On Determinisation of Pushdown Automata and Conversion of Regular Tree Expressions to Determinisable Pushdown Automata

by

Ing. Radomír Polách

A dissertation thesis submitted to the Faculty of Information Technology, Czech Technical University in Prague, in partial fulfilment of the requirements for the degree of Doctor.

Dissertation degree study programme: Informatics
Department of Theoretical Computer Science

Prague, June 2019
Supervisor:
Doc. Ing. Jan Janoušek, Ph.D.
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Copyright © 2019 Ing. Radomír Polách

ii
Abstract and contributions

Trees are one of the fundamental and well-studied data structures in Computer Science. Arbology uses pushdown automata (PDAs) which read linearised notations of trees, as its suitable model of computation.

Regular tree expressions (RTEs) are a formalism for describing regular tree languages (RTLs) whose standard model of computation is usually a finite tree automaton. It has been proven that the class of regular tree languages is a proper subclass of tree languages whose linear notations can be accepted by deterministic (string) PDAs.

In this dissertation thesis, two new algorithms for converting RTEs to equivalent real-time height-deterministic PDAs (rHPDAs) that accept the trees in their linear notation are presented.

Visibly pushdown automata (VPDAs) introduced by Alur and Madhusudan in 2004 [6], are PDA whose pushdown operations are determined by the input symbol, where the input alphabet is partitioned into three parts for the push, pop, and local pushdown operations. It is well known that nondeterministic VPDAs can be determinised.

Height-deterministic pushdown automata (HPDAs), a natural generalisation of VPDAs where for any given input string the stack heights during any (nondeterministic) computation are a priori fixed, were introduced by Dirk Nowotka and Jirí Srba in 2007 [37]. This can be seen as a generalisation of the property of VPDAs where the input alphabet is partitioned into three parts for a push, pop and local pushdown operations which implies this condition. The subgroup of rHPDAs was proven to be determinisable.

In this dissertation thesis, new algorithms for the determinisation of VPDAs and rHPDAs are presented. The algorithms are based upon theoretical proofs of determinisation and result in reasonably small deterministic PDAs for most practical examples. This is achieved in a way that only necessary and accessible states and pushdown symbols are computed and constructed during the determinisation.
In particular, the main contributions of the dissertation thesis are as follows:

- Two new algorithms for converting RTEs to equivalent rHPDAs that accept the trees in their linear notation. First is based on the Thompson’s construction algorithm for a regular expression (RE), the second is based on the Glushkov’s construction algorithm for a RE.

- A new algorithm for the VPDAs determinisation.

- A new algorithm for the rHPDAs determinisation.

Keywords:

Arbology, visibly pushdown automaton, height-deterministic pushdown automaton, real-time height-deterministic PDA, regular tree expression, conversion, determinisation, minimisation, trees.
Acknowledgements

First of all, I would like to express my gratitude to my dissertation dissertation thesis supervisor, doc. Ing. Jan Janoušek, Ph.D. He has been a constant source of encouragement and insight during my research and helped me with numerous problems and professional advancements.

I want to thank prof. Ing. Bořivoj Melichar, DrSc. for his help.

Special thanks go to the staff of the Department of Theoretical Computer Science, who maintained a pleasant and flexible environment for my research. My research has also been partially supported by several sources: by the Czech Science Foundation as project No. 13-03253S, by the Grant Agency of the Czech Technical University in Prague, grant No. SGS17/209/OHK3/3T/18.

I want to express thanks to my colleagues from Arbology Group and Prague Stringology Group.

Finally, my greatest thanks go to my family members, for their infinite patience and care.
Dedication

To all my family and friends.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Regular tree expressions</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 Pushdown automata determinisation</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Problem statement</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Formal definition of presented problems</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Related work/Previous results</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Contributions of the Dissertation thesis</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Structure of the Dissertation thesis</td>
<td>4</td>
</tr>
<tr>
<td><strong>2 Theoretical Background</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Alphabet, string</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Language</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Finite automaton</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Pushdown automaton</td>
<td>8</td>
</tr>
<tr>
<td>2.5 Tree, tree pattern, linear notations and their properties</td>
<td>9</td>
</tr>
<tr>
<td>2.6 Regular tree expression</td>
<td>11</td>
</tr>
<tr>
<td><strong>3 State-of-the-Art and Related work</strong></td>
<td>15</td>
</tr>
<tr>
<td>3.1 Determinisation of finite automaton</td>
<td>15</td>
</tr>
<tr>
<td>3.1.1 Finite automaton</td>
<td>15</td>
</tr>
<tr>
<td>3.1.2 Input-driven pushdown automaton</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Determination of pushdown automaton</td>
<td>17</td>
</tr>
<tr>
<td>3.2.1 Normalisation of pushdown automaton</td>
<td>18</td>
</tr>
</tbody>
</table>

---

**Glossary**

---

vii
List of Figures

4.1 Various $\lambda$ relations of states ............................................. 30
4.2 The $\lambda^*$ relation computed for state $9$ ................................ 30
4.3 The $\lambda^*$ relation computed for all states ............................... 31
4.4 Connecting $T_\sigma$ between locally connected states ....................... 31
4.5 Nondeterministic VPDA $a_1$ from Example 4.1 .............................. 35
4.6 The resulting deterministic PDA $d_1$ from Example 4.1 .................... 36
4.7 Nondeterministic rHPDA $a_2$ from Example 4.2 ............................. 42
4.8 The resulting deterministic PDA $d_2$ from Example 4.2 .................... 43
4.9 The resulting deterministic PDA $d_5$ from Example 4.2 .................... 43
4.10 Nondeterministic PDA for RTE1 from Example 4.3 ......................... 45
4.11 Deterministic PDA for RTE1 from Example 4.3 ............................. 45
4.12 Nondeterministic PDA for TP1 from Example 4.4 ......................... 46

5.1 Tree language described by RTE $r$ ............................................. 54
5.2 Conversion of basic symbols from RTE ....................................... 54
5.3 Converting $A + B$ to fragment of equivalent PDA ......................... 54
5.4 Converting $(A_0\Box_1A_1\Box_1A_5)\Box$ to fragment of equivalent PDA 55
5.5 Converting $(A_0\Box_1A_1\Box_1A_5)\Box_{i,*}$ to fragment of equivalent PDA 55
5.6 Nondeterministic PDA $M_r$ created from RTE $r$ ........................... 58
5.7 Deterministic PDA $M_d$ .............................................................. 59
5.8 A directed, rooted, labelled, ranked and ordered tree over $A = \{a2,b0\}$ 61
5.9 Examples of RTEs ................................................................. 61
5.10 RTE $E'$ from Figure 5.9a ....................................................... 64
5.11 A RTE and its equivalent PDA .................................................. 66
5.12 Sample RTE with First set and substitution mapping for important nodes 68
5.13 PDA equivalent to a RTE from Figure 5.12 with transitions grouped by type 69
List of Tables

4.1 Construction of deterministic PDA $d_1$ from Example 4.1 . . . . . . . . . . . . 35
4.2 Comparison of nondeterministic and deterministic PDA for RTE problem . . . 44
4.3 Comparison of nondeterministic and deterministic PDA for TP problem . . . 44
4.4 Comparison of the upper bound and the improved per PDA upper bound . . . 46
List of Algorithms

3.1 Finite automaton determinisation algorithm ........................................... 16
3.2 Input-driven pushdown automaton determinisation algorithm ...................... 17

4.1 Visibly pushdown automaton determinisation algorithm ........................... 33
4.2 Real-time height-deterministic PDA determinisation algorithm .................. 40
4.3 Real-time height-deterministic PDA determinisation algorithm 2 .................. 41

5.1 Construction of PDA from RTE ................................................................. 56
5.2 Computation of Follow($E'$, $a$) in a single pass ...................................... 63
5.3 PDA accepting linearised trees described by a RTE $E$ ................................. 65
5.4 Improved PDA accepting linearised trees described by a RTE $E$ .................... 67
**Glossary**

**Arbology** Arbology is an algorithmic discipline for dealing with processing trees, preferably by means of pushdown automata. It was officially introduced at London Stringology Days 2009 for the first time.

**CFL** context-free language.

**DFA** deterministic finite automaton.

**FA** finite automaton.

**FTA** finite tree automaton.

**HPDA** height-deterministic pushdown automaton.

**IDPDA** input-driven pushdown automaton.

**LBA** linear bounded automaton.

**NFA** nondeterministic finite automaton.

**PDA** pushdown automaton.

**RE** regular expression.

**rHPDA** real-time height-deterministic PDA.

**RL** regular language.

**RTE** regular tree expression.
**RTL** regular tree language.

**Stringology** The term (for the first time used by Zvi Galil in 1984) denotes a science on algorithms on strings and sequences.

**TA** tree automaton.

**TM** Turing machine.

**TP** tree pattern.

**VPDA** visibly pushdown automaton.

**VPL** visibly pushdown language.
Chapter 1

Introduction

1.1 Motivation

Trees, a fundamental structure in information processing, with their usages ranging from data storage in markup languages to internal representation of programs by compilers are a natural way of modelling structured data. Processing of trees and tree languages is, therefore, an essential part of Computer Science. Various formal models of computation for processing trees already exist - be it tree automaton (TA) \[16\] or Stringology-based algorithms. Arbology \[8\] presents a systematic approach to the study of algorithms for processing of trees by means of pushdown automata (PDAs).

1.1.1 Regular tree expressions

The theory of formal string (or word) languages and the theory of formal tree languages are essential parts of the theory of formal languages. While the models of computation of string languages theory are finite automata (FAs), pushdown automata (PDAs), linear bounded automata (LBAs) and Turing machines (TMs), the most famous models of computation of the tree language theory are various kinds of tree automata (TAs) \[16\]. Trees, however, can also be seen as strings, for example in their prefix (preorder) or postfix (postorder) notation. Recently, it has been shown that the deterministic PDA is an appropriate model of computation for labeled ordered, ranked trees in postfix notation, and that the trees in postfix notation, acceptable by deterministic PDAs form a proper superclass of the class of regular tree languages (RTLs) which are accepted by finite TAs \[20\ [19\].

Regular tree expressions (RTEs) are a natural formalism for describing RTLs. RTE matching is an important operation in Computer Science with a number of useful applications oriented on searching in trees, manipulation with tree content or describing tree-like structures. Description of such RTEs and their conversion to...
1. Introduction

PDAs is presented. The systematic approach to the construction of these RTE matchers is based on a combination of linear tree notation called bar notation with height-deterministic pushdown automata (HPDAs) and is very similar to matching regular expressions for strings using FAs.

RTE matching is often declared to be analogous to the problem of regular expression (RE) (string) matching. FAs constructed for REs represents one of the primary approaches used for RE matching. The main advantage of this kind of FAs is that deterministic string matching automaton can be constructed in time linear to the size of the RE and the search phase is in time linear to the input text. Although there are a few RTE matching methods [29, 30, 35], they fail to present a systematic and straightforward approach with a linear time searching phase, which would also be directly analogous to the basic RE matching methods. In this dissertation thesis, a systematic approach to the construction of RTE matchers represented by deterministic PDAs is presented. The PDA reads subject trees in their bar notation [25, 34].

The presented PDAs matching RTEs have the following property. They are height-deterministic, i.e., for each run on the same input the height of pushdown store is the same in all automata’s nondeterministic paths, meaning that even the automata are non-deterministic, their pushdown store height in every moment is predetermined by the input. This means that the presented PDAs can be converted into deterministic PDAs and match in time linear to the input text [37].

1.1.2 Pushdown automata determinisation

Pushdown automata (PDAs) which accept context-free languages (CFLs) are one of the fundamental models of computation of the Theory of formal languages and automata [24]. Every nondeterministic finite automaton (FA) which accepts a regular language, can be determinised and the theory of the determinisation of FAs is simple and well-researched: states of an equivalent deterministic FA represent so-called deterministic subsets of states of a given nondeterministic FA [24]. The general determinisability does not hold for the case of all types of nondeterministic PDAs. The class of deterministic CFLs is a proper subclass of CFLs, i.e., for some nondeterministic PDAs their equivalent deterministic versions do not exist. Generally, it is not known how to decide for a given nondeterministic PDA whether there exists a deterministic equivalent or not. There is a lack of results in the theory of the determinisation of nondeterministic PDAs although such results would be usable, e.g., when constructing practical deterministic algorithms from nondeterministic PDAs.

It is proven that several subgroups of PDAs can be determinised. The determinisation of input-driven PDAs is trivial as they are just the naive extension of the FAs. For other groups: visibly pushdown automata (VPDAs) and
height-deterministic pushdown automata (HPDAs) the problem is more challenging. VPDAs were proven determinisable in \[6, 45\]. Then later subgroup of HPDAs called real-time height-deterministic PDAs (rHPDAs) were proven determinisable in \[37\]. These determinisation methods use principles that are also used in the well-known determinisation of FAs: states of the equivalent deterministic automata are represented by so-called deterministic subsets \[24\]. Alur and Madhusudan \[6\] presented the proof of the determinisability of a given nondeterministic VPDA with \(n\) states by creating a cartesian product consisting of all possible states and then creating deterministic subsets, which resulted in \(2^{n^2+n}\) states of the deterministic version of the PDA. Nowotka and Srba \[37\] modified the proof for the rHPDAs with similar properties. The properties of determinisation of VPDAs was later improved in \[45\], where the upper bound for the number of states was improved from \(2^{n^2+n}\) to \(2^{n^2}\) and the upper bound for the number of pushdown store symbols was lowered from \(|A_c|2^{n^2+n}\) to \(|A_c|2^{n^2}\). These improvements are also applicable for rHPDAs.

In this dissertation thesis, new algorithms are presented. One for each group of the restricted PDAs classes: VPDAs and rHPDAs. These are the first practical algorithms for these restricted PDAs ever presented. They are based on theoretical background described in \[6, 37\]. The deterministic automata produced using described methods are comparably small on practical examples. Only necessary and accessible states and pushdown symbols of the deterministic PDA are computed and constructed during the determinisation, which is done by analysing which states are used in transitions on the same level of the nesting of pushdown operations and which pushdown store symbols can appear at the top of the pushdown store for each state.

\section{Problem statement}

Dissertation thesis is mainly focused on pushdown automata (PDAs) determinisation and regular tree expressions (RTEs) conversion into determinisable PDAs. A complete explanation of the topic is described in chapter 4 and chapter 5, respectively.

\subsection{Formal definition of presented problems}

The dissertation thesis proposes new solutions to the following problems:

1. Given a nondeterministic visibly pushdown automaton (VPDA) accepting language \(L\) construct a deterministic pushdown automaton (PDA) accepting language \(L\).
2. Given a nondeterministic real-time height-deterministic PDA (rHPDA) accepting language \(L\) construct a deterministic PDA accepting language \(L\).
1. Introduction


1.3 Related work/Previous results

This dissertation thesis is closely related to Arbology [8], which presents a systematic approach to the study of algorithms for processing of trees by means of pushdown automata (PDAs).

The chapter 5 of the dissertation thesis which focuses on determinisation of PDAs is closely related to work done by Alur and Madhusudan proposed the class of visibly pushdown languages (VPLs) [6] and to the work done by Nowotka and Srba [37].

The chapter 4 of the dissertation thesis which focuses on the conversion of regular tree expressions (RTEs) into determinisable PDAs is based primarily on TATA [16].

Related work and previous results are covered in detail in chapter 3.

1.4 Contributions of the Dissertation thesis

1. Two new methods for converting regular tree expressions (RTEs) into the determinisable classes.


3. A new method for determinisation of real-time height-deterministic PDAs (rHPDAs).

1.5 Structure of the Dissertation thesis

The dissertation thesis is organized into 6 chapters as follows:

1. Introduction: Describes the motivation behind our efforts together with our goals. There is also a list of contributions of this dissertation thesis.

2. Theoretical Background: Introduces the reader to the necessary theoretical background.


4. Regular tree expressions (RTEs) conversion: Presents the results of the author in the area of RTEs conversion to determinisable pushdown automata (PDAs). Two new algorithms are proposed and defined.
5. **Pushdown automata (PDAs) determinisation**: Presents the results of the author in the area of PDAs determinisation. Two new algorithms are proposed and defined for both the visibly pushdown automata (VPDAs) and height-deterministic pushdown automata (HPDAs).

6. **Conclusions**: Summarizes the results of our research, suggests possible topics for further research and concludes the dissertation thesis.
Chapter 2

Theoretical Background

Basic notions are defined by the theory of string languages similarly as they are defined in [1] [24], [22] [16], [41].

2.1 Alphabet, string

An alphabet is a finite nonempty set of symbols. A ranked alphabet is a finite nonempty set of symbols each of which has a unique nonnegative arity (or rank). Given a ranked alphabet \( A \), the arity of a symbol \( a \in A \) is denoted \( \text{Arity}(a) \). The set of symbols of arity \( p \) is denoted by \( A_p \). Elements of arity 0, 1, 2, \ldots, \( p \) are called nullary (constants), unary, binary, \ldots, \( p \)-ary symbols, respectively. It is assumed that \( A \) contains at least one constant. In the examples, the numbers at the end of identifiers are used for a short declaration of symbols with arity. For instance, \( a2 \) is a short declaration of a binary symbol \( a \). The notation \(|A|\) is used for the size of set \( A \).

An \( A_p \) where \( p \geq 0 \) is a ranked alphabet of symbols of arity \( p \) only.

A string \( x \) is a sequence of \( i \) symbols \( s_1s_2s_3\ldots s_i \) from a given alphabet, where \( i \) is the size of the string. A sequence of zero symbols is called the empty string. The empty string is denoted by symbol \( \varepsilon \).

2.2 Language

A language over an alphabet \( A \) is a set of strings over \( A \). Symbol \( A^* \) denotes the set of all strings over \( A \) including the empty string, denoted by \( \varepsilon \). Set \( A^+ \) is defined as \( A^+ = A^* \setminus \{\varepsilon\} \). Similarly, for string \( x \in A^* \), symbol \( x^m \), \( m \geq 0 \), denotes the \( m \)-fold concatenation of \( x \) with \( x^0 = \varepsilon \). Set \( x^* \) is defined as \( x^* = \{x^m : m \geq 0\} \), \( x^+ = \{x^m : m \geq 1\} \) and \( x^* = x^+ \cup \{\varepsilon\} \).
2. Theoretical Background

2.3 Finite automaton

A **nondeterministic finite automaton (NFA)** is a five-tuple $F_M = (Q, A, \delta, q_0, F)$, where $Q$ is a finite set of states, $A$ is an input alphabet, $\delta$ is a mapping from $Q \times A$ into a set of finite subsets of $Q$, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is the set of final (accepting) states. A finite automaton (FA) $F_M$ is deterministic – deterministic finite automaton (DFA) if $\delta(q, a)$ has no more than one member for any $q \in Q$, and $a \in A$, hence the $\delta$ mapping of a DFA is usually simplified to a mapping from $Q \times A$ into $q$, where $q \in Q$. It is noted that the mapping $\delta$ is often illustrated by its transition diagram.

Every NFA can be converted to an equivalent DFA [1]. The conversion constructs the states of the DFA as subsets of states of the NFA and selects only such accessible states (i.e. subsets). These subsets are called $d$–subsets. Although $d$–subsets are standard sets, they are often written in square brackets ([ ] ) instead of in braces ( { } ).

2.4 Pushdown automaton

A **pushdown automaton (PDA)** is a seven-tuple $M = (Q, A, G, \delta, q_0, \perp, F)$, where $Q$ is a finite set of states, $A$ is an input alphabet, $G$ is a pushdown store alphabet, $\delta$ is a mapping from $Q \times (A \cup \{ \varepsilon \}) \times G$ into a set of finite subsets of $Q \times G^*$, $q_0 \in Q$ is an initial state, $\perp \in G$ is the initial pushdown store symbol, and $F \subseteq Q$ is the set of final (accepting) states.

