BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AN EFFICIENT FUNCTIONAL LIBRARY FOR FINITE AUTOMATA

EFEKTIVNÍ FUNKCIONÁLNÍ KNIHOVNA PRO KONEČNÉ AUTOMATY

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JAKUB ŘÍHA**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDŘEJ LENGÁL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Zadání diplomové práce/19561/2016/xrihaj03

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Říha Jakub, Bc.**
Obor: Inteligentní systémy
Téma: **Efektivní funkcionální knihovna pro konečné automaty**
An Efficient Functional Library for Finite Automata
Kategorie: Formální verifikace

Pokyny:

1. Nastudujte programovací jazyk Haskell a problematiku efektivního funkcionálního programování.
2. Nastudujte problematiku konečných automatů nad konečnými slovy a stromy.
3. Vytvořte knihovnu pro praci s konečnými automaty v programovacím jazyce Haskell. Zaměřte se na efektivní implementaci operací sjednocení, průniku, doplňku, a problémů členství, prázdnosti, inkluze a univerzality. Kladťe důraz na efektivní využití lazy evaluace. Ke knihovně vytvořte textové rozhraní.
4. Výkon implementované knihovny srovnejte s jinými knihovnami, např. s knihovnou VATA.
5. Diskutujte další možnosti rozšíření.

Literatura:

- Miran Lipovača. Learn You a Haskell for Great Good! April 2011, 400 pp. ISBN: 978-1-59327-283-8
- Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In Proc. of TACAS'12. Springer
- P.A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr and T. Vojnar. **When Simulation Meets Antichains**. In *Tools and Algorithms for the Construction and Analysis of Systems*. *Lecture Notes in Computer Science*, vol. 6015, pp. 158-174. Springer Berlin Heidelberg.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Lengál Ondřej, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstract

Finite automata are an important mathematical abstraction, and in formal verification, they are used for a concise representation of regular languages. Operations often used on finite automata in this setting are testing their universality and language inclusion. A naive approach to implement these operations leads to an explicit determinization of the automata, which can be costly and undesirable. There is, however, a more advanced method for performing those operations, called the Antichains algorithm, which avoids such an explicit determinization. This work shows how finite automata operations can be effectively implemented in Haskell and compares several approaches of their implementation. The obtained results are compared with VATA, an imperative implementation of a finite automata library.

Abstrakt

Konečné automaty jsou důležitou matematickou abstrakcí. Ve formální verifikaci se konečné automaty používají ke stručné reprezentaci regulárních jazyků. V této souvislosti se používají operace nad konečnými automaty, jako je testování jazykové univerzality a inkluze. Naivní přístup k implementaci těchto operací vede k explicitní determinizaci konečného automatu, což může být nakladné a nežádoucí. Nicméně existuje pokročilejší metoda k vykonávání těchto operací nazývaná Antichains algoritmus, která se vyhýbá explicitní determinizaci. Tato práce se zabývá efektivní implementací operací nad konečnými automaty v Haskellu a také porovnává několik implementačních variant. Získané výsledky jsou poté porovnány s knihovnou VATA, což je imperativní implementace knihovny pro práci nad konečnými automaty.

Keywords

finite automata, antichain, library, functional language, Haskell, lazy evaluation.

Klíčová slova

konečné automaty, antichain, knihovna, funkcionální jazyk, Haskell, lazy evaluace.

Reference

ŘÍHA, Jakub. *An Efficient Functional Library for Finite Automata.* Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Lengál Ondřej.

An Efficient Functional Library for Finite Automata

Declaration

Hereby I declare that this Master's thesis was prepared as an original author's work under the supervision of Mr. Ondřej Lengál. All the relevant information sources that were used during preparation of this Master's thesis are properly cited and included in the list of references.

.....
Jakub Říha
May 23, 2017

Acknowledgements

I would like to thank my Master's thesis supervisor, Mr. Ondřej Lengál, for his professional help and support.

Contents

Chapter 1

Introduction

Finite automata are extensively used in various branches of computer science ranging from software engineering and compilers to hardware digital systems. This work focuses specifically on an implementation of a representation of finite word automata and operations on them used in model checking of computer programs. Such automata may be used to represent data structures of an unbounded size and their operations to check properties of such structures in an efficient way.

There are various finite automata libraries, which differ in their intended use, performance, and support of various operations. Our work deals with operations related to model checking. More specifically, these operations are union, intersection, determinization, complement, and decision problems of membership, emptiness, language inclusion, and universality. Even though there are specialized finite automata libraries focusing on model checking, our library will differ in that it will use an efficient lazy evaluation implementation. Lazy evaluation is an evaluation strategy that delays the evaluation of an expression until its value is needed and that also avoids repeated evaluations (known as sharing). Such an evaluation strategy could potentially lead to a more efficient implementation of finite automata operations because it allows to skip unnecessary computations and thus improve the performance. Our library, implemented as a part of this Master's thesis, uses Haskell, a well-known purely functional programming language, which is one of the few programming languages with non-strict semantics and lazy evaluation.

This thesis is structured into seven chapters. After this introduction, the following Chapter 2 is concerned with finite word automata. First, it describes theoretical foundations of formal languages and then the definition of a finite automaton and other relevant information is presented. Most of the chapter contains description of the operations that are to be implemented in our library. A pseudo-code for each operation is provided. Apart from the above mentioned operations, the chapter also elaborates on an antichain-based method for deciding universality and language inclusion and on a transformation of a non-deterministic finite automaton to a deterministic finite automaton using the subset construction. Finally, the chapter discusses other finite word automata libraries.

Chapter 3 discusses the Haskell programming language. First, it contains general information about the language. Then it discusses the non-strict semantics, lazy evaluation, and how to suppress these to avoid performance penalty in certain circumstances. A description of available tools that can be used to analyze and profile the performance of Haskell programs follows. The last part of the chapter is concerned with monads. It explains the purpose of monads and then further elaborates on types of monads that are used in our library. More specifically, these types of monads are used to implement computations

that may fail, computations that work with a global state, and parsing. For this purpose, we use Parsec, which is a library implementation of a parsing monad in Haskell.

Chapter 4 analyses and discusses the design of our finite automata library. It closely follows from the two previous theoretical chapters and from the information contained therein. Data structures, parsing of the input format, and finite automata operations are discussed there.

The next Chapter 5 is focused on the implementation of our library. The chapter first briefly describes the library structure and the toolset used to implement this library. Then, it discusses data structures and their implementation along with the rationale on why we decided for a such specific solution. The next section is concerted with both textual and graphical visualization of an arbitrary finite automaton using our library. The following section describes all variants of finite automata operations that were implemented in the library. The chapter finishes with a brief description about unit testing.

Chapter 6 deals with benchmarking of the implemented finite automata operations. We performed a set of benchmarks to compare performance of our solution to VATA, which is a highly optimized library for non-deterministic finite word and tree automata. The chapter contains benchmarking results, which compare both libraries as well as two versions of data structures we implemented.

The last Chapter 7 concludes this text and gives directions for future work beyond this Master's thesis. This thesis follows from the Term project we created at the final year. More specifically, Chapter 2 and 3 of this Master's thesis were taken from the Term project with small modifications and extensions.

Chapter 2

Finite automata

This chapter introduces finite automata, a mathematical abstraction used in the field of formal verification. It describes theoretical foundations of formal languages and then the definition of a finite automaton and other relevant information, which were taken from Meduna [17]. The second part of the chapter deals with finite automata operations. The description of those was taken from Meduna [17] and the supervisor of this thesis, Ondřej Lengál. Finally, the chapter mentions other existing finite automata libraries.

2.1 Formal languages

A *formal language* is a set of strings of symbols. Formal languages are used extensively in computer science, and, more specifically, in language theory and in formal theories, systems, and proofs.

Alphabets and strings

An *alphabet* is a finite nonempty set of elements that are called *symbols*.

A sequence of symbols forms a *string* (also called a *word*). A string that contains no symbols is called the *empty string*, and is denoted by ε. We can then recursively define strings over an alphabet Σ:

1. ε is a word over Σ.
2. If x is a word over Σ and $a \in \Sigma$, then xa is a word over Σ.

Let x and y be two strings over an alphabet Σ. Then, xy is the *concatenation* of x and y. It holds for every string x that $x\varepsilon = \varepsilon x = x$.

Formal languages

Given an alphabet Σ, let Σ^* denote the set of all strings over Σ. Let Σ^+, defined as $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$, denote the set of all nonempty strings over Σ. The following definition formalizes a language over Σ as a set of strings over Σ. Let Σ be an alphabet and let $L \subseteq \Sigma^*$. Then L is a *language* over Σ.

Observe that for every alphabet Σ, the set Σ^* represents a language over Σ containing all words over Σ. Such a language is called the *universal language* over Σ. Because languages are defined as sets, the operations and notions regarding sets also apply to them.

A language L is called a *finite language* if it has n members, for some $n \in \mathbb{N}_0$; otherwise, L is an *infinite language*.

Consider a language L over an alphabet Σ. The *complement of* L, denoted as $\overline{L}$, is defined as $\overline{L} = \Sigma^* \setminus L$.

2.2 Finite automata

A *finite automaton* (FA) is a 5-tuple:

$$M = (Q, \Sigma, \delta, s, F), \tag{2.1}$$

where

- Q is a finite set of *states*,
- Σ is the *input alphabet*,
- $\delta \colon Q \times \Sigma \to 2^Q$ is the *transition function*,
- $s \in Q$ is the *initial state*, and
- $F \subseteq Q$ is the set of *final states*.

For convenience, we use $p \xrightarrow{a} q$ to denote that $q \in \delta(p, a)$. The workings of a finite automaton are determined by a sequence of transitions. These transitions are given by the transition function δ. Given a current state $q \in Q$ and an input symbol $a \in \Sigma$, the transition function outputs a new set of states $R = \delta(q, a) \subseteq Q$.

For a finite automaton M, we define a *configuration* as $C = (q, w)$ from $Q \times \Sigma^*$. Given a word w on the input, the configuration (s, w) is called the *initial configuration* and the configuration (q, ε) for $q \in F$ is called a *final configuration*.

A *transition* of an automaton M is a binary relation on C defined as $(q, aw) \vdash_M (q', w)$ iff $q' \in \delta(q, a)$. Sometimes, we omit the M subscript from the $\vdash$ symbol when it is clear which automaton we are referring to. Symbol $\vdash_M^+$ is the transitive closure of $\vdash_M$ and $\vdash_M^*$ is its transitive and reflexive closure. A *path* from a state q to a state q' of finite automaton M is a sequence of transitions $(q, w) \vdash_M^* (q', w')$, where $w, w' \in \Sigma^*$. An *accepting path* is a path from a state q to a state f, such that $f \in F$.

For a set of states $S \in Q$, we define $post_a(S) = \bigcup_{s \in S} \{t \mid s \xrightarrow{a} t \in \delta\}$, which gives a set of states reachable from a set of states S using transitions that read the symbol a. We define $post(S) = \bigcup_{a \in \Sigma} post_a(S)$, which is a generalized version of the former as it reads an arbitrary symbol $a \in \Sigma$.

In the rest of this work, we will generalize the definition of a finite automaton to support multiple initial states so the automaton will be a 5-tuple $M = (Q, \Sigma, \delta, I, F)$ with the same meaning as in Equation 2.1 except for the I term, which is the set of initial states $I \subseteq Q$.

A string w is *accepted* by a finite automaton M iff $(s, w) \vdash^* (f, \varepsilon)$ for some $s \in I$ and $f \in F$. Then, we can define the set of strings accepted by the finite automaton M as $L(M)$. Formally,

$$L(M) = \{w \mid \exists s \in I, f \in F : (s, w) \vdash^* (f, \varepsilon)\}.$$

We will denote the language of a state $q \in Q$ of FA M as

$$L(M)(q) = \{w \mid \exists f \in F : (q, w) \vdash^* (f, \varepsilon)\}.$$

Types of finite automata

If the transition function of an automaton M is of the form $Q \times \Sigma \to Q$, i.e. $|\delta(q, a)| \leq 1$ for every $q \in Q$ and $a \in \Sigma$, then M is called a *deterministic finite automaton* (DFA); otherwise, M is called a *non-deterministic finite automaton* (NFA). The term deterministic refers to the fact that for each input string it accepts or rejects the input string using a unique run of the automaton.

A finite automaton $M = (Q, \Sigma, \delta, I, F)$ is called a *complete* FA if it cannot get stuck, that is, if for any $q \in Q$ and $a \in \Sigma$ there exists at least one transition rule $q \xrightarrow{a} r \in \delta$ for some $r \in Q$; otherwise, M is *incomplete*.

A state $q \in Q$ is *accessible* if it can be reached by following transitions from the initial state, i.e. $\exists s \in I : (s, w) \vdash^* (q, w')$ for some $w, w' \in \Sigma^*$. A state $q \in Q$ is *terminating* if it can reach a final state. Formally, a state $q \in \Sigma$ is terminating if there exists $w, w' \in \Sigma^*$, such that $(q, w) \vdash^* (f, w')$ where $f \in F$; otherwise, q is *non-terminating*.

A complete DFA $M = (Q, \Sigma, \delta, I, F)$ is called a *well-specified* FA (WSFA) iff 1) M has no inaccessible state, and 2) M has at most one non-terminating state.

