equivalent to the language of an automaton, where q is the only final state. Therefore the left language is defined as:

Definition 3.10 (Left language of a state)
The left language of q is the language of the automaton $\langle Q, \Sigma, \delta, s, \{q\} \rangle$. The left language of a state q is written $\overleftarrow{L}(q)$. □

Similarly, the right language of state q is the set of all strings accepted by the finite automaton, starting from state q. For this reason, the right language is defined as:

Definition 3.11 (Right language of a state)
The right language of q is the language of the automaton $\langle Q, \Sigma, \delta, q, F \rangle$. The right language of a state q is written $\overrightarrow{L}(q)$. □

3.3 Regular expressions

Regular expressions and finite state automata are related to each other in the sense that they both represent regular languages. The definition of a regular language, given below, is taken from [13] [10]:

Definition 3.12 (Regular language)
A language is *regular* if and only if it is accepted by a finite state automaton. □

Definition 3.13 (Regular expression)
A regular expression over an alphabet Σ is defined recursively as follows, with L giving the semantics of the regular expression representation:

- ***Empty String:*** ε is a regular expression, and $L(\varepsilon) = \{\varepsilon\}$
- ***Empty Language:*** The empty language, denoted by $\emptyset$, is a regular expression, and $L(\emptyset) = \emptyset$. Note that $L(\emptyset) \neq L(\varepsilon)$.
- ***Alphabet Symbols:*** An alphabet symbol $s \in \Sigma$ is a regular expression, and $L(s) = \{s\}$. Note that such a symbol also represents a string of length 1.
- ***Concatenation:*** If A and B are regular expressions then their concatenation, written as $A \cdot B$, is also a regular expression, and $L(A \cdot B) = \{vw | v \in L(A) \wedge w \in L(B)\}$.
- ***Union:*** The union of two regular *languages* A and B is a regular expression and is written as $A \cup B$. $L(A \cup B) = L(A) \cup L(B)$.
- ***Star Closure:*** The star closure of a regular expression A, written as A^*, is a regular expression, and $L(A^*) = L(\varepsilon) \cup L(A) \cup L(A \cdot A) \cup L(A \cdot A \cdot A) \cup \ldots$.

□

In some contexts, it is convenient to rely on the notion of extend regular expressions. The following extensions of regular expressions are encountered:

- ***Optional:*** If A is a regular expression then $A^?$ is also a regular expression, and $L(A^?) = L(A) \cup \{\varepsilon\}$.
- ***Plus Closure:*** The plus closure of a regular expression, A, written as A^+ is the regular expression $A \cdot A^*$. Thus $L(A^+) = L(A \cdot A^*)$.

In this dissertation, it is necessary to refer to the set of all regular languages and the set of all regular expressions. Therefore, the following notation needs to be defined:

Notation 3.14 (The set of regular languages)
The set of all regular languages is written R^l. □

Notation 3.15 (The set of regular expressions)
The set of all regular expressions is written R^e. □

3.4 Super-automata and exact automata

It is now possible to define what is meant by a super-automaton. Super-automata have previously informally been described in [23], where they were called approximate automata. The following definitions are given formally due to their importance to this dissertation. $r \in R^l$ denotes an *intended* regular language, which can be described in any regular language notation. These definitions are not taken from literature but were defined for the purpose of this dissertation.

Definition 3.16 (Exact automaton*)
If r is a regular language and FA is an automaton with $L(FA) = r$, then FA is an exact automaton of r. □

Definition 3.17 (Super-automaton*)
If r is a regular language and FA is an automaton with $L(FA) \supseteq r$, then FA is a super-automaton of r. □

Definition 3.18 (Sub-automaton*)
If r is a regular language and FA is an automaton with $L(FA) \subseteq r$, then FA is a sub-automaton of r. □

Note that a super-automaton is also a finite state automaton. Therefore it still represents a regular language, and not (for example) a context-free language [10].

3.5 Properties of regular expressions

The properties of regular languages in this section relate to Brzozowski's algorithm in the next chapter. The nullability of regular languages, left derivatives and first symbol sets are all involved in the algorithm and are presented in Definitions 3.19, 3.20 and 3.21.

Definition 3.19 (Nullable Regular Languages)
For a regular expression RE, $nullable(RE) = (\varepsilon \in L(RE))$. $nullable$ is defined [7] in terms of regular expressions as:

- $nullable(\emptyset) = false$
- $nullable(\varepsilon) = true$
- $nullable(s) = false$, if $s \in \Sigma$
- $nullable(A \cdot B) = nullable(A) \wedge nullable(B)$

- $nullable(A \cup B) = nullable(A) \vee nullable(B)$
- $nullable(A^*) = true$
- $nullable(A^+) = nullable(A)$
- $nullable(A^?) = true$

□

Definition 3.20 (First Symbol Sets of Regular Expressions)
The first symbol set $first(RE)$ of a regular expression RE is the set of first symbols of all the strings in $L(RE)$, and is given in [7] as:

- $first(\emptyset) = \emptyset$
- $first(\varepsilon) = \emptyset$
- $first(s) = \{s\}$
- $first(A \cdot B) = first(A) \cup \begin{cases} first(B) & \text{if } nullable(A) \\ \emptyset & \text{otherwise} \end{cases}$
- $first(A \cup B) = first(A) \cup first(B)$
- $first(A^*) = first(A)$
- $first(A^+) = first(A)$
- $first(A^?) = first(A)$

□

Definition 3.21 (Left Derivatives of Regular Expressions)
The left derivative of a regular expression RE with respect to a symbol $s \in first(RE)$ represents all the strings in $L(RE)$, starting with the first symbol s, but with the first symbol removed. The left derivative is recursively defined by these rules [1]:

- $s^{-1}\emptyset = \emptyset$
- $s^{-1}\varepsilon = \emptyset$
- $s^{-1}s = \varepsilon$
- $s^{-1}(A \cdot B) = (s^{-1}A) \cdot B \cup \begin{cases} (s^{-1}B) & \text{if } nullable(A) \\ \emptyset & \text{otherwise} \end{cases}$

- $s^{-1}(A \cup B) = (s^{-1}A) \cup (s^{-1}B)$
- $s^{-1}A^* = (s^{-1}A) \cdot A^*$
- $s^{-1}A^+ = (s^{-1}A) \cdot A^*$
- $s^{-1}A^? = s^{-1}A$

□

3.6 Summary

In this chapter, definitions and notation were given for strings, regular expressions and finite state automata. This concludes the basic definitions needed in the rest of the dissertation. In the next chapter, the left derivatives, first symbol sets and nullability tests are used in the presentation of Brzozowski's algorithm.

Chapter 4

Brzozowski's algorithm

Brzozowski's algorithm for constructing a deterministic finite state automaton from a regular expression forms the basis of this dissertation. In this chapter, Brzozowski's algorithm is presented.

4.1 Basic definitions for Brzozowski's algorithm

Notation 4.1 (Initial regular expression and Brzozowski's algorithm)
The initial regular expression RE_{init} is the expression from which an automaton is derived using Brzozowski's algorithm. □

Notation 4.2 (The state of a regular expression)
The abstract state s represented by a regular expression RE in Brzozowski's algorithm is written as $state(RE) \in Q$. □

The definition of the left and right language of a regular expression follows the same line of reasoning as the definition for the left and right language of a state:

Notation 4.3 (Left language of regular expressions)
If $Brz(RE_{init}) = \langle Q, \Sigma, \delta, state(RE_{init}), F \rangle$ and RE_s is a regular expression such that $state(RE_s) \in Q$ then the left language of RE_s is written as $\overleftarrow{L}(RE_s)$ and is equal to $\overleftarrow{L}(state(RE_s))$. □

Notation 4.4 (Right language of regular expressions)
If $Brz(RE_{init}) = \langle Q, \Sigma, \delta, state(RE_{init}), F \rangle$ and RE_s is a regular expression such that $state(RE_s) \in Q$ then the right language of RE_s is written as $\overrightarrow{L}(RE_s)$ and is equal to $\overrightarrow{L}(state(RE_s))$. □

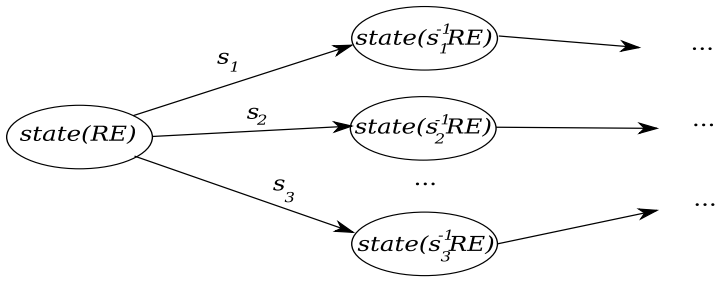

Figure 4.1: Brzozowski's algorithm without state merging

Note that the left language of a regular expression exists relative to RE_{init}: in order to have a left language, the regular expression RE must be in a finite state automaton. This language gives a set of strings from the start state that forms the left language of $state(RE)$.

The right language of the regular expression RE is equal to the language of the automaton $Brz(RE)$, which in turn has the same language as the input regular expression. For this reason, $L(RE)$ and $\overrightarrow{L}(RE)$ can be used interchangeably.

4.2 General algorithm overview

Broadly speaking Brzozowski's algorithm proceeds as follows. Given a regular expression RE_{init}, the start state of the deterministic finite automaton to be built is $state(RE_{init})$. The first symbol set $first(RE_{init})$ is then found. For each symbol s in $first(RE_{init})$, the left derivative $s^{-1}RE_{init}$ is calculated. s then becomes a label for an out-transition for $state(RE_{init})$, and $state(s^{-1}RE_{init})$ forms the next state for this out-transition. These steps are repeated for each derived regular expression in a recursive fashion.

In each case, each regular expression RE_j is subjected to the nullability test $nullable(RE_j)$. If $nullable(RE_j) = true$, then $\varepsilon \in L(RE_j)$ and $state(RE_j)$ is a final state. The input regular expression forms the start state in the resulting automaton. This process is depicted in Figure 4.1, with RE representing the initial regular expression.

After expanding $s^{-1}RE$ for the first symbol set $first(RE)$, the algorithm is repeated for $s^{-1}RE$. Note that cycles may form in the resulting automaton. This occurs when the left derivative corresponds to a regular expression that has previously been associated with a state. Such a regular expression is not further expanded, in

order to avoid non-termination.

4.3 Rewrite rules for algorithm termination

In order for Brzozowski's algorithm to terminate, equality must take commutativity, associativity and idempotence of the union operator into account. The following rewrite rules may also be applied to eliminate the empty set and the empty string:

- $RE \cup \emptyset = RE$
- $RE \cdot \emptyset = \emptyset$
- $\emptyset \cdot RE = \emptyset$
- $RE \cdot \varepsilon = RE$
- $\varepsilon \cdot RE = RE$

4.4 Remapping version of the algorithm

Algorithm 4.1 comes from [23] with slight modifications. It uses *remapping* (the *remap* function call) to determine a unique integer to represent the state of each regular expression. In the implementation, the initial regular expression RE_{init} is represented by 0, and all further assignments are incremented by one. These integers can be used as addresses in a state transition table, as will be shown in Section 4.5.

Note that the destination state for a given symbol is uniquely determined, due to the *remap* function. As a consequence, a *deterministic* finite state automaton is constructed. Note, also, that the test in the algorithm for membership of regular expression sets (e.g. $destination \notin (done \cup todo)$) takes commutativity, idempotence and the rewrite rules for the empty string and the empty set into account.

In the remap function, the test for equality must *also* take commutativity, idempotence and the rewrite rules into account, as is done in the set membership test. This keeps the remapped regular expressions, representing states, consistent with the *done* and the *todo* sets. This also influences the formation of cycles in the constructed automaton: when a regular expression is encountered more than once, it must be remapped to the same integer in order to form a cycle in the automaton.

Algorithm 4.1 (Brzozowski's algorithm with Remapping)

func $Brz(RE_{init})$
  $next, \delta, F, remap := 0, \emptyset, \emptyset, \emptyset;$
  $remap[RE_{init}], next := next, next + 1;$
  $done, todo := \emptyset, \{RE_{init}\};$
  do $todo \neq \emptyset \rightarrow$
    let RE_j be some regular expression such that $RE_j \in todo$;
    $done, todo := done \cup \{RE_j\}, todo \backslash \{RE_j\};$
    { Only expand out-transitions for symbols in the first symbol set ofRE_j }
    for $s : first(RE_j) \rightarrow$
      { Use the left derivatives to calculate the next state }
      $destination := s^{-1}RE_j;$
      if $destination \notin (done \cup todo) \rightarrow$
        { Update the todo set in order to expand the automaton for $destination$ }
        $todo := todo \cup \{destination\};$
        $remap[destination], next := next, next + 1$
      [] $destination \in (done \cup todo) \rightarrow$ **skip**
      fi;
      $\delta(remap[RE_j], s) := remap[destination]$
    rof;
    if $nullable(RE_j) \rightarrow$
      { A state is final if and only if it accepts the empty string }
      $F := F \cup \{remap[RE_j]\}$
    [] $\neg nullable(RE_j) \rightarrow$ **skip**
    fi
  od;
  return $\langle \{0, \ldots, next - 1\}, \delta, 0, F \rangle$
cnuf

4.5 Example of algorithm execution

In order to clarify the way the algorithm works, and to expose several finer points in its implementation, an example of the construction of an automaton from a regular expression is now presented, using $RE_{init} = (a \cdot b \cup b \cdot c)^*$.

In Algorithm 4.1, the *remap* variable associates a regular expression with an integer, as discussed in the above sections. The *next* variable stores the next value that will be mapped to a regular expression. The *done* set stores the regular expressions that have been processed by the algorithm's *for*- and *do*-loops. The *todo* set stores the regular expressions that must still be processed.

In order to initialize these variables, *remap* is set to 0, and *todo* is set to $\{(a \cdot b \cup b \cdot c)^*\}$. The other sets are set to $\emptyset$.

In each iteration of the algorithm's *do*-loop, a regular expression is removed from the *todo* set, and moved to the *done* set. This regular expression will from this point forward be referred to as the *current regular expression*. In this first iteration of the algorithm, the current regular expression is $(a \cdot b \cup b \cdot c)^*$. In order to expand this into a finite state automaton, the first symbol set is calculated. The calculation of the first symbol set, based on Definition 3.20, is:

$$\begin{aligned} first((a \cdot b \cup b \cdot c)^*) &= first(a \cdot b \cup b \cdot c) \\ &= first(a \cdot b) \cup first(b \cdot c) \\ &= first(a) \cup first(b) \\ &= \{a, b\} \end{aligned}$$

The *for*-loop iterates over all the symbols in the first symbol set, taking their left derivative. These left derivatives represent the next states in the finite state automaton. The guard of the *if*-statement in the *for*-loop of the algorithm prevents non-termination: when a regular expression have already been expanded in the algorithm, it must not be expanded again. This is prevented by the $destination \notin (done \cup todo)$ check. If this check was not present, the $todo \neq \emptyset$ test of the *do*-loop would never return false.