Triple $(q, w, x) \in Q \times A^* \times G^*$ denotes the configuration of a PDA. The top of the pushdown store $x$ is written on its left hand side. The initial configuration of a PDA is $(q_0, w, \perp)$. The relation $\vdash_M \subseteq (Q \times A^* \times G^*) \times (Q \times A^* \times G^*)$ is a transition of a PDA $M$. It holds that $(q, aw, \alpha \beta) \vdash_M (p, w, \gamma \beta)$ if $(p, \gamma) \in \delta(q, a, \alpha)$.

The $k$-th power, transitive closure, and transitive and reflexive closure of the relation $\vdash_M$ is denoted $\vdash_M^k$, $\vdash_M^+$, $\vdash_M^*$, respectively.

A PDA is input-driven if each of its pushdown operations is determined only by the input symbol.

A language $L$ accepted by a PDA $M$ is defined in two distinct ways:

1. **Accepting by final state:** $L(M) = \{ x : (q_0, x, \perp) \vdash_M^* (q, \varepsilon, \gamma) \land x \in A^* \land \gamma \in G^* \land q \in F \}$.

2. **Accepting by empty pushdown store:** $L_\varepsilon(M) = \{ x : (q_0, x, \perp) \vdash_M^* (q, \varepsilon, \varepsilon) \land x \in A^* \land q \in Q \}.

If the PDA accepts the language by empty pushdown store, then the set $F$ of final states is the empty set.

Unreachable states are states $p \in Q$ from automaton $M = (Q, A, G, \delta, q_0, \perp, F)$ which are not reachable from the initial state because there is no sequence of transitions from
2.5. Tree, tree pattern, linear notations and their properties

the initial state to that particular state \( p \). Formally, there are no transitions that allow \((q_0, kw, \perp) \vdash^+ M (p, w, \gamma)\).

Unnecessary states are states \( p \in Q \) from automaton \( M = (Q, \mathcal{A}, G, \delta, q_0, \perp, F) \) which are not connected to any final state \( f \in F \) if automaton accepts by final states, or not connected to any state, where \( \gamma \in G^* \) may be \( \varepsilon \), if automaton accepts by empty pushdown store.

\[ \text{PDA} \quad M = (Q, \mathcal{A}, G, \delta, q_0, \perp, F) \text{ is acyclic if it does not contain transitions } (q, x_1, \gamma_1) \vdash^+_M (q, x_2, \gamma_2), \text{ where } xx_2 = x_1, x \neq \varepsilon \text{ and } q \in Q. \]

\[ \text{PDA} \quad M \text{ is a deterministic PDA}, \text{ if it holds:} \]

\[ \forall q \in Q, \forall a \in \mathcal{A}, \forall z \in G, \delta(q, \varepsilon, z) = \emptyset : |\delta(q, a, z)| \leq 1, \]

\[ \forall a \in \mathcal{A}, |\delta(q, \varepsilon, z)| \geq 1 : \delta(q, a, z) = \emptyset. \]

The relation \( \vdash_M \subset (Q \times \mathcal{A}^* \times G^*) \times (Q \times \mathcal{A}^* \times G^*) \) is a transition of a \( \text{PDA} \) \( M \). It holds that \((q, aw, \alpha z) \vdash_M (p, w, \beta z)\) if \((p, \beta) \in \delta(q, a, \alpha)\), where \( z, \alpha, \beta \in G^* \). The \( k \)-th power, transitive closure, and transitive and reflexive closure of the relation \( \vdash_M \) is denoted \( \vdash^+_M, \vdash^*_M \), respectively.

Note that standard symbol \( Z_0 \) for initial pushdown store symbols is replaced by \( \perp \). This change is related to [6, 37]. This change is mostly cosmetic in nature.

### 2.5 Tree, tree pattern, linear notations and their properties

Based on concepts and notations from graph theory [1]:

A **graph** \( G \) is a pair \((N, R)\), where \( N \) is a set of nodes and \( R \) is a set of edges such that each element of \( R \) is of the form \((f, g)\), where \( f, g \in N \). This element will indicate that, for node \( f \), there is an edge between node \( f \) and node \( g \).

A **directed graph** \( G \) is a graph, where each element of \( R \) of the form \((f, g)\) indicates that there is an edge leaving node \( f \) and entering node \( g \). This edge is ordered from \( f \) to \( g \). An **undirected graph** \( G \) is a graph in which no such ordering of edges is given.

A sequence of nodes \((f_0, f_1, \ldots, f_n), n \geq 1\), is a **path** of length \( n \) from node \( f_0 \) to node \( f_n \) if there is an edge which leaves node \( f_{i-1} \) and enters node \( f_i \) for \( 1 \leq i \leq n \). A **labelling** of an ordered graph \( G = (N, R) \) is a mapping of \( N \) into a set of labels. In the examples \( a_f \) is used for a short declaration of node \( f \) labelled by symbol \( a \).

A directed graph is **connected** if there exists a path from \( f_u \) to \( f_v \) for each pair of nodes \((f_u, f_v), u \neq v\), of the graph.

A **cycle** is a path \((f_0, f_1, \ldots, f_n)\) in which \( f_0 = f_n \).
2. Theoretical Background

Given a node \( f \) of a directed graph, its \textit{out-degree} is the number of distinct pairs \((f, g)\) \( g \in N \), where \( g \in N \). By analogy, the \textit{in-degree} of node \( f \) is the number of distinct pairs \((g, f)\) \( g \in N \).

A \textit{tree} is a connected directed graph whose underlying undirected graph is without any cycle. The tree is assumed to have at least one node. A \textit{rooted tree} \( t \) is a tree with a special node \( r \in N \), called the \textit{root}.

The rooted tree \( t \) can be also defined by following:

1. \( r \in N \) (root) has in-degree 0,
2. all other nodes of \( t \) have in-degree 1,
3. there is just one path from the root \( r \) to every \( f \in N \), where \( f \neq r \).

Nodes of a tree with out-degree 0 are called \textit{leaves}.

A \textit{labelled and rooted tree} is a tree with the additional property: (4) every node \( f \in N \) is labelled by a symbol \( a \in A \), where \( A \) is an alphabet.

A node \( g \) is a \textit{direct descendant} of node \( f \) if a pair \((f, g)\) \( g \in N \).

An \textit{ordered, labelled and rooted tree} is a labelled and rooted tree where direct descendants of a node \( f \) are ordered.

An \textit{ordered, ranked, labelled and a rooted tree} is a labelled and rooted tree labelled by symbols from a ranked alphabet and where the out-degree of a node \( f \) labelled by symbol \( a \in A \) equals \( \text{Arity}(a) \). Nodes labelled by nullary symbols (constants) are leaves.

Throughout the text shorthand \textit{ranked tree} will be used in the context of ordered, ranked, labelled and rooted tree and \textit{unranked tree} in the context of ordered, labelled and rooted tree.

The \textit{prefix notation} \( \text{pref}(t) \) of a ranked tree \( t \) is defined as follows:

1. \( \text{pref}(a) = a0 \) if \( a \) is a leaf,
2. \( \text{pref}(t) = a \text{pref}(b_1) \text{pref}(b_2) \ldots \text{pref}(b_n) \), where \( a \) is the root of tree \( t \), \( n = \text{Arity}(a) \) and \( b_1, b_2, \ldots, b_n \) are direct descendants of \( a \).

The \textit{prefix bar notation} \( \text{pref\,\bar{\text{bar}}}(t) \) of a unranked tree \( t \) is defined as follows:

1. \( \text{pref\,\bar{\text{bar}}}(a) = a \uparrow \) if \( a \) is a leaf,
2. \( \text{pref\,\bar{\text{bar}}}(t) = a \text{pref\,\bar{\text{bar}}}(b_1) \text{pref\,\bar{\text{bar}}}(b_2) \ldots \text{pref\,\bar{\text{bar}}}(b_n) \uparrow \), where \( a \) is the root of tree \( t \) and \( b_1, b_2, \ldots, b_n \) are direct descendants of \( a \).

The \textit{postfix notation} \( \text{post}(t) \) of a ranked tree \( t \) is defined as follows:

1. \( \text{post}(a) = a0 \) if \( a \) is a leaf,
2. \( \text{post}(t) = \text{post}(b_1) \text{post}(b_2) \ldots \text{post}(b_n) a \), where \( a \) is the root of tree \( t \), \( n = \text{Arity}(a) \) and \( b_1, b_2, \ldots, b_n \) are direct descendants of \( a \).
2.6 Regular tree expression

The postfix bar notation \( \text{post}_{\bar{\text{bar}}}(t) \) of a unranked tree \( t \) is defined as follows:

1. \( \text{post}_{\bar{\text{bar}}}(a) = a \uparrow \) if \( a \) is a leaf,

2. \( \text{post}_{\bar{\text{bar}}}(t) = a \text{ post}_{\bar{\text{bar}}}(b_1) \text{ post}_{\bar{\text{bar}}}(b_2) \cdots \text{ post}_{\bar{\text{bar}}}(b_n) \uparrow \), where \( a \) is the root of tree \( t \) and \( b_1, b_2, \ldots, b_n \) are direct descendants of \( a \).

### 2.6 Regular tree expression

Regular tree expressions (RTEs) \([16, 3]\) and their relation to the standard regular expressions (REs) \([1]\) are defined using an operation substitution and generalisation of Kleene’s theorem for strings to trees. This definition of RTE is closely related to the definition of regular-like expressions \([42]\). For defining RTEs, basically the ground term notation, is used in Defs. \([2.1, 2.5, 2.6]\). Instead of the ground term notation, the corresponding postfix or postfix bar notations could also be used.

\( \text{RTEs} \) are over ranked alphabet denoted as \( \mathcal{F} \). Unlike strings, which can be concatenated only at its ends (and beginnings), trees can be connected in multiple places. A notion for regular tree expressions needs to be defined to mark some specific positions where concatenation and iteration can take place in trees. Those places are represented as elements from a set of constants (i.e. symbols with arity 0) \( \mathcal{K} = \{\square_1, \square_2, \ldots, \square_n\}, n \geq 1, \Sigma \cap \mathcal{K} = \emptyset \), is used in \( \text{RTEs} \) to indicate position where operations take place. Trees over alphabet \( \mathcal{F} \cup \mathcal{K} \) can be constructed.

Note that there is also different notation of regular tree languages. This different notation uses only one alphabet, \( \mathcal{F} \), and allows the substitution operation on any constant symbol from \( \mathcal{F} \) \([31]\). Both notations are equal in terms of expressiveness.

The substitution from string languages to tree languages is generalised by replacing some \( \square_i \in \mathcal{K} \) by a tree of some language \( L_i \). It holds that different occurrences of the same constant \( \square_i \) can be replaced by different terms from \( L_i \).

**Definition 2.1 (Substitution of tree language into tree).** Given a tree \( t \in T(\Sigma \cup \mathcal{K}), \square_1, \square_2, \ldots, \square_n \in \mathcal{K} \) and \( L_1, L_2, \ldots, L_n \) languages over \( T(\Sigma \cup \mathcal{K}) \), the tree substitution of \( \square_1, \square_2, \ldots, \square_n \) by \( L_1, L_2, \ldots, L_n \) in \( t \), denoted by \( t\{\square_1 \leftarrow L_1, \square_2 \leftarrow L_2, \ldots, \square_n \leftarrow L_n\} \), is the tree language defined as follows:

1. \( \varepsilon\{\square_1 \leftarrow L_1, \square_2 \leftarrow L_2, \ldots, \square_n \leftarrow L_n\} = \{\varepsilon\} \), for empty tree \( \varepsilon \in T(\Sigma \cup \mathcal{K}) \)

2. \( a\{\square_1 \leftarrow L_1, \square_2 \leftarrow L_2, \ldots, \square_n \leftarrow L_n\} = \{a\} \), for \( a \in T(\Sigma \cup \mathcal{K}) \), such that arity of \( a \) is 0 and \( a \neq \square_1, a \neq \square_2, \ldots, a \neq \square_n \),

3. \( \square_i\{\square_1 \leftarrow L_1, \square_2 \leftarrow L_2, \ldots, \square_n \leftarrow L_n\} = L_i \) for \( i = 1, \ldots, n \),
2. Theoretical Background

4. \( f(s_1, s_2, \ldots, s_n) \{ \square_1 \leftarrow L_1, \square_2 \leftarrow L_2, \ldots, \square_n \leftarrow L_n \} = \{ f(t_1, t_2, \ldots, t_n) \mid t_i \in s_i \{ \square_1 \leftarrow L_1, \square_2 \leftarrow L_2, \ldots, \square_n \leftarrow L_n \} \} \).

The substitution operation generalises languages in a straightforward way. When \( L, L_1, \ldots, L_n \) are languages over \( T(\Sigma \cup K) \) and \( \square_1, \square_2, \ldots, \square_n \in K \), the language \( L \{ \square_1 \leftarrow L_1, \square_2 \leftarrow L_2, \ldots, \square_n \leftarrow L_n \} \) is defined to be the set \( \bigcup_{t \in L} \{ t \{ \square_1 \leftarrow L_1, \square_2 \leftarrow L_2, \ldots, \square_n \leftarrow L_n \} \} \).

**Definition 2.2** (Union). Given languages \( L, M \subseteq T(\Sigma \cup \{ \square \}) \), the union of \( M \) and \( L \), denoted by \( L + M \), is a set of trees obtained by union of languages \( L \) and \( M \), i.e. \( L + M = L \cup M \).

**Definition 2.3** (Concatenation). Given languages \( L, M \subseteq T(\Sigma \cup \{ \square \}) \), and a symbol \( \square \in K \), the concatenation of \( M \) to \( L \) through symbol \( \square \), denoted by \( L \cdot \square M \), is a set of trees obtained by substituting the occurrences of \( \square \) in trees of \( L \) by trees of \( M \), i.e. \( L \cdot \square M = \bigcup_{t \in L} \{ t \{ \square \leftarrow M \} \} \).

**Definition 2.4** (Iteration). Given a language \( L \) over \( T(\Sigma \cup \{ \square \}) \) and a symbol \( \square \in K \), the sequence \( L^{*\square} \) of successive iterations is defined as follows:

- \( L^{0\square} = \{ \square \} \),
- \( L^{n+1\square} = L^{n\square} \cup L \cdot \square L^n\square \).

The closure \( L^{*\square} \) of \( L \) is the union of all \( L^{n\square} \) for non-negative \( n \), i.e. \( L^{*\square} = \bigcup_{n \in \mathbb{N}} L^{n\square} \). From the definition, one gets that \( \{ \square \} \subseteq L^{*\square} \) for any \( L \).

The **RTEs** are defined similar to regular (string) expressions with use of operators +, \( \cdot \), *, \( ^\square \).

**Definition 2.5** (RTEs). The set \( RTE(\Sigma, K) \) of all RTEs over \( \Sigma \) and \( K \) is defined as the minimal set as follows:

1. the empty set \( \emptyset \) is in \( RTE(\Sigma, K) \),
2. if \( a \in (\Sigma_0 \cup K) \) is a constant, then \( a \in RTE(\Sigma, K) \),
3. if \( f \in \Sigma_n \) has arity \( n > 0 \) and \( E_1, E_2, \ldots, E_n \) are RTEs of \( RTE(\Sigma, K) \), then \( f(E_1, E_2, \ldots, E_n) \) is a RTE of \( RTE(\Sigma, K) \),
4. if \( E_1, E_2 \) are RTEs of \( RTE(\Sigma, K) \), then \( (E_1 + E_2) \) is a RTE of \( RTE(\Sigma, K) \),
5. if \( E_1, E_2 \) are RTEs of \( RTE(\Sigma, K) \) and \( \square \in K \), then \( (E_1 \cdot \square E_2) \) is a RTE of \( RTE(\Sigma, K) \),
6. if \( E \) is a RTEs of \( RTE(\Sigma, K) \) and \( \square \in K \), then \( E^{\square} \) is a RTE of \( RTE(\Sigma, K) \).
2.6. Regular tree expression

Each RTE $E$ represents a set of terms over $T(\Sigma \cup K)$, which is denoted $[E]$ and which is formally defined by Def. 2.6.

**Definition 2.6 (Operations over RTEs).** Properties of operations over RTEs are defined as follows:

1. $[\emptyset] = \emptyset$,
2. $[a] = \{a\}$ for $a \in (\Sigma_0 \cup K)$,
3. $[f(E_1, E_2, \ldots, E_n)] = \{f(s_1, s_2, \ldots, s_n) | s_1 \in [E_1], s_2 \in [E_2], \ldots, s_n \in [E_n]\}$,
4. $[E_1 + E_2] = [E_1] \cup [E_2]$,
5. $[E_1 \cdot \triangleleft E_2] = [E_1]\{\triangleleft \leftarrow [E_2]\}$,
6. $[E^* \cdot \triangleleft] = [E]^* \cdot \triangleleft$. 

State-of-the-Art and Related work

This section surveys previous results in the field and related work. It builds upon the definitions from chapter 2. It summarises main sources of the dissertation thesis.

Most notably it contains the topics related to determinisation of finite automata (FAs), determinisation of input-driven pushdown automata (IDPDAs), determinisation of visibly pushdown automata (VPDAs), determinisation of real-time height-deterministic PDAs (rHPDAs), and regular tree expressions (RTEs) definition.

3.1 Determinisation of finite automaton

In this section, the determinisation of FAs is described. This follows with the determinisation of IDPDAs. IDPDAs are closely related to pushdown automata (PDAs).

3.1.1 Finite automaton

Let \( M = (Q, \mathcal{A}, \delta, Q_0, F) \) be a nondeterministic FA. An equivalent deterministic FA \( M' = (Q', \mathcal{A}, \delta', \{q_0', \}, F') \) can be constructed as follows:

\[
Q' = \mathcal{P}(Q),
\]

\[
q_0' = \{Q_0\},
\]

\[
F' = \{q' \subseteq Q | q' \cap F \neq \emptyset\},
\]

\[
\delta'(q', a) = \bigcup_{q \in q'} \delta(q, a), \forall q' \in Q', \forall a \in \mathcal{A}.
\]

Determinisation of FAs is described by Algorithm 3.1.
3. State-of-the-Art and Related work

Algorithm 3.1: Finite automaton determinisation algorithm

Input: Nondeterministic FA $M_n(Q, A, \delta, Q_0, F)$.
Output: Equivalent deterministic FA $M_d(Q', A, \delta', q'_0, F')$.

1. init $q'_0 \leftarrow \{Q_0\}$;
2. init $Q' \leftarrow \{q'_0\}$;
3. init $\delta' \leftarrow \emptyset$;
4. init $F' \leftarrow \emptyset$;
5. queue $Dirty \leftarrow (q'_0)$; // The set of states
6. set $Clean \leftarrow \emptyset$; // The set of states
7. while $Dirty$ is not empty do
   8. $q \leftarrow$ dequeue $Dirty$;
   9. if $q \notin Clean$ then
      10. foreach $a \in A$ do
         11. $q' \leftarrow \{q_1 : q_1 \in \delta(q, a)\}$;
         12. if $q' = \emptyset$ then continue; // Fail state
         13. if $q' \in Q'$ then continue; // Existing state
         14. add $q'$ to $Q'$;
         15. add $q'$ to $\delta'(q, a)$;
         16. enqueue $q'$ to $Dirty$;
   17. foreach $q' \in Q'$ do // Set final states
      18. if exists $q \in q'$ such that $q \in F$ then
      19. add $q'$ to $F'$;

3.1.2 Input-driven pushdown automaton

Let $M = (Q, A, G, \delta, Q_0, \bot, F)$ be a nondeterministic IDPDA. An equivalent deterministic IDPDA $M' = (Q', A, G, \delta', \{q'_0\}, \bot, F')$ can be constructed as follows:

$$Q' = \mathbb{P}(Q),$$

$$q'_0 = \{Q_0\},$$

$$F' = \{q' \subseteq Q | q' \cap F \neq \emptyset\},$$

$$\delta'(q', a, \Gamma) = \bigcup_{q \in q'} \delta(q, a, \Gamma), \forall q' \in Q', \forall a \in A, \forall \Gamma \in G^*. $$

Determinisation of FAs is described by Algorithm 3.2.
Algorithm 3.2: Input-driven pushdown automaton determinisation algorithm

Input : Nondeterministic IDPDA $M_n(Q, A, G, \delta, Q_0, \bot, F)$.
Output: Equivalent deterministic IDPDA $M_d(Q', A, G, \delta', q'_0, \bot, F')$.