2.3 The membership decision problem

Deciding whether given some $w \in \Sigma^*$ it holds that $w \in L(M)$ is called the *membership problem*. Algorithm 1 decides membership problem for a non-deterministic finite automaton M.

Algorithm 1 Deciding membership for a non-deterministic finite automaton

Input: A non-deterministic finite automaton $M = (Q, \Sigma, \delta, I, F)$ and a string $w \in \Sigma^*$.
Output: `true` if $w \in L(M)$; otherwise, `false`.

1: $Q_{current} \leftarrow I$
2: **while** $w \neq \varepsilon$ **do**
3: Let $w = aw'$ for $a \in \Sigma$ and $w' \in \Sigma^*$
4: $Q_{current} \leftarrow post_a(Q_{current})$
5: $w \leftarrow w'$
6: **end while**
7: **return** $Q_{current} \cap F \neq \emptyset$

2.4 Union of two languages

For two languages accepted by finite automata M_0 and M_1 over the same alphabet, we can easily construct a finite automaton M that accepts the union of the two languages $L(M_0) \cup L(M_1)$ as presented in Algorithm 2.

Algorithm 2 Construction of a finite automaton for union

Input: Two finite automata $M_0 = (Q_0, \Sigma, \delta_0, I_0, F_0)$ and $M_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$, such that $Q_0 \cap Q_1 = \emptyset$.
Output: A finite automaton $M = (Q, \Sigma, \delta, I, F)$, such that $L(M) = L(M_0) \cup L(M_1)$.

1: **return** $M = (Q_0 \cup Q_1, \Sigma, \delta_0 \cup \delta_1, I_0 \cup I_1, F_0 \cup F_1)$

2.5 Product union of two languages

The previous operation showed how to construct a finite automaton that accepts a union of two languages. The operation constructs a new set of states $Q_0 \cup Q_1$. If we implement this operation in some strongly typed programming language the set of states Q_0 and Q_1 have to be of the same type. This fact limits the use of the union operation for FAs that have the same type of states. For this reason, we present another variant of the union operation, called a *product union*, which does not have this limitation in Algorithm 3.

Algorithm 3 Construction of a finite automaton for product union

Input: Two complete finite automata, $M_0 = (Q_0, \Sigma_0, \delta_0, I_0, F_0)$ and $M_1 = (Q_1, \Sigma_1, \delta_1, I_1, F_1)$.
Output: A finite automaton, $M = (Q, \Sigma, \delta, I, F)$, such that $L(M) = L(M_0) \cup L(M_1)$.

1: $Q \leftarrow Q_0 \times Q_1$
2: $\delta \leftarrow \{(q_0, q_1) \xrightarrow{a} (q_0', q_1') \mid q_0 \xrightarrow{a} q_0' \in \delta_0 \wedge q_1 \xrightarrow{a} q_1' \in \delta_1\}$
3: $I \leftarrow I_0 \times I_1$
4: $F \leftarrow F_0 \times Q_1 \cup Q_0 \times F_1$
5: **return** $M = (Q, \Sigma, \delta, I, F)$

This algorithm creates a FA whose states are a product $Q_0 \times Q_1$. Both input automata must be complete. Algorithm 4 describes how to transform a FA M to an equivalent complete FA. It creates a special state called a *sink* to which all missing transitions to create a complete FA are redirected.

Algorithm 4 Conversion of a finite automaton to an equivalent complete finite automaton

Input: A finite automaton $M = (Q, \Sigma, \delta, I, F)$.
Output: A complete finite automaton $M' = (Q', \Sigma, \delta', I, F)$, such that $L(M') = L(M)$.

1: $Q' \leftarrow Q \cup sink$
2: $\delta' \leftarrow \delta \cup \{q \xrightarrow{a} sink \mid q \in Q', a \in \Sigma, |post_a(q)| = 0\}$
3: **return** $M' = (Q', \Sigma, \delta', I, F)$

2.6 Intersection

The family of regular languages is closed under intersection. The intersection can be described by DeMorgan's law as $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_1}}$ and, as a consequence, constructed using automaton complement and union. We, however, prefer a different solution presented in Algorithm 5, which is more suitable as it can easily be used when building the resulting automaton on the fly.

Algorithm 5 Construction of a finite automaton for intersection

Input: Two finite automata, $M_0 = (Q_0, \Sigma, \delta_0, I_0, F_0)$ and $M_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$.
Output: A finite automaton, $M = (Q, \Sigma, \delta, I, F)$, such that $L(M) = L(M_0) \cap L(M_1)$.

1: $Q \leftarrow Q_0 \times Q_1$
2: $\delta \leftarrow \{(q_0, q_1) \xrightarrow{a} (q_0', q_1') \mid q_0 \xrightarrow{a} q_0' \in \delta_0 \wedge q_1 \xrightarrow{a} q_1' \in \delta_1\}$
3: $I \leftarrow I_0 \times I_1$
4: $F \leftarrow F_0 \times F_1$
5: **return** $M = (Q, \Sigma, \delta, I, F)$

2.7 Determinization of a finite automaton

Later in this chapter, we will show Algorithm 7, which constructs the complement of a language. Before the construction of the complement, we, however, first need to perform determinization of a non-deterministic finite automaton using Algorithm 6 by implementing the so-called subset construction, also known as the Rabin-Scott subset construction [19, p. 210]. The subset construction is exponential in the worst case because the number of possible subsets of a set Q is 2^Q.

Algorithm 6 Conversion of a non-deterministic finite automaton to an equivalent deterministic finite automaton

Input: A finite automaton $M = (Q, \Sigma, \delta, I, F)$.
Output: A deterministic finite automaton $M' = (Q', \Sigma, \delta', \{I\}, F')$, such that $L(M') = L(M)$.

1: $Q' \leftarrow \emptyset$
2: $Next \leftarrow \{I\}$
3: $\delta' \leftarrow \emptyset$
4: **while** $Next \neq \emptyset$ **do**
5: Pick and remove a macro-state R from $Next$ and move it to Q'
6: **foreach** $a \in \Sigma$ **do**
7: $R' \leftarrow post_a(R)$
8: Add $R \xrightarrow{a} R'$ to δ'
9: **if** $R' \notin Q'$ **then** $Next \leftarrow Next \cup \{R'\}$
10: **end for**
11: **end while**
12: $F' \leftarrow \{q \mid q \in Q', q \cap F \neq \emptyset\}$
13: **return** $M' = (Q', \Sigma, \delta', \{I\}, F')$

The basic idea of this construction is to simultaneously follow all possible runs of an input string in M. It begins with the set of initial states and then considers which states can be reached when a certain symbol is read. This defines a new set of states that are obtained by following transitions from the initial states. The procedure is repeated whenever a new set of states is obtained.

Note that the resulting automaton is also complete. Utilizing the property of $post_a$ that $\forall a \in \Sigma : post_a(\emptyset) = \emptyset$, it implicitly creates a sink state $\emptyset$ and, for every pair $(s, a) \in Q \times \Sigma$ for which $\delta(s, a) = \emptyset$, it creates a transition $\delta'(s, a) = \emptyset$. The resulting automaton also does not contain inaccessible states because the set of states R in the algorithm contains only reachable states.

2.8 Complement of a language

It is easy to construct a finite automaton for the complement of the language accepted by a complete finite automaton as described by Algorithm 7. The resulting automaton is identical to the input automaton except for the final states as shown in the algorithm.

Algorithm 7 Construction of a finite automaton for the complement of the language accepted by a complete deterministic finite automaton

Input: A complete deterministic finite automaton $M = (Q, \Sigma, \delta, I, F)$.
Output: A complete deterministic finite automaton $M' = (Q, \Sigma, \delta, I, F')$, satisfying $L(M') = \Sigma^* \setminus L(M)$.

1: $F' \leftarrow Q \setminus F$
2: **return** $M' = (Q, \Sigma, \delta, I, F')$

2.9 Testing emptiness of a finite automaton

Testing whether $L(M) = \emptyset$ for some finite automaton M is called the *emptiness problem* for a FA M. A FA $M = (Q, \Sigma, \delta, I, F)$ is empty iff there is no accepting path from an initial state $s \in I$ to a final state $f \in F$. Algorithm 8 tests whether there is such a path.

Algorithm 8 Testing whether a finite automaton has any path from an initial state to a final state

Input: A finite automaton $M = (Q, \Sigma, \delta, I, F)$.
Output: `true` if $(i, w) \vdash^* (f, \varepsilon)$ where $i \in I, f \in F$; otherwise, `false`.

1: **if** $F = \emptyset$ **then return** `false`
2: $Processed \leftarrow \emptyset$
3: $Next \leftarrow I$
4: **while** $Next \neq \emptyset$ **do**
5: **if** $Next \cap F \neq \emptyset$ **then return** `true`
6: $Processed \leftarrow Processed \cup Next$
7: $Next \leftarrow post(Next) \setminus Processed$
8: **end while**
9: **return** `false`

At the beginning, the algorithm checks whether the input automaton contains any final state. If not, the algorithm terminates as there cannot be any accepting path without a final state. Otherwise, it follows all possible runs of an input string. It begins with the set of initial states and then follows states that can be reached when a certain symbol is read. The procedure, however, skips previously visited states. When the algorithm reaches a final state, it terminates as an accepting path was found. This procedure is repeated whenever a new set of states is obtained. If no final state was reached, the algorithm terminates as it did not find any accepting path.

2.10 Testing language inclusion of a pair of finite automata

The language inclusion decision problem decides whether $L(M_0) \subseteq L(M_1)$ for two finite automata M_0 and M_1. This problem is equivalent to testing whether $L(M_0) \cap \overline{L(M_1)} = \emptyset$,

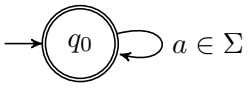


Figure 2.1: Finite automaton M_{Σ^*} accepting the language Σ^*.

which is composed of the operations of intersection, complement, and emptiness testing, which were described in the previous sections.

2.11 Testing universality of a finite automaton

Deciding whether $L(M) = \Sigma^*$ for some finite automaton M, that is whether M accepts every string over the input alphabet Σ, is called the *universality problem*. This problem is equivalent to testing whether $L(M_{\Sigma^*}) \subseteq L(M)$, which is using the language inclusion testing and the finite automaton M_{Σ^*} that accepts the language Σ^*. M_{Σ^*} is defined as $M_{\Sigma^*} = (\{q_0\}, \Sigma, \delta, \{q_0\}, \{q_0\})$ where $\delta = \{q_0 \xrightarrow{a} q_0 \mid a \in \Sigma\}$ and is shown in Figure 2.1.

2.12 Antichain-based approach

Apart from the simple classical algorithms for deciding language inclusion and universality (Sections 2.10 and 2.11, respectively), there exist more efficient approaches. One such a method is a family of algorithms called the *antichain-based* approach introduced by De Wulf et al. in [5]. The idea of these algorithms is to prune states that are being generated during the subset construction using subsumption [3].

There is, however, a newer solution that combines a simulation-based approach and the antichain-based approach. This solution utilizes a *simulation* relation to determine which states can be pruned. A simulation on an FA $M = (Q, \Sigma, \delta, I, F)$ is a relation $\preceq \subseteq Q \times Q$, such that $p \preceq r$ only if 1) $p \in F \implies r \in F$ and 2) for every transition $p \xrightarrow{a} p'$, there exists a transition $r \xrightarrow{a} r'$, such that $p' \preceq r'$. It holds that the simulation relation implies language inclusion, that is $p \preceq q \implies L(M)(p) \subseteq L(M)(q)$. The computed simulation relation is used for pruning unnecessary search paths of the antichain-based method.

The original antichain-based approach that was described by De Wulf et al. [5] is just a specific version of the more general approach when using the identity relation as the simulation relation. In the following two sections, we will use this general approach to describe algorithms that test language universality and inclusion for finite automata.

For this description, it is convenient to introduce some new notation. We will call a set of states of a FA M a *macro-state*. A macro-state is *accepting* if it contains at least one final state; otherwise, it is *rejecting*. Given a macro-state P, we define $L(M)(P) = \bigcup_{p \in P} L(M)(p)$. For two macro-states P and R and a binary relation $\sqsubseteq$, we write $P \sqsubseteq^{\forall\exists} S$ to denote $\forall p \in P. \exists s \in S : p \sqsubseteq s$. We define the *post-state* of a macro-state P as

$$post(P) = \{P' \mid \exists a \in \Sigma : P' = \{p' \mid \exists p \in P : p \xrightarrow{a} p' \in \delta\}\}.$$

Finally, we use $M^{\subseteq}$ to denote the set of relations over the states of a FA M that imply language inclusion.

2.12.1 Testing universality of a finite automaton

In Section 2.11, we discussed a naive algorithm for testing language universality. The algorithm performs $L(M_{\Sigma^*}) \subseteq L(M)$ using a naive algorithm, presented in Section 2.10, for

testing language inclusion. Such an algorithm can be inefficient due to likely fast growth of the number of states during the determinization. Note that in the case of universality checking, we can terminate the subset construction when we reach a rejecting macro-state.