The set membership test must take idempotence, commutativity and associativity into account: in an object oriented programming language such as *C#*, this modifies the definition of equality. This alters the design of an implementation of the algorithm. A method used to handle equality in the implementation is discussed in Appendix A.

The *do*-loop iterates over the first symbol set, calculating the left derivatives. First the left derivative with respect to a is calculated:

$$\begin{aligned} a^{-1}((a \cdot b \cup b \cdot c)^*) &= a^{-1}((a \cdot b \cup b \cdot c) \cdot (a \cdot b \cup b \cdot c)^*) \\ &= (b \cup \emptyset) \cdot (a \cdot b \cup b \cdot c)^* \\ &= b \cdot (a \cdot b \cup b \cdot c)^* \end{aligned}$$

Note that the rewrite rule $b \cup \emptyset = b$ has been applied. Now the left derivative with respect to b is calculated:

$$\begin{aligned} b^{-1}((a \cdot b \cup b \cdot c)^*) &= b^{-1}((a \cdot b \cup b \cdot c) \cdot (a \cdot b \cup b \cdot c)^*) \\ &= (\emptyset \cup c) \cdot (a \cdot b \cup b \cdot c)^* \\ &= c \cdot (a \cdot b \cup b \cdot c)^* \end{aligned}$$

This time the rewrite rule has been applied as $\emptyset \cup c = c$. When implementing these rules in a programming language, the commutativity, associativity and idempotence of the regular expression operators must be taken into account.

The left derivatives are remapped to integers, as discussed in previous sections. The remapping of all the regular expressions generated for the resulting automaton is given in the Table 4.1. When remapping the regular expressions, the modified definition of equality must be taken into account. This is important for updating the set of final states: the set of final states contains integers. In order to maintain consistency with the other sets (i.e. *done* and *todo*), the remap function must also take idempotence, commutativity and associativity into account.

Regular Expression	Remap Value
$(a \cdot b \cup b \cdot c)^*$	0
$b \cdot (a \cdot b \cup b \cdot c)^*$	1
$c \cdot (a \cdot b \cup b \cdot c)^*$	2

Table 4.1: Remapped values for regular expressions in Algorithm 4.1

After calculating $a^{-1}((a \cdot b \cup b \cdot c)^*)$ and $b^{-1}((a \cdot b \cup b \cdot c)^*)$, and calculating the remap values, the state transition function is updated. Then the last guard in the *do*-loop marks states as final states. This is done by using Definition 3.19:

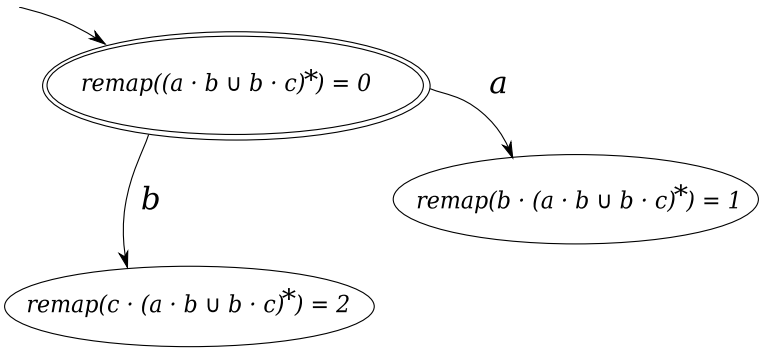

Figure 4.2: Partially constructed automaton for $(a \cdot b \cup b \cdot c)^*$

$$nullable((a \cdot b \cup b \cdot c)^*) = true$$

The calculation does not traverse the entire regular expression due to the star closure rule in Definition 3.19. The nullable values for all regular expressions in the automaton is given in Table 4.2.

Regular Expression	Nullable
$(a \cdot b \cup b \cdot c)^*$	$true$
$b \cdot (a \cdot b \cup b \cdot c)^*$	$false$
$c \cdot (a \cdot b \cup b \cdot c)^*$	$false$

Table 4.2: Nullable values for regular expressions in Algorithm 4.1

This concludes the first iteration of the *do*-loop. The automaton after the first iteration is depicted in Figure 4.2. The start state is also the final state and has been marked with a double circle.

The second iteration of the *do*-loop is the final iteration. In this iteration, $done = \{(a \cdot b \cup b \cdot c)^*\}$ and $todo = \{b \cdot (a \cdot b \cup b \cdot c)^*, c \cdot (a \cdot b \cup b \cdot c)^*\}$. When the *current* regular expression is $c \cdot (a \cdot b \cup b \cdot c)^*$, the left derivative is

$$c^{-1}(c \cdot (a \cdot b \cup b \cdot c)^*) = (a \cdot b \cup b \cdot c)^*$$

When the result of this derivative is remapped, the remapping algorithm must take into account that this regular expression has been processed before. For this reason,

$(a \cdot b \cup b \cdot c)^*$ is mapped to 0, as it was before. This causes a cycle to form in the finite state automaton, representing the *-closure operator. It is also the case that

$$b^{-1}(b \cdot (a \cdot b \cup b \cdot c)^*) = (a \cdot b \cup b \cdot c)^*$$

Therefore, another cycle forms from $remap(b \cdot (a \cdot b \cup b \cdot c)^*) = 1$ to $remap((a \cdot b \cup b \cdot c)^*) = 0$. The final state transition table is shown in Table 4.3. The final automaton is shown in Figure 4.3.

Remapped Value (state)	a	b	c
0	1	2	
1		0	
2			0

Table 4.3: The state transition table constructed for $(a \cdot b \cup b \cdot c)^*$

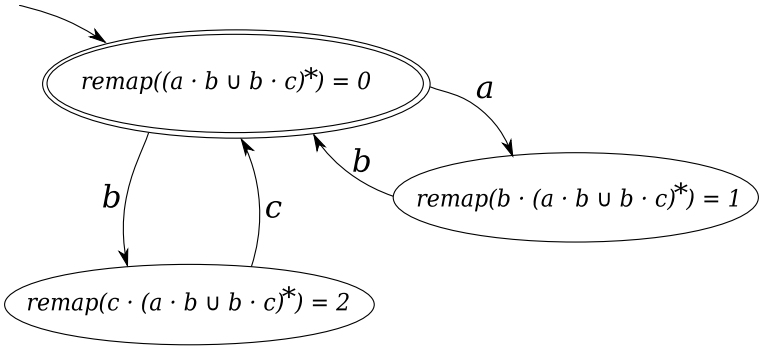

Figure 4.3: Complete automaton for $(a \cdot b \cup b \cdot c)^*$

4.6 Example of non-termination

In order to see how the *non*-application of the rewrite rules in Section 4.3 can lead to non-termination, consider the algorithm run for the regular expression a^*. Initially, a^* would be placed in the *todo* set, and then processed when it is selected in the *let*-statement.

When the left derivative is taken in the first iteration, we have

$$\begin{aligned} a^{-1}(a^*) &= a^{-1}a \cdot a^* \\ &= \varepsilon \cdot a^* \end{aligned}$$

Consider what happens if the rewrite rules are *not* applied as they should be. As a consequence, a^* and $\varepsilon \cdot a^*$ will not be remapped to the same integer, even though they represent the same language. On the second iteration, we would have

$$\begin{aligned} a^{-1}(\varepsilon \cdot a^*) &= (a^{-1}(\varepsilon) \cdot a^*) \cup (a^{-1}(a^*)) \\ &= \emptyset \cdot a^* \cup a^{-1}a \cdot a^* \\ &= \emptyset \cdot a^* \cup \varepsilon \cdot a^* \end{aligned}$$

As can be seen on the right hand side of the derivation, $\varepsilon \cdot a^*$ re-appears. It was also the input regular expression. After an arbitrary number of iterations, the regular expression would be expanded to

$$\emptyset \cdot a^* \cup \ldots \cup \emptyset \cdot a^* \cup \varepsilon \cdot a^*$$

Applying the rewrite rule $\emptyset \cdot RE = \emptyset$ at this stage would result in:

$$\emptyset \cup \emptyset \cup \ldots \cup \emptyset \cup \varepsilon \cdot a^*$$

If the $RE \cup \emptyset = RE$ rule, and then the $\varepsilon \cdot RE = RE$ rule were now to be applied, the resulting regular expression would be a^*. Placing this regular expression in the *done* set would prevent further expansion.

4.7 Summary

In this chapter, Brzozowski's algorithm is presented. An example run of the algorithm and an example of non-termination resulting from the non-application of the rewrite rules with regards to the empty set and the empty string is given. In the next chapters, modified versions of the algorithm are given which construct super-automata.

Chapter 5

Brzozowski's algorithm with state merging

In this chapter, modifications to the algorithm in the previous chapter are discussed that allows the construction of a super-automaton. These modifications were originally published in [23]. The effects of using the hashing approach is investigated. The hashing approach results in the construction of a super-automaton that is non-deterministic. In the next chapter, a modified version of the algorithm in this chapter will be given, which constructs a deterministic automaton.

5.1 General algorithm overview

In the hashing version of Brzozowski's algorithm (Algorithm 5.1), the remapping function is replaced with a hash function. The result might be that the resulting automaton is smaller. This happens when two regular expressions with the same hash code represent the same state, and states are merged. This algorithm was first presented in [23].

Algorithm 5.1 (Brzozowski's algorithm with Hashing — NFA version)

func $Brz\_hash\_NFA(RE_{init})$
  $Q, \delta, F := \emptyset, \emptyset, \emptyset$;
  $done, todo := \emptyset, \{RE_{init}\}$;
  do $todo \neq \emptyset \rightarrow$
    let RE_j be some regular expression such that $RE_j \in todo$;
    $done, todo, h := done \cup RE_j, todo \backslash RE_j, hash(RE_j)$;
    $Q := Q \cup \{h\}$;
    { Only expand out-transitions for symbols in the first symbol set ofRE_j }
    for $s : first(RE_j) \rightarrow$
      { Use the left derivatives to calculate the next state }
      $destination := s^{-1}RE_j$;
      if $destination \notin (done \cup todo) \rightarrow$
        { Update the todo set in order to expand the automaton for $destination$ }
        $todo := todo \cup \{destination\}$
      [] $destination \in (done \cup todo) \rightarrow$ **skip**
      fi;
      $\delta(h, s) := \delta(h, s) \cup \{hash(destination)\}$
    rof;
    if $nullable(RE_j) \rightarrow$
      { A state is final if and only if it accepts the empty string }
      $F := F \cup \{h\}$
    [] $\neg nullable(RE_j) \rightarrow$ **skip**
    fi
  od;
  return $\langle Q, \delta, hash(RE_{init}), F \rangle$
cnuf

When comparing Algorithm 4.1 and 5.1, the following general remarks can be made:

- In Algorithm 5.1, the state transition function δ goes to a set of integers, rather than to a single integer. Therefore the resulting automaton is non-deterministic[1].
- When updating the state transition function, the *hash* function is used instead of the *remap* function. This change also affects the start state and final states. The *done* and *todo* sets still contain regular expressions, where equality takes rewrite rules, idempotence, associativity and commutativity into account. The set of final states F contains integers from the hashed regular expressions.
- Because *destination* is added to the *todo* set, its out-transitions will be expanded in a later iteration. Note that an out-transition can originate not only from *destination* but also from the state with which it is merged.
- As a consequence of the possible merging of states, the automaton constructed in Algorithm 5.1 is a super-automaton of the automaton constructed in Algorithm 4.1.

These observations will be further explained below.

In Algorithm 4.1, 5.1 and 6.1 the alphabet of the automaton is implied by the alphabet of the input regular expression RE_{init}. Therefore it has been omitted in order to simplify the notation. The following notation is introduced for indicating that two states have been merged. It will be used in later sections.

Notation 5.1 (Merged states)
$merge(s_1, s_2)$ denotes the state that is constructed when merging states s_1 and s_2. The state $merge(s_1, s_2)$ replaces the states s_1 and s_2. The in- and out-transitions of state s_1 and s_2 become the in- and out-transitions of $merge(s_1, s_2)$. □

5.2 Example of the algorithm execution

In order to clarify the workings of Algorithm 5.1, it is now shown how an automaton is derived from a regular expression. The same regular expression from Section 4.5 is used: $(a \cdot b \cup b \cdot c)^*$. This simplifies the comparisons with Algorithm 4.1.

The main difference between algorithm 4.1 and 5.1 is that the *remap* function has been replaced by the *hash* function. Most of the data structures remain the same.

[1] A new deterministic version (Algorithm 6.1) is presented in the next chapter.

The *done* and the *todo* sets still store regular expressions, with the *done* set storing regular expressions that have already been processed, and the *todo* set still storing regular expressions that must still be processed.

The same considerations must still be taken into account with respect to the rewrite rules for the empty set and the empty string. Equality tests in an implementation must still take idempotence, associativity and commutativity of regular language operators into account.

When processing the regular expression $(a \cdot b \cup b \cdot c)^*$, it is loaded into the *todo* set. It is subsequently selected in the *let*-statement, and the first symbol set is calculated as is done in Section 4.5. The left derivatives of $(a \cdot b \cup b \cdot c)^*$ are also calculated, with respect to the first symbol set, as is done in Section 4.5, and subsequently added to the *todo* set.

Regular expression	Hash code
$(a \cdot b \cup b \cdot c)^*$	5
$b \cdot (a \cdot b \cup b \cdot c)^*$	5
$c \cdot (a \cdot b \cup b \cdot c)^*$	7

Table 5.1: Hash codes for example run of Algorithm 5.1

Next, the state transition function is updated. This is where the real difference between Algorithm 4.1 and 5.1 lies: the resulting automaton can be non-deterministic, as will be shown in a later section. For this reason the state transition function update takes the form

$$\delta(h, s) := \delta(h, s) \cup \{hash(destination)\}$$

There is no restriction placed on the hash function, accept that it must map a regular expression to an integer. For illustrative purposes, let the hash function generate the mappings given in Table 5.1. This hash function is chosen to show how certain hash functions can generate non-deterministic automata. Algorithm 6.1 is designed to construct a deterministic automaton, and will be discussed in the next chapter.