1. init $q'_0 \leftarrow \{Q_0\}$;
2. init $Q' \leftarrow \{q'_0\}$;
3. init $\delta' \leftarrow \emptyset$;
4. init $F' \leftarrow \emptyset$;
5. init queue Dirty $\leftarrow (q'_0)$; // The set of states
6. init set Clean $\leftarrow \emptyset$; // The set of states
7. while Dirty is not empty do
   8. $q \leftarrow$ dequeue Dirty;
   9. if $q \notin$ Clean then
      foreach $a \in A$ do
         10. $q' \leftarrow \{q_1 : (q_1, \Gamma_2) \in \delta(q, a, \Gamma_1)\}$;
         if $q' = \emptyset$ then continue; // Fail state
         if $q' \in Q'$ then continue; // Existing state
         add $q'$ to $Q'$;
         add $(q', \Gamma_2)$ to $\delta'(q, a, \Gamma_1)$;
         enqueue $q'$ to Dirty;
   17. foreach $q' \in Q'$ do // Set final states
      if exists $q \in q'$ such that $q \in F$ then
         19. add $q'$ to $F'$;

3.2 Determinisation of pushdown automaton

As history of VPDAs and related work is quite brief, the related papers are organized in chronological order beginning with foundational paper from Alur and Madhusudan. They introduced visibly pushdown languages, a subset of pushdown languages and their processing machine, VPDA, a special case of PDAs. They also presented a proof of determinisability of these automata of which all following works are based.

In 2004 a foundational paper introducing visibly pushdown languages was published by Alur and Madhusudan [6]. Visibly pushdown languages are class of languages directly associated with VPDAs. The paper incorporates a proof of VPDA determinisability. The proof itself could be directly translated into determinisation algorithm.

In 2006 Ha Nguyen created a library for nondeterministic VPDA determinisation called VPAlib [36]. The determinisation implemented in this library is based on cartesian product consisting of all possible states. This algorithm is directly based on the proof by Alur and Madhusudan [6].

As pointed by Nguyen Van Tang – in the first implementation of VPDA determin-
isation library, named VPAlib, the determinisation was performed in an exhaustive way. Namely, unreachable states and redundant transitions were also generated. Therefore their determinisation easily gets stuck with VPDAs of small size [45]. Details in [http://www.emn.fr/z-info/hnguyen/vpa/](http://www.emn.fr/z-info/hnguyen/vpa/).

In 2007 a paper about input driven VPDAs with two pushdown stores was published by Carotenuto, Murano and Peron. It has been shown by Madhusudan, that the determinisation construction in this paper is wrong, and in fact, 2-VPAs are not determinisable [32].

In 2007 Dirk Nowotka and Jiří Srba defined the notion of height-deterministic pushdown automata (HPDAs), a model where for any given input string the pushdown store heights during any computation on the input are a priori fixed. Several classes of these automata were described and decidability and complexity questions were considered [37].

These automata are closely related to VPDAs. They are their natural extension sharing various good properties.

In 2009 an improved upper bound of number of states has been found by Nguyen Van Tang [45]. Two optimisations for Alur-Madhusudan’s determinisation of VPDAs were introduced. Minimizing the set of summaries S component of state pair for some special cases concerning initial states and removing R component of state pair.

By removing R component of determinised VPDAs the upper bound for number of states was lowered from \(2^{n^2+n}\) to \(2^{n^2}\). The optimisation is based on observation that information stored in R component of state pair is already contained in S component of state pair [45].

### 3.2.1 Normalisation of pushdown automaton

**Definition 3.1** (Alur-Madhusudan normalisation [6]). The pushdown automaton \(A = (Q, A, G, \delta, Q_0, Z_0, F)\) is normalised if the transition relation is in the form \((q, X) \in \delta(p, a, \varepsilon)\) or \((q, \varepsilon) \in \delta(p, a, X)\) or \((q, \varepsilon) \in \delta(p, a, \varepsilon)\), where \(p, q \in Q, X \in G\) and \(a \in A\).

Note that automata with this normalisation are real-time.

**Definition 3.2** (Nowotka-Srba normalisation [37]). Pushdown automaton \(A = (Q, A, G, \delta, Q_0, Z_0, F)\) is normalised if:

1. \(\forall p \in Q\) transition relation \((q, a) \in \delta(p, a, X)\) either satisfy \(a \in A\) or all of them satisfy \(a = \varepsilon\) but not both for \(a \in G^*, X \in G\) and \(q \in Q\).

2. the transition relation is in the form \((q, Y X) \in \delta(p, a, X)\) or \((q, \varepsilon) \in \delta(p, a, X)\) or \((q, X) \in \delta(p, a, X)\), where \(p, q \in Q, X \in G\) and \(a \in (A \cup \{\varepsilon\})\).

Note that Nowotka and Srba do not consider the real-time property to be a part of normalisation.
3.2. Determinisation of pushdown automaton

3.2.2 Visibly pushdown automaton

VPDAs are defined as in [6, 45]. Let $\mathcal{A}$ be an input alphabet partitioned into three distinct parts $\mathcal{A}_c \cup \mathcal{A}_r \cup \mathcal{A}_l = \mathcal{A}$. The intuition behind the partitioning is: $\mathcal{A}_c$ is the set of call (push) symbols, $\mathcal{A}_r$ is the set of return (pop) symbols, and $\mathcal{A}_l$ is the set of local symbols.

A VPDAs $M$ is a seven-tuple $(Q, \mathcal{A}, G, \delta, Q_0, \bot, F)$, where $Q$ is a finite set of states, $\mathcal{A}$ is a finite input alphabet, $G$ is a finite pushdown store alphabet, $\delta$ is the transition mapping, $Q_0 \subseteq Q$ is a set of initial states, $\bot \in G$ is an initial pushdown symbol (bottom-of-the-pushdown-store) which can be popped multiple times, and $F \subseteq Q$ is a set of final (accepting) states.

VPDAs are normalised according to Definition 3.1 to create push, pop and local transitions accordingly.

3.2.3 Height-deterministic pushdown automaton

HPDAs and their real-time normalised variants are defined as in [37]. Instead of a partitioned alphabet, HPDAs follow the condition that for any given input string the pushdown store heights during any (nondeterministic) computation are fixed a priori.

A HPDA $M$ is a seven-tuple $(Q, \mathcal{A}, G, \delta, Q_0, \bot, F)$, where $Q$ is a finite set of states, $\mathcal{A}$ is a finite input alphabet, $G$ is a finite pushdown store alphabet, and $\delta$ is transition mapping.

HPDAs are real-time and are normalised according to Definition 3.2 to create push, pop and local transitions accordingly.

3.2.4 Directly related works

VPDAs were introduced in [6]. Moreover, it was shown that any nondeterministic VPDAs can be converted into an equivalent deterministic pushdown automaton. The determinisation principle is similar to the determinisation principle of FAs [24].

In [6], the states of the resulting deterministic VPDAs consist of two components $(S, R)$. Component $R \in \mathcal{P}(Q)$ is an element of the power set of the states of the original automaton. Component $S = \mathcal{P}(Q \times Q)$ is a power set of pairs of states of the original nondeterministic pushdown automaton that keeps tracking the beginning states on the path from push transitions to all states listed in the $R$ component. It is noted that, given union of the states in the second parts of the pairs in the $S$ component is equal to the $R$ component, the $R$ component can be omitted. However to keep the automata hierarchy simple this $R$ component is maintained in the following definition as a connection to FAs [45], where the states of the determinised automata correspond to the $R$ component.
Definition 3.3 (VPDAs). VPDAs are defined as in [6, 45].

Let $A$ be an input alphabet, $A_c \cup A_r \cup A_l = A$ be partitions of $A$. The intuition behind the partitioning is: $A_c$ is the finite set of call (push) symbols, $A_r$ is the finite set of return (pop) symbols, and $A_l$ is the finite set of local symbols.

A VPDA $M$ over $A$ is a seven-tuple $(Q, A, G, \delta, Q_0, \bot, F)$, where $Q$ is a finite set of states, $A$ is a finite input alphabet, $G$ is a finite pushdown store alphabet, $\delta \subseteq (Q \times A_c \times Q \times (G \setminus \bot)) \cup (Q \times A_r \times G \times Q) \cup (Q \times A_l \times Q)$ is the transition mapping, $Q_0 \subseteq Q$ is a set of initial states, $\bot \in G$ is initial pushdown store symbol (bottom-of-the-pushdown-store) which can be popped multiple times, and $F \subseteq Q$ is a set of final (accepting) states.

Let $M = (Q, A, G, \delta, Q_0, \bot, F)$ be a nondeterministic VPDA. For $A = A_c \cup A_r \cup A_l$. An equivalent deterministic VPDA $M' = (Q', A, G', \delta', \{q'_0\}, \bot, F')$ can be constructed as follows:

$$Q' = 2^Q \times Q \times 2^Q,$$

$$q'_0 = (Id_{Q_0}, Q_0) \text{ where } Id_X = \{(x, x) | x \in X\},$$

$$F' = \{(S, R) | R \cap F \neq \emptyset\},$$

$$G' = 2^Q \times Q \times 2^Q \times A_l.$$

The transition relation $\delta' = \delta'_l \cup \delta'_c \cup \delta'_r$ is given by:

**Local**: For every $l \in A_l$, $((S', R'), \varepsilon) \in \delta'_l((S, R), l, \varepsilon)$, where

$$S' = \{ (q, q') \exists q'' \in Q : (q, q'') \in S, (q', \varepsilon) \in \delta_l(q'', l, \varepsilon) \},$$

$$R' = \{ q' \exists q \in R : (q', \varepsilon) \in \delta_l(q, l, \varepsilon) \}.$$

**Push**: For every $c \in A_c$, $((Id_{R'}, R'), (S, R, c)) \in \delta'_c((S, R), c, \varepsilon)$, where

$$R' = \{ q' \exists q \in R : (q', \gamma) \in \delta_c(q, c, \varepsilon) \}.$$

**Pop**: For every $r \in A_r$,

- if the pushdown store is empty : $((S', R'), \varepsilon) \in \delta'_r((S, R), r, \bot)$, where $S' = \{ (q, q') \exists q'' \in Q : (q, q'') \in S, (q', \varepsilon) \in \delta_r(q'', r, \bot) \}$ and $R' = \{ q' \exists q \in R : (q', \varepsilon) \in \delta_r(q, r, \bot) \}$. 

20
- otherwise: \((S'', R''), (\epsilon, (S, R, r, S', R', c))\), where

\[
R'' = \{ q' | \exists q \in R' : (q, q') \in U \},
S'' = \{ (q, q') | \exists q_2 \in Q : (q, q_3) \in S', (q_3, q') \in U \},
U = \{ (q, q') | \exists q_1 \in Q,
\quad q_2 \in R : (q_1, q_2) \in S,
\quad (q_1, \gamma) \in \delta_r(q, c, \epsilon),
\quad (q', \epsilon) \in \delta_r(q_2, r, \gamma) \}.
\]

Note that Alur and Madhusudan allow for the bottom of the pushdown symbol \(\bot\) to be popped an unlimited number of times. The pushdown store never empties and automaton accepts by final state. This part of the definition handles this property and is not present in the HPDAs below. If this feature is not needed, this part of the algorithm can be omitted.

The equivalent deterministic automaton has at most \(2^{n^2} + n^n\) states and at most \(|A_c|2^{2n^2+n}\) pushdown symbols. The size of the transition relation can be at most \(|A_l|(2^{n^2} + n^n)^2 + |A_c|(2^{n^2} + n^n)^3 + |A_r||A_c|(2^{n^2} + n^n)^3\).

An improved upper bound of the number of states has been presented in [45]. Two optimisations for Alur-Madhusudan’s determinisation of VPDA were introduced. The \(S\) component for some special cases concerning initial states was minimized. The \(R\) component was removed. By removing the \(R\) component of the determinised VPDA the upper bound for the number of states was lowered to \(2^{n^2}\) and the number of pushdown symbols was lowered to \(|A_c|2^{n^2}\). The optimisation is based on the observation that information stored in the \(R\) component is already contained in the \(S\) component [45]. However, that determinisation algorithm is still not practical. As was pointed by Nguyen Van Tang in his implementation of a VPDA determinisation library, called VPAlib, the determinisation was performed in an exhaustive way. Therefore, his determinisation easily fails with VPDA of small size [45].

HPDAs were introduced in [37]. Subgroup HHPDAs have been proven to be determinisable [37], but no practical algorithm has been offered so far. The improvements presented in [45] are also directly applicable for HPDAs as well.

The difference between HHPDA and VPDA is that instead of partitioning the whole alphabet, the alphabet is partitioned for each group of states in which any (nondeterministic) computation may end for the same input string.

**Definition 3.4 (RHPDAs).** HPDAs and their real-time normalised variants are defined as in [37].

Instead of partitioned alphabet the HPDAs follow the condition that for any given input string the pushdown store heights during any (nondeterministic) computation are a priori fixed.

A \(\text{rHPDA}\) \(M\) over \(\mathcal{A}\) is a seven-tuple \((Q, \mathcal{A}, G, \delta, Q_0, \bot, F)\), where \(Q\) is a finite set of states, \(\mathcal{A}\) is a finite input alphabet, \(G\) is a finite pushdown store alphabet, \(\delta \subseteq (Q \times \mathcal{A} \times \mathcal{G})\)
3. State-of-the-Art and Related work

\[ G \times Q \times (G \setminus \perp) \times (G \setminus \perp) \cup (Q \times \mathcal{A} \times G \times Q) \cup (Q \times \mathcal{A} \times G \times Q \times (G \setminus \perp)) \] is the transition mapping, \((Q, \alpha) \in \delta_r(Q, \mathcal{A}, \gamma),\) where \(\gamma \in G \cup \{\varepsilon\}\) and \(\alpha \in G^*\) is a push, local, or pop transition if \(|\alpha| = 2, 1,\) or 0 respectively, \(Q_0 \subseteq Q\) is a set of initial states, \(\perp \notin G\) is initial pushdown store symbol, and \(F \subseteq Q\) is a set of final (accepting) states.

Let \(M = (Q, \mathcal{A}, G, \delta, Q_0, \perp, F)\) be a nondeterministic HPDA. An equivalent deterministic HPDA \(M' = (Q', \mathcal{A}, G', \delta', \{q_0\}, \perp, F')\) can be constructed as follows:

- \(Q' = 2^{Q \times Q} \times 2^Q,\)
  - \(q_0' = (\{(q_0, (q_0, \perp))|q_0 \in Q_0\}, \{(q_0, \perp)|q_0 \in Q_0\}),\)
  - \(F' = \{(S, R) \in R \cap \{f, \gamma)|f \in F, \gamma \in G\} \neq \emptyset\},\)
  - \(G' = 2^{Q \times Q} \times 2^Q \times \mathcal{A}.\)

The transition relation \(\delta'\) is given by:

**Local:** For every \(l \in \mathcal{A},\)

\((S', R'), (q, \gamma) \in \delta'_l((S, R), l, \gamma),\)

\[
S' = \{ (q, (q', \gamma'))|\exists (q'', \gamma'') \in Q : (q, (q'', \gamma'')) \in S, (q', \gamma') \in \delta_l(q'', l, \gamma') \},
\]

\[
R' = \{ (q', \gamma')|\exists (q, \gamma') \in R : (q', \gamma') \in \delta_l(q, l, \gamma') \}.
\]

**Push:** For every \(c \in \mathcal{A},\)

\((Id_{R'}, R'), (S, R, c) \gamma) \in \delta'_c((S, R), c, \gamma),\)

\[
R' = \{ (q', \gamma'')|\exists (q, \gamma') \in R : (q', \gamma'' \gamma') \in \delta_c(q, c, \gamma') \}.
\]

**Pop:** For every \(r \in \mathcal{A},\)

- \((S'', R''), \varepsilon) \in \delta_r((S, R), r, (S', R', c)),\)

\[
\begin{align*}
R'' &= \{ q''|\exists q' \in R' : (q', q'') \in U \}, \\
S'' &= \{ (q, q'')|\exists q_3 \in Q : (q, q_3) \in S', (q_3, q'') \in U \}, \\
U &= \{ ((q, \gamma'), (q'', \gamma''))|\exists (q_1, \gamma'''), (q, \gamma') \in Q, (q_2, \gamma'') \in R : ((q_1, \gamma'''), (q_2, \gamma'')) \in S, (q_1, \gamma'' \gamma') \in \delta_c(q, c, \gamma'), (q', \varepsilon) \in \delta_r(q_2, r, \gamma'') \}.
\end{align*}
\]

The equivalent deterministic PDA has at most \(2^{n^2 + n}\) states and at most \(|\mathcal{A}|2^{n^2 + n}\) pushdown symbols. The optimisation for the VPDAs mentioned above can also be used for HPDAs by removing the \(R\) component of the determinised automaton. The upper bound for the number of states can be lowered to \(2^{n^2}\) and the number of pushdown symbols can be lowered to \(|\mathcal{A}|2^{n^2}\).
3.3 Regular tree expression conversion

Regular tree expressions (RTEs) are an important part of Computer Science with a long history. The basic summary of sources is provided on which are based the presented definition of RTEs and also the methods of their conversion into the PDAs including some similar or connected works.

The theory of tree automata (TAs) and tree languages emerged in the 1960's from the algebraic approach to FAs advocated by J. R. Büchi and J. B. Wright [13]. The kind of trees studied in this theory appear in mathematics, computer science and other areas as formal terms, algebraic expressions, parse and derivation trees, computation trees, and generally as representations of hierarchically organised structures. The first applications of TAs were some decidability results in logic. Tree transducers offer mathematical models for the theories of syntax direction translations and program schemata. Syntactic pattern recognition, logic programming, term rewriting and linguistics are some other areas in which TAs also have played a role.

Tree Automata Techniques and Applications commonly abbreviated as TATA [16] is the current bible of automaton based arbology, it summarizes and connects many works of many authors. The book is still maintained and continuously updated. Most notable authors are Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison and Marc Tommasi.

TATA provides a comprehensive definition of RTEs by generalisation of Kleene’s theorem for words to trees. It introduces the operation substitution over the languages of trees and on this basis it defines concatenation and iteration of trees similar to the ones defined for strings. It is one of the main sources of the dissertation thesis and more is introduced in following chapters. The RTEs from TATA are followed and described later.

Implementing RTEs [3] is a paper by Alexander Aiken focused on RTEs as a natural formalism for describing the sets of tree-structured values that commonly arise in programs; thus they are well-suited to applications in program analysis. He describes an implementation of RTEs, his experience with that implementation in the context of the FL type system [9]. A combination of algorithms, optimisations, and fast heuristics for computationally difficult problems yields an implementation efficient enough for practical use.

It follows his previous paper describing the theoretical basis for FL inference system, in which types are represented by RTEs [2]. The definition of used RTEs is heavily based on description of RTEs contained in TATA.