In this work, we implemented a more efficient algorithm for testing universality using the antichain-based approach [3]. It runs in the similar manner as the naive algorithm by performing the subset construction but prunes unnecessary paths and therefore reduces the number of states that would be processed otherwise. This optimization is based on the fact that when the algorithm reaches a macro-state R whose language is a superset of the language of a visited macro-state P, then there is no need to continue the search from R. If a string is not accepted from R, it is also not accepted from P, and, moreover, because P is smaller, it is more likely to reach a rejecting macro-state.

Unfortunately, it is generally hard to test $L(M)(P) \subseteq L(M)(R)$ and, therefore, we use an easy-to-compute alternative: given P, R to be two macro-states, M to be a FA, and $\preceq$ to be a relation in $M^{\subseteq}$, then $P \sqsubseteq^{\forall\exists} R$ implies $L(M)(P) \subseteq L(M)(R)$. The $\preceq$ relation can be any relation that implies language inclusion like the identity relation or a simulation relation.

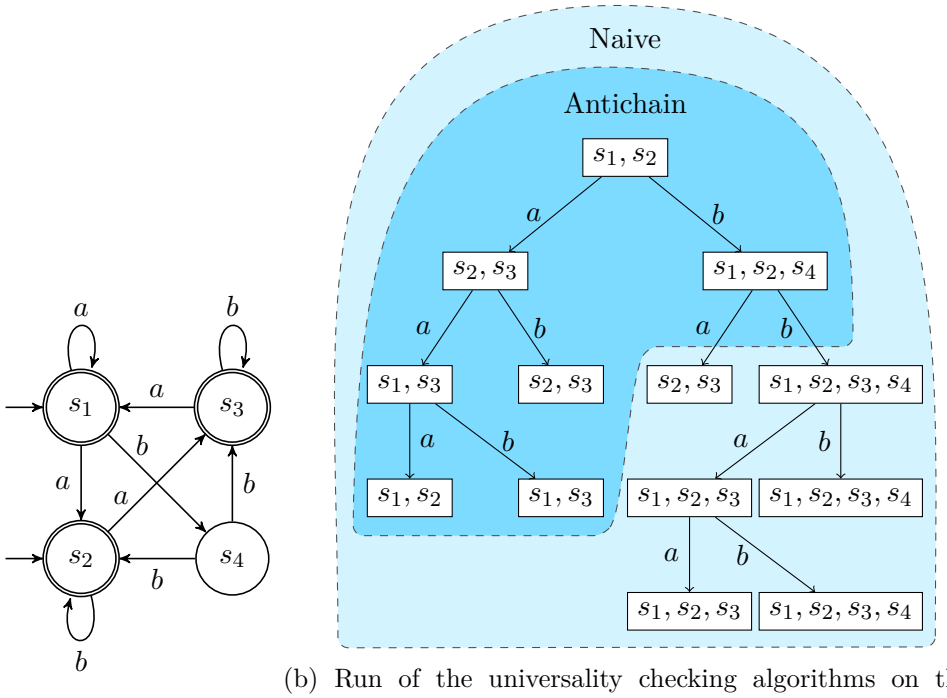
Algorithm 9 Testing universality of a finite automaton using the antichain-based approach

Input: A finite automaton $M = (Q, \Sigma, \delta, I, F)$ and a relation $\preceq \in M^{\subseteq}$.
Output: `true` if M is universal; otherwise, `false`.

```
1:  if I is rejecting then return false
2:  Processed ← ∅
3:  Next ← {I}
4:  while Next ≠ ∅ do
5:      Pick and remove a macro-state R from Next and move it to the Processed
6:      foreach P ∈ {R' | R' ∈ post(R)} do
7:          if P is an rejecting macro-state then return false
8:          else if ∄S ∈ Processed ∪ Next s.t. S ⪯^∀∃ P then
9:              Remove all S from Processed ∪ Next s.t. P ⪯^∀∃ S
10:             Add P to Next
11:         end if
12:     end for
13: end while
14: return true
```

Algorithm 9 tests the universality of a finite automaton M using the antichain-based approach we just discussed. It works as follows: *Next* is a set of macro-states to be processed and, at the beginning, it contains the initial states of M. The algorithm runs until the set *Next* of macro-states is not empty or a rejecting macro-state is found. Meanwhile, it picks one macro-state R from *Next* and moves it to *Processed*. The algorithm then creates all successors of the macro-state R and terminates if any of these successors are rejecting. The lines 8 to 10 of the algorithm implement the pruning using the relation $\preceq$ with regard to each successor P of the macro-state R. If there are no macro-states in both *Processed* and *Next* that are $\preceq^{\forall\exists}$-smaller than P, then it removes all items of *Processed* and *Next* that are $\preceq^{\forall\exists}$-larger than P. Finally it adds P into *Next*.

Figure 2.2 shows runs of the classical and the antichain-based universality checking algorithms. The classical approach generated 13 macro-states, whereas, thanks to macro-state pruning, only 7 states were generated by the antichain-based approach. In this example,



(a) Input finite automaton M.

(b) Run of the universality checking algorithms on the FA M. Areas labelled "Naive" and "Antichain" represent the naive and antichain-based algorithms, respectively.

Figure 2.2: An example of universality checking of a FA M (taken from [3]).

we used identity as the relation $\preceq$. The search can stop at the state $\{s_1, s_2, s_4\}$ because it is a superset of the initial state, i.e. $\{s_1, s_2\} \subseteq \{s_1, s_2, s_4\}$.

2.12.2 Testing language inclusion of a finite automaton

We can use the same techniques we learned in the previous section to implement an algorithm that tests language inclusion using the antichain-based approach. The language inclusion decision problem decides whether $L(M_0) \subseteq L(M_1)$ for two finite automata M_0 and M_1. The naive algorithm that tests language inclusion, presented in Section 2.10, builds the product automaton $M_0 \times \overline{M_1}$ of M_0 and the complement of M_1 and searches for an accepting state. A state of the product automaton $M_0 \times \overline{M_1}$ is a pair (p, P) where p is a state of M_0 and P is a macro-state of M_1. For convenience, we will call the pair (p, P) a *product-state*. Such a product-state (p, P) is accepting iff the state p is accepting and the macro-state P is rejecting. We will use $L(M_0, M_1)(p, P)$ to denote the language of the product-state (p, P). The language of M_0 is not contained in the language of M_1 iff there exists some accepting product-state (p, P) reachable from some initial product-state. It holds that $L(M_0, M_1)(p, P) = L(M_0)(p) \setminus L(M_1)(P)$. We define a *post-state* of a product-state (p, P) as

$$post((p, P)) = \{(p', P') \mid \exists a \in \Sigma : p \xrightarrow{a} p' \in \delta_{M_0}, P' = \{p'' \mid \exists p \in P : p \xrightarrow{a} p'' \in \delta_{M_1}\}\}$$

where δ_{M_0} and δ_{M_1} are the transition functions of M_0 and M_1, respectively.

Algorithm 10 tests the language inclusion of two finite automata using the antichain-based approach we have just discussed. The algorithm has a similar structure as Algorithm 9, which tests universality. Both algorithms terminate when the set *Next* is empty.

Similarly, Algorithm 10 builds the product automaton and searches for an accepting product-state. It can stop and conclude that the language inclusion does not hold when it encounters such an accepting product-state.

Algorithm 10 Testing language inclusion of two finite automata using the antichain-based approach

Input: Two finite automata, $M_0 = (Q_0, \Sigma, \delta_0, I_0, F_0)$, $M_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$ and a relation $\preceq \in (M_0 \cup M_1)^{\subseteq}$.
Output: `true` if $L(M_0) \subseteq L(M_1)$; otherwise, `false`.

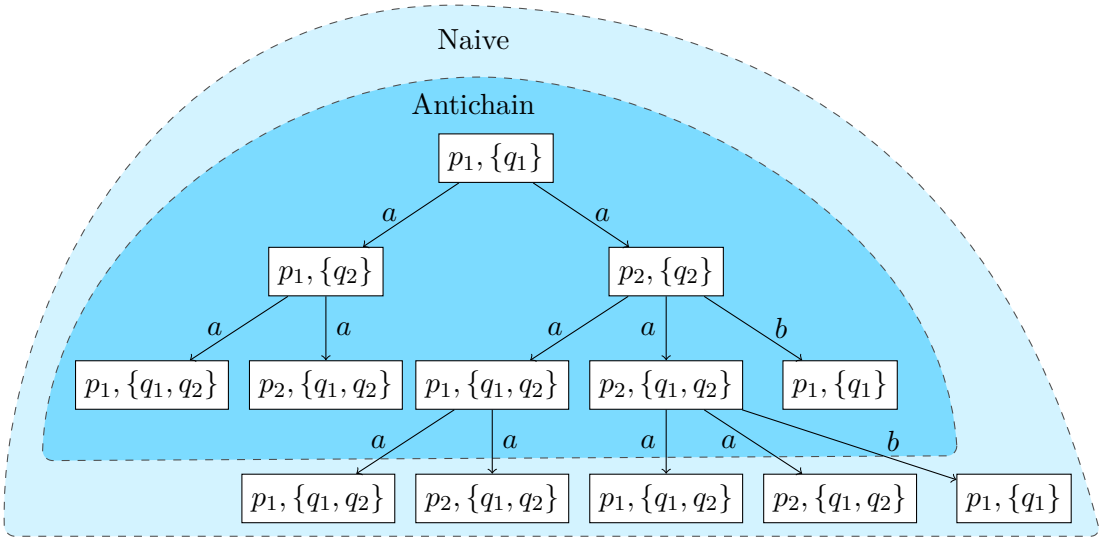
if there is an accepting product-state in $\{(i, I_1) \mid i \in I_0\}$ **then return** `false`
$Processed \leftarrow \emptyset$
$Next \leftarrow \{(i, I_1) \mid i \in I_0\}$
while $Next \neq \emptyset$ **do**
    Pick and remove a product-state (r, R) from $Next$ and move it to the $Processed$
    foreach $(p, P) \in \{(r', R') \mid (r', R') \in post((r, R))\}$ **do**
        if (p, P) is an accepting product-state **then return** `false`
        else if $\nexists p' \in P$ s.t. $p \preceq p'$ **then**
            if $\nexists (s, S) \in Processed \cup Next$ s.t. $p \preceq s \wedge S \preceq^{\forall\exists} P$ **then**
                Remove all (s, S) from $Processed \cup Next$ s.t. $s \preceq p \wedge P \preceq^{\forall\exists} S$
                Add (p, P) to $Next$
            end if
        end if
    end for
end while
return `true`

The universality checking algorithm utilized the simulation relation to prune unnecessary paths during the macro-state construction. This optimization can also be generalized for language inclusion checking. Let $M_0 = (Q, \Sigma_0, \delta_0, I_0, F_0)$ and $M_1 = (Q, \Sigma_1, \delta_1, I_1, F_1)$ be two FAs and it holds that $M_0 \cap M_1 = \emptyset$. Let $M_0 \cup M_1$ be the union of automata M_0 and M_1. Let $\preceq$ be a relation in $(M_0 \cup M_1)^{\subseteq}$. During the construction of product-states, we can halt the algorithm in a product-state (p, P) if (a) there exists some visited product-state (r, R), such that $p \preceq r$ and $R \preceq^{\exists\forall} P$, or (b) $p' \in P : p \preceq p'$. In other words, $p \preceq r$ and $R \preceq^{\exists\forall} P$ implies $L(M_0, M_1)(p, P) \subseteq L(M_0, M_1)(r, R)$.

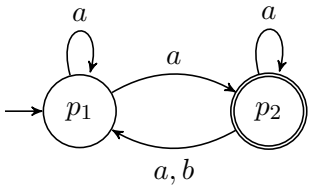
Figure 2.3 shows run of the naive and the antichain-based language inclusion checking algorithms. The naive approach generated 13 macro-states, and, thanks to the macro-state pruning, only 8 states were generated by the antichain-based approach. We also used $\preceq$ as the identity relation in this example. The algorithm does not need to continue the search from the product-states $(p_1, \{q_1, q_2\})$ and $(p_2, \{q_1, q_2\})$ because $Processed \cup Next$ already contains product-states $(p_1, \{q_1\})$ and $(p_2, \{q_2\})$.

2.13 Finite automata libraries

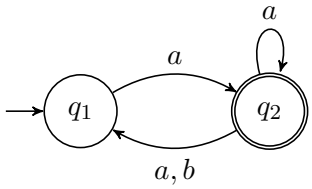
This section discusses other finite automata libraries with similar aims and properties as the library proposed in this work. As mentioned in the introduction, the goal of this work was to implement an efficient finite automata library for certain operations on automata that are relevant to formal analysis of computer programs.



(a) Run of the language inclusion checking algorithms on FAs M_0 and M_1. Areas labelled "Naive" and "Antichain" represent the naive and antichain-based algorithms, respectively.



(b) Finite automaton M_0.



(c) Finite automaton M_1.

Figure 2.3: An example of language inclusion checking of FAs M_0 and M_1 (taken from [3]).

The first library mentioned here is *VATA* [15]. It is a highly optimized library for non-deterministic finite word and tree automata. The focus of the library is to be used in formal verification. The library utilizes the antichain-based and a bisimulation up to congruence approaches to perform language inclusion checking. The library also allows to work with finite automata and internally uses an explicit representation of a FA.

Another library called *Automata* was created by Margus Veanes and Loris D'Antoni [1]. According to the project description, Automata is a .NET library for composing and analyzing regular expressions, automata, and transducers over symbolic alphabets, i.e. alphabets with potentially infinitely many symbols. Apart from word automata, this library also contains algorithms for analysis of tree automata. The library can work with finite alphabets and also with their symbolic counterparts.