During the first iteration of the *do*-loop, the *current regular expression* is set to $(a \cdot b \cup b \cdot c)^*$. When the state transition function is updated with the *hash* function, the difference between the automata produced by Algorithm 4.1 and 5.1 becomes apparent: compare Figure 4.2 and Figure 5.1. Initially, $(a \cdot b \cup b \cdot c)^*$ and $b \cdot (a \cdot b \cup b \cdot c)^*$ remapped to different values. Here they are hashed to the same integer, 5. This causes $state((a \cdot b \cup b \cdot c)^*)$ and $state(b \cdot (a \cdot b \cup b \cdot c)^*)$ to become merged.

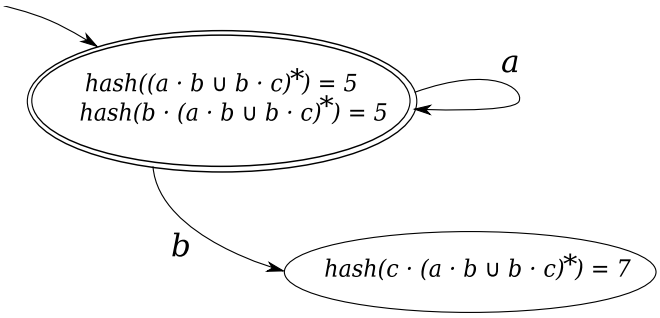

Figure 5.1: Partial automaton constructed using Algorithm 5.1

In the second iteration of the *do*-loop:

$$todo = \{b \cdot (a \cdot b \ \cup \ b \cdot c)^*, c \cdot (a \cdot b \ \cup \ b \cdot c)^*\}$$

Algorithm 4.1 and 5.1 proceed in the same way at this point, expanding the automaton for each of the elements of the *todo* set. When the left derivative of $b \cdot (a \cdot b \cup b \cdot c)^*$ is calculated with respect to b, the next state is $(a \cdot b \ \cup \ b \cdot c)^*$. The state for which the hash code is 5 already has an out-transition on the b symbol: the state transition function is updated in the following manner:

$$\begin{aligned} \delta(5, b) &:= \delta(5, b) \cup \{hash((a \cdot b \ \cup \ b \cdot c)^*) = 5\} \\ &= \{7, 5\} \end{aligned}$$

For this reason the automaton becomes non-deterministic. The complete automaton is shown in Figure 5.2.

The final state transition table for the automaton in Figure 5.2 is given in Table 5.2.

Hash code (State)	a	b	c
5	$\{5\}$	$\{5, 7\}$	$\emptyset$
7	$\emptyset$	$\emptyset$	$\{5\}$

Table 5.2: State transition table for example run of Algorithm 5.1

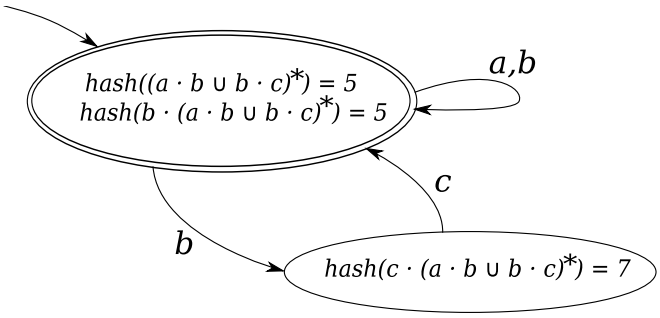

Figure 5.2: Complete automaton constructed using Algorithm 5.1

5.3 Algorithm 5.1 constructs a super-automaton

In Chapter 3, the notion of a super-automaton FA^s, constructed from regular expression RE is defined as a finite automaton having the following property:

$$L(FA^s) \supseteq L(RE)$$

It must be proven that the automaton constructed in the hash version of Brzozowski's algorithm satisfies this property. This will be done by first providing an informal illustration and then a proof.

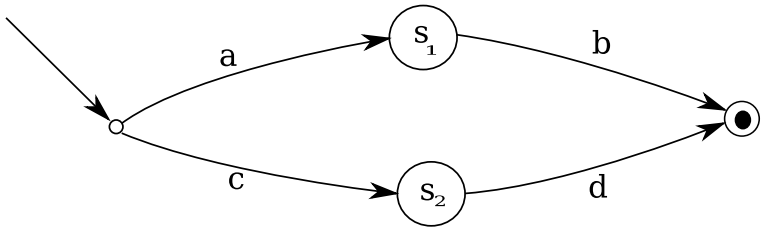

Figure 5.3: Before state merging

Consider the finite automaton depicted in Figure 5.3. Suppose that this automaton is generated by Algorithm 4.1 and that RE_1 and RE_2 are regular expressions such that $s_1 = state(RE_1)$ and $s_2 = state(RE_2)$. The language of the automaton is $L(FA) = \{ab, cd\}$. Now suppose that Algorithm 5.1 is used to generate an automaton, based on the same regular expression that gave rise to Figure 5.3. Suppose too, that there was a collision between the hash values of RE_1 and RE_2. Then states s_1 and s_2 would be merged by Algorithm 5.1, producing an automaton depicted in Figure 5.4.

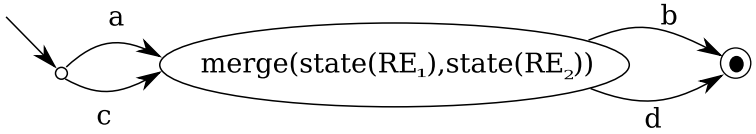

Figure 5.4: After state merging

The language of the new automaton with $merge(s_1, s_2)$ is:

$$L(FA^s) = \{ab, ad, cb, cd\} \supseteq L(FA)$$

Note that, a, the in-transition of s_1, has been connected to the out-transition of s_2, namely d. Similarly, the in-transition of s_2, namely c, has been connected to the out-transition of s_1, namely b. As a consequence the language of FA^s is a superset of the language of FA — new strings are added to the original language and none of the old strings in the language are removed. We now present a new proof that a super-automaton is constructed.

Theorem 5.1 The automaton FA^s, produced by Algorithm 5.1 with input regular expression RE_{init}, is a super-automaton of $L(RE_{init})$, i.e. $L(FA^s) \supseteq L(RE_{init})$.
Proof Consider any two states s_1 and s_2 of $Brz(RE_{init})$. The full language of s_1 is $\overleftrightarrow{L}(s_1) = \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_1)$ and similarly, the language of s_2 is $\overleftrightarrow{L}(s_2) = \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_2)$. If Algorithm 5.1 merges these states into one called $merge(s_1, s_2)$, then its language is:

$$\overleftrightarrow{L}(merge(s_1, s_2)) = (\overleftarrow{L}(s_1) \cup \overleftarrow{L}(s_2)) \cdot (\overrightarrow{L}(s_1) \cup \overrightarrow{L}(s_2))$$

Distributing $\cdot$ over $\cup$ gives

$$\begin{array}{rc} & \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_1) \cup \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_2) \cup \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_1) \cup \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_2) \\ \supseteq & \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_1) \cup \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_2) \\ = & \overleftrightarrow{L}(s_1) \cup \overleftrightarrow{L}(s_2) \end{array}$$

Since $\overleftrightarrow{L}(merge(s_1, s_2)) \supseteq \overleftrightarrow{L}(s_1) \cup \overleftrightarrow{L}(s_2)$ for any two states s_1 and s_2 that are merged by Algorithm 5.1, it follows that $L(FA^s) \supseteq L(RE_{init})$. □

Note that Theorem 5.1 does not exclude the possibility of producing an automaton FA^s with the same language as the initial regular expression RE_{init}.

This happens when $\overrightarrow{L}(s_1) = \overrightarrow{L}(s_2)$. Let $\overrightarrow{L}(s_1) = \overrightarrow{L}(s_2) = \overrightarrow{L}(s_x)$, then:

$$\begin{aligned} & \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_1) \cup \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_2) \\ = \quad & (\overleftarrow{L}(s_1) \cup \overleftarrow{L}(s_2)) \cdot \overrightarrow{L}(s_x) \\ = \quad & \overleftrightarrow{L}(merge(s_1, s_2)) \end{aligned}$$

5.4 Summary

In this chapter, the original hashing version of Brzozowski's algorithm is presented. An example is given of an execution run of the algorithm resulting in the construction of a non-deterministic automaton. It is also proven that, in general, the algorithm constructs a super-automaton. In the next chapter, a new algorithm is proposed that will construct a deterministic automaton.

Chapter 6

Brzozowski's algorithm with state merging — the DFA version

In previous chapters, it was shown that the algorithm proposed in [23] has to be designed to construct a non-deterministic super-automaton of the input regular expression. In this chapter, the algorithm is modified to construct a deterministic automaton, which is still a super-automaton. Note that the language of the two super-automata need not be the same.

6.1 The source of non-determinism

Algorithm 5.1 deliberately constructed a non-deterministic finite state automaton. This is manifested in the fact that its transition function was designed to map to a set of states, rather than to a single state. This was done in anticipation of the possibility that the algorithm generates a transition on a given symbol from a given state to more than one state.

Algorithm 5.1 places no restrictions on the properties of states that are merged. Merging is simply the consequence of unforeseen hash collisions. Furthermore, the algorithm computes the out-transitions on all symbols of states involved in the merging, and combines them as out-transitions of the single merged state. The source of the non-determinism is when symbols of the out-transitions of two or more merged states coincide, and these symbols go to *different* next states.

This is expressed in formal terms in the following theorem:

Theorem 6.1 Suppose that two states of the automaton produced in Algorithm

4.1, $state(RE_i)$ and $state(RE_j)$, are merged in Algorithm 5.1, into a state[1] $h = merge(state(RE_i), state(RE_j))$. Then $|\delta(h, a)| > 1$ if and only if

$$a \in (first(RE_j) \cap first(RE_i)) \wedge (hash(a^{-1}RE_i) \neq hash(a^{-1}RE_j))$$

Proof The theorem follows from the following observations regarding left derivatives and first symbol sets of regular expressions:

- The first symbol sets of RE_i and RE_j provide the out-transition labels for $state(RE_i)$ and $state(RE_j)$ respectively.
- The intersection $(first(RE_i) \cap first(RE_j))$ contains the out-transition symbols shared by $state(RE_j)$ and $state(RE_i)$.
- The left derivatives of RE_i and RE_j with respect to a provide the respective destination states if a transition on a is made. If these destination states differ in Algorithm 5.1, then the assignment in Algorithm 5.1:

  $$\delta(h, a) := \delta(h, a) \cup hash(destination)$$

  will insert two different values into $\delta(h, a)$, resulting in $|\delta(h, a)| > 1$.

□

The foregoing theorem clearly points to the source of the non-determinism in Algorithm 5.1. It is generally recognised that it is more practical to rely on a deterministic automaton for string recognition, both in terms of simplified data structures for storage (the NFA cannot be represented as a two dimensional state transition table) and in terms of greater efficiency in the associated recognition algorithm. For this reason it would be desirable to have a deterministic version of Algorithm 5.1.

One route would be to simply deploy the classical determinising algorithm on the output of Algorithm 5.1, thus obtaining an equivalent deterministic finite state automaton. (See for example [13] and [10].) However this kind of algorithm is not only rather inefficient, but also increases the number of states (i.e. the size) of the resulting input automaton. Since much of this dissertation is concerned with deriving automata that are smaller than those from the conventional Brzozowski Algorithm, such an approach to determinising is self-defeating. An alternative to Algorithm 5.1 has therefore been derived. It generates directly a deterministic super-automaton of the language of the input regular expression, and thus, by implication, complies with the objective of limiting the size of the generated automata.

[1]Note that h denotes a *state* here. In the Section 5.2, states are represented by hash codes. Therefore h was used.

6.2 General algorithm overview

Note that an alternate notation for the state transition function is used in Algorithm 6.1:

Notation 6.1 (The function δ as a set of pairs)
The state transition function can be seen as a set of nested pairs. Thus, if the pair consisting of input state h and state transition symbol a and destination state d (i.e. if $\delta(h, a) = d$), then an element of the set of nested pairs can be viewed as $\langle\langle h, a\rangle, d\rangle$. □

This convenient notation allows the manipulation of the state transition function with set operators, such as the relative set difference.

It will be seen that the automaton derived by Algorithm 6.1 is identical to that derived by Algorithm 5.1, provided that no opportunities for non-determinism as described in Theorem 6.1 arise. It will also be seen that under certain circumstances, the language of the super-automaton derived by Algorithm 6.1 is identical to that derived by Algorithm 5.1. However, even when these conditions do not arise, Algorithm 6.1 will produce a super-automaton of the language of its input regular expression.

Figure 6.1 illustrates the scenario that can arise in Algorithm 5.1 — i.e. the kind of scenario which Algorithm 6.1 has to overcome. The figure assumes several iterations of Algorithm 5.1's *do*-loop has been performed, and that the following sequence of events occurred:

1. At some stage of the computation, RE_i has been identified as the left derivative with respect to some symbol of some regular expression. Its hash function value is determined as a state of the automaton. Subsequently RE_i is placed in the *todo* list for further handling.

2. At some later stage, RE_j too is identified as the left derivative with respect to some symbol of some regular expression. Its hash code is also determined, but found to collide with that of RE_i. Thus state merging occurs as explained before. Therefore a single merged state has been created, as shown on the left side of the figure. Also, at this point, RE_j is placed on the *todo* list for subsequent processing.

3. Later, the regular expression selected for processing from the *todo* list is RE_i. As part of the inner *for*-loop of Algorithm 6.1, s was identified as an element of $first(RE_i)$ and the next sate for this symbol was generated. This is indicated in the figure as the state marked $hash(s^{-1}RE_i)$, which has an inbound arc labelled s.

4. Some time thereafter, the regular expression selected for processing from the *todo* list is RE_j. Again, in the for-loop, s was identified as an element of $first(RE_j)$ and consequently, the state marked $hash(s^{-1}RE_j)$ was defined, with the inbound arc s. This state, too is shown in the figure.

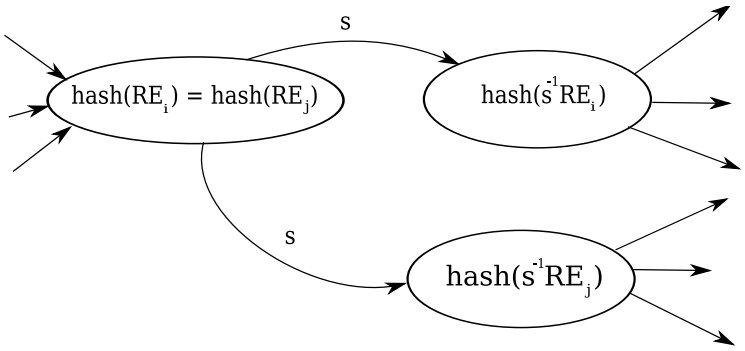

Figure 6.1: Non-determinism resulting from Algorithm 5.1

Algorithm 6.1 effectively carries out the following steps to avoid this kind of situation arising.