3.3.1 Regular tree expression to finite tree automaton

The research in the area of conversions of RTEs to finite tree automata (FTAs) was enabled by the fact, that Kleene’s theorem can be lifted from strings to trees [16 Th. 2.2.8]. Recall
3. State-of-the-Art and Related work

that RTEs and FTAs recognize exactly the class of regular tree languages [16].

3.3.1.1 Construction of (Finite) Tree Automata from Regular (Tree) Expressions

Kuske and Meinecke published a method [29] of creating FTAs from RTEs. This research was inspired by the work of Champaurnaud and Ziadi [14, 15] in the area of strings. Kuske and Meinecke adapted Antimirov’s partial derivatives method [7] for regular (string) expressions to work with RTEs. It is also important to mention, that this work uses the c-product definition of the RTEs. The resulting FTA can be created in $O(R \cdot |E|^2)$ time where $|E|$ is the size of the syntax tree of the input RTE and $R$ is the maximum rank appearing in the ranked alphabet.

The paper mainly deals with the extension of Antimirov’s partial derivatives to work with RTEs. The concept of partial derivatives yields a FTA with at most $\|E\|$ states and $\|E^2\|$ transitions where $\|E\|$ is the number of occurrences of symbols from the ranked alphabet in $E$.

It is an interesting fact that, as far as our research went, derivatives in the sense defined by Brzozowski are not adapted to tree languages. However, Antimirov’s partial derivatives seem like an interesting topic for a future research, mainly in the area of a conversion algorithm that would result in deterministic PDA.

3.3.1.2 From Regular Tree Expression to Position Tree Automaton

The adaptation of the position automaton for RTEs was presented in the work of Laugerotte et al. [30]. They achieved to create a FTA with the properties similar to the position automaton by Glushkov. In order to do that, the basic functions appearing in the Glushkov’s algorithm (First, Follow, Pos), were adapted to work with RTEs.

The result of this algorithm is a position (finite) TA. This automaton is a homogeneous non-deterministic bottom-up FTA. Note that bottom-up TAs can always be determinised [16].

As in the work of Kuske and Meinecke, the alternative definition of RTEs is used, i.e. there is no alphabet $K$, only $F$, and every symbol with zero arity can appear in the substitution. In order to achieve good complexity, they also adapted so-called ZPC structure to trees. This enables the computation of the Follow function to be asymptotically faster. Using the ZPC data structure to allow fast queries for the computation of Follow, the FTA equivalent to the RTE $E$ is created in $O(|E| \cdot \|F\|)$ time. This is asymptotically faster than the work of Kuske and Meinecke [29].
3.3. Regular tree expression conversion

3.3.1.3 From Regular Tree Expression to Position Tree Automaton

Very recently, new study of Mignot et al. appeared [35]. In strings, there exist several methods of computing an FA recognizing the same language as some regular expression (RE). Some of them were proven to be isomorphic and some of them were proven to be a quotient of others.

This paper discuss that the mentioned relations are still valid in the domain of trees and RTE and FTA.

3.3.2 Automaton to regular tree expression

In the domain of strings, there are algorithms for the opposite direction of conversion, i.e. from FAs to REs. The basic algorithms for this conversion in the string domain are:

- State Removal Technique (by Brzozowski and McCluskey [12], also in [11]),
- Algebraic Method (by Brzozowski [11], also in [26]) and
- Transitive Closure (by Kleene [27]).

Although the survey for existing similar algorithms in the tree domain was done thoroughly, no algorithm for the conversion of PDAs or FTAs to RTEs was found. The conversion from PDAs would be difficult as the classes of regular tree languages and of deterministic context-free languages (CFLs) are not equal. Firstly, at least some kind of check whether the input PDA accepts a regular tree language would have to be invented. There is however a good opportunity for a future research in the area of converting FTAs to RTEs [16, Prop. 2.2.7].
This section is split into two parts. The first section summarises the progress in visibly pushdown automata (VPDAs) determinisation. The second section summarises the real-time height-deterministic PDAs (rHPDAs) determinisation.

All necessary background and basic notions were explained in previous chapters.

This chapter contains the main results presented as a conference paper [A.2] with contributions made by Jan Trávníček as a second author, and the main results presented as a journal paper [A.3].

4.1 Introduction

Pushdown automata (PDAs) which accept context-free languages (CFLs) are one of the fundamental models of computation of the Theory of formal languages and automata [24]. Every nondeterministic finite automaton (FA) which accepts a regular language (RL) can be determinised and the theory of the determinisation of FAs is simple and well-researched: states of an equivalent deterministic FA represent so-called deterministic subsets of states of a given nondeterministic FA [24]. The general determinisability does not hold for the case of all types of nondeterministic PDAs. The class of deterministic CFLs is a proper subclass of CFLs i.e. for some nondeterministic PDAs, their equivalent deterministic versions do not exist. Generally, it is not known how to decide for a given nondeterministic PDA whether there exists a deterministic equivalent or not. There is a lack of results in the theory of the determinisation of determinisable classes of PDAs (general case is undecidable), although such results would be usable, e.g. when constructing practical deterministic algorithms from nondeterministic PDAs.
4.2 Determinisation of the visibly pushdown automaton

4.2.1 Introduction

Visibly pushdown automata (VPDAs) \cite{6} are an important and well-motivated subclass of pushdown automata (PDAs), where pushdown operations are determined by the input symbol: the input alphabet is partitioned into three parts $A_c$, $A_r$, and $A_l$ for a push, pop and local pushdown operations, respectively. This relates, for example, to function calls. A function call is represented by push operation, local operations executed in the context of the called function are represented by local transitions, and, finally, the return from the function is represented by the pop operation. The push, pop and local operations are sometimes referred to as call, return and internal operations. VPDAs are widely used, researched and known to be used in many practical applications, such as XML processing \cite{4, 5, 10, 17, 18, 44}.

It is well known that nondeterministic VPDAs can be determinised \cite{6, 45}. These determinisations use principles that are also used in the well-known determinisation of finite automata (FAs): states of the equivalent deterministic FAs are represented by so-called deterministic subsets \cite{24}. Alur and Madhusudan \cite{6} presented the proof of the determinisability of a given nondeterministic VPDA with $n$ states by creating a cartesian product consisting of all possible states and then creating deterministic subsets, which resulted in $2^{n^2+n}$ states of the deterministic version of the PDA. This was improved in \cite{45}, where the upper bound for the number of states was lowered from $2^{n^2+n}$ to $2^{n^2}$ and the upper bound for the number of pushdown store symbols was lowered from $|A_c|2^{n^2+n}$ to $|A_c|2^{n^2}$.

The class of visibly pushdown languages (VPLs) accepted by VPDAs explains, unifies, and extends a natural model of control flow of computation in a typical programming language with nested invocations of code organized in functions and methods. Yet there are numerous problems that can be formulated as decision problems for PDAs like program analyses, compiler optimisation as well as other stringological and arbolescent problems.

Since Alur and Madhusudan proposed the class of VPLs \cite{6}, a few algorithms for VPDAs determinisation were published. However, because of shortcomings in these algorithms, the only reasonable way of processing calculations over VPLs was a simulation of nondeterministic VPDAs. The determinisation of these PDAs was proven, but the algorithm directly derived from this proof was too exhaustive. Namely, unreachable states and redundant transitions were also generated. Therefore this algorithm was ineffective to be of much use for the practical problems.

In this section, the algorithm for the determinisation of nondeterministic VPDAs is presented. The determinisation algorithm improves the original determinisation algorithm \cite{6, 45}. As was stated in chapter 1, the algorithm computes and constructs only accessible
4.2. Determinisation of the visibly pushdown automaton

states and necessary pushdown symbols of the deterministic pushdown automaton. The basic idea of this improvement is to analyze and to track the pushdown symbols that can appear on the top of the pushdown store for a particular state. With this information, the pop transitions that are used for the state can be reduced to only those that correspond to the possible pushdown store top symbols. It is also shown below that this information on the possible pushdown store top symbols for a state has certain interesting properties, which can be exploited for an effective way to calculate and store this information for all states in the automaton.

This section contains main results presented as a conference paper [A.2].

4.2.2 Determinisation algorithm

The \( \mathcal{T}_q \) is used to denote pushdown store top symbols. The pushdown store top symbols of state \( q \), \( \mathcal{T}_q \subseteq G' \), are the set of all pushdown symbols that could appear at the top of the pushdown store for a state \( q \in Q' \).

The symbol \( \lambda \) is used for a local connection: State \( q' \) is locally connected to \( q'' \) if there is a sequence of transitions from state \( q'' \) to state \( q' \), the pushdown store depth in both states is the same and the pushdown store depth for all other states along the sequence of transitions is greater than the pushdown store depth in \( q'' \). This relation between the two states is not symmetric but is transitive. For an example of various local connections, see Figure 4.1. The notation \( a|\alpha \mapsto \beta \) denotes a transition that reads symbol \( a \in A \) and replaces \( \alpha \in G'^* \) with \( \beta \in G'^* \) on the top of the pushdown store. This notation will be used in the figures throughout the paper. Note that for example path \((2, x, 3)\) in Figure 4.1 is not a local connection.

It can be seen easily that the pushdown store top symbols are shared between locally connected states.

With a local connection from \( q'' \) to \( q' \) and a local connection from \( q''' \) to \( q'' \), \( q' \) is locally connected to \( q''' \) transitively. If \( \mathcal{T}_q \) is the set of pushdown store top symbols of state \( q \), then \( \mathcal{T}_{q'} \subseteq \mathcal{T}_{q''} \) and \( \mathcal{T}_{q'} \subseteq \mathcal{T}_{q''} \) and also \( \mathcal{T}_{q'} \subseteq \mathcal{T}_{q''} \).

The local closure of state \( q \) is \( \lambda^*(q) \). See Figure 4.2 as an example of a local closure. See Figure 4.1 and note that paths \((0, 1, 2, x, z, 3, 4, 5, 6, 7, 8, 9), (1, 2, x, 3, 4) \) and others are in fact local closures.

Closing all states under the \( \lambda^*(q) \) connects all \( \mathcal{T}_q \). See Figure 4.3.

These notions are formally defined:

**Definition 4.1** (Local connection \( \lambda(q) \) of state \( q \)). Given a deterministic VPDA \( M' = (Q', A, G', \delta', \{q_0\}, \perp, F') \), where \( A = A_c \cup A_r \cup A_l \), states \( q, q' \in Q' \), then \( \lambda(q) = \{q' : (q, uw, \gamma_0) \vdash^k (q', w, \gamma_k), u, w \in A^*, \gamma_0, \gamma_k \in G'^*, 1 \leq k, |\gamma_0| = |\gamma_k|, |\gamma_0| < |\gamma_i|, 1 \leq i < k\} \).
Definition 4.2 (Local connection closure \( \lambda^*(q) \) of state \( q \)). The local connection closure \( \lambda^* \) is defined by the following equalities:

\[
\begin{align*}
\lambda^0(q) &= q, \\
\lambda^{i+1}(q) &= \bigcup_{q' \in \lambda^i(q)} \lambda(q'), \\
\lambda^*(q) &= \bigcup_{i \geq 0} \lambda^i(q).
\end{align*}
\]

Definition 4.3 (Set of pushdown store top symbols \( T_q \) of state \( q \)). Given a deterministic visibly pushdown automaton \( M' = (Q', A, G', \delta', \{q_0'\}, \perp, F') \), where \( A = A_c \cup A_r \cup A_l \), states \( q, q', q'' \in Q' \), then \( T_q = \{ g : (q', g) \in \delta(q'', c, \epsilon), c \in A_c, g \in G', q' \in \lambda^*(q) \} \).

Due to the convenient properties of \( \lambda^*(q) \), \( T_q \) can be stored in a space-optimal way. Given \( q, q' \in Q' \), then \( \forall q' \in \lambda^*(q) \) holds that \( T_{q'} \subseteq T_q \), i.e. parts of \( T_q \) can be shared.
4.2. Determinisation of the visibly pushdown automaton

Figure 4.3: The \( \lambda^* \) relation computed for all states

Figure 4.4: Connecting \( T_q \) between locally connected states

between locally connected and locally closed states. See Figure 4.4 as an example of locally connected states.

**Pushdown store top symbols** \( T \). Pushdown store top symbols \( T_q \) are given by the \( \lambda^*(q) \) and states that are the source and target of the appropriate push and pop transition.

**Induction, step one:**

\[
e \vdash \bot \in T_{q_0}.
\]  

Assume that pushdown store top symbols are given by local closure for the first \( i \) symbols of the input string. Symbol \( a \) is the \((i + 1)\)-th symbol of the input string. Given \( q_1, q_2, q_3, r_1, r_2, r_3 \in Q', a \in A_c \), there are three distinct cases:

\[
\lambda(r_1) = \{q_1 : (q_1, aw, \gamma) \vdash (r_1, w, \gamma)\},
\]  

\[
T_{r_2} = \{(q_2, a) : (q_2, aw, \gamma) \vdash (r_2, w, (q_2, a)\gamma)\},
\]  

\[
\lambda(r_3) = \{q_2 : (q_3, aw, (q_2, a)\gamma) \vdash (r_3, w, \gamma)\}.
\]
Note that $\gamma$ does not change between states $q_1$ and $r_1$. The pushdown store top symbol $(q_2, a)$ was pushed in state $q_2$ and was popped in state $q_3$, so $\gamma$ also does not change between states $q_2$ and $r_3$. Pushdown store top symbols are given by the $\lambda^*(q)$ and states that are the source and target of an appropriate push and pop transition for the first $i + 1$ symbols of the input string.

Induction holds for $i + 1$. \qed

More informally: The deterministic automaton is constructed starting from the initial state $q_0'$. The deterministic subset of the initial state is created from all initial states of the original automaton. The initial pushdown store symbol $\bot \in T_{q_0'}$ forms the set of pushdown store top symbols.

In each iteration, all local and push transitions are explored for known states. The base set of possible pushdown store top symbols $T_q$ of the pushdown store for a given state $q \in Q'$ is given by the push transitions. Then, the pushdown store top symbols for each state are tracked.

Any two states $q, q'$, where a local transition exists from state $q'$ to state $q$, share part of $T_{q'}$ in the form that $T_{q'} \subseteq T_q$.

Any two states $q, q'$, where a pop transition popping symbol $(q', r)$ exists from state $x$ to state $q$, share part of $T_{q'}$ in form of that everything from $T_{q'}$ is in $T_q$. Given $q, q', q'' \in Q', l \in \mathcal{A}_l, c \in \mathcal{A}_c, r \in \mathcal{A}_r$, then for $T$ the following properties hold:

\begin{align}
\bot & \in T_{q_0'}, \quad (4.8) \\
\forall (q, \varepsilon) \in \delta(q', l, \varepsilon) \Rightarrow T_{q'} \subseteq T_q, \quad (4.9) \\
\forall (q, (q'', c)) \in \delta(q'', c, \varepsilon) \Rightarrow (q'', c) \subseteq T_q, \quad (4.10) \\
\forall (q, \varepsilon) \in \delta(q'', r, (q', c)) \Rightarrow T_{q'} \subseteq T_q. \quad (4.11)
\end{align}

Only a pair of states $(q', q)$, $q' \in \lambda^*(q)$ can appear as an element in the $S$ component of a state. As is described in Chapter 2, new pairs of $S$ components are only created from the push and local transitions. Push transitions create only elements of the $S$ component of state based on identity. However, the local transitions create exactly the pairs that conform to a local closure, because they connect appropriate targets of push and sources of pop operations.

The determinisation algorithm is formally described in Algorithm 4.1. The algorithm uses two main data structures: a queue $\text{Dirty}$, which holds discovered states, and a set $\text{Clean}$, which holds states for which push and local transitions were successfully created.

Its main part can be informally described as follows:

1. Create the initial state.

2. Initialize queue $\text{Dirty}$ with the initial state.
4.2. Determinisation of the visibly pushdown automaton

\textbf{Algorithm 4.1: Visibly pushdown automaton determinisation algorithm}

\textbf{Input:} Nondeterministic $\text{VPDA}(M_0(Q, A_c \cup A_t \cup A_r, G, \delta, Q_0, \bot, F))$.

\textbf{Output:} Equivalent deterministic $\text{VPDA}(M_d(Q', A_c \cup A_t \cup A_r, G', \delta', \{q'_0\}, \bot, F'))$.

1. $q'_0 \leftarrow \{(x, x) : x \in Q_0\}$, $G' \leftarrow \{q'_0\}$, $G' \leftarrow \emptyset$, $\delta' \leftarrow \emptyset$, $F' \leftarrow \emptyset$.

2. \textbf{init} $\text{queue} \text{Dirty} \leftarrow ((q'_0, \bot))$; // The set of pairs (state, symbol)

3. \textbf{init} set $\text{Clean} \leftarrow \emptyset$; // The set of states

4. \textbf{while} $\text{Dirty}$ is not empty do // Main loop over dirty states

5. \hspace{1em} $(q, \gamma) \leftarrow \text{dequeue} \text{Dirty}$;

6. \hspace{1em} \text{init} set $\text{States} \leftarrow \emptyset$;

7. \hspace{1em} if $q \notin \text{Clean}$ then

8. \hspace{2em} \textbf{foreach} $l \in A_l$ do // Local transitions

9. \hspace{3em} \textbf{foreach} $(q'', q_2) \in q$ do

10. \hspace{4em} $q' \leftarrow \{(q', q_1) : (q_1, \varepsilon) \in \delta(q_2, l, \varepsilon)\}$;

11. \hspace{4em} if $q' = \emptyset$ then continue;

12. \hspace{4em} add $q'$ to $Q'$; add $(q', \varepsilon)$ to $\delta'(q, l, \varepsilon)$; add $q'$ to $\text{States}$;

13. \hspace{4em} update $T_x, x \in \lambda^*(q')$;

14. \hspace{2em} \textbf{foreach} $c \in A_c$ do // Push transitions

15. \hspace{3em} \textbf{foreach} $(q'', q_2) \in q$ do

16. \hspace{4em} $q' \leftarrow \{(q_1, q_1) : (q_1, \varepsilon) \in \delta(q_2, c, \varepsilon)\}$;

17. \hspace{4em} if $q' = \emptyset$ then continue;

18. \hspace{4em} add $q'$ to $Q'$; add $(q', (q, c))$ to $\delta'(q, c, \varepsilon)$; add $q'$ to $\text{States}$;

19. \hspace{4em} add $(q, c)$ to $G'$;

20. \hspace{4em} update $T_x, x \in \lambda^*(q')$;

21. \hspace{2em} add $q$ to $\text{Clean}$;

22. \hspace{1em} \textbf{foreach} $r \in A_r$ do // Pop transitions

23. \hspace{2em} if $\gamma = \bot$ then // Bottom of the stack

24. \hspace{3em} \textbf{foreach} $(q'', q_2) \in \text{state}$ do

25. \hspace{4em} $q' \leftarrow \{(q'', q_1) : (q_1, \varepsilon) \in \delta(q_2, r, \bot)\}$;

26. \hspace{4em} if $q' = \emptyset$ then continue;

27. \hspace{4em} add $q'$ to $Q'$; add $(q', (q, c))$ to $\delta'(q, r, \gamma)$; add $q'$ to $\text{States}$; update $T_x, x \in \lambda^*(q')$;

28. \hspace{1em} else

29. \hspace{2em} \textbf{init} set $\text{Update} \leftarrow \emptyset$; \textbf{foreach} $(q_1, q_2) \in q$ do

30. \hspace{3em} add $\{(q'', q_1) : (q_1, g) \in \delta(q_2, r, \varepsilon), (q_1, \varepsilon) \in \delta(q'', \text{second}(\gamma), g)\}$ to $\text{Update}$;

31. \hspace{2em} \textbf{foreach} $(q'', q_3) \in \text{car}(\gamma)$ do

32. \hspace{3em} $q' \leftarrow \{(q'', q_2) : (q_2) \in \text{Update}\}$;

33. \hspace{3em} if $q' = \emptyset$ then continue;

34. \hspace{3em} add $q'$ to $Q'$; add $(q', (q, \varepsilon))$ to $\delta'(q, r, \gamma)$; add $q'$ to $\text{States}$;

35. \hspace{3em} update $T_x, x \in \lambda^*(q')$;

36. \hspace{2em} \textbf{foreach} $q' \in \text{States}$ do // Enqueue new dirty states

37. \hspace{3em} \textbf{foreach} $q \in T_q \setminus T_q'$ do

38. \hspace{4em} enqueue $(q', q)$ to $\text{Dirty}$;

39. \hspace{2em} \textbf{foreach} $q' \in Q'$ do // Set final states

40. \hspace{3em} if exists $(q, q) \in q'$ such that $q \in F$ then

41. \hspace{4em} add $q'$ to $F'$;
4. Pushdown Automata Determinisation

3. Initialize set Clean as an empty set.

4. Dequeue state from queue Dirty until it is not empty:
   
   o If a state is not in Clean:
     
     – Create local transitions, enqueue new states in queue Dirty.
     – Create push transitions, enqueue new states in queue Dirty.
     – Put the state in set Clean.
   
   o Create pop transitions.