Automaton [2] is a Java library that contains deterministic and non-deterministic finite automata implementations using a symbolic representation of an alphabet. The library contains standard finite automata operations as well as some non-standard ones (intersection, complement, etc.).

Chapter 3

Haskell

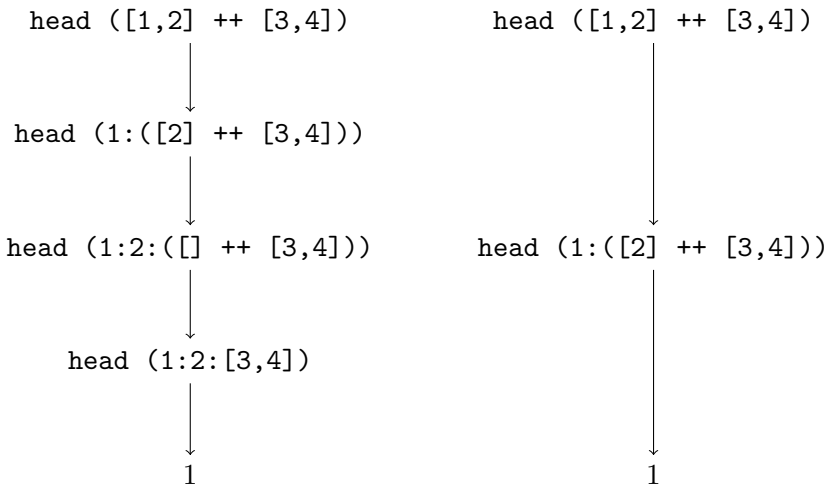
Haskell is a standardized, general-purpose purely functional language, which has non-strict semantics and strong static typing. It is one of the most popular functional programming languages. The language could be described according to its main properties [10]:

- *Static typing.* All Haskell expressions have types that are checked during the compilation time. All expressions must logically fit together according to the type system. Otherwise, the compiler will reject the program.
- *Type inference.* When writing Haskell programs, one does not have to explicitly indicate all types. The compiler will try to automatically infer types in the program. Of course, types can be written out explicitly if desired.
- *Laziness.* Function arguments are not evaluated when they are bound to variables but their evaluation is postponed until their results are needed by other computations. This enables to write clearer code, custom control structures, or to achieve performance benefits.
- *Purity.* Every Haskell function is a pure function, forbidding to write code that performs side-effects or mutates state. This leads to a much more maintainable and error-prone code. Side-effects are performed inside *monads*. Monad is a mechanism of how to build program components by joining simple components in robust ways. By using this mechanism, all side-effects are explicitly separated from a pure code.
- *Concurrency.* Thanks to the Haskell's purity, the language is suitable for concurrent programming due to its explicit handling of effects. The language comes with a lightweight concurrency library and an efficient parallel garbage collector.

3.1 Non-strict semantics

In the following explanation, we will use a mathematical symbol $\perp$ that is called the *bottom*, and refers to a computation that never completes successfully. It includes computations that fail due to some kind of an error, or computations that just falls into an infinite loop (without returning any data).

Haskell is an expression language, which means the program run corresponds to a reduction of an expression. There are two relevant reduction strategies called the *strict semantics* and the *non-strict semantics*. Here, the reduction strategies refer to the semantics of the



(a) Strict evaluation. (b) Non-strict evaluation.

Figure 3.1: Strict and non-strict reduction strategies on the Haskell expression `head ([1,2] ++ [3,4])`. In this case the non-strict reduction skips the reduction of the innermost sub-expression `[2] ++ [3,4]`.

expression. They simply define what kinds of values in the domain map to what kinds of values in the codomain [22]. In particular, a strict function must map the value $\perp$ to $\perp$; a non-strict function is not required to do this.

A strict function must map the value $\perp$ to $\perp$. That effectively means that an expression language has a strict semantics if all sub-expressions of the expression are reduced before the expression. That is, the reduction starts with the innermost sub-expressions and then gradually works outwards. On the other hand, an expression language has a non-strict semantics if expressions can have a value even though some of their sub-expressions do not. This means that the reduction starts with the outermost expression and gradually reduces sub-expressions inwards. Figure 3.1 represents both reduction strategies on a Haskell expression `head ([1,2] ++ [3,4])`.

Haskell expressions can have certain properties based on how much they are reduced. An expression is in the *weak head normal form* (WHNF), if it is either:

- a constructor (eventually applied to arguments) like `Just (square 16)` or `(:) 1`;
- a built-in function applied to too few arguments (perhaps none) like `(+) 2` or `sqrt`;
- or a lambda abstraction.

An expression is in the *normal form* (NF), if it is fully reduced, and no sub-expression could be evaluated any further [11]. For example, the normal form of the expression `Just (square 16)` is `Just 4`.

Haskell is one of the few languages that utilize the non-strict semantics. This allows programs to work with conceptually infinite data structures. Generally, the main advantage of non-strict languages over strict languages is cleaner, more maintainable code, and also that they allow lazy evaluation.

3.2 Lazy evaluation

Strict and non-strict semantics, which were discussed in the previous section, were concerned with the mathematical meaning of an expression [22]. However, the previous section was not discussing such concepts as the running time of a function, memory consumption, or even a computer.

On the other hand, this section discusses *operational semantics*, i.e. how the program is executed on a real computer. One such a strategy is called *lazy evaluation* [11]. Lazy evaluation means that expressions are not evaluated when they are bound to variables but their evaluation is deferred until their results are needed by other computations. As a consequence, function arguments are not evaluated before they are passed to a function but only when their values are actually needed. Haskell uses lazy evaluation to implement non-strict semantics.

Such deferred values are stored as thunks. A *thunk* is a value that is yet to be evaluated. It is essentially a data structure containing values that are needed to evaluate an expression, plus a pointer to the expression itself. When the result is needed, the program evaluates the expression and then replaces the thunk with the result.

Suppressing non-strict semantics

Non-strict evaluation can sometimes hurt run-time performance if not used carefully. Specifically, the program needs to hold on to data passed into an expression to evaluate them later if needed. If such data would be evaluated anyway, it is more economical to evaluate them immediately. Luckily, Haskell has mechanisms to eagerly evaluate sub-expressions and thus to suppress non-strict evaluation if needed [11].

First such a mechanism is called the *strictness analysis*. Haskell compilers like GHC are performing strictness analysis, which attempts to determine which sub-expressions are always evaluated by the expression and therefore can be evaluated by the caller instead.

Strictness analysis can improve the run-time performance by eagerly evaluating some sub-expressions. Unfortunately, the analysis often cannot optimize all sub-expressions suitable for eager evaluation. Therefore, Haskell enables to explicitly annotate sub-expressions to be evaluated eagerly. The basic method to explicitly introduce strictness to Haskell program is the `seq` function. Its type annotation is `seq :: a -> b -> b` and has an important property that it is strict on its first argument. `seq` is given by the following two definitions:

$\bot$ `` `seq` b = $\bot$ `` a `` `seq` b = b.``

These two definitions are all `seq` must satisfy. If the compiler can statically prove that the first argument is not $\bot$ or that its second argument is $\bot$, it does not have to evaluate anything. However, such a situation almost never happens in practice. For example, `seq` function is used in a definition of a standard library function `foldl'`:

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' _ z [] = z
foldl' f z (x:xs) =
  let
    c = f z x
  in
    c `seq` foldl' f c xs
```

This function is commonly called the *strict fold*. As opposed to the ordinary `foldl` function, the strict fold eagerly evaluates its accumulator parameter.

Another way of how to add required strictness to a code is the GHC's language extension called the *bang patterns*. By using this extension, we can suggest a function parameter to be strict by annotating the parameter with the `!` symbol. For example, `f !x !y = z` defines a function `f` with both its parameters to be evaluated strictly. Specifically,

```
f !x !y = z
```

is semantically equivalent to

```
f x y
  | (x `seq` y) `seq` False = undefined
  | otherwise = z
```

Bang patterns are the easiest way to modify strictness properties of some code as they are syntactically less invasive than other methods [18, p. 574].

The last mechanism to explicitly introduce strictness that we will mention is the *strict infix application operator* `($!)` [11]. It differs from the standard application operator `($)` in that it evaluates its arguments strictly. So instead of writing

```
f (g x)
```

if 1) you were to evaluate `g x` anyway, and 2) `f` is not visibly strict or inlined, it is more efficient to write

```
f $! (g x).
```

3.3 Profiling

Haskell is a high-level language that allows to program in abstractions like functors, monoids, monads, and others. The language specification goes to great lengths to avoid prescribing any specific evaluation model. Most of the time, it is beneficial to free ourselves from low-level details and, instead, focus on the essence of the problem we are trying to solve [18, p. 561-578].

Unfortunately, in real-world programming, it is often necessary to concentrate on lower-level details in order to optimize the program's performance. Haskell programs do not run on abstract machines but on real hardware with time and space constraints. Therefore, it is essential to use the right profiling tools when performance matters to analyze run-time behaviour of a program. Luckily, the GHC platform provides these tools.

GHC allows to collect runtime statistics by passing specific flags to the Haskell runtime, using a special `+RTS` flag to introduce arguments reserved for the runtime system. Some other useful flags are:

- The `-s` flag instructs the runtime to gather statistics about the memory and garbage collector. It gives information about what a program is doing. In particular, it tells how much time was spent in garbage collection and what the maximum live memory usage was.
- The `-p` flag enables to get the time and allocation profiling report. Such a report gives information about the proportion of time and memory each function consumed in relation to all other functions. This report is especially useful to quickly narrow down a problematic location in a code that is responsible for a performance bottleneck.

- The `-K` flag is used to set a specific stack limit for a program. A non-optimized program or profiling can often lead to a stack overflow and, therefore, it is sometimes useful to set a larger stack limit.
- The `-hc` flag turns on the heap profile information gathering of a program. Heap profile is useful for revealing memory leaks in programs, which is a situation when memory is consumed and not disposed afterwards. This leads to a heavy garbage collector activity.

Apart from the runtime statistics gained during profiling, it is often useful to analyze lazy evaluation. To check the laziness of a program, we can use the Hood debugger [8]. This debugger is based on the idea of observing functions and structures as they are evaluated. It provides information about which part of a data structure is evaluated and which part is not. Hood provides the function `observe :: String -> a -> a`. The first parameter just defines a name that is associated with the observation. The second parameter is then passed unmodified as a result and additionally records to which result its parameter is evaluated. In the end, we can check the records and find out which parts of a data structure passed into `observe` were evaluated and which were not.

3.4 Monad

In functional programming, *monads* are a way of how to join computer program components in robust ways. A monad often encapsulates values of a particular datatype, creating a new type with an associated computation. The computation on that particular datatype depends on a specific type of the monad. Monads find many uses in functional programming as they provide a convenient framework for simulating effects found in other languages, such as global state, exception handling, output, or non-determinism. Monads allow to represent a wide variety of concepts commonly found in imperative languages in a pure, functional way without the need for special syntactic constructs [16] [18, p. 325-357].

The following definition is a simplified version of a `Monad` type class:

```
class Monad m where
  return :: a -> m a

  (>>=) :: m a -> (a -> m b) -> m b
```

Many types have monadic behaviour and thus implement this type class, such as `Maybe`, `[]`, `IO` and many others. The first function that the `Monad` type class defines is `return`. It receives a value and puts it in a minimal default monadic context that still holds that value. In other words, it takes some value and "wraps" it in a monad. Such a minimal default context of course depends on a specific type of a monad represented by the type parameter `m`. In case of the `Maybe` monad, where `m` is `Maybe`, the function `return` is defined as follows:

```
return a = Just a
```

When using the `[]` monad, it is defined as:

```
return a = [a]
```

The second function, `(>>=)`, also called the *bind operator*, takes a monadic value (that is, a value with a context) and feeds it to a function that takes a normal value but returns

a monadic value. Utilizing this knowledge, we can compose a sequence of function calls (forming a pipeline) with several bind operators chained together in an expression. In this pipeline, the bind operator unwraps the value of the previous monadic context and passes it to function `a -> m b`, which creates a new monad with a type parameter `b`. Afterwards, this monad is fed to the next bind operators composed in the pipeline.

The following code snippet is an example of such a pipeline:

```
return 5
  >>= (\x -> if x > 10 then Just x else Nothing)
    >>= (\y -> Just y)
```

Initially, the value `5` is wrapped in a default context (in this case `Just 5`) and send to the pipeline. The first bind operator passes the value into the first transformation function, which returns `Nothing`. Afterwards, this monad is passed into the second bind operator, which skips transformation function and immediately returns `Nothing`. Bind operation implements logic that decides whether to unwrap the value and pass it to the transformation function or to skip this step completely.

Haskell provides a special syntactic construct called the *`do`-notation*, which allows to write monadic pipelines in a such a way that mimics an appearance of imperative languages. The compiler translates `do`-notation to expressions involving chained bind operators. The pipeline showed in the previous snippet can be rewritten in a `do`-notation as:

```
do
  x <- return 5
  y <- if x > 10 then Just x else Nothing
  Just y
```

3.4.1 Monad laws

Every monad implementation should be compatible with certain laws that specify what behaviour is expected from a monad. Haskell language itself does not enforce these laws through the compiler—it is up to the author of a `Monad` instance to follow them. Abiding to these laws ensures that we can expect some common behaviour when using monads. Every law below can be read as: the expression on the left-hand side of `(==)` is equivalent to the expression on the right-hand side.