1. At step 3 above, a check is carried out to identify whether an s-transition from the merged state already exists. If not, computation proceeds as normal.

2. If, however, such a transition already exists, then the following sequence of actions takes place:

   (a) The transition function is provisionally modified so that it no longer indicates the previously derived s-transition from merged state to successor state. This corresponds to removing the top arc labelled s in Figure 6.1.

   (b) The regular expression for generating a destination node is revised from $s^{-1}RE_i$ to $(s^{-1}RE_i \cup s^{-1}RE_j)$

   (c) This regular expression is inserted into the *todo* list so that it will subsequently be used to create a state from its hash value.

   (d) The transition function is updated so that it indicates an s-transition from the merged state to the new state identified by $hash(s^{-1}RE_i \cup s^{-1}RE_j)$

3. Thereafter, computation proceeds as normal. However, note that this means that at some stage, the regular expression $(s^{-1}RE_i \cup s^{-1}RE_j)$ will be found in the *todo* list, and its successors will be determined in the normal manner.

Note also, that because of the union operator, the first symbol set of this regular expression will correspond to the union of the first symbol sets of its constituents. As a result, there will be a transition from this new state for each symbol for which there would have been a transition in Algorithm 5.1.

Figure 6.2 is now changed to represent a deterministic finite state automaton, constructed using the above changes, starting from Figure 6.1.

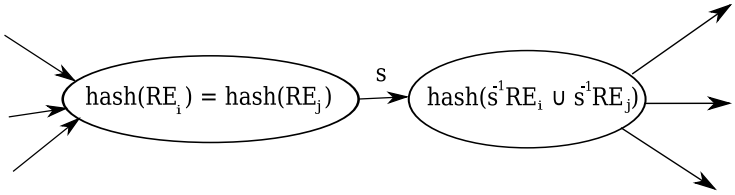

Figure 6.2: Determinised results from Algorithm 6.1

In order for the automata produced by Algorithm 5.1 and 6.1 to represent the same super language of the automaton from Algorithm 4.1, the following constraint must be met:

$$h(s^{-1}RE_i) = h(s^{-1}RE_j) = h(s^{-1}RE_j \cup s^{-1}RE_i)$$

This will be the case when the hash function is an injection from the regular languages to the natural numbers. This will be further discussed in the next chapter. Even if this condition is not satisfied, a super-automaton of the input regular expression for Algorithm 6.1 is still constructed, because of state merging, as shown in Theorem 5.1.

6.3 Example of the algorithm execution

An example run of the algorithm is now presented, in order to make comparisons between Algorithm 6.1 and 5.1. The same regular expression that was used to illustrate the other algorithms (Section 4.5 and 5.2) is used here: $(a \cdot b \ \cup \ b \cdot c)^*$.

For the first iteration of the *do*-loop, the same automaton is constructed that is constructed in Section 5.2, using the same hash function: the same derivatives and first symbols are calculated. The automaton at the end of the first iteration of the *do*-loop is depicted in Figure 6.3. At this point, $done = \{(a \cdot b \ \cup \ b \cdot c)^*\}$ and

$$todo = \{b \cdot (a \cdot b \ \cup \ b \cdot c)^*, c \cdot (a \cdot b \ \cup \ b \cdot c)^*\}$$

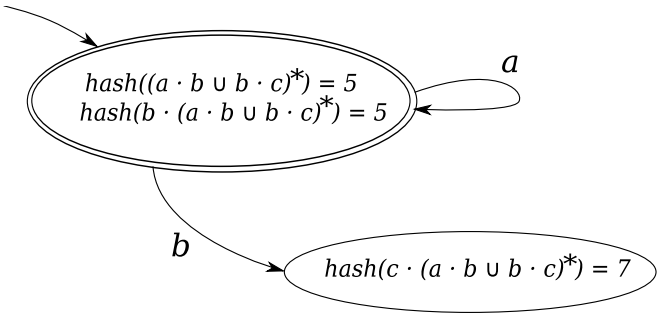

Figure 6.3: Partially constructed automaton constructed using Algorithm 6.1

The key to understanding this algorithm is that a NFA is never constructed. Even in a partially complete state, the automaton is always deterministic. In the second iteration of the *do*-loop, suppose that the *current regular expression* selected by the *let*-statement, is $b \cdot (a \cdot b \ \cup \ b \cdot c)^*$. Note that $first(b \cdot (a \cdot b \cup b \cdot c)^*) = b$. When this regular expression's left derivative with respect to b is calculated, the next state is

$$hash(b^{-1}(b \cdot (a \cdot b \ \cup \ b \cdot c)^*)) = hash((a \cdot b \ \cup \ b \cdot c)^*) = 5$$

At this point, the state transition function has not been updated yet. Therefore the current state transition table is the one in Table 6.1.

Remapped Value (state)	a	b	c
5	5	7	
7			

Table 6.1: Partially constructed transition table for Algorithm 6.1

If the transformations from Section 6.2 are not applied a NFA is constructed. The update of the state transition table would be:

$$\delta(5, b) := \delta(5, b) \cup \{hash((a \cdot b \ \cup \ b \cdot c)^*)\} = \{5, 7\}$$

This is as it is in Algorithm 5.1. In order to prevent this, the transformations from Section 6.2 are applied: first the condition of the first guard of the first *if*-statement in the *for*-loop of Algorithm 6.1 is satisfied:

$$(\exists d : \ \langle\langle h, s\rangle, d\rangle \ \in \delta) = true$$

This is because there is already a transition from state

$$h = hash(b \cdot (a \cdot b \ \cup \ b \cdot c)^*)) = 5$$

on symbol $s = b$ to the destination state

$$d = hash(c \cdot (a \cdot b \ \cup \ b \cdot c)^*) = 7$$

With the guard satisfied, the state transition function is rewritten:

$$\begin{aligned} \delta \ &:= \ \delta \setminus \{\langle\langle h, s\rangle, d\rangle\} \\ &= \ \delta \setminus \{\langle\langle 5, b\rangle, 7\rangle\} \\ &= \ \{\langle\langle 5, b\rangle, 5\rangle\} \end{aligned}$$

The automaton now has an unconnected state $d = 7$. In the implementation discussed in Appendix A, this state is not removed from the automaton, because there may be other states that still point to it. In a system implementing garbage collection, such a state could be cleaned up automatically when garbage is collected in order to free memory, if there are no pointers pointing to it.

A new regular expression is now constructed from the *current regular expression*'s left derivative with respect to b, and the destination state for which the transition have been removed:

$$\begin{aligned} destination \ &:= \ destination \ \cup \ regex(d) \\ &= \ ((a \cdot b \ \cup \ b \cdot c)^*) \cup (c \cdot (a \cdot b \ \cup \ b \cdot c)^*) \end{aligned}$$

Note that the $\cup$ operator is overloaded here: $\cup$ refers to the *regular expression* operator and not the *set union* operator. This new regular expression represents the new destination state of the automaton that is determinised. Next the *done* and *todo* sets are updated. Note that the *todo* set now contains the new destination state, which allows for further expansion of the automaton. The state transition function is updated next. For this purpose, a new hash value for the destination state is given in Table 6.2.

The updated automaton is given in Figure 6.4, which also includes the part of the automaton constructed when the current regular expression is $c \cdot (a \cdot b \ \cup \ b \cdot c)^*$: this represents the rest of the automaton constructed in the *do*-loop. Bear in mind that $c \cdot (a \cdot b \ \cup \ b \cdot c)^*$ is now not reachable from the rest of the automaton, but that it

Regular expression	Hash code
$(a \cdot b \cup b \cdot c)^*) \cup (c \cdot (a \cdot b \cup b \cdot c)^*)$	3
$(a \cdot b \cup b \cdot c)^*$	5
$b \cdot (a \cdot b \cup b \cdot c)^*$	5
$c \cdot (a \cdot b \cup b \cdot c)^*$	7

Table 6.2: Hash codes for example run of Algorithm 6.1

will still be expanded because it is in the *todo* set. In other scenarios, it may be the case that there are other states pointing to $state(c \cdot (a \cdot b \cup b \cdot c)^*)$. Therefore this expansion is still necessary. It is possible to clean up the automaton, but this is not shown here.

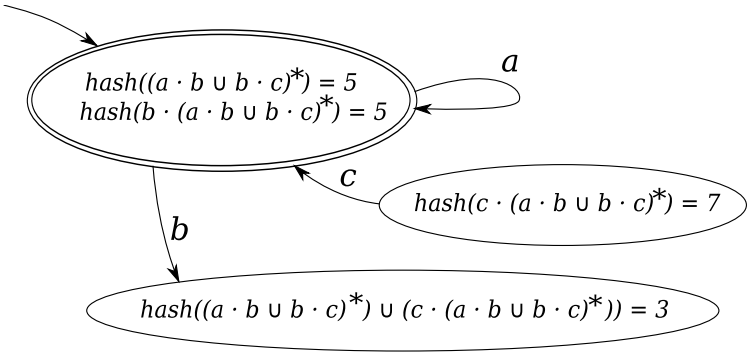

Figure 6.4: Determinised automaton for example run of Algorithm 6.1

The rest of the automaton construction process is not shown in this example.

6.4 Summary

In this chapter, a new version Brzozowski's algorithm is presented that constructs a deterministic automaton, where the version in the previous chapter constructed a non-deterministic automaton. In the next chapter, the question of how this automaton differs from the exact automaton will be addressed.

Algorithm 6.1 (Brzozowski's algorithm with Hashing — DFA version)

func $Brz\_hash\_DFA(RE_{init})$
  $Q, \delta, F := \emptyset, \emptyset, \emptyset;$
  $done, todo, := \emptyset, \{RE_{init}\};$
  do $todo \neq \emptyset \rightarrow$
    let RE_j be some regular expression such that $RE_j \in todo$;
    $done, todo := done \cup \{RE_j\}, todo \backslash \{RE_j\};$
    $h := hash(RE_j);$
    $Q := Q \cup \{h\};$
    { Expand out-transitions for symbols in the first symbol set of RE_j }
    for $s : first(RE_j) \rightarrow$
      { Compute the left derivative of RE_j with respect to s }
      $destination := s^{-1}RE_j;$
      if $(\exists d : \langle\langle h, s\rangle, d\rangle \in \delta) \rightarrow$
        $\delta := \delta \setminus \{\langle\langle h, s\rangle, d\rangle\};$
        $destination := destination \cup regex(d)$
      [] $(\nexists d : \langle\langle h, s\rangle, d\rangle \in \delta) \rightarrow$ **skip**
      fi;
      if $destination \notin (done \cup todo) \rightarrow$
        $todo := todo \cup \{destination\}$
      [] $destination \in (done \cup todo) \rightarrow$ **skip**
      fi;
      $\delta := \delta \cup \{\langle\langle h, s\rangle, hash(destination)\rangle\}$
    rof;
    if $nullable(RE_j) \rightarrow$
      { A state is final if and only if it accepts the empty string }
      $F := F \cup \{h\}$
    [] $\neg nullable(RE_j) \rightarrow$ **skip**
    fi
  od;
  return $\langle Q, \delta, hash(RE_{init}), F\rangle$
cnuf

Chapter 7

Equivalence classes of regular languages

In previous chapters, regular expressions and their properties were constructed from basic definitions. Following these definitions, Brzozowski's algorithm was presented, along with an existing variant constructing a non-deterministic super-automaton, and a new variant constructing a deterministic super-automaton. The question now naturally arises: how does the super-automaton of an input regular expression constructed by the hashing versions of the algorithm differ from an exact automaton of the same input regular expression? How can the difference be measured? These questions are adressed in this chapter.

7.1 k-equivalence classes

Equivalence classes of states of finite state automata are generally used in automaton minimization algorithms [21]: states are divided into these classes to determine which states are equivalent.

One approach to test whether two states are in the same equivalence class, is to iterate through all strings accepted by the two states. If these strings form the same regular language, the two states can be combined in the minimized automaton. This can be done recursively, using the following definition [6]:

Definition 7.1 (k-equivalence classes for states in a finite state automaton)
Two states t_1 and t_2 in a finite state automaton are:

- 0-equivalent if and only if both t_1 and t_2 are either accepting or rejecting states.

- k-equivalent if, for each transition symbol s from the states t_1 and t_2, the next states $\delta(t_1, s)$ and $\delta(t_2, s)$ are $(k-1)$-equivalent.

The predicate $equiv(t_1, t_2, k)$ asserts that t_1 and t_2 are k-equivalent. □

The value of k gives the length for up to which the strings accepted by two states are equal. When two states are equivalent for *all* values of k, they are *-equivalent.

In Brzozowski's algorithm, an automaton is constructed from regular expressions which are associated with states. Therefore it is convenient to extend the notation of equivalence to regular expressions, as follows:

Definition 7.2 (k-equivalence classes for regular expressions)
Two regular expressions RE_i and RE_j are:

- 0-equivalent if and only if $nullable(RE_i) = nullable(RE_j)$
- k-equivalent if and only if $first(RE_i) = first(RE_j)$ and for all $s \in first(RE_i)$, $s^{-1}RE_i$ and $s^{-1}RE_j$ are $(k-1)$-equivalent.

The predicate $equiv(RE_i, RE_j, k)$ tests whether RE_i and RE_j are equivalent. □

Definition 7.2 giving k-equivalence classes on regular expressions, is derived from Definition 7.1 giving k-equivalence on the states of finite state automata. In order to illustrate the connection between these two definitions, consider Figure 7.1 and 7.2.

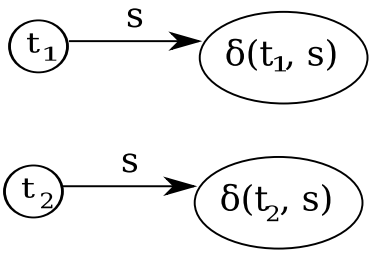

Figure 7.1: k-equivalence and finite state automata

When comparing Figure 7.1 and 7.2, it can be seen how regular expressions are substituted for states. In order to address the case where $k = 0$, it should be taken into account that when two states are either accepting or rejecting states, their right languages either accept or reject the empty string. *nullable* (see Section 3.19) provides a method for testing whether a regular expression represents a regular language that contains the empty string.