5. Create the set of final states.

Correctness of the algorithm can be easily checked. Algorithm uses rules for creating initial state as defined by Nguyen Van Tang [45] and rules for creating local, push, and pop transitions as defined by Alur and Madhusudan [6]. The R component is omitted [45]. It maintains a $T_q$ set of pushdown store top symbols for each state as defined above. And only those pop transitions with pop symbol that belongs in the $T_q$ set for a processed state are created. The $T_q$ was proven to be enough to satisfy all states reachable through pop transitions by Figure 4.2.2.

Two structures are used to keep track of states processed: a queue Dirty, which holds discovered states, and a set Clean, which holds states for which all push and local transitions were successfully created. Creating local and push transitions is a one time operation for a state and all reachable states are guaranteed to pass this operation, because all states reached are queued in the queue Dirty and algorithm does not stop until the queue Dirty is empty. Creating pop transitions for a state is done every time that state is processed (taken from a queue Dirty) and new pushdown store top symbols are added into pushdown store top symbols set $T_q$. These pushdown store top symbols need to be processed. After creating all of the pop transitions for a state, targets of these pop transitions are added to the queue Dirty, meaning that all affected states – states for which the set of pushdown store top symbols $T_q$ may have changed are processed for a new pop transitions. All possible pop transitions are guaranteed to be create, because all states that may have been affected will be queued in the queue Dirty and algorithm does not stop until the queue Dirty is empty.

If the algorithm creates all of the possible push, local, and pop transitions (with regards to $T_q$ set) for all reachable states, then it must be correct.

4.2.3 Example

Determinisation of a simple nondeterministic visibly pushdown automaton is demonstrated. For the sake of clarity, the $R$ component of the states and pushdown symbols is omitted,
because the information it holds is already contained in the $S$ component (it is the second value of each pair), as was described in [45].

**Example 4.1.** Non-deterministic visibly pushdown automaton $a_1$ is shown in Figure 4.5.

Let $a_1 = (Q, \mathcal{A}_c \cup \mathcal{A}_r \cup \mathcal{A}_l, G, \delta, Q_0, \bot, F)$ be a non-deterministic VPDA. The equivalent deterministic VPDA $d_1 = (Q', \mathcal{A}_c \cup \mathcal{A}_r \cup \mathcal{A}_l, G', \delta', \{q'_0\}, \bot, F')$ is constructed as follows.

The initial state is constructed as a power set of all identity pairs of initial states of automaton $a_1$, so $q'_0 = \{(0, 0)\}$. 

![Figure 4.5: The nondeterministic VPDA $a_1$ from Example 4.1](image)

<table>
<thead>
<tr>
<th>State $q$</th>
<th>${(0, 0)}$</th>
<th>${(0, 0), (1, 1)}$</th>
<th>${(0, 1)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pushdown store top symbols $T_q$</strong></td>
<td>${}</td>
<td>\bot, (a, {(0, 0)})$, $(a, {(0, 0), (1, 1)})$</td>
<td>$(a, {(0, 0)})$, $(a, {(0, 0), (1, 1)})$</td>
</tr>
<tr>
<td>1. $a</td>
<td>\varepsilon \rightarrow (a, {(0, 0)})$</td>
<td>${(0, 0), (1, 1)}$</td>
<td>${(0, 1)}$</td>
</tr>
<tr>
<td>2. $c</td>
<td>\varepsilon \rightarrow \varepsilon$</td>
<td>${(0, 1)}$</td>
<td>${(0, 1)}$</td>
</tr>
<tr>
<td>3. $a</td>
<td>\varepsilon \rightarrow (a, {(0, 0), (1, 1)})$</td>
<td>${(0, 0), (1, 1)}$</td>
<td>${(0, 1)}$</td>
</tr>
<tr>
<td>4. $d</td>
<td>(a, {(0, 0), (1, 1)}) \varepsilon \rightarrow \varepsilon$</td>
<td>${(0, 1)}$</td>
<td>${(0, 1)}$</td>
</tr>
<tr>
<td>5. $d</td>
<td>(a, {(0, 0)}) \varepsilon \rightarrow \varepsilon$</td>
<td>${(0, 1)}$</td>
<td>${(0, 1)}$</td>
</tr>
<tr>
<td>6. $b</td>
<td>(a, {(0, 0)}) \varepsilon \rightarrow \varepsilon$</td>
<td>${(0, 1)}$</td>
<td>${(0, 1)}$</td>
</tr>
</tbody>
</table>
4. Pushdown Automata Determinisation

In each push transition, the pushdown store top symbol is tracked for the target state of the push transition. When a local transition occurs, all pushdown store top symbols are shared from a source state of the local transition to a target state of the local transition and all other locally connected states. Tracking of all locally connected states may be achieved by creating virtual transitions serving as a transitive closure.

Then, pop transitions are created on the basis of known input symbols (from the non-deterministic automaton) and tracked pushdown store top symbols. The transitions are created according to the determinisation rules above.

For an illustration of the determinisation, see Table 4.1.

The resulting deterministic PDA $d_1$ is shown in Figure 4.6.

The resulting deterministic PDA was produced by the algorithm shown in Algorithm 4.1.

4.2.4 Evaluation of results

For the nondeterministic pushdown automaton from the example above, the Nguyen2015 algorithm [36] constructs a deterministic pushdown automaton with 45 states and 1206 transitions. It is noted that 45 (states) is not a power of 2. This is caused by the fact that the implementation of Nguyen2015 library does not consider states in which $R$ or $S$ components are empty sets (it optimises the determinisation algorithm [45]). For comparison, our algorithm constructs an equivalent deterministic pushdown automaton with only 3 states and 8 transitions. This is a significant improvement compared to previously existing determinisation algorithms [6,45,36].
4.2.5 Conclusion

A new incremental algorithm of the determinisation of nondeterministic VPDAs has been described. The algorithm creates only necessary states and pushdown symbols by analysing and tracking which states are achievable by computing transitions on the same depths of the pushdown store. Possible tops of the pushdown store are stored for each state when a pop transition is in progress and then they are shared through local transitions with states on the same depths of the pushdown store. The behaviour of the algorithm is inspired by the behaviour of the incremental construction of the deterministic FA.

The algorithm has been implemented as a part of an experimental automata library [47]. Although the number of states of the deterministic PDA for the worst case is still $2^{n^2}$ for a given nondeterministic VPDAs with $n$ states, it has been shown that the upper bound of the number of states of the deterministic PDA is dependant on the number of distinct pairs of locally connected states. For those and other reasons in many practical cases, the new algorithm provides significantly smaller deterministic PDAs than the previously existing determinisation algorithms.
4.3 Determinisation of the real-time height-deterministic pushdown automaton

4.3.1 Introduction

Height-deterministic pushdown automata (HPDAs) are a natural generalisation of the visibly pushdown automata (VPDAs) proposed by Nowotka and Srba [37]. This fact leads to a method of real-time height-deterministic PDAs (rHPDAs) determinisation being very similar to a VPDAs in particular, the method is just more general. A similar incremental algorithm of the determinisation of rHPDAs has been described. The algorithm creates only necessary states and pushdown symbols by analysing and tracking which states are achievable by computing transitions on the same depths of the pushdown store. Possible tops of the pushdown store are stored for each state when a pop transition is in progress and then they are shared through local transitions with states on the same depths of the pushdown store. The behaviour of the algorithm is inspired by the behaviour of the VPDAs determinisation or more generally on the incremental construction of the deterministic finite automata (FAs).

In this section, the algorithm for the determinisation of nondeterministic rHPDAs is presented. The algorithm improves the original determinisation algorithm [37, 6, 45]. As stated in the introduction the algorithm computes and constructs only necessary and accessible states and pushdown symbols of the deterministic pushdown automaton (PDA). The basic idea of this improvement is analysing and tracking pushdown symbols that can appear on the top of the pushdown store for a particular state. With this information, the explored pop transitions for the state can be reduced to only those that correspond to the possible pushdown store top symbols. Also, it is shown below that this information on the possible pushdown store top symbols for a state has certain interesting properties, which can be exploited for an effective way of calculation and storing this information for all states in the PDA.

This section contains main results presented as journal paper [A.3] and is also based on the results presented as a conference paper [A.2].

4.3.2 Determinisation algorithm

As was defined above, VPDAs use different normalisation for proofs of their determinisability. To bridge this gap, the Alur-Madhusudan normalisation is used for the determination algorithm. A rHPDA with Alur-Madhusudan normalisation can be renormalised to Srba-Nowotka normalisation.

An Alur-Madhusudan normalised pushdown automaton is automaton $M = (Q, A, G, \delta, Q_0, Z_0, F)$ with a transition relation in the form $(q, X) \in \delta(p, a, \varepsilon)$
4.3. Determinisation of the real-time height-deterministic pushdown automaton

or \((q, \varepsilon) \in \delta(p, a, X)\) or \((q, \varepsilon) \in \delta(p, a, \varepsilon)\), where \(p, q \in Q\), \(X \in G\) and \(a \in A\), which means that the automaton is already real-time. A pushdown automaton can be renormalised into Srba-Nowotka normalisation to automaton \(M = (Q, A, G, \delta, Q_0, Z_0, F)\) with a transition relation in the form \((q, YX) \in \delta(p, a, X)\) or \((q, \varepsilon) \in \delta(p, a, X)\) or \((q, X) \in \delta(p, a, X)\), where \(p, q \in Q\), \(X \in G\) and \(a \in A\), by appending all \(T_p\) to all push and local transition.

This conversion shows that the proof of the determinisability of \(rHPDAs\) also holds for \(rHPDAs\) with Alur-Madhusudan normalisation.

There seems to be a flaw in the theoretical proof for the determinisation of \(rHPDAs\) which is indifferent to normalisation. As it stands, the algorithm directly derived from the proof generates transitions to the garbage state of the automaton which violates determinism. This can be handled by removing all transitions to the garbage state and the garbage state itself. In reality, these transitions are superfluous and do not lead to the acceptance of any string, but this should be clearly mentioned. Determinisation algorithm is formally described in Algorithm 4.2

The most notable difference between the determinisation of \(VPDAs\) and \(rHPDAs\) is in the fact, that for \(rHPDAs\) there is no alphabet partitioning and a cleanup of garbage states and transitions is needed at the end of the computation.

The algorithm is very similar to Algorithm 4.1 and its correctness can be checked in similar way. Algorithm uses rules for creating initial state as defined by Nowotka and Srba \[37\] with respect to the improvement by Nguyen Van Tang \[45\] and rules for creating local, push, and pop transitions as defined by Nowotka and Srba \[37\]. The \(R\) component is omitted \[45\].

Otherwise the algorithm uses set of pushdown store top symbols \(T_q\) in a very similar way as Algorithm 4.1

4.3.3 Nowotka-Srba normalisation

The algorithm described in Algorithm 4.2 can be extended for the Nowotka-Srba normalisation of the \(rHPDA\). The modified algorithm needs to handle the additional pushdown store symbol. Determinisation algorithm for \(rHPDAs\) in Nowotka-Srba normalisation is formally described in Algorithm 4.3

4.3.4 Example

Determinisation of a simple nondeterministic \(VPDA\) is demonstrated. For the sake of clarity, the \(R\) component of the states and pushdown symbols is omitted, because the information it holds is already contained in the \(S\) component (it is the second value of each pair), as was described in \[45\].
4. Pushdown Automata Determinisation

**Algorithm 4.2: Real-time height-deterministic PDA determinisation algorithm**

- **Input:** Nondeterministic rHPDA \( M_n(Q, A, G, \delta, Q_0, \bot, F) \).
- **Output:** Equivalent deterministic rHPDA \( M_d(Q', A, G', \delta', \{q'_0\}, \bot, F') \).

1. \( q'_0 \leftarrow \{(x, x) : x \in Q_0\}, Q' \leftarrow \{q'_0\}, G' \leftarrow \emptyset, \delta' \leftarrow \emptyset, F' \leftarrow \emptyset; \)
2. \( \text{init queue Dirty} \leftarrow ((q'_0, \bot)); \) // The set of pairs (state, symbol)
3. \( \text{init set Clean} \leftarrow \emptyset; \) // The set of states
4. while Dirty is not empty do // Main loop over dirty states
5. \( (q, \gamma) \leftarrow \text{dequeue Dirty}; \)
6. \( \text{init set States} \leftarrow \emptyset; \)
7. if \( q \notin \text{Clean} \) then
8. \( \text{foreach } a \in A \text{ do // Local & push transitions} \)
9. \( \text{foreach } (q'', q_2) \in q \text{ do // Local transitions} \)
10. \( q' \leftarrow \{(q'', q_1) : (q_1, \varepsilon) \in \delta(q_2, a, \varepsilon)\}; \)
11. if \( q' = \emptyset \) then continue;
12. add \( q' \) to \( Q' \); add \( (q', \varepsilon) \) to \( \delta'(q, a, \varepsilon) \); add \( q' \) to States;
13. update \( T_x, x \in \lambda^*(q') \);
14. \( \text{foreach } (q'', q_2) \in q \text{ do // Push transitions} \)
15. \( q' \leftarrow \{(q_1, q_1) : (q_1, g) \in \delta(q_2, a, \varepsilon)\}; \)
16. if \( q' = \emptyset \) then continue;
17. add \( q' \) to \( Q' \); add \( (q', (q, a)) \) to \( \delta'(q, a, \varepsilon) \); add \( q' \) to States;
18. add \( (q, a) \) to \( G' \);
19. update \( T_x, x \in \lambda^*(q') \);
20. add \( q \) to \( \text{Clean} \);
21. \( \text{foreach } a \in A \text{ do // Pop transitions} \)
22. \( \text{init Update} \leftarrow \emptyset; \)
23. \( \text{foreach } (q_1, q_2) \in q \text{ do } \)
24. \( \text{add } \{(q'', q_1) : (q_1, g) \in \delta(q_2, a, \varepsilon), (q_1, \varepsilon) \in \delta(q'', \text{car}(\gamma), g)\} \text{ to Update}; \)
25. \( \text{foreach } (q'', q_3) \in \text{car}(\gamma) \text{ do } \)
26. \( q' \leftarrow \{(q'', q_3) : (q_3, q_2) \in \text{Update}\}; \)
27. if \( q' = \emptyset \) then continue;
28. add \( q' \) to \( Q' \); add \( (q', (q, \varepsilon)) \) to \( \delta'(q, a, \gamma) \); add \( q' \) to States;
29. update \( T_x, x \in \lambda^*(q') \);
30. \( \text{foreach } q' \in \text{States do // Enqueue new dirty states} \)
31. \( \text{foreach } g \in T_q \setminus T_{q'} \text{ do } \)
32. \( \text{enqueue } (q', g) \text{ to Dirty}; \)
33. \( \text{foreach } q' \in Q' \text{ do // Set final states} \)
34. \( \text{if exists } (q_1, q) \in q' \text{ such that } q \in F \text{ then } \)
35. \( \text{add } q' \text{ to } F'; \)
36. \( \text{remove all transition leading to state } \emptyset \text{ from } \delta'; \) // Garbage transitions
37. \( \text{remove state } \emptyset \text{ from } Q'; \) // Garbage state
Algorithm 4.3: Real-time height-deterministic PDA determinisation algorithm (Nowotka-Srba normalisation)

Input: Nondeterministic \(\mathcal{HPDA} M_0(Q, A, G, \delta, Q_0, \bot, F)\).

Output: Equivalent deterministic \(\mathcal{HPDA} M_d(Q', A, G', \delta', \{q'_0\}, \bot, F')\).

1. init \(q'_0 \leftarrow \{(x, (x, \bot)) : x \in Q_0\}, Q' \leftarrow \{q'_0\}, G' \leftarrow \emptyset, \delta' \leftarrow \emptyset, F' \leftarrow \emptyset\);
2. init queue Dirty \(\leftarrow ((q'_0, \bot))\); // The set of pairs (state, symbol)
3. init set Clean \(\leftarrow \emptyset\); // The set of states
4. while Dirty is not empty do // Main loop over dirty states
   5. \((q', \gamma) \leftarrow \text{dequeue Dirty};\)
   6. init set States \(\leftarrow \emptyset\);
   7. if \(q \notin \text{Clean then}\)
      8. foreach \(a \in A\) do // Local & push transitions
         9. foreach \((q'', (q_2, g)) \in q\) do // Local transitions
            10. \(q' \leftarrow \{(q'', (q_1, g)) : (q_1, q_1, g_1) \in \delta(q_2, a, g_1)\};
            11. if \(q' = \emptyset\) then continue;
            12. add \(q'\) to \(Q'\); add \((q', \gamma_1)\) to \(\delta'(q_2, a, g_1)\); add \(q'\) to States;
            13. update \(T_x, x \in \lambda^*(q)\);
      14. foreach \((q'', (q_2, g)) \in q\) do // Push transitions
            15. \(q' \leftarrow \{(q_1, (q_1, g)) : (q_1, g_2, g_1) \in \delta(q_2, a, g_1)\};
            16. if \(q' = \emptyset\) then continue;
            17. add \(q'\) to \(Q'\); add \((q', (q, a), g)\) to \(\delta'(q_2, a, g_1)\); add \(q'\) to States;
            18. add \((q, a)\) to \(G'\);
            19. update \(T_x, x \in \lambda^*(q)\);
      20. add \(q\) to \(\text{Clean};\)
   21. foreach \(a \in A\) do // Pop transitions
      22. init set Update \(\leftarrow \emptyset\); foreach \((q_1, (q_2, g)) \in \text{state}\) do
         23. add \{\((q'', \gamma_1, g) : (q_1, g_2, g_1) \in \delta(q_2, a, g_1), (q_1, \varepsilon) \in \delta(q'', \text{car}(\gamma), g_2)\}\ to Update;
      24. foreach \((q'', q_3) \in \text{car}(\gamma)\) do
         25. \(q' \leftarrow \{(q'', q_3, q_2) : (q_3, q_2) \in \text{Update}\};
         26. if \(q' = \emptyset\) then continue;
         27. add \(q'\) to \(Q'\); add \((q', (q, a), \gamma)\) to \(\delta'(q_2, a, g_1)\); add \(q'\) to States;
         28. update \(T_x, x \in \lambda^*(q')\);
      29. foreach \(q' \in \text{States}\) do // Enqueue new dirty states
         30. foreach \(g \in T_q \setminus T_{q'}\) do
            31. enqueue \((q', g)\) to Dirty;
   32. foreach \(q' \in Q'\) do // Set final states
      33. if exists \((q_1, (q, \gamma)) \in q'\) such that \(q \in F\) then
         34. add \(q'\) to \(F'\);
   35. remove all transition leading to state \(\emptyset\) from \(\delta'\); // Garbage transitions
   36. remove state \(\emptyset\) from \(Q'\); // Garbage state
Example 4.2. A nondeterministic rHPDA \(a_2\) is shown in Figure 4.7.