```
(return x) >>= f  ==  f x
   m >>= return   ==  m
 (m >>= f) >>= g  ==  m >>= (\x -> (f x >>= g))
```

Intuitively, the first and the second law state that `return` acts as a neutral element of `(>>=)`. The third law states that binding two functions in succession is the same as binding a composition of the functions.

There are many types of monads found in the Haskell standard library and many more in other, third-party packages. In the following sections, we will explore three of them that are useful when implementing our library.

3.4.2 Maybe

The `Maybe` monad is used to create a chain of sequential computations that may fail. If one computation in such a chain fails, the rest of the chain is skipped and `Nothing` is returned. The implementation of this monad has the following definition:

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x
```

The `return` function only wraps the value in a default context. The behaviour of the bind operator depends on the input. If the input is `Nothing`, the bind operator returns `Nothing`. When the input is `Just x`, the value `x` is fed to the transformation function and the result is returned.

3.4.3 State

Because Haskell is a pure language, programs are made of functions that cannot change global state or variables. They can only perform computations based on their inputs and then return results. This restriction is actually useful most of the time as it simplifies the reasoning about programs. We do not need to keep in mind the values of all variables at a specific point of time. However, this property comes with a price as some problems are inherently stateful, which can be tedious to model in a pure language.

Due to the above-mentioned problem, Haskell provides the `State` monad, which allows to attach state information of any type to a calculation and, thus, to imitate a mutable environment. Together with the `do`-notation, a code using the `State` monad resembles an imperative one. A state is implicitly handled inside the `State` monad and the need to transfer all state through parameters of pure functions is unnecessary.

Basically, the `State` monad is a function that takes some state `s` and returns a value `a` along with some new state [16]:

```
newtype State s a = state { runState :: s -> (a,s) }
```

Now, that we described the `State` monad, we can move to its implementation:

```
instance Monad (State s) where
  return x = state $ \s -> (x,s)
  (state h) >>= f = state $ \s -> let (a, newState) = h s
                                      (state g) = f a
                                  in  g newState
```

The `return` function always wraps the passed value in a default minimal monadic context. It is also the case of the `State` monad in which `return` takes a value and makes a stateful computation that always has that value as its result. So `return` creates a stateful computation that presents a certain value as the result and keeps the state unchanged. Now, we will move on to `(>>=)`. The bind operator must return some stateful computation, and, therefore, it returns a lambda wrapped in the `state` data constructor. This lambda represents a stateful computation that will be returned. Generally, a bind operator usually unwraps the value from the previous context and applies the transformation function on it. It works exactly in the same way with the `State` monad. By unwrapping a stateful computation `h` out of a previous context and passing the current state `s` to it, a new intermediate

result `(a, newState)` is produced. Afterwards, the transformation function `f` is applied and a new stateful computation `g` is finally returned. `(>>=)` kind of glues two stateful computations together, only the second one is hidden inside a function that takes the previous one's result.

As a simple demonstration of the `State` monad, let us consider an abstract data type, stack, implemented using a single-linked list where list's head represents the top element of the stack:

```
type Stack = [Int]
```

We can then define the two essential stack operations, `pop` and `push`, in terms of the `State` monad as follows:

```
pop :: State Stack Int
pop = state $ \(x:xs) -> (x,xs)

push :: Int -> State Stack ()
push a = state $ \xs -> ((),a:xs)
```

`pop` is a stateful computation with a state of type `Stack` that returns a value of type `Int`. Given the input stack `(x:xs)`, `pop` just removes the top element `x` from the stack and returns it. `push` requires a parameter of type `Int` to become a stateful computation with a state of type `Stack` that does not return any value (expressed as the `()` type). `push` simply adds the parameter to the stack.

Given these operations, we can conveniently use `do`-notation to implement a function, which manipulates a stack without the need to explicitly pass a state of type `Stack` around:

```
computation :: State Stack Int
computation = do
  push 3
  a <- pop
  pop
```

Specifically, the function is a stateful computation on the `Stack` type that pushes value `3`, pops a value to `a`, and finally pops again.

3.4.4 Parsec

Parsec is a monadic parser combinator library for Haskell. It can parse context-sensitive, infinite look-ahead grammars but reaches the best performance on predictive (LL) grammars. LL grammars are grammars that can be parsed by an LL parser, which parses the input from left to right and constructs a leftmost derivation of the sentence. According to Daan Leijen [13], combinator parsing, which is also used in Parsec, offers several advantages over YACC and event-based parsing as:

- The combinator parsers are developed in the same language as the rest of the program. The user does not have to learn both parser's domain specific language (e.g. Yacc) and the actual programming language (e.g. C). This offers several benefits, such as simplicity for the user as well as availability of developer tools of the host language.
- Parsers are first-class objects of the programming language. Therefore, they can be passed as parameters, returned as values, or put into lists. A user can also easily extend available parsers to meet his specific needs.

In the previous section, we saw that the `State` monad is used to carry an implicit state. The Parsec library utilizes the `ParsecT` monad, which is similar to the `State` monad in the sense that it also carries a state. The actual `ParsecT` monad used by the Parsec library is rather complex. In order to introduce its structure without going into unnecessary details, we will present a simpler variant called the `Parser` monad:

```
type Error = String

type ParserState = String

newtype Parser a = Parser {
  runParse :: ParserState -> Either Error (a, ParserState)
  }
```

This monad has two logically distinct aspects. The first one is the idea of a parse failing and providing a message with the details. We represent this using the `Error` type found on the left-hand side of the `Either` type. The other idea involves carrying around a piece of an implicit state (`ParserState`) with a partially consumed input of the type `String`. The following definition contains the bind operation. It simply chains two parsers, `firstParser` and `secondParser`, together:

```
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
firstParser >>= secondParser = Parser chainedParser
  where chainedParser initState =
    case runParse firstParser initState of
      Left errMessage ->
        Left errMessage
      Right (firstResult, newState) ->
        runParse (secondParser firstResult) newState
```

It wraps the function `chainedParser` to a new parser and returns this parser. Inside `chainedParser`, the `firstParser` parser is run with the following two possible outcomes: 1) it returns an error, which is then passed into the resulting `chainedParser` parser or 2) it returns a successful result, which is then run by the `secondParser` parser.

The original monad used in Parsec is declared as `ParsecT s u m a` with

1. a stream type `s`, which determines a type of data to be parsed (e.g. `String`, `ByteString`, and others),
2. a user state type `u`, because `ParsecT` is also a state monad,
3. an underlying monad `m`,
4. and a return type `a`, which is returned from the monad if the parsing was successful.

The library also defines these, more specialized, versions of `ParsecT` among others:

```
type Parsec s u = ParsecT s u Identity

type Parser = Parsec ByteString ()
```

The type `Parser`, which represents a parser with a `ByteString` stream type without a user state, is used in our library.

Sequence and choice

Apart from the bind operator for chaining two parsers, the Parsec library contains many utility functions and constructs to define custom parsing logic in a simpler and more elegant way. Two important operations for specifying grammars are *sequence* and *choice*. In Parsec, we can use the monadic `do`-notation to perform sequencing because sequencing is implemented using the bind operator. The following parser parses an opening parenthesis immediately followed by a closing parenthesis:

```
openClose :: Parser Char
openClose = do
  char '('
  char ')'
```

Note that the `char` parser parses a single character, which specified by its parameter.

To define a choice parser, we can use the choice operator (`<|>`), which applies its second operand unless the first operand succeeds. Let us move on to an example of a parser that recognizes all inputs with matching pairs of parentheses:

```
parens :: Parser ()
parens = do
  char '('
  parens
  char ')'
  parens
  <|> return ()
```

Matching parentheses consist of either an open parenthesis followed by a matching pair of parenthesis, a closing parenthesis and another matching pair of parentheses, or it is empty; the empty alternative is implemented using the `return x` parser, which always succeeds with the value x without consuming any input.

Adding semantics

In the previous example, we built the `parens` parser, which processes some text according to predefined syntax rules. Given an input text, the parser would then return information whether the parsing succeeded or did not. The parser simply recognizes matching pairs of parenthesis but does not return a useful value. We can extend this parser by adding some semantic actions. These will compute the maximal nesting level of the parentheses. The `<-` construct will be used to bind the intermediate values returned by the parsers:

```
nesting :: Parser Int
nesting = do
  char '('
  n <- nesting
  char ')'
  m <- nesting
  return (max (n+1) m)
  <|> return 0
```

Notice that now the `nesting` parser has the `Int` data type. It is a type that contains the maximal nesting level value. This value is then returned when parsing is invoked.

Sequences and separators

Let us consider an example of a parser that parses a word, that is a sequence of one or more characters:

```
word :: Parser String
word = do
  c <- letter
  do
    cs <- word
    return (c:cs)
  <|> return [c]
```

After parsing a letter, it either parses the rest of the word or returns the single letter as a string. It is not particularly hard to follow the logic of this parser but it can quickly become unclear with a more complicated parsing logic. Therefore, the Parsec library provides some useful abstractions to clarify and simplify such logic. One such abstraction is the `many1` parser that parses a sequence of one or more parsers. Using this parser, the previous `word` parser can be substantially simplified:

```
word :: Parser String
word = many1 letter
```

Apart from the `many1` parser, the library provides the `many` parser, which parses a sequence of *zero* or more parsers.

Parsec also provides parsers that allow to parse a sequence of parsers delimined by some separator. These parsers are called `sepBy` and `sepBy1`. We can implement a parser that parses a sequence of words in which each word is separated by a comma:

```
words :: Parser [String]
words = sepBy1 word separator

separator :: Parser Char
separator = char ','
```

By utilizing the Parsec library, one can easily build-up a hierarchy of parsers from simple to more complex ones. Such a hierarchy of parsers can then express parsing logic in a clear, declarative fashion. The library provides many basic parsers to simplify its usage. We use the Parsec library in our library to parse the Timbuk text format, which is elaborated on in Section 4.2.

Chapter 4

Analysis

This chapter focuses on the design and analysis of the proposed finite automata library written in Haskell. First, data structures that are used for representing automata will be presented. Then, the textual input format for the library will be described. In the end, supported operations over finite automata (union, intersection and complement) and decision problems (membership, emptiness, language inclusion, and universality) will be discussed.

4.1 Data structures

It is important to carefully encode mathematical structures to their data structure counterparts in Haskell, as the choice of data structures is crucial to the performance of the implemented operations. Mathematically, a non-deterministic finite automaton is a 5-tuple $M = (Q, \Sigma, \delta, I, F)$ consisting of states, alphabet, transitions, initial states, and final states. Each string of the alphabet Σ is made of symbols $a \in \Sigma$. We want our design of data structures to be as general as possible to cover various use cases in which users may use the library. Instead of hard-coding some of the types representing individual components of a FA, it is more appropriate to generalize such types utilizing Haskell's support for type parameters.

We can take a simple approach to encode a single transition in Haskell as a record data structure:

```
data Transition sym sta =
  Transition
    { symbol :: sym
    , source :: sta
    , target :: sta
    }
```

This approach follows the mathematical notation of a transition presented in Chapter 2. There, we used the notation $p \xrightarrow{a} q$ to denote that $q \in \delta(p, a)$. Note, that the Haskell types to represent symbol and state are represented as type parameters `sym` and `sta`, respectively.

Similarly, we can use a simple approach to represent a finite automaton in Haskell in a way that closely follows its mathematical definition:

Table 4.1: Asymptotic time complexity of access, insertion, and deletion operations on a random element of `[]` and `Set` [23, 24]. The number of elements in a container is represented by n. For the union operation, the number of elements in containers is represented by m and n.

Operation	`[]`	`Set`
Access	$O(m)$	$O(\log m)$
Insertion	$O(m)$	$O(\log m)$
Deletion	$O(m)$	$O(\log m)$
Union	$O(m^2 + mn)$	$O(m \log(n/m + 1)), m \leq n$

```
data Fa sym sta =
  Fa
    { initialStates :: [sta]
    , finalStates :: [sta]
    , transitions :: [Transition sym sta]
    }
```

A finite automaton is represented as a record data structure `Fa`. As in the previous code snippet, types for representing symbols and states are given as type parameters. We need to keep information about sets of initial and final states to be able to distinguish them from other states. However, it is not necessary to store the set of states itself because it can be retrieved explicitly by enumerating the states stored inside transitions. A similar point holds for the alphabet—it is not always necessary to store the alphabet explicitly. The alphabet can be retrieved from transitions. However, for certain operations like complement, it is important to be able to pass a custom alphabet into the operation. Therefore, we provide two variants of all FA operations: 1) a variant that uses an alphabet retrieved from FA transitions, and 2) a variant that uses an external alphabet passed as an argument to the operation.

One problem with the above-mentioned approach is performance. Operations on a finite automaton heavily use container and set operations, such as accessing a specific element, inserting and deleting an element, or a set union. In the previous code snippet, the sets of initial states, final states, and transitions were represented by the Haskell's list data structure (denoted as `[]` in Haskell). Internally, `[]` is implemented as an immutable single-linked list. Such a list is a non-trivial constant-factor faster for operations at the head, the first element of a list, making it a more efficient choice for stack-like and stream-like access patterns [11].