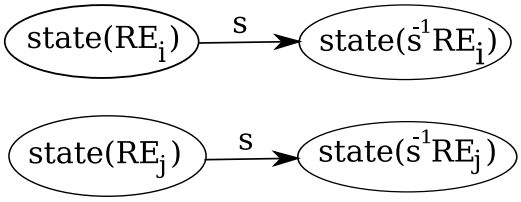

Figure 7.2: k-equivalence and regular expressions

7.2 Exact *versus* super-automaton minimization

The k-equivalence concept can be used to distinguish between a conventional minimization algorithm, not admitting extra strings into the regular language, and a reduction approach constructing a super-automaton.

In the exact automaton minimization scenario, all strings that are accepted by two states, which are candidates to be merged in the minimization process, are generated. The length of these strings must be limited in order to prevent the generation of an infinite number of strings. This is done by factoring in a result from [21]: when the string length is equal to $\mathbf{max}(|Q|-2, 0)$, with Q representing the number of states in the exact automaton, enough strings have been compared to conclude whether two states accept the same language for strings of all length. The *max* function is used to prevent comparisons with negative string lengths: when the length is less than or equal to 2, the test only tests whether the two states accept the empty string. In this case, the states must be both accepting or rejecting states in order to be equal.

The string length in the exact automaton minimization scenario corresponds with the k variable from Definitions 7.1 and 7.2. Therefore, for exact automaton minimization, it *must* be the case that $k \geq \mathbf{max}(|Q| - 2, 0)$.

In the super-automaton scenario, it is acceptable that $k < \mathbf{max}(|Q| - 2, 0)$: any two states can be merged together. The value of k is determined by the languages of the states being merged together, rather than being prescribed to a value of $\mathbf{max}(|Q| - 2, 0)$: k is a measure of the length up to which the strings of the two regular expressions are equal.

In order to formalize this relation between exact and super-automata, a Boolean expression that indicates whether two regular expressions are k-equivalent or not can be formulated. This bears a relation to existing work in [21]: reference is made to the equivalence relation defined in Definition 7.1 and 7.2 with $E \subseteq Q \times Q$ and

$$\langle p, q \rangle \ : E \equiv [\overrightarrow{L}(p) = \overrightarrow{L}(q)]$$

In Brzozowski's algorithm, two regular expressions RE_i and RE_j represent $state(RE_i)$ and $state(RE_j)$. In the case of super-automata, let $p = state(RE_i)$ and $q = state(RE_j)$, then the above equivalence relation is changed to

$$\langle p, q \rangle \ : E \equiv equiv(RE_i, RE_j, k)$$

The usage of regular expressions avoids the need to take the differences between a NFA and a DFA into account: they both represent regular languages.

7.3 Categories of hash functions

The k-equivalence notion provides an opportunity to divide hash functions into three categories, in order to judge hash function quality:

- ***Worst Case:*** The merged states are not even 0-equivalent;
- ***Intermediate Case:*** Some merged states are $0 \leq k < \mathbf{max}(|Q| - 2, 0)$ equivalent and some are $*$-equivalent; and
- ***Best Case:*** For all merged states, $k \geq \mathbf{max}(|Q| - 2, 0)$.

In the worst case, the automaton accepts all strings in the regular language over the alphabet. However, this is not the measure for the worst case: it might be the case that the automaton being constructed must accept the language Σ^*. The k-equivalence measure defined in Definition 7.2 offers an alternative means of assessing deviation from the exact automaton.

7.4 Ideal hash functions

The foregoing raises the question: what are the characteristics of an ideal hash function? To give such a characterisation, note that the signature of hash functions in Algorithm 5.1 and 6.1 is of the form $hash : R^e \rightarrow \mathbb{N}$, where R^e is the set of regular expressions.

Let R^ℓ be the set of regular languages, and $L : R^e \rightarrow R^\ell$, and let $L(RE)$ be the regular language associated with regular expression RE. Finally, let f denote a function $f : R^\ell \rightarrow \mathbb{N}$. An ideal hash function may now be defined as the composition of the latter two functions as follows:

Definition 7.3 (The ideal hash function*)
hash is an ideal hash function for R^e if and only if $hash = f \cdot L$ and f is an injection. □

This means an ideal hash function maps all regular expressions that have the same language (and only those expressions) to the same natural number. Put differently, an ideal hash function maps regular expressions to the same value if and only if they are *-equivalent. Thus, if it were possible to find an ideal hash function for use in Algorithm 6.1, then the algorithm would be guaranteed to produce the *minimum exact* DFA for the input regular expression.

7.5 There is no ideal hash function *modulo* n

Generally, when a hash function is used to hash a data structure in a programming language, the hash code is calculated, and then modulo n is taken. This maps the hash code to a small address space in the range $[0, n)$, which is used in an array data structure.

In Brzozowski's algorithm, the state transition function can be represented by a state transition table. A state transition table is a two dimensional array, where one dimension represents a state address, and the other represents alphabet symbols. An entry in the state transition table is the next state for a given symbol and a current state.

This array representation creates an opportunity to "project" the regular expression's automaton being constructed into a predetermined number of states, using the DFA version of Brzozowski's algorithm with hashing, presented in this dissertation. The DFA version has to be used, because a NFA cannot be represented by a state transition table in the form of a two dimensional array.

The projection is performed by taking *modulo* n of the hash code generated in the algorithm. This limits the number of states in the resulting automaton to a maximum of n. Due to Theorem 5.1, a super-automaton will still be constructed.

From the point of view of k-equivalence classes, this creates an additional constraint on the hash function: the *modulo* n function reduces the number of states from Algorithm 5.1 and 6.1 to a maximum of n. In finite state automaton reduction, each equivalence class represents an abstract state. Therefore there is now an upper bound on the number of equivalence classes $|[Q]_{equiv(RE_i,RE_j,k)}|$ induced by the equivalence relation $\langle p, q \rangle : E \equiv equiv(RE_i, RE_j, k)$. This gives the following inequality

that must be satisfied if the merged together states must be at least k-equivalent, for a given hash function:

$$|[Q]_{equiv(RE_i,RE_j,k)}| \leq n$$

As long as this inequality is satisfied, the number of equivalence classes is potentially less than the number of available addresses: it is therefore up to the hash function to hash together regular expressions that are equivalent up to the value k.

It is however possible that the hash function will produce more values than the value of n. In this case the modulo function will force together hash values representing different regular expressions that are not k-equivalent. When the maximum value of k is calculated for the case where the hash values have been forced together, the resulting value of k will be lower than in the case where the values have not been forced together. This results in a super-automaton that has more additional strings, in addition to those of the regular language represented by the input regular expression.

When $n < |[Q]_{equiv(RE_i,RE_j,k)}|$ for the case where the states in the equivalence classes are *-equivalent, non-equivalent states will become merged due to the pigeon hole principle [15]. For this reason there is no ideal hash function for any n.

7.6 Summary

In this chapter, finite state automaton reduction through state merging has been related back to traditional finite state automaton minimization through the concept of k-equivalence classes. This serves as a measure of the difference between exact and super-automata. The number of equivalence classes gives a way of interpreting the effects of taking *modulo* n in the hash function: when n is less than the number of equivalence classes, no ideal hash function exists.

Chapter 8

Implementation

This chapter brings together concepts from all preceding chapters in an implementation. Nine hash functions were tested on a large set of small regular expressions, and a small set of large regular expressions. This chapter provides the basis for the next chapter, which will focus on the results.

8.1 General implementation outline

Figure 8.1 gives a high level overview of the process followed in the implementation. This process is designed to test the k-equivalence of regular expressions generated with the algorithms in this dissertation.

In this process, there is a main loop iterating over a Gödel Numbering from 0 to 50 000 000. Gödel numbers are briefly discussed in Section 8.2. Each number is converted into a string using Algorithm 8.1. This string is parsed as a prefix notation string representing a regular expression.

Some of the strings generated from the Gödel numbers do not parse and have to be discarded. The parsed regular expressions are rewritten based on the rewrite rules under in Section 4.2. As an example of this, consider the following regular expression:

$$(\varepsilon \cdot b)^* \cup c \cdot d \cup \emptyset$$

This is rewritten to:

$$b^* \cup c \cdot d$$

which may already have occurred in the input. Duplicate input strings are eliminated at this point in order to guarantee that the same test regular expressions are not

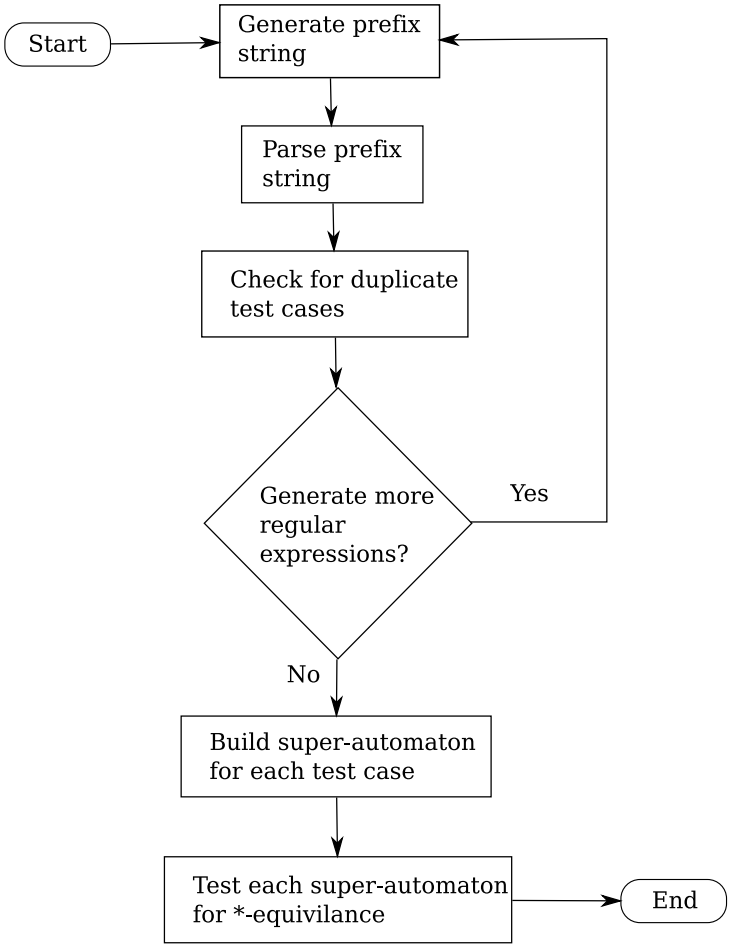

Figure 8.1: The procedure for testing hash functions and *-equivalence

used twice. The outcome of this process was the generation of 272 850 regular expressions.

Using the test regular expressions as input, the super-automata are build based on the handpicked hash functions in Section 8.5 and 8.6. These hash functions were converted into C# source code, and compiled into the implementation.

The k-equivalence of all merged states are measured as outlined in Section 7.1. The number of automata where all merged states are *-equivalent is reported as a percentage of the number of test expressions, along with statistics on size reductions.

8.2 Gödel numbering and regular expression generation

A Gödel numbering is a method for assigning a unique integer to each string over an alphabet. There are two definitions for a Gödel numbering in literature. The first is [11]:

Definition 8.1 (Gödel numbering with prime numbers)
Given a string $w = w_0 \dots w_{n-1}$ with alphabet Σ and a mapping $i : \Sigma \to \mathbb{N}$ with i assigning a uniquely increasing number in increments of 1 to each alphabet symbol, with the first symbol labelled 1, the Gödel number of w is

$$gn_{prime}(w) = p_0^{i(w_0)} \times p_1^{i(w_1)} \times \ldots \times p_{n-1}^{i(w_{n-1})}$$

where p_k represents the k^{th} prime number and $p_0 = 2$. □

The following definition [9] was preferred over Definition 8.1. This is for two reasons. The first is that the inverse mapping from numbers to strings can easily be calculated using Algorithm 8.1. The second is that the sizes of the numbers involved are smaller:

Definition 8.2 (Gödel numbering without prime numbers)
Given a string $w = w_0 \dots w_{n-1}$ with alphabet Σ and a mapping $i : \Sigma \to \mathbb{N}$ with i assigning a uniquely increasing number to each alphabet symbol, with the first symbol labelled 1, and $\beta = |\Sigma| + 1$, the Gödel number gn for string w is

$$gn(w) = \beta^{n-1} i(w_{n-1}) + \beta^{n-2} i(w_{n-2}) + \ldots + \beta^1 i(w_2) + \beta^0 i(w_0)$$

□

It should be noted that in the numbering scheme above, 0 is not assigned by the mapping i. The function gn does not generate the number 0. 0 forms a *null* character. If 0 was recognized some numbers would be assigned to multiple strings because any integer m prefixed by any number of zeros remains equal to m. This is taken into account in Algorithm 8.1, which converts an integer into a string.

In Algorithm 8.1, $numChar(n)$ is the inverse of the function i in Definition 8.2.

In this dissertation, the alphabet is set to $\{0, 1, 2, 3, \cdot, \cup, *, +, ?, \varepsilon, \emptyset\} = \Sigma$ for the purpose of generating regular expressions from Gödel numbers. Therefore, the alphabet of the automata constructed is $\{0, 1, 2, 3\}$. The strings generated from the numbers are parsed as prefix notation strings. When a string does not parse into a regular expression, it is discarded. When this happens, Algorithm 8.1 returns the empty set, rather than a string. This empty set it different from the empty set character, which forms part of the input alphabet.

Algorithm 8.1 (Converting a Gödel number into a string)

```
func gnToString(gNum)
    retStr := ε;
    β := |Σ| + 1;
    do gNum > 0 →
        d := gNum mod β;
        gNum := gNum div β;
        if d = 0 →
            retStr := ∅;
            gNum := 0
        [] d ≠ 0 →
            retStr := retStr · numChar(d)
        fi
    od;
    return retStr
cnuf
```

8.3 Short *versus* long regular expressions

The process for measuring the hash function quality exhaustively enumerates all short regular expressions up to a length of 6 characters, and some regular expressions of 7 characters. These lengths are the result of choosing the upper bound value of 50 000 000 when iterating over the Gödel numbers to generate test data. This process yielded only 272 850 regular expressions.

Even though the sample is representative of all short regular expressions, and therefore presents a good measure of the performance of the hash functions with respect to other regular expressions, it is not so simple to choose a representative sample of long regular expressions. Some regular expressions, especially those with a lot of plus operators, causes Brzozowski's algorithm to execute over a long period of time, making it unfeasible to test the hash function for these regular expressions.

Another problem with long regular expressions is that it takes very long to measure k-equivalence of these regular expressions. In effect the process of measuring k-equivalence generates all strings in the regular language of an automaton up to length k. The number of these strings is potentially exponential, depending on the first symbol sets of the input regular expressions.