Let \(a_1 = (Q, A, G, \delta, Q_0, \bot, F)\) be a nondeterministic rHPDA. An equivalent deterministic rHPDA \(d_2 = (Q', A, G', \delta', \{q'_0\}, \bot, F')\) is constructed as follows.

The initial state is constructed as a power set of all identity pairs of the initial states of automaton \(a_2\), so \(q'_0 = \{(0, 0)\}\).

![Figure 4.7: Nondeterministic HPDA \(a_2\) from Example 4.2](image)

In each push transition, the pushdown store top symbol is tracked for the target state of the push transition. When a local transition occurs, all pushdown store top symbols are shared from a source state of the local transition to a target state of the local transition and all other locally connected states. Tracking of all locally connected states can be achieved by creating virtual transitions serving as a transitive closure.

Pop transitions are then created on the basis of known input symbols (from the nondeterministic PDA) and tracked pushdown store top symbols. The transitions are created according to the determinisation rules above.

The resulting deterministic PDA \(d_2\) is shown in Figure 4.8.

The resulting deterministic PDA can also be reproduced by using the algorithm in Algorithm 4.2.

If the last two steps of the algorithm in Algorithm 4.2 are omitted the resulting PDA may not be deterministic. This is because it may contain garbage return transitions, leading to garbage state, which are not able to accept any string. However, they are not deterministic, because the return transitions for the same input symbols are already present in the PDA.

As was mentioned above, these transitions can be safely removed. A partial visualisation of these transitions in PDA \(d_2\) is shown in Figure 4.9.

4.3.5 Evaluation of results

This section presents the results of the determinisation of PDAs that are produced as a solution of two problem types. The first problem type is to convert regular tree expressions (RTEs) [16] to rHPDAs. The second problem type is a search
4.3. Determinisation of the real-time height-deterministic pushdown automaton

Figure 4.8: The resulting deterministic PDA $d_2$ from Example 4.2 created by determinisation of automaton $a_2$

Figure 4.9: The resulting deterministic PDA $d_2$ from Example 4.2 with nondeterministic garbage transitions and garbage state

of tree patterns with arbitrary subtrees in trees which are converted to PDAs. These two problem types are very different, but both of them yield nondeterministic PDAs determinisable by presented methods.

Solutions to these problem types as PDAs are the results of our related works and they are the reason for pursuing the efficient implementation of PDAs determinisation.

For the comparison tables shown in Table 4.2 and Table 4.4, eight samples of the first problem RTE and eight samples of the second problem tree pattern (TP) were selected.
RTE automata are always two state automata with one initial state and one final state. TP automata have a varying number of states (which increase with the complexity of the pattern), but they also have one initial state and one final state. The presented samples were selected from the sets of 100 handwritten problems of both types. They were selected to behave with ascending complexity.

Table 4.2: Comparison of nondeterministic and deterministic PDA for RTE problem

<table>
<thead>
<tr>
<th>Sample</th>
<th>RTE1</th>
<th>RTE2</th>
<th>RTE3</th>
<th>RTE4</th>
<th>RTE5</th>
<th>RTE6</th>
<th>RTE7</th>
<th>RTE8</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Transitions</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>16</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>Initial states</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Final states</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Deterministic PDAs properties

| States | 2    | 2    | 2    | 2    | 2    | 2    | 2    | 2    |
| Transitions | 12  | 31   | 32   | 66   | 129  | 213  | 320  | 692  |
| Initial states | 1   | 1    | 1    | 1    | 1    | 1    | 1    | 1    |
| Final states | 1   | 1    | 1    | 1    | 1    | 1    | 1    | 1    |

Table 4.3: Comparison of nondeterministic and deterministic PDA for TP problem

<table>
<thead>
<tr>
<th>Sample</th>
<th>TP1</th>
<th>TP2</th>
<th>TP3</th>
<th>TP4</th>
<th>TP5</th>
<th>TP6</th>
<th>TP7</th>
<th>TP8</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td>Transitions</td>
<td>12</td>
<td>12</td>
<td>20</td>
<td>26</td>
<td>32</td>
<td>42</td>
<td>54</td>
<td>60</td>
</tr>
<tr>
<td>Initial states</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Final states</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Deterministic PDAs properties

| States | 20   | 22   | 110  | 458  | 2056 | 8084 | 21384 | 88720 |
| Transitions | 147  | 162  | 1881 | 10006| 88524| 901450| 6220454| 32245320|
| Initial states | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   |
| Final states | 5   | 9   | 33   | 145  | 510  | 1824 | 3485 | 13245 |

For an RTE problem type Table 4.2, the number of transitions of a deterministic PDA behaves very well, and the number of states does not change at all. For a TP problem Table 4.4, the number of transitions and states is clearly exponential, but not nearly as high as the theoretical upper bound $O(2^{n^2})$.

Example 4.3. See Figure 4.10 and Figure 4.11 for the deterministic and nondeterministic PDA for the RTE1 problem from the comparison table Table 4.2.
4.3. Determinisation of the real-time height-deterministic pushdown automaton

Figure 4.10: Nondeterministic PDA for RTE1 from Example 4.3

Figure 4.11: Deterministic PDA for RTE1 from Example 4.3

Example 4.4. See Figure 4.12 for the nondeterministic PDA for the TP1 problem from the comparison table in Table 4.4. Note that a deterministic PDA would be too large to visualize reasonably.

Even though the general upper bound for transitions, states and pushdown store symbols is still $O(2^n)$, there is an important observation on locally closed states. Let $L$ be the set of all distinct pairs of locally closed states of a nondeterministic PDA. $2^{|L|}$ is the maximum number of states of the deterministic PDA. The $|L|$ is at most $n^2$ when all states in the nondeterministic PDA are locally closed, but it is normal behaviour. Several problems were investigated, and the states are usually grouped into several groups of locally closed states. These groups represent states on the same levels of nesting. This leads to the number of locally closed states being significantly lower than $n^2$. This upper bound also directly affects the number of pushdown store symbols and the number of transitions.
4. Pushdown Automata Determinisation

For the PDA in Figure 4.12, the set of locally closed states is as follows:

\[
L = \{(0,0), (1,1), (2,2), (3,3), (4,4), (0,3), (1,2)\}.
\]

The upper bound is \(2^{5^2} = 33\,554\,432\), but a better upper bound can be created \(2^{|L|} = 128\). The actual number of states in a created deterministic PDA is 20, see ???. For a comparison of the upper bound and the improved upper bound, see ???, it contains first four RTE problems from Table 4.2 and first four TP problems from Table 4.4.

Table 4.4: Comparison of the upper bound and the improved per PDA upper bound

<table>
<thead>
<tr>
<th>Sample</th>
<th>RTE1</th>
<th>RTE2</th>
<th>RTE3</th>
<th>RTE4</th>
</tr>
</thead>
<tbody>
<tr>
<td>States (NPDA)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>States (DPDA)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Improved upper bound</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Upper bound</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample</th>
<th>TP1</th>
<th>TP2</th>
<th>TP3</th>
<th>TP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>States (NPDA)</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>States (DPDA)</td>
<td>20</td>
<td>22</td>
<td>110</td>
<td>458</td>
</tr>
<tr>
<td>Improved upper bound</td>
<td>128</td>
<td>1048576</td>
<td>(3.36 \times 10^7)</td>
<td>(3.44 \times 10^{10})</td>
</tr>
<tr>
<td>Upper bound</td>
<td>(3.36 \times 10^7)</td>
<td>(3.36 \times 10^7)</td>
<td>(1.84 \times 10^{19})</td>
<td>(2.42 \times 10^{24})</td>
</tr>
</tbody>
</table>

The new improved upper bound calculated per PDA is significantly lower, but still much higher than the actual result. This upper bound can be calculated by a modified version of the pushdown store top symbol tracking described in subsection 4.3.2. Additional parameters can be considered to improve the upper bound even more.

4.3.6 Conclusion

RHPDAs are a natural generalisation of the VPDAs. This fact leads to a method of RHPDAs determinisation being very similar to a VPDAs. An incremental algorithm of the determinisation of nondeterministic RHPDAs has been described. The algorithm creates
only necessary states and pushdown symbols by analysing and tracking which states are achievable by computing transitions on the same depths of the pushdown store. Possible tops of the pushdown store are stored for each state when a pop transition is in progress and then they are shared through local transitions with states on the same depths of the pushdown store. The behaviour of the algorithm is inspired by the behaviour of the visibly PDAs determinisation [A.2] or more generally on the incremental construction of the deterministic FA.

The algorithm has been implemented as a part of an experimental automata library [17].

Although the number of states of the deterministic PDA for the worst case is still $2^{n^2}$ for a given nondeterministic VPDA with $n$ states, it has been shown that the upper bound of the number of states of the deterministic PDA is dependant on the number of distinct pairs of locally connected states. For those and other reasons in many practical cases, the new algorithm provides significantly smaller deterministic PDAs than the previously existing determinisation algorithms.

Furthermore, it may be possible to extend this approach to other kinds of PDAs. The $k$-Height-deterministic PDAs are proposed as PDAs where for any given input string the difference between pushdown store heights during any (nondeterministic) computation is a priori bounded by a constant $k$. 
4. Pushdown Automata Determinisation

4.4 Summary

In this section, additional informations relevant to the chapter are presented. It includes timeline for the State-of-the-Art and new results including the information about the implementation of the algorithms used for evaluation. It also contains small comparison of the three types of determinisable pushdown automata (PDAs) and an observation about visibly pushdown automata (VPDAs) and real-time height-deterministic PDAs (rHPDAs) minimization based on analysis of pushdown store top symbols set.

4.4.1 Timeline and attributions

Here follows a timeline of related to VPDAs and rHPDAs State-of-the-Art:

2004

- Alur and Madhusudan defined visibly pushdown languages (VPLs) \[6\] and presented the proof of the determinisability of a given nondeterministic [VPDA] The proof was correct, but not practical for direct implementation of the determinisation algorithm due to its exhaustive nature. The proof enumerates all possible states (cartesian product), all possible pushdown store symbols and all possible transitions.

2006

- Ha Nguyen created a library for nondeterministic [VPDAs] determinisation called VPAlib \[36\]. The determinisation implemented in this library is based on exhaustive enumeration of all possible states, all possible pushdown store symbols and all possible transitions. This algorithm is directly based on the proof by Alur and Madhusudan \[6\].

2007

- Nowotka and Srba defined [rHPDAs] \[37\] and presented the proof of the determinisability of a given nondeterministic [rHPDA]. The proof is very similar to the proof for determinisability of [VPDAs] and is similarly impractical for direct implementation of the determinisation algorithm.

2009

- Nguyen Van tang improved upper bound of the number of states \[45\]. Two optimisations for Alur-Madhusudan’s determinisation of [VPDAs] were introduced. Minimizing the set of summaries S component of state pair for some special cases concerning initial states and removing R component of state pair.
Here follows a timeline related to the new results present in this Dissertation thesis:

2012
- Radomír Polách and Jan Trávníček implemented a proof of concept of section 4.2 in Java programming language.
- Jan Veselý implemented determinisation of the finite automata and pushdown automata in his Master’s Thesis [48] based on profs [6, 37, 45] into the Automata Library Toolkit [47] with similar results as [36]. Veselý’s Masters Thesis was supervised by Jan Trávníček and opposed by Radomír Polách.

2014
- Radomír Polách implemented proof of concept of section 4.3 in Java programming language.

2014
- Radomír Polách et al. published A new algorithm for the determinisation of visibly pushdown automata [A.2]. For the evaluation was used the implementation in Java programming language above.

2015
- The determinisations of pushdown automata in the Automata Library Toolkit [47] adopted section 4.2 and section 4.3.

2016
- Radomír Polách et al. published an Efficient determinization of visibly and height-deterministic pushdown automata [A.3]. For the final evaluation was used the implemented in [47].

### 4.4.2 Comparison of the determinisable pushdown automata

The relations between all of the mentioned determinisable classes of PDAs are as follows:

1. **Input-driven pushdown automata (IDPDAs)** have their pushdown store operation solely determined by the input symbol.

2. **Visibly pushdown automata (VPDAs)** have their input alphabet partitioned into three parts, each part of the alphabet representing one of the pushdown store operations push, pop, and local respectively.

3. **Real-time height-deterministic PDAs (rHPDAs)** have their pushdown store operation determined by input symbol for groups of corresponding states (d–subsets). For
every input word, the depth of pushdown store is the same for all nondeterministic
paths. A VPDA is a RHPDA where all states are part of a single group of corresponding
states.

4.4.3 Minimisation of the deterministic pushdown automata

Based on the information gathered in pushdown store top symbols during the de-
terminisation of the VPDA S and RHPDA S, the notion of simple minimisation of the
pushdown automata (PDAs) can be derived.

All states with disjunct pushdown store top symbols found in the deterministic PDA S can
be merged together to a single state. Such change will not change the language accepted
by the deterministic PDA S.
Chapter 5

Regular Tree Expression Conversion

This section is split into two parts, one for each of the conversion methods. The first section summarises the progress in regular tree expressions (RTEs) conversion similar to Thompson’s construction algorithm for regular expressions (REs). The second section summarises the progress in RTEs conversion similar to Glushkov’s construction algorithm for REs.

All necessary background and basic notions were explained in previous chapters.

This chapter contains main results presented as a conference paper [A.1] and main results from joint work with Tomáš Pecka, and Jan Trávníček presented as a conference paper [A.4]. The original author of most parts of the algorithm is Tomáš Pecka.

5.1 Introduction

The theories of formal string languages and formal tree languages are important parts of Computer Science. Strings and trees are fundamental data structures. Tree languages processing has become very popular in the recent years. For example, the practical usages can be found in the area of processing markup languages (like XML) or abstract syntax trees. Traditionally, problems on trees are solved using various kinds of tree automata (TAs) [16]. However, trees can also be represented by strings, for instance in their prefix or postfix notation obtained by preorder or postorder traversal of the tree, respectively. It was proven by Janoušek and Melichar [25] that the class of regular tree languages is a proper subclass of tree languages whose linear notations can be accepted by deterministic pushdown automata (PDAs). Thus, the standard (string) PDA is another suitable model of computation for processing regular tree languages in a linear notation. For example, algorithms processing XML with the use of PDAs have been investigated [28, 43].

RTEs are a natural formalism for the description of regular tree languages [16]. They are analogous to regular (string) expressions. It is well known that regular (string) expressions
describe regular languages and can be converted to finite automata (FAs). In the case of trees, RTEs can be converted to corresponding finite tree automata (FTAs).

FAs and regular (string) expressions are well-studied [24, 41]. A string language membership problem is a decision problem. Given a regular (string) expression $E$ and a string $w$, decide whether $w$ is in the language described by the regular (string) expression $E$. This problem can be decided by converting the expression to an equivalent FA and running the automaton on the input word $w$.

Many algorithms deal with a problem of converting regular (string) expressions to FAs in the string domain. Three algorithms by Brzozowski [11], Thompson [46] and Glushkov [23] (also known as a position automaton) are the basic ones. Antimirov’s partial derivatives method [7] (which can be seen as a non-deterministic extension of Brzozowski’s algorithm) must be also mentioned. Conversions by Glushkov’s and Antimirov’s can be done in polynomial time w.r.t. the number of occurrences of symbols in the regular expression.

RTE is a formalism for describing regular tree languages. The language membership problem for trees and RTEs is analogous: Given a regular tree expression $E$ and a tree $t$, decide whether $t$ is in the language described by the regular tree expression $E$. As in the string case, one can create an FTA (or a PDA) equivalent to the RTE $E$ and let the automaton run on (linearised) tree $t$.

Algorithms for the conversion of RTEs to FTAs are inspired by the mentioned algorithms from the string domain. Antimirov’s and Glushkov’s algorithms were adapted to RTEs by Kuske and Meinecke [29] and also later by Laugerotte et al. [30]. The FTA is constructed in polynomial time w.r.t. the size of the RTE in both adaptations.
5.2 Thompson-like method

5.2.1 Introduction

In this section, a simple and straightforward way of converting a
regular tree expression (RTE) to an equivalent deterministic pushdown automaton (PDA)
is presented. The PDA reads input ranked and unranked ordered trees in postfix and
postfix bar notation, respectively. No method of directly converting RTE to an equivalent
deterministic PDA is known.

The presented PDAs constructed for RTEs are height-deterministic \[37\], i.e. given a
nondeterministic PDA, the height of contents of the pushdown store for each run on the
same input is the same in all nondeterministic paths. Consequently, any such nondetermin-
istic PDA can be converted into an equivalent deterministic PDA.

The rest of this section is organised as follows. The second subsection describes some
basic notions and notations. The third subsection deals with constructing the nondetermin-
istic real-time height-deterministic PDA (rHPDA) for a given RTE. The fourth subsection
discusses the determinisation of the constructed nondeterministic rHPDA. The last sub-
section is a conclusion.

This section contains main results presented as conference paper \[A.1\].

5.2.2 Pushdown automaton for an RTE

This section deals with a method for converting a regular tree expression (RTE) into an
equivalent nondeterministic pushdown automaton (PDA). There are two variants of our
method, one for the postfix notation in the case of ranked ordered trees and the other
one for the postfix bar notation in the case of unranked ordered trees. In this dissertation
thesis, the case of the postfix bar notation is described in details because this case is more
general. Our method could also be modified for trees in prefix notation or prefix bar
notation. The constructed PDAs accept the input by the empty pushdown store.

Example 5.1. Given a RTE \(r\), post\(\bar{\text{bar}}(r) = ( | | □_1 | b | □_1 a ) □_1 ( | | a b + | c ),\)
post\(r = ( □_10 b0 □_10 a3 ) □_10 ( a0 b1 + c0 ),\) the resulting set of trees represented by
\(r\) is illustrated in Figure 5.1.

In the first phase, these basic symbols of RTEs are converted to corresponding fragments
of a nondeterministic PDA. Notation in figures uses \(a|\alpha \rightarrow \beta\) denotes a transition which
read input symbol \(a\), pops \(\alpha\) and pushes \(\beta\) on the top of the pushdown store. Figure 5.2
illustrates the principles of these conversions. It is noted that there are different rules for
postfix notation and postfix bar notation. Note that \(| □_i| \) is taken as one input symbol
because the arity of □_i is always 0 and it simplifies the method.
5. Regular Tree Expression Conversion

In the second phase, the rules for converting RTE operations union, concatenation and iteration are applied. These conversions are the same for both postfix notation and postfix bar notation.

![Diagram](image)

Figure 5.3: Converting $A + B$ to fragment of equivalent PDA

Figure 5.3 illustrates the union of two RTEs $A$ and $B$, i.e. $A + B$.

Figure 5.4 illustrates the concatenation of $B$ into expression $A$ containing two substitution symbols $\square_i \in K$ on this substitution symbol, i.e. $A \cdot \square_i B$. 
Algorithm 5.1 describes the conversion of RTE into its equivalent real-time height-deterministic PDA (rHPDA) formally. The algorithm is based on the Thompson’s algorithm [46] for construction string nondeterministic finite automaton (FA) from regular expression (RE).

**Theorem 5.1** (Every RTE is rHPDA recognisable.) Given an RTE $r$, Algorithm 5.1 correctly computes PDA $M_r$, which accepts the postfix bar notation of the tree language described by $r$.

Let $r$ be an RTE. In needs to proven that $L(r)$ is $r$HPDA recognisable. In the proof, the inductive definition given for the language $L(r)$ from the previous section is followed and prove that $L(r)$ is $r$HPDA recognisable. The proof uses $r$HPDA with transitions.