Haskell allows to work with other containers apart from lists. One such a container is called `Set` located in the library `containers` (in module `Data.Set`). The implementation of `Set` is based on *size-balanced* binary trees (or trees of *bounded balance*) [24]. Table 4.1 compares time complexity for certain operations of `[]` and `Set`. It is clear that for the listed operations, `Set` has better asymptotic time complexity than the list. Apart from this, `Set` implicitly stores only unique elements. On the other hand, it is necessary to manually remove duplicate elements after performing certain list operations to keep elements of the list unique. Function `nub` removes duplicate elements in a list but its time complexity is quadratic.

There exist other ways of how to implement individual components of a finite automaton apart from the `[]` and `Set` containers. For example, one such a way is to use a lookup table to implement a container of transitions.

4.1.1 Typeclasses

Haskell provides a way to specify that some data structure supports some specific abstraction. This functionality is provided through typeclasses. Haskell has many build-in typeclasses like `Functor`, `Applicative`, `Monad` or `Monoid`. We will not explain these individual typeclasses here. In our setting, it could be useful for a finite automaton `Fa` to support the `Functor` typeclass. The `fmap` function could map over states of `Fa`. Its type declaration would be

```
fmap :: (sta1 -> sta2) -> Fa sym sta1 -> Fa sym sta2
```

This would open up new possibilities, such as transforming all states `sta` of `Fa` to a different type. For example, states of a string type could be mapped to a numerical type, which could in turn cause performance improvement when doing FA operations. The string would be transformed by some hashing function to a number. In this case, however, we would need to be aware of collisions, which can occur when doing hashing.

A finite automaton `Fa` can be also considered a member of the `Monoid` typeclass with respect to the union operation. Monoids are structures with an associative binary operation and an identity element. The union operation on two FAs is an associative binary operation. Its identity element is a FA accepting an empty language.

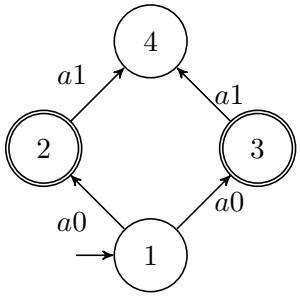
4.2 Input format

One requirement of the library is the possibility to load a finite automaton stored in the Timbuk text format, which is also used by the VATA library [15]. We use the Parsec library, described in section 3.4.4, to parse the Timbuk input format into our data structures. Note that the Timbuk format stores a *tree automaton*, a more general type of automaton than a FA. Such a tree automaton has to be then converted to a FA.

Figure 4.2 shows an example of a simple FA represented in the Timbuk file format. Line 1 starts with the keyword `Ops` and then enumerates all alphabet symbols of the FA. The number on the right-hand side of the colon symbol represents the *arity* of a symbol. Symbol arity is essential for tree automata. In the case of FAs, though, symbol arities are irrelevant and can be safely ignored. The name of the automaton and enumeration of states, final states, and transitions are presented on lines 3–11. Every transition in the Timbuk format has the following structure: `<label> (<src-states>) -> <tgt-state>`, where `<src-states>`, in the FA setting, is a single `source` state and the whole string represents the FA transition `<source>` $\xrightarrow{\texttt{<label>}}$ `<tgt-state>`. The `<src-states>` component may also be omited, in which case it denotes that `<tgt-state>` is an initial state of the FA. For instance, a Timbuk transition on line 7 of Figure 4.2 denotes that `1` is an initial state of the automaton. On the other hand, line 8 represents the transition $1 \xrightarrow{a0} 2$.

```
1     Ops a0:1 a1:1
2
3     Automaton A
4     States 1 2 3 4
5     Final States 2 3
6     Transitions
7     x -> 1
8     a0(1) -> 2
9     a0(1) -> 3
10    a1(2) -> 4
11    a1(3) -> 4
```

(a) A FA encoded in the Timbuk file format.



(b) A FA corresponding to its textual representation presented in (a).

Figure 4.1: Example of a simple finite automaton represented in the Timbuk file format.

The following example is a parser processing one transition encoded in the Timbuk format using Parsec:

```
transition :: Parser Transition
transition = do
  { label <- label
  ; inputStates <- transitionStateList
  ; string " -> "
  ; finalState <- state
  ; return (Transition label inputStates finalState)
  }
```

This parser contains a `do`-block, which sequentially parses individual sub-components of a Timbuk transition. `label` and `state` are simple parsers that parse one or more alphanumeric symbols. `transitionStateList` is a more complex parser composed of other simple parsers. Finally, a new `Transition` data record is then wrapped in a monadic value and returned using the `return` function.

4.3 Operations

One of the main tasks of this thesis is to implement certain operations over finite automata. These operations and their respective algorithms were described in Chapter 2. Table 4.2 lists all the algorithms for reference. As can be seen in the table, we presented two variants of the problems of language inclusion and universality testing. Therefore, we implemented both variants of these algorithms to compare their performance. The naive versions are expected to be inefficient as these algorithms internally use determinization, which has exponential time complexity. Antichain-based versions also internally determinize automata but they try to minimize this costly operation by skipping parts of automata if possible.

All algorithms were described in the form of an imperative pseudocode in Chapter 2. We need to take into account some issues that arise when using Haskell to implement these algorithms. One such an issue is the absence of `while`, `for`, and other iterative constructs in Haskell. In this language, the said constructs are instead implemented using recursion.

Table 4.2: Algorithms presented in Chapter 2.

Operation		Location
Membership		Algorithm 1
Union		Algorithm 2
Intersection		Algorithm 5
Determinization		Algorithm 6
Complement		Algorithm 7
Emptiness		Section 2.9
Language inclusion	Naive	Section 2.10
	Antichain	Algorithm 10
Universality	Naive	Section 2.11
	Antichain	Algorithm 9

This brings up another issue—efficiency. Haskell provides the *tail-recursion* optimization that can eliminate this problem.

A *tail call* is a function call performed as the final action of some function. If such a tail call can be executed recursively, we talk about a *tail-recursive call* [18]. A tail-recursive function is a function with a tail-recursive call. Such a function performs its calculation first, and then it executes the tail-recursive call, passing the results of the current step to the next recursive step. Once the tail-recursive function is ready to perform the next recursive step, it does not need to keep the current stack frame anymore. This allows for a considerable optimization. A compiler can optimize away the creation of stack frames in such a situation and use a much more effective jump instruction instead. This technique resembles a classical loop constructs found in imperative languages. The technique avoids stack overflow and reduces memory consumption. It is therefore vital to write all recursive logic as tail-recursive if possible. However, some recursive constructs are not possible to express as tail-recursive.

Another issue arising from the use of Haskell is that the language uses lazy evaluation by default. It is therefore important to work with lazy evaluation carefully. Such a form of evaluation can lead to performance gains if used correctly by skipping evaluation of parts of finite automata that are unnecessary. When used incorrectly, lazy evaluation can, however, contribute to higher memory consumption as the Haskell runtime stores thunks in memory. Thunks were discussed in Section 3.2. These thunks can then be kept in memory unnecessarily, leading to a considerable memory overhead. Section 3.3 showed some useful tools on how to analyze performance of Haskell programs. Such tools can be also used to track down a location in the code that is responsible for memory leaks caused by lazy evaluation. When such a location is found, we can use certain techniques to suppress lazy evaluation as described in Section 3.2.

Finally, the last issue involves a global state. In imperative languages, the global state can often be accessed from an arbitrary location of the program. This is a huge disadvantage as it can greatly complicate the program logic: we have to worry about every variable's value at some point in time in order to understand the program logic and to ensure it works properly. This problem is greatly reduced in purely functional languages like Haskell. Haskell programs are made of functions that cannot change any global state or variables,

they can only do some computation and then return a result. This restriction actually makes it easier to think about such programs. It also allows to perform aggressive optimizations by compiler as many of those rely on pureness. However, in pure languages, we cannot rely on a global state at various locations of a program to store our data into it. All state must be explicitly passed from one location to another, which can quickly become burdensome. This is especially true for imperative-like algorithms presented in Chapter 2. Therefore, it could be useful to utilize the `State` monad presented in Section 3.4.3 when implementing these algorithms. With the `State` monad, one can define a state that is accessible at any location inside the state monad context without the need to explicitly pass such a state around the program.

Chapter 5

Implementation

In the previous chapter, we discussed the issues encountered when using Haskell to implement certain finite automata operations. We performed analysis and presented two variants of how to implement data structures representing a finite automaton. We also described the Timbuk file format and the Parsec library, which we use to perform parsing of this format. In the last part, we talked about operations that are implemented in the library. Due to the fact that Haskell is a purely functional and lazily evaluated language, we will need to pay special attention during the implementation to be sure that the code is both readable and efficient. This chapter is concerned with the actual implementation of the finite automaton library and is implemented as a part of this Master's thesis. The source code of this library is available on Github [25].

5.1 Library structure

The library uses *Stack* as a development tool [21]. Stack makes developing in Haskell easier by taking care of installing a Haskell compiler, required packages, as well as building, testing, and benchmarking Haskell projects. We use the *GHC* compiler, which is the *de facto* standard compiler for Haskell [7]. The project is split into two main directories: `src` directory containing source code and `tests` directory containing unit tests and benchmarking scripts. For a better explanation, individual parts of the library will be referenced by their Haskell module names in the following sections.

5.2 Data structures

We implemented data structures to represent a tree automaton (module `Types.Fta`) and a finite automaton (module `Types.Fa`). The tree automaton data structure is only used while parsing the Timbuk file format. The file is parsed into a tree automaton, which is then transformed into a corresponding finite automaton. Of course, the file could be parsed into the finite automaton directly. The reason for this detour is to be able to load a tree automaton from a Timbuk format file. We can then simply extend the library to support various tree automata operations in the future.

We implemented two variants of the finite automaton data structure. The first variant is implemented in terms of an immutable single-linked list (`[]`) and the second one in terms of `Set` located in the Haskell library `containers`. We implemented all the FA operations

we discussed in this Master's thesis for both `[]` and `Set` variants. Comparison of both versions in terms of performance, as well as other testing, is available in Chapter 6.

The `Fa` data type (module `Types`) is declared as follows:

```
data Fa sym sta =
  Fa
    { initialStates :: Set sta
    , finalStates :: Set sta
    , transitions :: Set (Transition sym sta)
    } deriving (NFData, Generic)
```

The type derives two typeclasses: `NFData` and `Generic`. They specify that the `Fa` type can be evaluated to the normal form. `Fa` also derives the `Show` typeclass using the Haskell's `instance` keyword. As described in the previous Chapter 4, we do not need to explicitly store the set of all FA states and the alphabet. These can be retrieved from initial and final states and transitions. Alphabet can also be passed externally if needed by certain operations.

Originally, we did not intend to use the `Set` data type from the `containers` library but wanted to use the `Set` data type from the `set-monad` library instead. The `set-monad` library exports `Set` data type and set-manipulating functions [9]. This data type and the functions behave exactly as their counterparts from the `containers` library. In addition, the `set-monad` library extends `Set` by providing `Functor`, `Applicative`, `Monad`, and other instances. The original `Set` of the `containers` library is not a monad due to the limits of Haskell's type system and the way things are currently structured. Specifically, `Set` operations require elements to be instances of the `Ord` typeclass but the `Monad` typeclass signature does not allow for that [6]. Because `Set` from the `set-monad` library is a `Monad` instance, we can conveniently use some of Haskell's language constructs to easily implement set-manipulation logic in our algorithms. More specifically, we can use the `do`-notation and list comprehension to simulate a mathematical set-builder notation. This notation is also used in description of finite automata operations found in Chapter 2. To be able to use list comprehension with `Set`, one can enable the `MonadComprehensions` language extension.

We originally implemented the library with the `set-monad` version of `Set`. Unfortunately, after the implementation, we realized that the `set-monad` library has a significant performance overhead. The library is just a wrapper over the `Set` data type of the `containers` library. When executing any operation, this wrapper converts `Set` from the one representation to another, which is extremely inefficient. Therefore, we finally decided to use the `containers` library version of `Set` despite not having `Functor`, `Applicative`, `Monad` instances available at our disposal.

To overcome the inconvenience of not having the `Monad` instance, we implemented our own version of monad operations: functions `return` and `andThen` (module `Helpers`). The `andThen` function represents the bind operation and is defined as follows:

```
andThen :: Set a -> (a -> Set b) -> Set b
andThen monad f =
  (unions . Set.map f) monad
```

The parameter `monad` is mapped over the function `f` and the result is concatenated. We cannot use `do`-notation nor list comprehension in conjunction with `andThen` but we can use this function to form a chain of computations that simulates these constructs.