It might be possible to argue that the Gödel numbers used in the short regular

expressions should be used again: first generate a very large random number and convert it into a regular expressions. The problem with this process is that the prefix notation representation of the regular expressions causes the generated numbers to have a very low probability of parsing into a valid regular expressions. This can be seen in the 50 000 000 numbers that were reduced to 272 850 regular expressions.

For this reason, the large regular expressions are generated by randomly selecting short regular expressions from the exhaustively enumerated set of short regular expressions, and then connecting these regular expressions using regular language operators. The regular expression operators used to connect the short regular expressions are chosen randomly. The long regular expressions are given in Appendix B.

8.4 Choosing hash functions operators

In order to select the mappings from the regular expressions to the natural numbers, the recommendations from [23] were adopted, as follows:

- ε and $\emptyset$ are mapped to $000\ldots000_{16}$ or $FFF\ldots FFF_{16}$ respectively;
- unary and binary regular expression operators are mapped to unary and binary bit string operators;
- commutative idempotent regular expression operators are mapped to commutative idempotent bit string operators; and
- the alphabet is mapped to $1\ldots|\Sigma|$.

The operators used to select hash functions are given in Tables 8.1 and 8.2. Their basic algebraic properties are also given.

Operator	Commutativity	Associativity	Idempotence
$x_1 \vee (1 << (n-1))$	No	No	Yes
$x_1 \wedge x_2$	Yes	Yes	Yes
$x_1 \vee x_2$	Yes	Yes	Yes
$x_1 \rightarrow x_2 \equiv \neg x_1 \vee x_2$	No	Yes	No

Table 8.1: Bit string operator properties

The first operator was found in [23]. n is the number of bits used to represent an integer in a machine register, $<<$ represents the left shift operation.

Operator	Commutativity	Associativity	Idempotence
RE_1^*	No	No	Yes
$RE_1 \cup RE_2$	Yes	Yes	Yes
$RE_1 \cdot RE_2$	No	Yes	No

Table 8.2: Regular expression operator properties

All the operators in Table 8.2 are non-extended regular expression operators. The extended operators are constructed from non-extended operators and then hashed, using the method illustrated in the next section.

8.5 Applying operator mappings

The operator mappings used to gather the test results in Tables 9.1 and 9.2 are given in Tables 8.3 and 8.4. Each set of operator mappings represents a hash function. Hash functions in Tables 8.3 and 8.4 are numbered to correspond with results in Tables 9.1 and 9.2.

An example of calculating the hash code using mapping 6 is now given, using the following regular expression:

$$(\varepsilon \cdot b)^* \cup c \cdot d \cup \emptyset$$

Alphabet symbols are mapped to increasing integers starting from 1 (because $\emptyset$ already uses the number 0). This substitution can be seen in this intermediate step:

$$((FFF \ldots FFF_{16}) \cdot (000 \ldots 001_{16}))^* \cup (000 \ldots 002_{16}) \cdot (000 \ldots 003_{16}) \cup (000 \ldots 000_{16})$$

Substituting the operator mappings for the *concatenation*, *union* and *star closure* operators gives the final expression that can be evaluated in order to get the hash code:

$$\begin{aligned} & (((\neg(FFF \ldots FFF_{16}) \vee (000 \ldots 001_{16}))) \vee (1 << (n-1))) \\ \vee \quad & (\neg(000 \ldots 002_{16}) \vee (000 \ldots 003_{16})) \\ \vee \quad & (000 \ldots 000_{16}) \end{aligned}$$

In the implementation, this substitution is done by recursively moving over the regular expression using the hash function. This is indicated by the h in Tables 8.3 and 8.4. Extended operators are handled as a separate case: this can be seen by

comparing the optional regular expression operator $E^?$ for mapping 5 and 6. From Section 3.3, $RE^? = \varepsilon \cup RE$. Applying the substitutions in mapping 6 gives:

$$RE^? \mapsto FFF \ldots FFF_{16} \vee h(RE)$$

but applying the substitutions in mapping 5 gives:

$$RE^? \mapsto FFF \ldots FFF_{16} \wedge h(RE)$$

The main difference lies in the mapping for the union regular expression operator: for mapping 5, $RE_1 \cup RE_2 \mapsto h(RE_1) \wedge h(RE_2)$, but for mapping 6, $RE_1 \cup RE_2 \mapsto h(RE_1) \vee h(RE_2)$.

8.6 An alternative hash function

An alternative hash function was also tested, that guarantees that all merged states are at least 1-equivalent. It uses the following definitions and notation:

Definition 8.3 (Longest substring length)
The longest substring length is inductively defined by the following rules:

- $len\_subseq(\emptyset) = 0$
- $len\_subseq(\varepsilon) = 0$
- $len\_subseq(s) = 1$
- $len\_subseq(A \cdot B) = len\_subseq(A) + len\_subseq(B)$
- $len\_subseq(A \cup B) = \begin{cases} len\_subseq(A) & \text{if } len\_subseq(A) \geq len\_subseq(B) \\ len\_subseq(B) & \text{otherwise} \end{cases}$
- $len\_subseq(A^*) = len\_subseq(A)$
- $len\_subseq(A^+) = len\_subseq(A)$
- $len\_subseq(A^?) = len\_subseq(A)$

The longest substring indicates the length of the longest string in the regular expression, ignoring *-closure. □

Definition 8.4 (First symbol set bit string of a regular expression)
The first symbol set bit string $first\_b(RE)$ of a regular expression RE is a bit string b such that

$$b[i(s)] = \begin{cases} 1 & \text{if } s \in first(RE) \\ 0 & \text{otherwise} \end{cases}$$

where i is a function assigning an array index for a symbol s. This array index references a position in bit string b. □

The hash code itself it denoted by h_Σ:

Definition 8.5 (The hash function h_Σ (*))
The function h_Σ is a concatenation of three bit strings $b_1 \cdot b_2 \cdot b_3$ with

- $b_1 = first\_b(RE)$;
- $b_2 = \begin{cases} 1 & \text{if } nullable(RE) \\ 0 & \text{otherwise} \end{cases}$; and
- $b_3 = len\_subseq(RE)$

where b_3 is a binary representation of the value of $len\_subseq(RE)$. □

The reasoning behind the construction of this hash code is that, in order to guarantee 1-equivalence of the regular expressions being hashed together, they must at least accept the empty string, in addition to the same first symbol sets. This is guaranteed by the first two parts of the hash code $h_\Sigma = b_1 \cdot b_2 \cdot b_3$.

This concept has been extended to *at least* 1-equivalence by appending the longest substring length b_3 to $b_1 \cdot b_2$: consider the regular expression $a \cdot b \cup b \cdot a$: because the first symbol set is $\{a, b\}$ and the regular expression is not nullable, and the longest substring has a length of 2:

$$\begin{aligned} h_\Sigma(a \cdot b \cup b \cdot a) &= b_1 \cdot b_2 \cdot b_3 \\ &= 11 \cdot 0 \cdot 10 \\ &= 11010 \end{aligned}$$

Now consider the regular expression $a \cup b$: this regular expression also has the first symbol set $\{a, b\}$. If hashing only took the first symbol set into account,

$$h_\Sigma(a \cdot b \cup b \cdot a) = h_\Sigma(a \cup b)$$

This is undesirable, since the regular expressions have different regular languages. The difference is in the b_3 value: for $h_\Sigma(a \cup b)$, $len\_subseq(a \cup b) = 1$.

This inclusion of the length generates better hash codes, but is not sufficient to guarantee *-equivalence. The regular expressions $a^* \cdot a \cdot a$ and $(a^* \cdot a \cdot a) \cup a$ generate the same hash codes, even though they are not *-equivalent.

8.7 Summary

In this chapter, a method is put forward for measuring and comparing hash functions in the hashing versions of Brzozowski's algorithm. Gödel numbers are used to generate short test regular expressions. Long test regular expressions are constructed from the short test regular expressions. The hash functions that are tested in the implementation are presented. In the next chapter, the results of this experiment is presented.

Mapping 1
$\emptyset \mapsto 000 \ldots 000_{16}$
$\varepsilon \mapsto 000 \ldots 000_{16}$
$RE^{?} \mapsto 000 \ldots 000_{16} \wedge h(RE)$
$RE^{+} \mapsto \neg h(RE) \vee (h(RE) \vee (1 << (n-1)))$
$RE^{*} \mapsto h(RE) \vee (1 << (n-1))$
$RE_1 \cup RE_2 \mapsto h(RE_1) \wedge h(RE_2)$
$RE_1 \cdot RE_2 \mapsto \neg h(RE_1) \vee h(RE_2)$
Mapping 2
$\emptyset \mapsto 000 \ldots 000_{16}$
$\varepsilon \mapsto 000 \ldots 000_{16}$
$RE^{?} \mapsto 000 \ldots 000_{16} \vee h(RE)$
$RE^{+} \mapsto \neg h(RE) \vee (h(RE) \vee (1 << (n-1)))$
$RE^{*} \mapsto h(RE) \vee (1 << (n-1))$
$RE_1 \cup RE_2 \mapsto h(RE_1) \vee h(RE_2)$
$RE_1 \cdot RE_2 \mapsto \neg h(RE_1) \vee h(RE_2)$
Mapping 3
$\emptyset \mapsto FFF \ldots FFF_{16}$
$\varepsilon \mapsto 000 \ldots 000_{16}$
$RE^{?} \mapsto 000 \ldots 000_{16} \wedge h(RE)$
$RE^{+} \mapsto \neg h(RE) \vee (h(RE) \vee (1 << (n-1)))$
$RE^{*} \mapsto h(RE) \vee (1 << (n-1))$
$RE_1 \cup RE_2 \mapsto h(RE_1) \wedge h(RE_2)$
$RE_1 \cdot RE_2 \mapsto \neg h(RE_1) \vee h(RE_2)$
Mapping 4
$\emptyset \mapsto FFF \ldots FFF_{16}$
$\varepsilon \mapsto 000 \ldots 000_{16}$
$RE^{?} \mapsto 000 \ldots 000_{16} \vee h(RE)$
$RE^{+} \mapsto \neg h(RE) \vee (h(RE) \vee (1 << (n-1)))$
$RE^{*} \mapsto h(RE) \vee (1 << (n-1))$
$RE_1 \cup RE_2 \mapsto h(RE_1) \vee h(RE_2)$
$RE_1 \cdot RE_2 \mapsto \neg h(RE_1) \vee h(RE_2)$

Table 8.3: Hash function operator mappings

Mapping 5
$\emptyset \mapsto 000\ldots000_{16}$
$\varepsilon \mapsto FFF\ldots FFF_{16}$
$RE^{?} \mapsto FFF\ldots FFF_{16} \wedge h(RE)$
$RE^{+} \mapsto \neg h(RE) \vee (h(RE) \vee (1 << (n-1)))$
$RE^{*} \mapsto h(RE) \vee (1 << (n-1))$
$RE_1 \cup RE_2 \mapsto h(RE_1) \wedge h(RE_2)$
$RE_1 \cdot RE_2 \mapsto \neg h(RE_1) \vee h(RE_2)$
Mapping 6
$\emptyset \mapsto 000\ldots000_{16}$
$\varepsilon \mapsto FFF\ldots FFF_{16}$
$RE^{?} \mapsto FFF\ldots FFF_{16} \vee h(RE)$
$RE^{+} \mapsto \neg h(RE) \vee (h(RE) \vee (1 << (n-1)))$
$RE^{*} \mapsto h(RE) \vee (1 << (n-1))$
$RE_1 \cup RE_2 \mapsto h(RE_1) \vee h(RE_2)$
$RE_1 \cdot RE_2 \mapsto \neg h(RE_1) \vee h(RE_2)$
Mapping 7
$\emptyset \mapsto FFF\ldots FFF_{16}$
$\varepsilon \mapsto FFF\ldots FFF_{16}$
$RE^{?} \mapsto FFF\ldots FFF_{16} \wedge h(RE)$
$RE^{+} \mapsto \neg h(RE) \vee (h(RE) \vee (1 << (n-1)))$
$RE^{*} \mapsto h(RE) \vee (1 << (n-1))$
$RE_1 \cup RE_2 \mapsto h(RE_1) \wedge h(RE_2)$
$RE_1 \cdot RE_2 \mapsto \neg h(RE_1) \vee h(RE_2)$
Mapping 8
$\emptyset \mapsto FFF\ldots FFF_{16}$
$\varepsilon \mapsto FFF\ldots FFF_{16}$
$RE^{?} \mapsto FFF\ldots FFF_{16} \vee h(RE)$
$RE^{+} \mapsto \neg h(RE) \vee (h(RE) \vee (1 << (n-1)))$
$RE^{*} \mapsto h(RE) \vee (1 << (n-1))$
$RE_1 \cup RE_2 \mapsto h(RE_1) \vee h(RE_2)$
$RE_1 \cdot RE_2 \mapsto \neg h(RE_1) \vee h(RE_2)$

Table 8.4: Hash function operator mappings, continued

Chapter 9

Empirical results

In the previous chapter, a high level overview is given of the implementation, in addition to hash functions that are tested with the k-equivalence measure. In this chapter, the results of this experiment are given.

9.1 Measured results for short regular expressions

Table 9.1 gives the following information:

- The *Hash Function* column corresponds to the hash functions which were presented in Tables 8.3 and 8.4.
- The *Exact Automaton* column gives the percentage of exact automata constructed from the 272850 short regular expressions.
- The *Size Reduction* column gives the percentages of test cases where the constructed automaton had less states than the remap case (Algorithm 4.1).
- The *Both* column gives the percentage of exact automata constructed that also had a reduced size.

The table rows were ordered by the *Exact Automaton* column.

Both the NFA and DFA hashing versions of Brzozowski's algorithm were tested: both algorithms produced practically identical results. Table 9.1 reflects data for both algorithms. The difference in the algorithms lies in the memory used: the NFA version used less memory than the DFA version. This is to be expected, as the DFA version involves rewriting the destination state's regular expression when the conditions are satisfied that would cause a NFA to be constructed.

The similarity between the algorithms' results is attributed to the small size of the input regular expressions: when the regular expressions are small, there is a good chance that the DFA version's reconstructed target regular expression will hash to the same state as the state the original destination regular expression would have hashed to, in the NFA case. This is not the case where the long regular expressions are tested. This will be discussed in the next section.

The following general observations can be made from Table 9.1:

- The data in the *Size Reduction* column increases consistently as the *Exact Automata* column decreases. This can be explained in terms of the k-equivalence measure: for good hash functions, a large number of states are hashed together that represent the same regular language. This means less states are wrongly combined, resulting in larger automata. When the hash functions perform badly in terms of percentage exact automata, the number of states that are merged that do not represent equal right languages increases and the size of the constructed automata decreases.