**Proof.** Every RTE is rHPDA recognisable.
Algorithm 5.1: Construction of PDA from RTE

Input: Sequence \( P = p_1 p_2 \ldots p_n \) containing \( \text{RTE} \) in postfix bar notation with \( \text{RTE} \) operations converted into postfix notation (i.e. \( \text{RTE} \) operation followed by its operands), input alphabet \( \mathcal{A} \).

Output: Equivalent \( \text{rHPDA} \) \( M_r = (Q, \mathcal{A}, G, \delta, \{q_0\}, \bot, F) \).

1. \( X \leftarrow \text{New-Pushdown-Store();} \)
2. for \( i \leftarrow 1 \) to \( n \) do
   3. if \( p_i = \varepsilon \) then \( \text{Push}(X, \langle \{i\}, \mathcal{A}, \bot \rangle, \varepsilon \mapsto f, i, \bot, \{f\}) \);
   4. if \( p_i \in \mathcal{A} \) then \( \text{Push}(X, \langle \{i\}, \mathcal{A}, \bot, S \rangle, \varepsilon \mapsto f, i, \bot, \{f\}) \);
   5. if \( p_i = \mid \) then \( \text{Push}(X, \langle \{i\}, \mathcal{A}, \bot, S \rangle, \bot \mapsto f, i, \bot, \{f\}) \);
   6. if \( p_i \in K \) then \( \text{Push}(X, \langle \{i\}, \mathcal{A}, \bot \rangle, p_i \mapsto f, i, \bot, \{f\}) \);
   7. if \( p_i = \cdot \) then
      8. \( A \leftarrow \text{Pop}(X); \)
      9. \( B \leftarrow \text{Pop}(X); \)
      10. \( \text{Push}(X, \langle A.Q \cup B.Q, A.G \cup B.G, A.\delta(f_0 \leftarrow B.q_0) \cup B.\delta, A.q_0, \bot, \{B.f_0\}\rangle); \)
   8. if \( p_i = + \) then
      9. \( A \leftarrow \text{Pop}(X); \)
     10. \( B \leftarrow \text{Pop}(X); \)
     11. \( \text{Push}(X, \langle A.Q \cup B.Q \cup \{i, A.f_0, B.f_0\}, A.G \cup B.G, A.\delta \cup B.\delta \cup \{i\} \cup \{⊥\} \cup \{B.f_0\}\rangle); \)
     12. \( \text{Push}(X, \langle A.q_0, i \mapsto f, B.q_0, A.f_0 \mapsto f, B.f_0, \bot \mapsto f, q, \bot, \{f\}\rangle); \)
   9. if \( p_i \in \{\square_k\}, \square_k \in K \) then
      10. \( A \leftarrow \text{Pop}(X); \)
      11. \( B \leftarrow \text{Pop}(X); \)
      12. \( \delta \leftarrow A.\delta \cup B.\delta; Y \leftarrow S; \)
      13. foreach \( r \square_k \rightarrow s \) in \( \delta \) do
         14. if \( k = l \) then
            15. \( Y \leftarrow Y'; \delta \leftarrow \delta \cup r \mapsto Y; \delta \leftarrow \delta \cup B.f_0 \mapsto s; \delta \leftarrow \delta \setminus \{r \square_k \rightarrow s\}; \)
            16. \( \text{Push}(X, \langle A.Q \cup B.Q, A.G \cup B.G, \delta, A.q_0, \bot, \{A.f_0\}\rangle); \)
      17. if \( p_i \in \{\ast, \square_k\}, \square_k \in K \) then
         18. \( A \leftarrow \text{Pop}(X); \)
         19. \( \delta \leftarrow A.\delta; Y \leftarrow S; \) foreach \( r \square_k \rightarrow s \) in \( \delta \) do
            20. if \( k = l \) then
               21. \( Y \leftarrow Y'; \delta \leftarrow \delta \cup r \mapsto Y; \delta \leftarrow \delta \cup A.f_0 \mapsto s; \delta \leftarrow \delta \setminus \{r \square_k \rightarrow s\}; \)
               22. \( \text{Push}(X, \langle A.Q, A.G, \delta, A.q_0, \bot, \{A.f_0\}\rangle); \)
      23. \( M_r \leftarrow \text{Pop}(X); \)
      24. \( M_r.Q = M_r.Q \cup M_r.F; M_r.F \leftarrow \emptyset; \)
5.2. Thompson-like method

**Base case 1** (Basic RTEs). Assume that \( r \) is defined in the base case. Then \( L(r) \) is either \( \{a\} \) for some \( a \in A \) or \( \emptyset \) or \( \{\varepsilon\} \). Clearly, in each case, \( L(r) \) is \( r\text{HPDA} \) recognisable.

**Inductive step 1** (Composite RTEs). There are three cases to be considered.

Case 1. Assume that \( r \) is of the form \( r_1 + r_2 \). The inductive hypothesis is applied to \( r_1 \) and \( r_2 \). By the hypothesis, the languages \( L(r_1) \) and \( L(r_2) \) are by definition \( L(r_1) \cup L(r_2) \). Both automata are \( r\text{HPDA} \) that may have transitions. It is assumed that the state sets \( S_1 \) and \( S_2 \) have no states in common. Using these two automata a finite automaton recognizing \( L(r_1) \cup L(r_2) \) can be built.

Let \( A_1 = (Q_1, \mathcal{A}_1, G_1, \delta_1, \bot, q_1, f_1) \) and \( A_2 = (Q_2, \mathcal{A}_2, G_2, \delta_2, \bot, q_2, f_2) \) be automata recognizing \( L(r_1) \) and \( L(r_2) \), respectively. Both automata are \( r\text{HPDA} \) that may have transitions. It may be assumed that the state sets \( S_1 \) and \( S_2 \) have no states in common. Using these two automata the goal is to build a \( FA \) recognizing \( L(r_1) \cup L(r_2) \).

It can now pretend to be a machine accepting \( L(r_1) \cup L(r_2) \). Suppose \( t \) is an input tree.

Nondeterministically it is both machine \( A_1 \) and \( A_2 \) making transitions virtual initial state to initial states of \( A_1 \) and \( A_2 \). It continues running on both machines let them both lead in virtual final state from their final states. Thus, the automaton \( A \) recognizing \( L(r_1) \cdot \Box_i \) is constructed as described before in the subsection 5.2.2.

It is not difficult to check that \( A \) recognizes \( L(r_1) \cup L(r_2) \).

Case 2. Assume that \( r \) is of the form \( r_1 \cdot \Box_i \), where \( \Box_i \) is a symbol of \( K \). By the inductive hypothesis, the languages \( L(r_1) \) and \( L(r_2) \) are \( r\text{HPDA} \) recognisable. The language \( L(r_1) \cdot \Box_i \) \( L(r_2) \). The goal is to show that \( L(r_1) \cdot \Box_i \) \( L(r_2) \) is \( r\text{HPDA} \) recognisable.

Let \( A_1 = (Q_1, \mathcal{A}_1, G_1, \delta_1, \bot, q_1, f_1) \) and \( A_2 = (Q_2, \mathcal{A}_2, G_2, \delta_2, \bot, q_2, f_2) \) be automata recognizing \( L(r_1) \) and \( L(r_2) \), respectively. Both automata are \( r\text{HPDA} \) that may have transitions. It may be assumed that the state sets \( S_1 \) and \( S_2 \) have no states in common. Using these two automata the goal is to build a \( FA \) recognizing \( L(r_1) \cdot \Box_i \) \( L(r_2) \).

As above, it now pretends to be a machine accepting tress in \( L(r_1) \cdot \Box_i \) \( L(r_2) \). Suppose \( t \) is an input tree. It first pretends that it is the machine \( A_1 \) and run \( A_1 \) on \( t \). It simulates \( A_1 \) and read \( t \). Every time it reaches a state of \( A_1 \) which desired substitution move leads from, it has to make a nondeterministic choice. it either continues running \( A_1 \) or makes a transition to the initial state of \( A_2 \) and another \( \varepsilon \)-transition from the final state of \( A_2 \) to the other site of the substitution move. The \( \varepsilon \)-transition is marked by creating a new pushdown symbol, replacing the default pushdown symbol by this new symbol on the first move and this new symbol by default pushdown symbol on the second move. Thus, the automaton \( A \) recognizing \( L(r_1) \cdot \Box_i \) is constructed as described before in the subsection 5.2.2.

It is not difficult to check that \( A \) recognizes \( L(r_1) \cdot \Box_i \) \( L(r_2) \).

Case 3. Assume that \( r \) is of the form \( r_1 \cdot \Box_i \), where \( \Box_i \) is a symbol of \( K \). By the inductive hypothesis, the language \( L(r_1) \) is \( r\text{HPDA} \) recognisable. The language \( L(r_1 \cdot \Box_i) \) is
by definition \((r_1)^* \square_i\). The goal is to show that \((r_1)^* \square_i\) is \(r\text{HPDA}\) recognisable.

Let \(A_1 = (Q_1, A_1, G_1, \delta_1, \bot, q_1, f_1)\) be automata recognizing \(L(r_1)\). The goal is to construct a \(r\text{HPDA}\) that recognizes the language \(L(r_1)^* \square_i\), where \(\square_i\) is a symbol of \(K\).

As above, it pretends to be a machine that accepts trees in \(L(r_1)^* \square_i\). For an input tree \(t\), if it is the empty tree then it accepts it because the star of every language contains \(\varepsilon\). Otherwise it simulates \(A_1\) and read \(t\). Every time it reaches a state of \(A_1\) from which desired substitution move leads from, it has to make a nondeterministic choice. It either continues running \(A_1\) or make a transition to the initial state of \(A_1\) and another \(\varepsilon\)-transition from the final state of \(A_1\) to the other side of the substitution move. The \(\varepsilon\)-transition is marked by creating a new pushdown symbol, replacing default pushdown symbol by this new symbol on the first move and this new symbol by default pushdown symbol on the second move. Thus, the automaton \(A\) recognizing \(L(r_1)^* \square_i\) is constructed as described before in the subsection 5.2.2.

It is not difficult check that \(A\) recognizes \((r_1)^* \square_i\).

Example 5.2. Given RTE from Example 5.1 \(r\), Algorithm 5.1 constructs nondeterministic PDA \(M_r\) illustrated in Figure 5.6.

Nondeterministic PDA constructed by Algorithm 5.1 is height-deterministic and therefore can be determinised [37]. The determinisation is similar to the method of determinisation of visibly PDAs.

Example 5.3. Given nondeterministic PDA \(M_r\) from Example 5.2, its deterministic version is illustrated in Figure 5.7.
5.2. Thompson-like method

Given input tree in postfix bar notation $| \ c \ | \ b \ | \ c \ a$, the corresponding sequence of transitions by the deterministic PDA is given by the sequence of states $0, 0^00^00^014, 11''$, $22^244'', 3, 000^00^014, 11'', 22^244''$ and $56$, in which the acceptance by the empty pushdown store occurs.

5.2.3 Conclusion

A simple and straightforward method of constructing a deterministic pushdown automaton (PDA) for a given regular tree expression (RTE) was presented. It considers both ranked ordered trees in postfix notation and unranked ordered trees in postfix bar notation, which are read by the resulting deterministic PDA. These presented methods can also be modified for trees in prefix notations. The principle of this modification lies in the change of corresponding pushdown operations, by analogy to other Arbology algorithms for processing trees in various linear notations by the means of string PDA [8].
5.3 Glushkov-like method

5.3.1 Introduction

In this section, a new method of creating a real-time height-deterministic PDA (rHPDA) from regular tree expression (RTE) is proposed. The pushdown automaton (PDA) constructed accepts postfix ranked linear notation of trees.

The presented PDAs constructed for RTEs are height-deterministic [37], i.e. given a nondeterministic PDA, the height of contents of the pushdown store for each run on the same input is the same in all nondeterministic paths. Consequently, any such nondeterministic PDA can be converted into an equivalent deterministic PDA.

The rest of this section is organised as follows. The second subsection describes analyzing RTEs. The third subsection deals with constructing the nondeterministic rHPDA for a given RTE. The fourth subsection discusses the reduction of the number of transitions of the of the constructed nondeterministic rHPDA. The last subsection is a conclusion.

This section contains main results from joint work with Tomáš Pecka and Jan Trávníček presented as conference paper [A.4]. The original author of most parts of the algorithm is Tomáš Pecka and the unrefined version of the algorithm was already published in his Master’s Thesis [38] which was supervised by Jan Trávníček and opposed by Radomír Polách. Pecka’s Master Thesis was then extended into the article with refined algorithm and expanded proofs. Author of this Dissertation thesis most notably influenced definitions of the rHPDAs, their normalizations and proofs published in the article.

5.3.2 Analysing regular tree expression

To analyse the structure of the expression, the regular tree expression (RTE) has to be preprocessed similarly to Glushkov’s algorithm. Firstly, every occurrence of symbol from \( \mathcal{F} \) alphabet of the RTE is subscripted with an unique symbol. Subscripted RTE \( E \) is denoted as \( E' \).

Functions First, Follow and Pos are defined to analyse the RTE \( E' \). Function Pos returns a set of occurrences of symbols from \( \mathcal{F} \) alphabet of \( E' \). Function First computes a set of symbols that can be a root of a tree described by \( E' \). Function Follow returns tuples of children of a given symbol. Unlike strings, a symbol can be followed by more than a single symbol. The size of the children tuple is defined by the arity of the symbol.

Example 5.4. Let \( t \) from Figure 5.8 be a directed, rooted, labelled, ranked and ordered tree with labels from ranked alphabet \( \mathcal{A} = \{a2, b0\} \). The root of \( t \) is a node \( a2 \) with an ordered 2-tuple of children \( (a2, b0) \). Postfix notation of \( t \) is \( \text{post}(t) = b0 b0 a2 b0 b0 a2 a2 b0 a2 \).
5.3. Glushkov-like method

\[
\begin{align*}
& a^2 \\
& \quad \downarrow \\
& a^2 \quad b^0 \\
& \quad \downarrow \quad \downarrow \\
& b^0 \quad b^0 \quad b^0 \quad b^0
\end{align*}
\]

Figure 5.8: A directed, rooted, labelled, ranked and ordered tree over \( \mathcal{A} = \{a^2, b^0\} \)

\[
\begin{align*}
& \cdot \square \\
& \quad \downarrow \\
& \cdot \square \quad b^0 \\
& \quad \downarrow \\
& \ast \square \quad a^2 \\
& \quad \downarrow \\
& + \quad b^0 \quad \square \\
& \quad \downarrow \\
& a^2 \quad b^0 \\
& \quad \downarrow \quad \downarrow \\
& \square \quad \square
\end{align*}
\]

(a) A sample RTE

\[
\begin{align*}
& \cdot \square_1 \\
& \quad \downarrow \\
& \cdot \square_2 \quad \text{int} \\
& \quad \downarrow \\
& \ast \square_2 \quad \text{nil} \\
& \quad \downarrow \\
& \text{cons} \\
& \quad \downarrow \\
& \square_1 \quad \square_2
\end{align*}
\]

(b) An RTE denoting the language of integer lists in LISP

Figure 5.9: Examples of RTEs

**Definition 5.2.** Based on the definition of RTEs, the function First is defined recursively:

\[
\begin{align*}
\text{First}(\emptyset) &= \emptyset \\
\text{First}(a(E_1, E_2, \ldots, E_n)) &= \{a\} \\
\text{First}(E_1 + E_2) &= \text{First}(E_1) \cup \text{First}(E_2) \\
\text{First}(E_1 \cdot \square E_2) &= \begin{cases} \\
\text{First}(E_1) & \text{if } \square \notin \text{First}(E_1) \\
(\text{First}(E_1) \setminus \{\square\}) \cup \text{First}(E_2) & \text{if } \square \in \text{First}(E_1)
\end{cases} \\
\text{First}(E^* \square) &= \\{\square\} \cup \text{First}(E)
\end{align*}
\]

**Theorem 5.3.** The function First\( (E') \) returns the set of symbols that can be the root of any tree described by an RTE \( E \).

**Proof.** The proof is done by induction on the structure of the RTE. The basis: If \( E = a(E_1, E_2, \ldots, E_n) \), \( a \in \mathcal{F} \cup \mathcal{K} \), \( n \geq 0 \): Only \( a \) can be a root. If \( E = \emptyset \): There is no root.
Now, assume that the theorem holds for any $E_1$ and $E_2$. If $E = E_1 + E_2$: This operator unifies two sets of trees. Therefore the roots of trees from $L(E)$ are either from First($E_1$) or First($E_2$). If $E = E_1^* □$: The roots can be only elements from First($E_1$) or the substitution symbol □. If $E = E_1 \cdot □ E_2$: Initially, suppose □ $\notin$ First($E_1$). Then the root must be from First($E_1$). If □ $\in$ First($E_1$), then □ gets substituted by the roots of the trees from $E_2$.

The function Follow returns a set of tuples of symbols which can be direct descendants (children) of a symbol $a \in \mathcal{F}$. The computation of the function is defined using Algorithm 5.2. The algorithm recursively traverses the syntax tree of the RTE and maintains a substitution map. The map contains roots of all possible trees that can be substituted for each □ $\in \mathcal{K}$. If □ occurs as a child of the symbol $a$ when computing Follow($E', a$), it gets substituted by elements of a substitution map for a given □. It is possible that □$_2 \in \mathcal{K}$ is present in the subMap[□$_1$] of any node. Then it is required to include the contents of mapping for key □$_2$ into □$_1$ set of that node. Also, if □ $\in$ subMap[□] then □ element can be discarded from the set as it brings no new information.

For the purpose of proving the correctness of the computation, the algorithm can be split in a two pass algorithm. In the first pass, the substitution mapping for each node is computed. In the second pass, the computation of Follow can use the computed mapping.

**Theorem 5.4.** Algorithm 5.2 computes a substitution mapping of every node of the RTE.

*Proof.* If the substitution operation takes place (in concatenation and iteration nodes), it alters the substitution map. The changes in substitution mapping come from the definitions of RTEs. Case $E_1 \cdot □ E_3$: Roots from trees described by $E_2$ may appear in the place of □ symbols in $E_1$. Therefore the mapping for the □ symbol in $E_1$ is replaced. The substitutions for □ symbols in $E_2$ are determined by the same mapping as in the parent node. Case $E_1^* □$. Symbol □ is to be replaced by roots of $E_1$ (this implements the actual iteration) and the iteration is terminated by concatenating a tree from the right operand of the closest substitution or iteration node. In other cases, the existing mapping is simply passed to children as no substitution happens.

**Theorem 5.5.** Function Follow($E', a$) (defined by Algorithm 5.2) correctly returns a set of tuples representing all possible tuples of direct children of a node $a$.