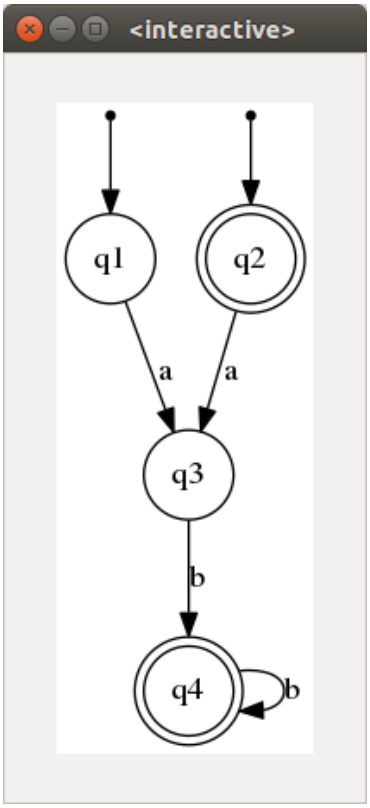


Figure 5.1: Graphical visualization of an example FA using the `displayFA` function.

5.3 Visualization

One of the useful functionalities of a finite automaton library is the ability to visualize and display a finite automaton to the user. We implemented two visualization variants—textual and graphical. Textual visualization simply prints arbitrary automaton on the standard output in the Timbuk format. For this reason, the type `Fa` (module `Types`) is a member of the `Show` typeclass. On the other hand, to display any FA graphically, one can use the unction `displayFa` (module `Vizualization`), which opens a new window and renders an image of the automaton inside the window. This functionality is implemented using the `gtk` library, a Haskell binding to the Gtk+ graphical user interface library [20]. Figure 5.1 shows an example of a simple FA displayed in a window using the function `displayFa`.

5.4 Operations

Finite automata operations are located in the following five different modules:

1. `Operations.Regular` contains all FA operations supported by the library: union, intersection, determinization, complement, and decision problems of membership, emptiness, language inclusion, and universality. The decision problems of language inclusion and universality are implemented using the naive algorithm versions (Sections 2.10 and 2.11).

2. `Operations.WithExternalSymbols` is similar to the module `Operations.Regular`. There is one important difference though: every operation in this module has one

extra parameter of the type `Set sym`, which is used to pass an external alphabet to the operation. This external alphabet is then used instead of the implicit alphabet, which is retrieved from transitions. This module does not contain operations for which this extra parameter is irrelevant (union and the decision problem of membership).

3. `Operations.Product` contains a single operation, `union`. This version of union is called the product union and was introduced in Algorithm 3. Its type definition is `Fa sym sta1 -> Fa sym sta2 -> Fa sym (Set sta1, Set sta2)`. Unlike the classical version of union, this operation accepts two FAs that may have different types of states `sta1` and `sta2`, respectively.

4. `Operations.Antichain.Universality` contains the antichain-based version of the language universality testing operation (Algorithm 2.2).

5. `Operations.Antichain.Inclusion` contains the antichain-based version of the language inclusion testing operation (Algorithm 2.3).

The operations shown in Chapter 2 are specified in the form of an imperative pseudocode. Due to this fact, our implementation uses list comprehension and a state monad. These techniques simplify writing of imperative-like code that performs set manipulation. Especially, the determinization and antichain-based language inclusion and universality operations use a state to track which FA states have been already visited and which are about to be processed. Therefore, we use the state monad to keep such a state inside the monad context. This renders an explicit passing of the state unnecessary.

In Section 3.2, we talked about lazy evaluation where we discussed its advantages and disadvantages with regard to performance. In the next chapter, we will discuss performance results of lazy evaluation as well as influence of suppressing lazy evaluation on performance. Our implementation uses two methods to suppress lazy evaluation: the first method is the use of bang patterns. Bang patterns offer a convenient way how to force evaluation of function parameters to normal form. The second method is the use of the `deepseq` library that contains the function `force` [4]. This function forces evaluation of an expression to the normal form. This function is elaborated on in the next chapter.

5.5 Unit testing

To have higher confidence that the implemented operations were implemented correctly, we wrote a set of unit tests to check the functionality of the operations. We chose Hspec [12] as a testing framework that integrates other Haskell testing frameworks (QuickCheck, SmallCheck, and HUnit) under the one set of API. Our unit tests are located in modules `Tests.Operations` and `Tests.Parsing`. All tests follow the same pattern of loading some Timbuk format files, which are located in the directory `tests/Examples`, and subsequently testing some functionality. Tests in the module `Tests.Parsing` check whether the parsing functionality is correct. Tests in the module `Tests.Operations` always perform some specific operation on FAs and check afterwards the correctness by making sure that the resulting automaton accepts or rejects certain strings.

Chapter 6

Evaluation

The previous chapter discussed the library implementation. This chapter is concerned with performance testing of this implemented library. We call this library "Automata" in the following text. The testing was performed on a laptop with Intel(R) Core(TM) i5-2410M CPU at 2.30 GHz and 8 GiB of memory at 1333 MHz running Ubuntu 16.04.2 LTS.

During testing, we had to take into account the fact that the Haskell language is evaluated lazily by default. This can considerably complicate the testing procedure due to the property of lazy evaluation: the fact that code is evaluated only when its result is actually needed. We paid special attention to the issue of lazy evaluation and therefore created the following function to evaluate our tests:

```
benchmarkForce :: (NFData a, NFData p)
               => (p -> a)
               -> p
               -> IO a
benchmarkForce action param = ...
```

The `NFData` typeclass marks types that have ability to be evaluated to normal form and the parameter `action` is the function to be benchmarked. `benchmarkForce` first ensures that the function's parameters are evaluated to the normal form. This is an important step, because, for example, if a benchmarking function starts to measure the duration of a complement operation of a finite automaton, the library at that moment can just start parsing the input automaton from a file. This is possible due to the nature of lazy evaluation, in which a function is evaluated at the last possible moment, only when its result is requested. This can render the testing results useless as the testing procedure could be measuring execution time of the complement operation as well as other things irrelevant to the actual benchmarking (like automaton parsing or hard drive access times). The execution time of automaton parsing is indeed mostly irrelevant to the overall efficiency. What matters for us is the efficiency of the FA operations. To avoid this problem, we use the library `deepseq` [4]. This library contains the function `force :: NFData a => a -> a`, which evaluates its argument `a` to the normal form. Therefore, the `benchmarkForce` function first evaluates its argument `param` to the normal form before doing anything else. Afterwards, `benchmarkForce` starts measuring the execution time, passes the argument `param` into the `action` function, and forces evaluation of this function into the normal form. After the function output is available, the measurement stops. The same logic applies to testing arbitrary binary functions, in which case the function has the following declaration:

```
benchmarkForce :: (NFData a, NFData p, NFData r)
               => (p -> r -> a)
               -> p
               -> r
               -> IO a
benchmarkForce action param1 param2 = ...
```

In this case, parameters `param1` and `param2` are to be passed into the benchmarked function `action`.

We performed testing on a sample of 20 finite automata saved in the Timbuk file format from [14]. These automata have an evenly distributed number of states in the range of 1 to 400. We used the `benchmarkForce` function to test the implemented library operations. In the previous chapter, we noted that the data structures of the Automata library were implemented using both `[]` and `Set` with the intention to compare performance of those data structures. Therefore, benchmarking tests of both solutions are presented.

We performed identical performance testing with the VATA library to compare it with the Automata library. VATA implements its own benchmarking functionality, which measures execution time of an operation to be executed. During benchmarking, VATA was executed as follows:

```
vata -t -r expl_fa <operation> <file(s)>
```

This command turns on a benchmarking mode, instructs VATA to use the explicit representation of a finite automaton, and executes the operation `<operation>` on file(s) `<file(s)>`. Note that VATA does not support all the operations that Automata supports. Therefore, we benchmarked VATA with regard to union, intersection, and antichain-based language inclusion and universality testing operations. VATA also does not directly support the universality testing operation. To overcome this problem, we reduced the universality testing operation to testing language inclusion as presented in Section 2.11, to support this operation in VATA.

We tried to present the benchmark results in the best possible way to give a good overview of the performance of individual operations. When we executed some benchmark with FAs that had an increasing number of states, the results had outliers. As expected, the execution time of individual operations depends more on the structure of input automata than on the number of states. Therefore, our results are presented as medians, last deciles, sorted sequences, etc. to suppress the noise in the data and to give a clear picture.

Table 6.1 presents a performance overview. To get an intuition of an average performance behaviour of FA operations, we ran every operation on the sample set and computed the presented indicators. We used the simple version of the union operation instead of the product union version presented in Section 2.5. We also used the antichain-based versions of the language inclusion and universality operations instead of the naive versions. The naive variants were much slower than the corresponding antichain-based operations. Actually, the execution time of a test suite for them would be extremely large and, therefore, we did not consider them in this table. However, the naive variants are plotted in graphs that are shown later in this chapter. The first column of the table represents operations that were tested. The second column of the table represents a library on which the test suite was performed: 1) VATA: the VATA library, 2) Automata (`[]`): the Automata library implemented in terms of the `[]` data structure, and 3) Automata (`Set`): the Automata library implemented in terms of the `Set` data structure. Other columns of the table present the median values (50%), the last deciles (90%), the last percentiles (99%), and the maxi-

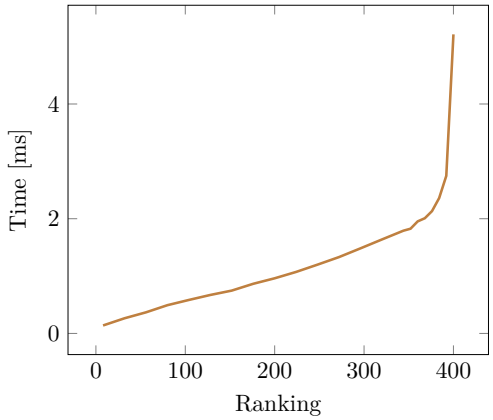
Table 6.1: Overview of the FA operations performance results.

Operation	Library	Required time (milliseconds)			
		50%	90%	99%	100%
Union	VATA	0.549	0.930	1.054	1.085
	Automata (`[]`)	14.465	51.042	75.785	81.346
	Automata (`Set`)	5.967	18.320	21.785	22.310
Intersection	VATA	0.043	1.257	3.143	4.097
	Automata (`[]`)	20.128	88.160	135.992	149.308
Determinization	Automata (`[]`)	122.238	573.196	684.678	799.672
	Automata (`Set`)	59.781	211.919	248.382	287.973
Complement	Automata (`[]`)	125.299	579.269	693.527	810.212
	Automata (`Set`)	58.491	220.620	258.893	296.217
Inclusion	VATA	3.251	5.231	5.839	5.995
	Automata (`[]`)	3.953	14.890	49.470	71.703
Universality	VATA	0.666	1.091	1.201	1.218
	Automata (`[]`)	4.591	13.113	15.460	15.853
	Automata (`Set`)	4.502	13.379	15.939	16.306

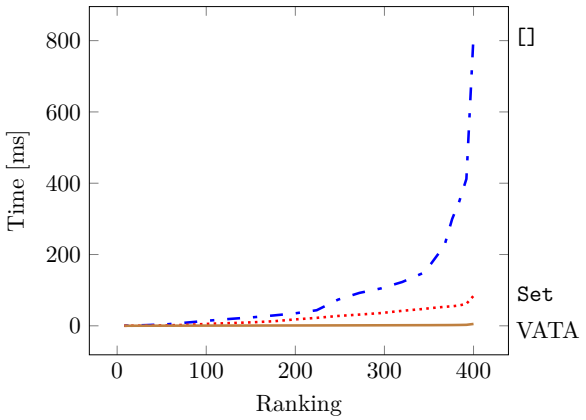
mum values (100%) of the execution time. For example, the value 0.549 in the first row of the column 50% means that 50% examples of the sample set required less than 0.549 ms to execute when performing union operation using the VATA library.

The VATA library is clearly many times faster than the Automata library in the all cases. The Automata (`Set`) variant of the library is considerably faster in union, determinization and complement operations then the `[]` variant. However, this does not hold for the universality operation. In the case of universality, both variants have almost identical performance.

We also focused on the suppression of lazy evaluation to improve performance of the FA operations. In Section 3.2, we discussed the reason why it can be beneficial to resort to such a thing. We used the profiling tools presented in Section 3.3 to detect problematic locations with regard to performance. It turns out that by suppressing lazy evaluation in any location in the source code did not have any significant influence on performance. Therefore, we do not present benchmarking results of such optimization here. To detect problematic source code locations, we used profiling tools provided by the GHC platform, which were discussed in Section 3.3. We mostly used the `-p` switch to collect the time and allocation profiling report. This report provides an overview of the program's runtime behaviour. It gives us information of the proportion of time and space each function was responsible for. It also contains the cost center report, structured as a call graph. This cost center report is valuable tool to track down a problematic source code location with regard to performance. The following sections discuss performance results of individual operations that were benchmarked.



(a) VATA



(b) VATA (brown, solid) and Automata with `[]` (blue, dash-dotted) and `Set` (red, dotted) variants

Figure 6.1: Performance results of the union operation.

6.1 Language union

Figure 6.1 shows performance results of the union operation, which was presented in Algorithm 2. We took our sample set of 20 FAs and created 400 FA pairs out of them. These pairs were run through the union operation and the execution time of each pair was recorded. Afterwards, the execution times were sorted to get an increasing function. The horizontal axis of both plots of the figure represents the ranking of pairs from fastest to slowest. The vertical axis represents the execution time in milliseconds. The VATA library is orders of magnitude faster than the Automata library for both variants of data structures (`[]` and `Set`). However, `Set` variant is substantially faster than the `[]` variant. We suppose that the difference in performance of these variants is due to the expected asymptotic time complexity of operations of both data structures as we described in Chapter 4.