- The values under the *Both* column header decreases and then increases, relative to the *Exact Automaton* column. When the number of exact automata constructed decreases, the chances of reducing the size of the output automaton increases.

- The hash function h_Σ had the best results. Only in 1 % of the cases did it produce super-automata — almost all automata were exact.

- The performance of a hash function seems to be related to the relation between the choice of bit string operator for the $\cup$ regular expression operator, and the associated choice of bit string operators for the *empty set* and the *empty string*.

The last point under the observations is made with respect to the behaviour of the $\vee$ and $\wedge$ bit string operators, and the choice of operator for $\emptyset$ and ε. When the $\vee$ operator is used in conjunction with $FFF \ldots FFF_{16}$, generated hash codes tend to favour the generation of 1's instead of 0's: taking bitwise *or*, for *any value* and 1, will be equal to 1.

The hash function recursively traverses the regular expression's expression tree. Therefore results from hashing sub-expressions in the recursive step will be destroyed by the $\vee$ operator, when a 1 is encountered. This is also true for the $\wedge$ operator and $000 \ldots 000_{16}$.

Bad operator mappings cause the same hash value to be generated repeatedly. When this happens, many regular expressions are hashed to the same value, causing states

Hash Function	Exact Automaton	Size Reduction	Both
h_Σ	99	21	20
2	70	52	22
4	69	54	23
5	68	54	22
7	67	55	22
3	58	55	13
1	58	55	13
6	44	79	23
8	43	80	23

Table 9.1: Statistics for short regular expressions

to be merged that do not represent the same regular language. This causes a low score for a hash function in terms of percentage exact automata constructed, as can be seen for hash function 8. In hash function 2, better results are achieved because the operator mappings preserve results.

9.2 Measured results for long regular expressions

The hash functions were also tested on longer regular expressions resulting in larger automata, as pointed out in Section 8.3. The difficulties associated with generating a representative set of large regular expressions makes it unclear how to define a statistically sound input sample. Nevertheless it seemed relevant to gain some idea of how the respective algorithms performed on larger regular expressions.

The main gain from the large regular expressions is that they prove that the NFA and DFA versions of Brzozowski's algorithm with hashing do not always produce identical results, as seemed to be the case with the short regular expressions.

The statistics for the long regular expressions are presented in Table 9.2. No exact automaton were constructed for the bit string operator based hash functions. As a result the tests for automaton quality in comparison to the exact automaton case could not be re-used. The *average k-equivalence* value was defined in order to deal with this problem:

Definition 9.1 (Average k-equivalence level*)
The *average k-equivalence level* value per state in Algorithm 5.1 and 6.1 is calculated in the following manner:

- When no collision occurred for the state, the average k-equivalence value of the state is taken as $\mathbf{max}(|Q-2|, 0)$.

- When the state represents a collection of regular expressions that are merged together by the hash function, the k-equivalence value of each pair of these regular expressions is found. These k values are added up, and divided by the number of regular expression pairs. When a pair of regular expressions is not even 0-equivalent, their equivalence level is taken to be 0. This happens when one regular expression accepts the empty string, and another does not.

□

Definition 9.2 (k-equivalence potential of an FA*)
The *k-equivalence potential of an FA* generated by Algorithm 5.1 or 6.1 is calculated by finding the average k-equivalence level of every *state* in the FA, and then computing the average of those averages. □

The 31 long regular expressions that were tested resulted in automata with a minimum of 6 and a maximum of 17 states in the remap case. In order to make these values comparable to each other, the k-equivalence potential value are divided by $|Q-2|$, for each automaton. As discussed in Chapter 7, the $|Q-2|$ value represents the value of k used to construct a minimum automaton. This give 31 normalized k-equivalence potential values between 0 and 1, where 1 represents the case where an exact automaton is constructed.

Table 9.2 represents the minimum, lower quartile, median, upper quartile and maximum [12] of the normalized values. The h_Σ hash function had by far the best performance: it had a median normalized k-equivalence potential value of 1. Therefore, at least half the automata constructed were exact automata.

The bit string operator based hash functions perform badly for long regular expressions: no exact automata are constructed, and the maximum score never exceeds 0.55 or 0.71 of the $|Q-2|$ value for each automaton tested, for the DFA and NFA cases respectively. The medial values is always between 0.2 and 0.3, regardless of whether a NFA or DFA is constructed.

The choice of commutative operators may be the main influencing factor: when the regular expression union is taken repeatedly, the substitution on the bitwise *or* operator results in values getting "stuck" on 1 when a 1 is generated by the hash function. This is also true for the bitwise *and* operator and 0.

In order to improve these results, operator mappings to the integer arithmetic operators may be considered. The problem with this alternative approach is idempotence: integer addition is not idempotent, where bitwise *or* is. This will cause regular expressions that are equal to be hashed to different hash codes if these operators are mapped to the regular expression *union* operator.

	DFA Version					NFA Version				
function	min	lower quart.	median	upper quart.	max	min	lower quart.	median	upper quart.	max
h_Σ	0.52	0.94	1.00	1.00	1.00	0.55	0.92	1.00	1.00	1.00
1	0.10	0.20	0.29	0.44	0.71	0.09	0.16	0.24	0.32	0.55
2	0.11	0.17	0.28	0.35	0.71	0.11	0.16	0.23	0.31	0.52
3	0.10	0.20	0.29	0.44	0.71	0.09	0.16	0.24	0.32	0.55
4	0.11	0.17	0.28	0.35	0.71	0.11	0.16	0.23	0.31	0.52
5	0.11	0.21	0.29	0.39	0.71	0.11	0.17	0.24	0.34	0.55
6	0.11	0.17	0.26	0.35	0.71	0.11	0.16	0.23	0.31	0.52
7	0.11	0.21	0.29	0.39	0.71	0.11	0.17	0.24	0.34	0.55
8	0.11	0.17	0.26	0.35	0.71	0.11	0.16	0.23	0.31	0.52

Table 9.2: Average k-equivalence for long regular expressions

9.3 Summary

In this chapter, the results are presented that were generated in the implementation that is described in the previous chapter. Statistics were taken for short and long regular expressions. The short regular expressions yielded the same results in the NFA and the DFA case of the hashing version of Brzozowski's algorithm. Exact automata were constructed for the short regular expressions, but not for the long regular expressions in the case of the bit string operator based hash functions. In order to compare automata for the long regular expressions, *average k-equivalence* was defined. Where the short regular expressions had the same results in the DFA and the NFA case, the long regular expressions had different results. The h_Σ function outperformed the bit string operator hash functions.

Chapter 10

Conclusion and future directions

In this chapter the results obtained in this dissertation are summarised, and future directions that may yield interesting results are discussed.

10.1 Results

The following new results are presented in this dissertation:

- A definition of a super-, exact and sub-automaton is put forward.
- A method for measuring the difference between an exact and a super-automaton is put forward. This is the k-equivalence measure.
- A new algorithm is presented that is based on the hashing version of Brzozowski's algorithm that first appeared in [23]. Where the original algorithm constructed a NFA, this algorithm constructs a DFA.
- A proof is put forward that, following the hashing approach, an exact or super-automaton is always constructed.
- Hash functions based on bit string operator mappings have been proposed.
- A hash function that guarantees the construction of an automaton with states that are at least 1-equivalent was proposed.

In terms of statistical results, the following conclusions are drawn:

- The bit string operator mapping based hash functions constructed a large number of small exact automata for the input 272 850 short test regular expressions. Between 43% and 71% of these regular expressions resulted in exact automata being constructed.

- The hash function guaranteeing at least 1 equivalence out-performed the bit string operator based hash functions. 99% of the automata constructed were exact automata.
- In terms the long regular expressions, the bit string operator based hash functions did not perform well: no exact automata were constructed and the maximum median for the k-equivalence potential value was 0.29 for the DFA case, and 0.24 in the NFA case.
- The hash function guaranteeing at least 1-equivalence performs very well for the long regular expressions. The h_Σ hash function produced exact automata in at least 50% of the test cases.
- It is difficult to draw meaningful statistical conclusions from the long regular expressions, because it is not clear how to select a representative sample. Another obstacle is performance limitations in terms of the k-equivalence measure. Despite these drawbacks, experimenting with longer regular expressions has shown that hash function performance can be differentiated using the normalised k-equivalence potential measure. It has also confirmed h_Σ as an effective approach to hashing.

10.2 Future directions

The following areas can be further researched and may yield results:

- A theory should be created that explains why the h_Σ function out-performed the bit string operator based hash functions.
- The performance of the algorithms presented in this dissertation can be compared and analyzed.
- The performance of the hash functions should be measured for a wide range of alphabet sizes.
- The prefix notation encoding of regular expressions resulting from Gödel numbers caused a large number of numbers to be wasted: iterating through 50 000 000 numbers only yielded 272 850 useful regular expressions. A more efficient encoding should be found that generates 50 000 000 useful regular expressions from 50 000 000 numbers. If such an encoding was to be found, it will also simplify the selection of a random large regular expression: firstly one would select a large random number, and then turn it into a regular expression in the encoding.

- Find a process that takes the hash space size n (i.e. number of states) and then produces the best possible hash function.
- Experiment with hash function operator substitutions for the regular expressions other than the bit string operators.
- Create a theory that explains the performance of hash functions based on axiomatic semantics. The axiomatization of the regular languages found in [17] and Peano's axioms of the integers found in [18] and in [8] can form a basis of such a theory.
- The ideal hash function can be sought, at first ignoring the effects of the application of the modulo function in hashing, or the limitations imposed by the bit string representation. Term rewriting systems [20] may present a viable avenue of exploration, starting from the axiomatization presented in [17]. The rewritten regular expression can be converted into a Gödel number that will serve as a hash code.

Bibliography

[1] Brzozowski J.A. Derivatives of Regular Expressions. *Journal of the Association of Computing Machinery*, 11:481–494, October 1964.

[2] Câmpeanu C., Sântean N. and Yu S. Minimal Cover-Automata for Finite Languages. *Theoretical Computer Science*, 267:3–16, 2001.

[3] Cleophas L., Zwaan G. and Watson B.W. Constructing factor oracles. *Journal of Automata, Languages and Combinatorics*, 10(5/6):627–640, 2005.

[4] Coetser W., Kourie D.G. and Watson B.W. On Regular Expression Hashing to Reduce FA Size. In *Proceedings of the Prague Stringology Workshop 2008*, pages 227–241, Czech Technical University in Prague, Czech Republic, 2008.

[5] Cyril A., Crochemore M. and Mathieu R. Factor oracle: A new structure for pattern matching. In *SOFSEM '99: Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*, pages 295–310, London, UK, 1999. Springer-Verlag.

[6] Epp S.S. *Discrete Mathematics with Applications*, pages 576–583. International Tomson Publishing, Inc., 1995.

[7] Frishert M. FIRE Works & FIRE Station, A Finite Automata & Regular Expression Playground. Master's thesis, Technical University Eindhoven, 2004.

[8] Bergstra J.A. *Algebraic Specification*. ACM, 1989.

[9] Lewis H.R. and Papadimitriou C.H. *Elements of the Theory of Computation*. Prentice-Hall, Inc., 1981.

[10] Linz P. *An Introduction to Formal Language and Automata*. Jones and Bartlett Publishers, Inc., 2006.

[11] McNaughton R. *Elementary Computability, Formal Languages and Automata*. Prentice Hall, 1981.

[12] Mendenhall W, Beaver R.J. and Beaver B.M. *Introduction to Probability and Statistics*. Duxbary, 2006.

[13] Michael S. *Introduction to the Theory of Computation*. Tompson Learning Inc., 2006.

[14] Můzátco P. Approximate Regular Expression Matching. In *Proceedings of the Prague Stringology Workshop '96*, pages 39–41, Czech Technical University in Prague, Czech Republic, 1996.

[15] *Private communication*.

[16] Grimaldi R.P. *Discrete and Combinatorial Mathematics; An Applied Introduction*. Addison-Wesley Longman Publishing Co., Inc., 1985.

[17] Salomaa A. Two Complete Axiom Systems for the Algebra of Regular Events. *Journal of the Association for Computing Machinery*, 13:158–169, January 1966.

[18] Scheurer T. *Foundations of Computing. System Development with Set Theory and Logic*. Addison-Wesley Publishing Company, 1994.

[19] Strauss T., Kourie D.G., and Watson B.W. A Concurrent Specification of Brzozowski's DFA Construction Algorithm. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference '06*, pages 90–99, Czech Technical University in Prague, Czech Republic, 2006.

[20] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[21] Watson B.W. A Taxonomy of Finite Automata Minimization Algorithms. Technical report, Technical University Eindhoven, 1994.

[22] Watson B.W. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, 1995.

[23] Watson B.W., Kourie D.G., Ngassam E.K., Strauss T. and Cleophas L. Efficient Automata Constructions and Approximate Automata. *International Journal of Foundations of Computer Science*, Vol. 19(1):185–193, 2008.

Appendix A

Code overview

This dissertation comes with a compact disc containing the implementation used to generate the statistics and test the ideas that are presented. In order to make it possible for future researchers to validate, and potentially falsify the findings in this dissertation, a code overview documenting design decisions in the implementation is presented in this appendix. It may also happen in the future that the compact disk format becomes unavailable and outdated, and that this will be the only record of this research. Placing some code in the dissertation also makes it database searchable.

The C# .NET version 3.5 programming language is used in the implementation.

A.1 High level design overview

The general design is presented in the class diagram depicted in Figure A.1. The most important artefact in the system is the generic $IRegex\langle T\rangle$ interface, which presents a contract to the rest of the regular expression classes in the system. In the implementation each regular expression operator from Section 3.3 is represented as a class.

Note that some *data type* and *visibility* information with regards to the class members have been omitted. This is done to make the diagram more readable.

The $IRegex\langle T\rangle$ interface is generic, allowing an automaton constructed from the regular expression to parse a sequence of data of any type. In the .NET framework, all types inherit from the *Object* class. Therefore these types can overload the equality methods, making objects of these types distinguishable, as would be the case with characters in a character stream.

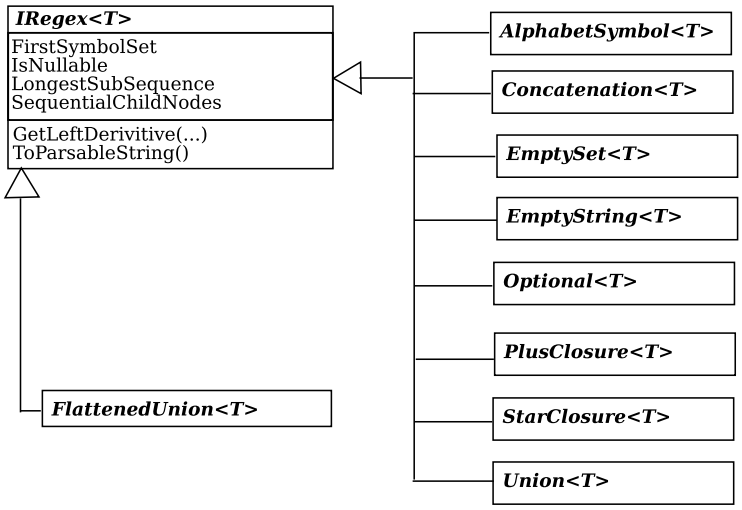

Figure A.1: Class diagram of the regular expression classes

A.2 Rewriting regular expressions

As pointed out in Section 4.3 and 4.6, it is necessary to take idempotence, associativity and commutativity of regular expression operators into account during the application of Brzozowski's algorithm to a regular expression. For this purpose, the expression trees representing regular expressions must be flattened for the union operator, which is implemented by the $Union\langle T\rangle$ class. This can be seen in the following algorithm:

```
private static IRegex<T> GetFlattened<T>(this IRegex<T> regexIn)
{
  if (regexIn is Union<T> || regexIn is Intersection<T>
    || regexIn is FlattenedUnion<T> || regexIn is FlattenedIntersection<T>)
  {
    Stack<IRegex<T>> evalStack = new Stack<IRegex<T>>();
    evalStack.Push(regexIn);
    HashSet<IRegex<T>> flattenedChildren = new HashSet<IRegex<T>>();
    while (evalStack.Count > 0)
    {
      foreach (var childNode in evalStack.Pop().SequentialChildNodes)
        if (childNode is AlphabetSymbol<T>) flattenedChildren.Add(childNode);
        else if (OperatorsIsFlattenCompatible(regexIn,childNode))
         evalStack.Push(childNode);
        else flattenedChildren.Add(childNode);
    }
    if (regexIn is Union<T>) return new FlattenedUnion<T>()
     { ChildNodes = flattenedChildren };
```

```
    else if (regexIn is Intersection<T>) return new FlattenedIntersection<T>()
     { ChildNodes = flattenedChildren };
    else if (regexIn is FlattenedIntersection<T>) return new FlattenedIntersection<T>()
     { ChildNodes = flattenedChildren };
    else if (regexIn is FlattenedUnion<T>) return new FlattenedUnion<T>()
     { ChildNodes = flattenedChildren };
    else throw new NotImplementedException();
  }
  else return regexIn;
}
```

In the *GetFlattened* function, all $Union\langle T\rangle$ nodes in the expression tree are converted to $FlattenedUnion\langle T\rangle$ nodes.

A.3 Equality and hashing

The *GetFlattened* function takes *associativity* into account by placing all union child nodes in the expression tree on the same level. This leaves *idempotence* and *commutativity*.

In order to take *idempotence* into account, the child nodes of the flattened union operator is stored in a $HashSet\langle T\rangle$ member:

```
public HashSet<IRegex<T>> ChildNodes { get; set; }
```

The $HashSet\langle T\rangle$ class represents a set as a hash table, by hashing the set members as keys, and storing the set members as values. This allows for fast insertion and lookup of set members. All set members are unique: this takes *idempotence* into account.

In order to take *commutativity* into account, the *Equals* and *GetHashCode* members inherited from the *Object* class are overridden in the $FlattenedUnion\langle T\rangle$ class:

```
public override bool Equals(object obj)
{
  FlattenedUnion<T> other = obj as FlattenedUnion<T>;
  if (other != null)
  {
    HashSet<IRegex<T>> compareSet = new HashSet<IRegex<T>>();
    foreach (var member in this.ChildNodes.Concat(other.ChildNodes))
      compareSet.Add(member);
    return this.ChildNodes.Count == compareSet.Count
```

```
    && compareSet.Count == other.ChildNodes.Count;
  }
  else return false;
}
```

It is important to maintain the relation between hashing and equality when overriding these members:

- Equal objects must have equal hash codes.
- It is *not* the case that equal hash codes always imply equal objects.

For this reason, the hash code for $FlattendUnion\langle T\rangle$ respects equality, in terms of commutativity, idempotense and associativity and is given as:

```
public override int GetHashCode()
{
  long code = 0;
  foreach (var node in ChildNodes) code += node.GetHashCode();
  return Convert.ToInt32(code % ChildNodes.Count)
   ^ this.GetType().Name.GetHashCode();
}
```

Note that it is only necessary to implement equality with associativity, commutativity and idempotece in order to avoid non-termination in the Brzozowski's algorithm, with respect to the *done* and *todo* sets in Algorithm 4.1, 5.1 and 6.1. Therefore, equality was not implemented in terms of *-equivalence classes on regular expressions.

A.4 Rewrite rules for the empty set and the empty string

In order for Brzozowski's algorithm to terminate (see for example Section 4.6), input regular expressions must be rewritten with respect to the *empty string* and the *empty set*. This is done with the *GetRewritten* function.

```
private static IRegex<T> GetRewritten<T>(this IRegex<T> regexIn)
{
  // Rewrite this
  IRegex<T> retVal = null;
  if (regexIn is Union<T>)
```

```
  {
    Union<T> current = regexIn as Union<T>;
    if (current.LHS is EmptySet<T>) retVal = current.RHS;
    else if (current.RHS is EmptySet<T>) retVal = current.LHS;
    else retVal = regexIn;
  }
  else if (regexIn is FlattenedUnion<T>)
  {
    FlattenedUnion<T> current = regexIn as FlattenedUnion<T>;
    // Get everything accept the empty set: if nothing: get the empty set only
    HashSet<IRegex<T>> newChildNodes = new HashSet<IRegex<T>>();
    foreach (IRegex<T> childNode in current.ChildNodes)
      if (!(childNode is EmptySet<T>)) newChildNodes.Add(childNode);
    if (newChildNodes.Count == 0) retVal = new EmptySet<T>();
    else if (newChildNodes.Count == 1) retVal = newChildNodes.First();
    else retVal = new FlattenedUnion<T>() { ChildNodes = newChildNodes };
  }
  else if (regexIn is Concatenation<T>)
  {
    Concatenation<T> current = regexIn as Concatenation<T>;
    if (current.Head is EmptyString<T>) retVal = current.Tail;
    else if (current.Tail is EmptyString<T>) retVal = current.Head;
    else if (current.Head is EmptySet<T>) retVal = new EmptySet<T>();
    else if (current.Tail is EmptySet<T>) retVal = new EmptySet<T>();
    else retVal = regexIn;
  }
  else retVal = regexIn;
  return retVal;
}
```

A.5 Brzozowski's algorithm with remapping

The original remapping version of Brzozowski's algorithm is given as (See also Algorithm 4.1):

```
public static FiniteStateAutomaton<IRegex<T>, T>
GetExactAutomaton<T>(this IRegex<T> regexIn)
{
  IEqualityComparer<IRegex<T>> stateComparer = new RemapEqualityComparer<T>();
  HashSet<IRegex<T>> todo = new HashSet<IRegex<T>>(stateComparer) { regexIn };
  HashSet<IRegex<T>> done = new HashSet<IRegex<T>>(stateComparer);
  FiniteStateAutomaton<IRegex<T>, T> dfa =
   new FiniteStateAutomaton<IRegex<T>, T>(stateComparer,EqualityComparer<T>.Default);
  dfa.StartStates.Add(regexIn);
```

```
  while (todo.Count > 0)
  {
    IRegex<T> current = todo.First();
    todo.Remove(current);
    done.Add(current);
    foreach (var firstSymbol in current.FirstSymbolSet)
    {
      IRegex<T> derivative = current.GetLeftDerivative(firstSymbol);
      if (!todo.Contains(derivative) && !done.Contains(derivative))
      {
        todo.Add(derivative);
      }
      dfa.AddTransition(current,firstSymbol,derivative);
    }
    if (current.IsNullable) dfa.EndStates.Add(current);
  }
  return dfa;
}
```

A question that may be asked when looking at the implementation of Algorithm 4.1 is: where is the *remap* variable? Remapping is implicit in the use of the $RemapEqualityComparer\langle T\rangle$ equality comparer. It utilizes the overloaded $Equals$ and $GetHashCode$ methods on the implementations of the $IRegex\langle T\rangle$ interface.

A.6 Brzozowski's algorithm with hashing

The original hashing version of Brzozowski's algorithm (see also Algorithm 5.1) is implemented as:

```
public static FiniteStateAutomaton<IRegex<T>, T>
GetNonDeterministicSuperAutomaton<T>(this IRegex<T> regex,
IEqualityComparer<IRegex<T>> hashFunction)
{
  IRegex<T> regexIn = regex.GetRewriteFlattened();
  HashSet<IRegex<T>> todo = new HashSet<IRegex<T>>(new RemapEqualityComparer<T>())
   { regexIn };
  HashSet<IRegex<T>> done = new HashSet<IRegex<T>>(new RemapEqualityComparer<T>());
  FiniteStateAutomaton<IRegex<T>, T> nfa =
   new FiniteStateAutomaton<IRegex<T>, T>(hashFunction, EqualityComparer<T>.Default);
  nfa.StartStates.Add(regexIn);
  while (todo.Count > 0)
  {
    IRegex<T> current = todo.First();
    todo.Remove(current);
    done.Add(current);
```

```
    foreach (var firstSymbol in current.FirstSymbolSet)
    {
      IRegex<T> derivative = current.GetLeftDerivative(firstSymbol);
      if (!todo.Contains(derivative) && !done.Contains(derivative))
      {
        todo.Add(derivative);
      }
      nfa.AddTransition(current, firstSymbol, derivative);
    }
    if (current.IsNullable) nfa.EndStates.Add(current);
  }
  return nfa;
}
```

The hash function being used was passed in as an implementation of the $IEqualityComparer\langle IRegex\langle T\rangle\rangle$ interface.

A.7 Brzozowski's algorithm with hashing, the DFA version

The new deterministic version of Brzozowski's algorithm with hashing proposed in this dissertation is implemented as (see also Algorithm 6.1):

```
public static FiniteStateAutomaton<IRegex<T>, T>
GetDeterministicSuperAutomaton<T>(this IRegex<T> regex,
IEqualityComparer<IRegex<T>> hashFunction)
{
  IRegex<T> regexIn = regex.GetRewriteFlattened();
  Dictionary<int, IRegex<T>> inverseHashMappings = new Dictionary<int, IRegex<T>>();
  HashSet<IRegex<T>> todo = new HashSet<IRegex<T>>(new RemapEqualityComparer<T>())
   { regexIn };
  HashSet<IRegex<T>> done = new HashSet<IRegex<T>>(new RemapEqualityComparer<T>());
  FiniteStateAutomaton<IRegex<T>, T> dfa =
   new FiniteStateAutomaton<IRegex<T>, T>(hashFunction, EqualityComparer<T>.Default);
  dfa.StartStates.Add(regexIn);
  while (todo.Count > 0)
  {
    IRegex<T> current = todo.First();
    todo.Remove(current);
    done.Add(current);
    foreach (var firstSymbol in current.FirstSymbolSet)
    {
      IRegex<T> derivative = current.GetLeftDerivative(firstSymbol);
      IEnumerable<IRegex<T>> nextStates = dfa.GetNextStates(current, firstSymbol);
```

```
      if (nextStates.Count() == 1)
      {
        IRegex<T> existingNextState = nextStates.First();
        dfa.RemoveTransition(current,firstSymbol,derivative);
        derivative = new Union<T>()
         { LHS = derivative, RHS = existingNextState}.GetRewriteFlattened();
      }
      if (!todo.Contains(derivative) && !done.Contains(derivative))
      {
        todo.Add(derivative);
      }
      dfa.AddTransition(current, firstSymbol, derivative);
    }
    if (current.IsNullable) dfa.EndStates.Add(current);
  }
  return dfa;
}
```

The main point of interest in this code is the first *if*-statement in the *foreach*-loop:

```
IRegex<T> derivative = current.GetLeftDerivative(firstSymbol);
IEnumerable<IRegex<T>> nextStates = dfa.GetNextStates(current, firstSymbol);
if (nextStates.Count() == 1)
{
  IRegex<T> existingNextState = nextStates.First();
  dfa.RemoveTransition(current,firstSymbol,derivative);
  derivative = new Union<T>()
   { LHS = derivative, RHS = existingNextState}.GetRewriteFlattened();
}
```

Note that the destination state, which is in the *derivative* variable, is rewritten to prevent non-determinism in the resulting automaton. It is also important that this new constructed next state (which is also a regular expression) is flattened and rewritten in order to avoid non-termination.

Appendix B

Long regular expression test cases

The test sample of long regular expressions are small: it consist of 31 test expressions. The sample is kept small due to the limitations discussed in Section 8.3. Operators are given in Table B.1.

Operator	Symbol
Concatenation	.
Union	|
Plus Closure	+
Star Closure	*
Optional	?

Table B.1: Prefix notation operators

The empty set and the empty string have been removed from the test cases using the rewrite rules from Section 4.3. The long regular expressions are written in prefix notation. Therefore, taking Table B.1 into account, the prefix string

```
|0*1
```

represents the regular expression $0 \cup 1^*$. The long regular expression test cases are:

```
.*.+0.00|+*+.0+0+++.1+1
|.**+.+33+.0.0?0*..?000
|+.3.3?3|+.1.+11+*+.0+0
||***.+33***.0+0+..11+1
|***.0+0.+.2.+22+++.1+1
|*..?222.*..?222*.?2.22
.+.0.0?0|*..2+22+.3.3+3
|+.+1.11.*..+333**+.3+3
```

```
..+.|0+11+.1.1?1*..0+00
|.*+*+.+00*..0+00*.?0.00
.*.?0.00|*.3.3?3*.3.3?3
..+..?000*+*.0+0+..?222
|*..+000|+*+.+11+..33?3
|*.2.2+2.*.0.?00*..33+3
|+..00?0|++*.+22+.2.2+2
|+.|1+33.+..+000+.?1.11
.+.|+323|*..2?22*++.+11
..*.0.+00++*.2+2*+*.3+3
||***.0+0*+*.+00*..00+0
.***.+11|+.3.?33*.1.1+1
|*.0.0?0.*..?222*.1.1?1
|+.|+131*.+..+000*..?000
.*..11+1.***.+33*+*.3+3
.*..3?33.***.0+0+++.3+3
|.+.|0+22**.?1.11***.+33
.|*++.+00+.3.3?3+..+333
||+.|+020+.|0+33+.2.2?2
..*..?111+..?111*..+111
||+++.2+2+.2.2?2+.|1+22
..*.0.0+0+..+111*+*.1+1
..+++.1+1+*+.2+2*..?000
```