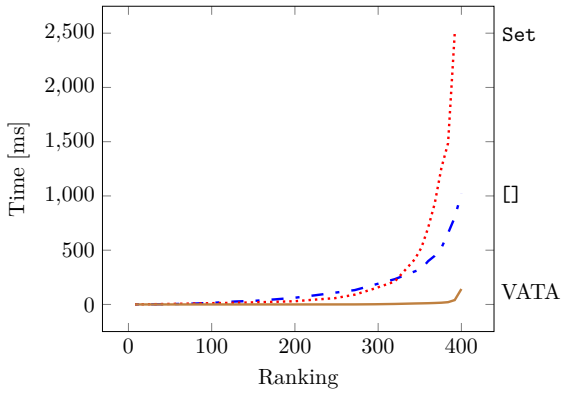


Figure 6.2: Performance results of the intersection operation for VATA (brown, solid) and Automata with `[]` (blue, dash-dotted) and `Set` (red, dotted) variants.

6.2 Language intersection

Figure 6.2 shows performance results of the intersection operation, which was presented in Algorithm 5. The graph was created in the same way as the figure in the previous section. Similarly to the union operation results, the VATA library outperforms the both Automata library variants with regard to the intersection operation. The `Set` variant is, however, substantially slower than the `[]` variant. GHC's profiler showed that the the `Set` variant spent most resources in the `andThen` function. We discussed this function in Section 5.2. We were not able to identify the reason why `andThen` was performing so poorly.

6.3 Language complement

We do not show graph of the determinization operation. Complement operation is implemented using determinization and, therefore, performance results are almost identical. Figure 6.3 shows performance results of the complement operation, which was presented in Algorithm 7. Because it is a unary operation, we did not have to create pairs of FAs from the sample set. We only ran the sample set through the complement operation and recorded the execution time for each FA. Afterwards, we sorted the results and plotted them in the graph. Because VATA does not support the complement operation, we only tested Automata with the `[]` and `Set` variants. The `Set` variant was faster for FAs with more than 5 states.

6.4 Language inclusion

Figure 6.4 shows performance results of the naive and the antichain-based language inclusion testing variants, which were presented in Section 2.10 and Algorithm 10, respectively. This figure was created in the same way as Figure 6.1 and 6.2. As with the other performance results, this operation implemented in VATA was significantly faster than in Automata. The `Set` variant was slower than the `[]` variant due to bad performance of the `andThen` function. As expected, the naive variant was slowest.

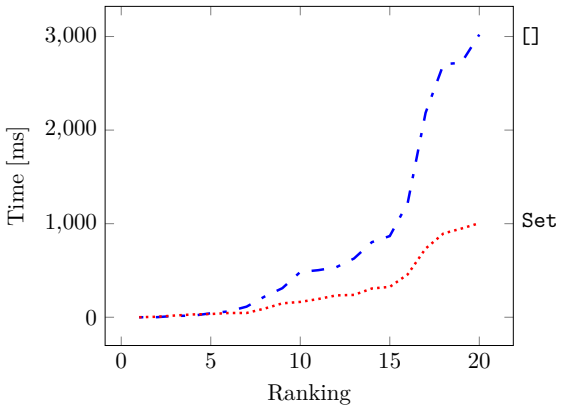
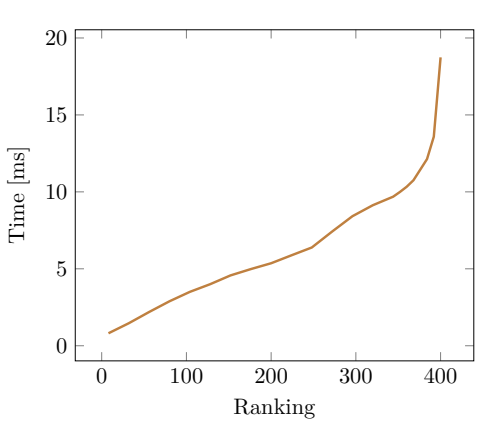
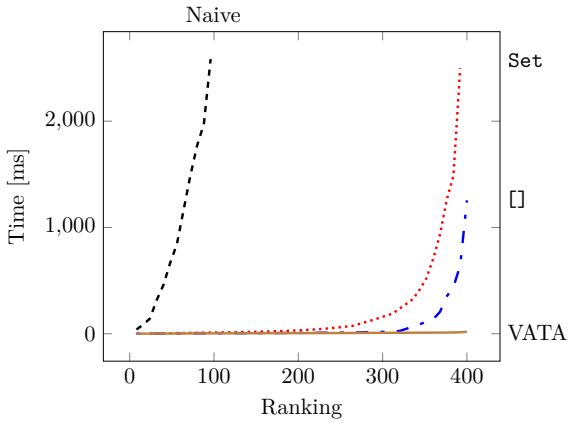


Figure 6.3: Performance results of the complement operation for Automata with `[]` (blue, dash-dotted) and `Set` (red, dotted) variants.



(a) VATA

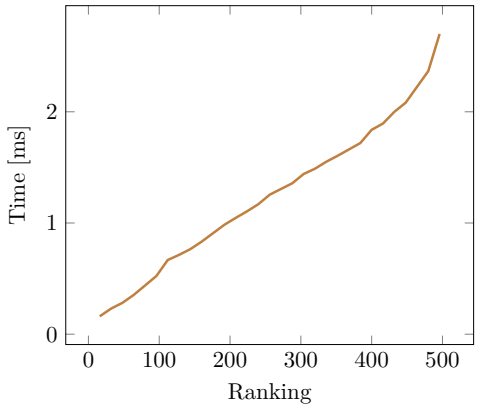


(b) VATA (brown, solid) and Automata with `[]` (blue, dash-dotted), `Set` (red, dotted), and naive (black, dashed) variants

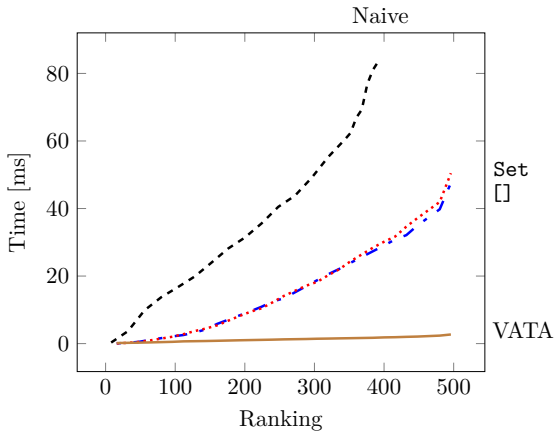
Figure 6.4: Performance results of the language inclusion testing operation.

6.5 Language universality

Figure 6.5 shows performance results of the naive and the antichain-based language inclusion testing variants, which were presented in Section sec:universality and Algorithm 10, respectively. We used different data to conduct this benchmark. We created 500 universal FAs with an ascending number of states in the range 1–500. Every created FA is a single-linked list where the head element is the initial state. Beside that, there are no other initial nor final states. We processed these FAs in the same way as in the other benchmarks. Figure 6.5 shows that VATA was significantly faster than both Automata variants. Performance of `[]` and `Set` variants were almost identical. As expected, the naive variant was slowest.



(a) VATA



(b) VATA (brown, solid) and Automata with `[]` (blue, dash-dotted), `Set` (red, dotted), and naive (black, dashed) variants

Figure 6.5: Performance results of the universality testing operation.

Chapter 7

Conclusion

The aim of this Master's thesis was to introduce, discuss, and analyze the theoretical and design considerations regarding a finite automata library for the use in the field of formal verification. We also implemented this library in Haskell, a popular non-strict functional language. The library supports operations of union, intersection, determinization, complement, and the decision problems of membership, emptiness, language inclusion, and universality. We also implemented more efficient variants of testing language inclusion, and universality using the antichain-based approach. This Master's thesis also explained selected topics of the Haskell programming language relevant to the library implementation. It elaborated on the reduction and evaluation logic of Haskell's runtime environment, on non-strict semantics, and lazy evaluation. Profiling tools for Haskell were presented as well as a description of monads.

The main purpose was to implement an efficient purely functional library written in Haskell, which effectively uses lazy evaluation in library operations. Beside that, we performed a benchmarking of the implemented library and compared its performance to VATA. Benchmarks demonstrated that VATA was significantly faster in all operations despite optimizations we implemented. The greatest influence on performance of the implemented library was the right choice of an efficient data structure. Therefore, beside the implementation using an immutable single-linked list, we also implemented a variant of the library that utilizes `Set`, an efficient implementation of a set using a size-balanced binary tree. We also focused on a correct use of lazy evaluation in the library. We found out that our experiments with suppressing lazy evaluation at certain locations did not yield any significant improvement in performance of the finite automata operations.

Future work can be oriented in more directions: we can implement and test the library with other purely functional data structures, such as some kind of a lookup table or a hash set. The library can also offer more FA operations, such as FA minimization. Finally, we could also extend the library with tree automata operations.

Bibliography

[1] *Automata*. [Online; visited 01.01.2017].
Retrieved from: https://github.com/AutomataDotNet/Automata

[2] Aarhus University: *Automaton*. [Online; visited 01.01.2017].
Retrieved from: http://www.brics.dk/automaton/

[3] Abdulla, P. A.; Chen, Y.-F.; Holík, L.; Mayr, R.; Vojnar, T.: *When Simulation Meets Antichains*. In Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 6015, pp. 158-174, Springer Berlin Heidelberg. 2010.

[4] Collective of authors: *deepseq: Deep evaluation of data structures*. [Online; visited 06.04.2017].
Retrieved from: https://hackage.haskell.org/package/deepseq

[5] De Wulf, M.; Doyen, L.; Henzinger, T. A.; Raskin, J.-F.: *Antichains: A New Algorithm for Checking Universality of Finite Automata*. In Proc. of CAV'06, LNCS 4144, Springer. 2006.

[6] Eaton, F.: *Why is Data.Set not a Monad?* [Online; visited 29.04.2017].
Retrieved from:
https://mail.haskell.org/pipermail/libraries/2007-May/007486.html

[7] Gamari, B.: *The Glasgow Haskell Compiler*. [Online; visited 27.04.2017].
Retrieved from: https://www.haskell.org/ghc/

[8] Gill, A.: *Hood: Debugging by observing in place*. [Online; visited 26.12.2016].
Retrieved from: https://hackage.haskell.org/package/hood

[9] Giorgidze, G.: *Haskell: Data.Set.Monad*. [Online; visited 11.05.2017].
Retrieved from: https:
//hackage.haskell.org/package/set-monad-0.2.0.0/docs/Data-Set-Monad.html

[10] haskell.org: *Haskell Language*. [Online; visited 02.01.2017].
Retrieved from: https://www.haskell.org/

[11] HaskellWiki: *HaskellWiki*. [Online; visited 25.12.2016].
Retrieved from: https://wiki.haskell.org/Haskell

[12] Hspec contributors: *Hspec: A Testing Framework for Haskell*. [Online; visited 30.04.2017].
Retrieved from: https://hspec.github.io/

[13] Leijen, D.: *Parsec, a fast combinator parser.* [Online; visited 01.01.2017]. Retrieved from: http://research.microsoft.com/en-us/um/people/daan/download/parsec/parsec-letter.pdf

[14] Lengál, O.: *automata-benchmarks.* [Online; visited 18.05.2017]. Retrieved from: https://github.com/ondrik/automata-benchmarks/tree/master/nfa

[15] Lengál, O.; Šimáček, J.; Vojnar, T.: *The VATA Tree Automata Library.* [Online; visited 03.01.2017]. Retrieved from: http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/

[16] Lipovača, M.: *Learn You a Haskell for Great Good!* [Online; visited 26.03.2017]. Retrieved from: http://learnyouahaskell.com

[17] Meduna, A.: *Automata and Languages: Theory and Applications.* Springer. 2000. ISBN 978-1-4471-0501-5.

[18] O'Sullivan, B.; Goerzen, J.; Stewart, D.: *Real World Haskell.* O'Reilly. 2009. ISBN 978-0-596-51498-3.

[19] Schneider, K.: *Verification of Reactive Systems.* Springer. 2013. ISBN 978-3-6621-0778-2.

[20] Simon, A.; Coutts, D.: *gtk: Binding to the Gtk+ graphical user library interface.* [Online; visited 10.04.2017]. Retrieved from: https://hackage.haskell.org/package/gtk

[21] Stack contributors: *The Haskell Tool Stack.* [Online; visited 27.04.2017]. Retrieved from: https://haskellstack.org

[22] Stack Overflow: *Haskell: How does non-strict and lazy differ?* [Online; visited 25.12.2016]. Retrieved from: http://stackoverflow.com/questions/7140978/haskell-how-does-non-strict-and-lazy-differ

[23] University of Glasgow: *Haskell: Data.List.* [Online; visited 06.04.2017]. Retrieved from: https://hackage.haskell.org/package/base-4.9.1.0/docs/Data-List.html

[24] University of Glasgow: *Haskell: Data.Set.* [Online; visited 06.04.2017]. Retrieved from: https://hackage.haskell.org/package/containers-0.5.10.2/docs/Data-Set.html

[25] Říha, J.: *An efficient functional library for finite automata.* [Online; visited 27.04.2017]. Retrieved from: https://github.com/jakubriha/automata/