*Proof.* The proof by induction is straightforward with the use of the previous theorem.
5.3. Glushkov-like method

Algorithm 5.2: Computation of Follow($E', a$) in a single pass

1. Function Follow($E, a$)
   2. return FollowRec($E, a, NewMap()$);
3. Function FollowRec($E, a, subMap$)
   4. switch $E$ do
      5. case $E_1 + E_2$ do
         6. return FollowRec($E_1, a, subMap$) $\cup$ FollowRec($E_2, a, subMap$);
      7. case $E_1 \cdot □E_2$ do
         8. subMapL $\leftarrow$ subMap /* copy map */;
         9. subMapL[□] $\leftarrow$ First($E_2$) /* replace mapping for □ */;
        10. return FollowRec($E_1, a, subMapL$) $\cup$ FollowRec($E_2, a, subMap$);
      11. case $E_1^* □$ do
         12. subMap[□] $\leftarrow$ subMap[□] $\cup$ First($E_1$);
        13. return FollowRec($E_1, a, subMap$);
      14. case $f(E_1, E_2, \ldots, E_n)$ do
         15. if $a = f$ then return ReplaceConstants($subMap, E_1, E_2, \ldots, E_n$);
         16. else return $\bigcup_{i=1}^{n}$ FollowRec($E_i, a, subMap$);
      17. case $\emptyset$ do
         18. return $\emptyset$;
19. Function ReplaceConstants($subMap, E_1, E_2, \ldots, E_n$)
   20. lst $\leftarrow$ NewList() ;
   21. for $E_i$ in $E_1, E_2, \ldots, E_n$ do
   22. if $E_i \in \mathcal{K}$ then lst $\leftarrow$ Append(lst, subMap[c]) /* child is a □ */;
   23. else lst $\leftarrow$ Append(lst, First($E_i$)) ;
   24. return CartesianProduct(lst);

5.3.3 Pushdown automaton construction

In the previous section, it was shown how to compute First and Follow sets. The First set determines what symbols are the last to be read in the postfix notation. The Follow sets store the information about the direct children of a node. This information is used to create transitions of the two-state pushdown automaton (PDA) that accepts by final state. The automaton reads a linear postfix notation of a tree with an end-of-string marker $\triangledown$ appended to the end of the input. Technical helper functions $\varphi$ and $\sigma$ are presented first.

Definition 5.6. Function $\varphi$ maps an element (tuple) of Follow($E', a$) to a string of pushdown store symbols. The resulting string is $\varepsilon$ if the size of the tuple is zero. Mapping $\sigma$ strips the unique index from the subscripted symbol.
Proof.
The RTE from Figure 5.9a describes, for instance, the tree from Example 5.6. This tree in Figure 5.8 is constructed by Algorithm 5.3.

Example 5.6. Let $E'$ be a subscripted regular tree expression (RTE) and let $f = (a_2, b_2, c_0)$ be a follow tuple of some node. Then $\varphi(f) = a_2 b_2 c_0$. Also $\sigma(a_2) = a_2$.

Roots of subtrees are stored on the pushdown store to keep track of which subtrees have been read so far. When the root of a subtree is read, its children have to be on the top of the pushdown store. They are replaced by a pushdown store symbol corresponding to the read symbol. [PDA] recognising postfix notations of trees described by $E$ (post($L(E)$)) is constructed by Algorithm 5.3.

Example 5.7. This example expands on Example 5.5. The PDA $A = (\{q, f\}, \{a_2, b_0, \bot\}, \{\bot, a_2, b_0, a_2, b_0, b_0, b_0\}, \delta, \bot, \{f\})$ is constructed by Algorithm 5.3. $\delta$ is defined as follows:

$\delta(q, a_2, a_2 a_2) = \{(q, a_2)\}$
$\delta(q, a_2, b_0 a_2) = \{(q, a_2)\}$
$\delta(q, a_2, b_0 b_0) = \{(q, a_2)\}$
$\delta(q, a_2, a_2 a_2 a_2) = \{(q, a_2)\}$
$\delta(q, a_2, a_2, b_0) = \{(q, a_2)\}$
$\delta(q, a_2, a_2, b_0, b_0) = \{(q, a_2)\}$
$\delta(q, a_2, a_2, b_0, b_0, b_0) = \{(q, a_2)\}$
$\delta(q, b_0, \epsilon) = \{(q, b_0), (q, b_0), (q, b_0)\}$
$\delta(q, \bot, \bot, b_0) = \{(f, \epsilon)\}$
$\delta(q, \bot, \bot, a_2) = \{(f, \epsilon)\}$

The RTE from Figure 5.9a describes, for instance, the tree from Figure 5.8. This tree in its postfix notation (with $\bot$ symbol appended) is accepted by the automaton.

Theorem 5.7. Algorithm 5.3 creates PDA $A$ such that $L(A) = \text{post}(L(E))$.

Proof. The proof consists of two parts: $\text{post}(L(E)) \subseteq L(A)$ and $L(A) \subseteq \text{post}(L(E))$.
Algorithm 5.3: PDA accepting linearised trees described by an RTE $E$

**Input:** RTE $E$.

**Output:** PDA $A$ such that $L(A) = \text{post}(L(E))$.

1. Create PDA with the following properties:
   - Set of states is equal to $\{q, f\}$,
   - Alphabet is equal to the $F$ alphabet of $E$, and $\perp$ symbol,
   - Pushdown store alphabet is equal to the symbols $\text{Pos}(E') \cup \{\perp\}$,
   - Mapping $\delta$ can be created by these rules:
     - $\forall a_i \in \text{Pos}(E'), \, \sigma(a_i) \in \text{F}_0$, add transition $\delta(q, \sigma(a_i), \varepsilon) = \{(q, a_i)\}$,
     - $\forall a_i \in \text{Pos}(E'), \, \sigma(a_i) \notin \text{F}_0$, $\forall f \in \text{Follow}(E', a_i)$, add $\delta(q, \sigma(a_i), \varphi(f)) = \{(q, a_i)\}$,
     - $\forall a_i \in \text{First}(E')$ add transition $\delta(q, \perp, \perp a_i) = \{(f, \varepsilon)\}$.

Resulting PDA is $A = (\{q, f\}, \text{F} \cup \{\perp\}, \text{Pos}(E') \cup \{\perp\}, \delta, \perp, q, \{f\})$. Automaton accepts by the final state. The top of the pushdown store is on the right.

Case $\text{post}(L(E)) \subseteq L(A)$: Proof comes directly from the proof of the Follow and First functions. The functions analyse all possible combinations of parent-children relations. The relations are used in the transition function of the PDA. When an input tree (except $\perp$ symbol) is read, the automaton can continue only if the pushdown store content equals to the string $\perp f$ ($f \in \text{First}(E')$ set) to ensure that whole tree was read.

Case $L(A) \subseteq \text{post}(L(E))$: If there is a word from $L(A)$ that is not in $\text{post}(L(E))$ then either the computation of First or Follow functions were wrong or the transitions created from Follow sets would allow the automaton to accept something more. The functions First and Follow are proven to be correct.

Theorem 5.8. Algorithm 5.3 creates a real-time height-deterministic PDA (rHPDA).

Proof. Transitions PDA always pops $\text{arity}(a)$ symbols from the pushdown store and push one symbol when reading symbol $a$. Reading symbol $\perp$ pops two symbols and pushes none. The pushdown store height is predetermined and same for all nondeterministic computations of the PDA on any string. This fulfils the conditions of height-determinism. The PDA never reads $\varepsilon$, therefore it is also real-time [37, A.3].

The PDA has two properties worth mentioning: The function ReplaceConstants from Algorithm 5.2 has an exponential output with the number of node’s children that are from $K$ and the size of $\text{subMap}^{[\Box]}$ for given node. As every element from the Follow set results in one transition, the PDA has an exponential amount of transitions. Also,
5. Regular Tree Expression Conversion

(a) A RTE

(b) The PDA for the RTE from Figure 5.11a

Figure 5.11: A RTE and its equivalent PDA

Lemma 5.8 shows that the PDA is determinisable because the PDA is real-time height-deterministic [37, A.3].

Example 5.8. RTE $E'$ (Figure 5.11a) has the following properties: First($E'$) = \{a2, b0, c0\}, Follow($E'$, a2) = \{(a2, a2), (a2, b0), (a2, c0), (b0, a2), (b0, b0), (b0, c0)\} and Follow($E'$, b0) = Follow($E'$, c0) = \emptyset. The equivalent PDA is illustrated in Figure 5.11b.

5.3.4 Reducing the number of transitions

The pushdown automaton (PDA) created by Algorithm 5.3 has an exponential number of transitions. The transition function $\delta$ enumerated all possible tuples of children for every node in the tree.

The idea behind the improvement is to make better use of the pushdown store. New pushdown store symbols representing all possible symbols that can appear in the place of a symbol $\square \in K$ are introduced. These symbols effectively represent the complete substitution mapping. For every occurrence of the symbol $\square \in K$ the substitution mapping set for this occurrence is to be added as a new pushdown store symbol.

The only difference in the analysis of the regular tree expression (RTE) is the Follow algorithm. On line 24 the computation is altered by removing the computation of Cartesian product and returning the list $lst$ instead. This excludes the need for computing the Cartesian product. Furthermore, every symbol of $\mathcal{F}$ alphabet is now followed by exactly one tuple.
5.3. Glushkov-like method

The ideas from previous paragraphs are applied in the Algorithm 5.4. The algorithm constructs an improved PDA which has an asymptotically lower amount of transitions.

**Definition 5.9.** Let $\text{subMap}_\triangle \square$ return the substitution mapping for symbol $\square \in \mathcal{K}$ inside the $\triangle$ node of the syntax tree.

**Algorithm 5.4:** Improved PDA accepting linearised trees described by a RTE $E$

<table>
<thead>
<tr>
<th>Input</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>PDA $A$ such that $L(A) = \text{post}(L(E))$</td>
</tr>
</tbody>
</table>

1. Create PDA with following properties:
   - Set of states is equal to $\{q, f\}$,
   - input alphabet is $\mathcal{F} \cup \{\perp\}$,
   - pushdown store alphabet consists of all sets that appear in substitution mapping in $\square \in \mathcal{K}$ nodes and singletons consisting of indexed occurrences of symbols from $\mathcal{F}$,
   - transitions ($\delta$) are created by these rules:
     1. for all symbols $a_i \in \mathcal{F}$ add transition $\delta(q, \sigma(a_i), \varphi(\text{Follow}(E, a_i))) = \{(q, \{a_i\})\}$,
     2. for all nodes $\square_i$ labelled with $\square \in \mathcal{K}$, for all symbols $a_i \in \text{subMap}_\square \square$ add $\delta(q, \sigma(a_i), \varphi(\text{Follow}(E, a_i))) = \{(q, \text{subMap}_\square \square)\}$,
     3. for all symbols $a_i \in \text{First}(E')$ add transition $\delta(q, \perp, \perp \{a_i\}) = \{(f, \varepsilon)\}$.

Resulting PDA is $A = (\{q, f\}, \mathcal{F} \cup \{\perp\}, \{\{a_i\} | a_i \in \mathcal{F} \cup \text{subMap}_\square \square \} \setminus \perp \{f\}$). Automaton accepts by the final state. The top of the pushdown store is on the right.

**Theorem 5.10.** Algorithm 5.4 creates a real-time height-deterministic PDA.

**Proof.** Similar to the proof of Lemma 5.8.

The automaton created by Algorithm 5.4 is determinisable.

**Theorem 5.11.** Algorithm 5.4 creates PDA equivalent to RTE $E$ in $O(|E|^2)$ time and the number of transitions of the PDA is $O(|\mathcal{F}_E| \|\mathcal{K}_E\|)$.

**Proof.** Overall time complexity can be determined from the efficient implementation of the algorithm. Computing and saving the First set takes $O(|E| \|\mathcal{F}_E\|)$ time. The substitution mapping can be computed in one traversal over the syntax tree of RTE. It is saved as a mapping from every occurrence of a node from $\mathcal{K}_E$ alphabet to the set of elements from $\mathcal{F}_E$. The Follow elements can be computed in the same traversal. This takes $O(|E|^2)$ time.
5. Regular Tree Expression Conversion

Rules of type 1 and 3 are created in $O(|\mathcal{F}_E|)$ time from the Follow mapping and First set, respectively. While creating type 2 rules, for every $\mathcal{F}_E$ node it is required to iterate over the saved substitution mapping. Therefore, creating type 2 rules takes $O(|\mathcal{F}_E||\mathcal{K}_E|)$ time.

The overall time complexity is $O(|E| |\mathcal{F}_E| + |E| |E| + |\mathcal{F}_E||\mathcal{K}_E|) = O(|E|^2)$ as $|\mathcal{K}_E| \leq |E|$ and $|\mathcal{F}_E| \leq |E|$. The number of transitions is $O(|\mathcal{F}_E||\mathcal{K}_E|)$ because there are $O(|\mathcal{F}_E|)$ transitions of types 1 and 3, and $O(|\mathcal{F}_E||\mathcal{K}_E|)$ transitions of type 2.

![Diagram of RTE with First set and substitution mapping](image)

Figure 5.12: Sample RTE with First set and substitution mapping for important nodes

**Example 5.9.** RTE $E'$ from Figure 5.12 converted to equivalent DDA (Figure 5.13). First$(E') = \{a_{41}, b_{02}, c_{03}, d_{04}, e_{05}, p_{06}, q_{07}, r_{08}\}$. Follow$(E', a_{41}) = (\{a_{41}, b_{02}, c_{03}, d_{04}, e_{05}, p_{06}, q_{07}, r_{08}\}, \{a_{41}, b_{02}, c_{03}, d_{04}, e_{05}\}, \{a_{41}, b_{02}, c_{03}, d_{04}, e_{05}, p_{06}, q_{07}, r_{08}\})$. Follow of other symbols (leaves) is $\emptyset$. Note that if the Follow was computed by Algorithm 5.2 then $|\text{Follow}(E', a_{41})| = 1600$. 

68
5.3. Glushkov-like method

\[
\begin{align*}
\text{start} & \rightarrow q \\
\left\{ 
\begin{array}{l}
\vdash a_4 \rightarrow \{a_4\} \\
\vdash b_0 \rightarrow \{b_0\} \\
\vdash c_0 \rightarrow \{c_0\} \\
\vdash d_0 \rightarrow \{d_0\} \\
\vdash e_0 \rightarrow \{e_0\} \\
\vdash p_0 \rightarrow \{p_0\} \\
\vdash q_0 \rightarrow \{q_0\} \\
\vdash r_0 \rightarrow \{r_0\}
\end{array}
\right. \\
\end{align*}
\]

\[
\begin{align*}
\vdash a_4 \rightarrow \{a_4\} & \rightarrow \varepsilon, \\
\vdash b_0 \rightarrow \{b_0\} & \rightarrow \varepsilon, \\
\vdash c_0 \rightarrow \{c_0\} & \rightarrow \varepsilon, \\
\vdash d_0 \rightarrow \{d_0\} & \rightarrow \varepsilon, \\
\vdash e_0 \rightarrow \{e_0\} & \rightarrow \varepsilon, \\
\vdash p_0 \rightarrow \{p_0\} & \rightarrow \varepsilon, \\
\vdash q_0 \rightarrow \{q_0\} & \rightarrow \varepsilon, \\
\vdash r_0 \rightarrow \{r_0\} & \rightarrow \varepsilon
\end{align*}
\]

Figure 5.13: PDA equivalent to a RTE from Figure 5.12 with transitions grouped by type. For readability, symbol \( \heartsuit \) stands for \( \{a_4, b_0, c_0, d_0, e_0, p_0, q_0, r_0\}\) and symbol \( \spadesuit \) stands for \( \{a_4, b_0, c_0, d_0, e_0, p_0, q_0, r_0\}\).

5.3.5 Conclusion and future work

A new algorithm for the conversion of a regular tree expression (RTE) to a pushdown automaton (PDA) has been described. The resulted PDA accepts all trees from the language described by the RTE in their linear postfix notation. Present PDA belongs to the class of real-time height-deterministic PDAs. Therefore it can always be determinised [A.3].

The presented algorithm creates the PDA in quadratic time w.r.t. to the size of input RTEs syntax tree, i.e. in \( O(|E|^2) \) time. The number of transitions in the PDA is \( O(|F| |K|) \).

There is also a number of interesting open problems. As the processing of the RTE is similar to processing the regular expression for the Glushkov’s algorithm, the hope is to explore more similarities with this algorithm. The reverse conversion (from an finite tree automaton (FTA) or a PDA to an RTE) can also be investigated. Finally, it would be interesting to explore the tree pattern matching problem where the definition of a set of tree patterns is represented by RTEs.
5.4 Summary

In this section, additional informations relevant to the chapter are presented. It includes timeline for the new results including the information about the implementation of the algorithms.

5.4.1 Timeline and attributions

Here follows a timeline related to the new results present in this Dissertation thesis:

- 2011
  - Radomír Polák et al. published Regular Tree Expressions and Deterministic Pushdown Automata which presents a method for transformation of regular tree expressions (RTEs) into pushdown automata (PDAs). The proof of concept was implemented in Java programming language [40].

- 2016
  - Tomáš Pecka presented his Master's Thesis Construction of a Pushdown Automaton Accepting a Language Given by Regular Tree Expression with a Glushkov like method for transformation of RTEs into PDAs. Pecka's Master's Thesis was supervised by Jan Trávníček and opposed by Radomír Polák [38].

- 2018
  - Tomáš Pecka and Jan Trávníček and Radomír Polák et al. published Construction of a Pushdown Automaton Accepting a Postfix Notation of a Tree Language Given by a Regular Tree Expression which presents a Glushkov like method for transformation of RTEs into PDAs [39]. It was implemented in Automata Library Toolkit [47].
Conclusions

This dissertation thesis contains results presented as conference papers [A.1, A.2, A.4] and results presented as a journal paper [A.3].

A new incremental algorithm of the determinisation of nondeterministic visibly pushdown automata (VPDAs) has been described. The algorithm creates only necessary states and pushdown symbols by analysing and tracking which states are achievable by computing transitions on the same levels of pushdown operations nesting. Possible tops of the pushdown store are stored for each state when a pop transition is in progress and then they are shared through local transitions with states on the same levels of the nesting. The behaviour of the algorithm is inspired by the behaviour of the incremental construction of the deterministic finite automata (FAs). The algorithm has been implemented as a part of an experimental automata library [47].

Height-deterministic pushdown automata (HPDAs) are a natural generalisation of the VPDAs which leads to a determinisation method being very similar. A similar incremental algorithm of the determinisation of real-time height-deterministic PDAs (rHPDAs) has been described. The algorithm creates only necessary states and pushdown symbols by analysing and tracking which states are achievable by computing transitions on the same depths of the pushdown store. Possible tops of the pushdown store are stored for each state when a pop transition is in progress and then they are shared through local transitions with states on the same depths of the pushdown store. The behaviour of the algorithm is inspired by the behaviour of the VPDAs determinisation or more generally on the incremental construction of the deterministic FAs. The algorithm has been implemented as a part of an experimental automata library [47].

Although the number of states of the deterministic pushdown automaton (PDA) for the worst case for both of the algorithms is still $2^n$ for a given nondeterministic VPDAs with $n$ states, it has been shown that the upper bound of the number of states of the deterministic PDA is dependant on the number of distinct pairs of locally connected states. For those
6. Conclusions

and other reasons in many practical cases, the algorithm provides significantly smaller
deterministic PDA than the previously existing determinisation algorithms.

Furthermore, it may be possible to extend this approach to other kinds of pushdown store automata. The $k$-HPDAs are proposed as PDAs where for any given input string the difference between pushdown store heights during any (nondeterministic) computation is a priori bounded by a constant $k$. It would also be interesting to provide a determinisation algorithm for rHPDA with different normalisation.

Two new algorithms for the conversion of a regular tree expression (RTE) to a PDA has been described. The resulted PDAs accepts all trees from the language described by the RTE regular tree expression in their linear postfix notation. Presented PDAs belongs to the class of rHPDAs. Therefore it can always be determinised by presented algorithms.

These presented methods can also be modified for trees in prefix notations. The principle of this modification lies in the change of corresponding pushdown operations, by analogy to other arbology algorithms for processing trees in various linear notations by means of string PDAs [8]. Principles have been explained regarding arbitrary subtree matching by HPDAs which lie down basics for Approximate pattern matching with arbitrary subtrees, combining papers Template Matching in Ranked Trees [21] and Approximate Subtree Matching by Pushdown Automata [33].

The presented conversions contribute to a better understanding of the theories of the tree and string formal languages and allow any solution of a problem described by finite tree automaton (FTA) to be converted into the equivalent solution described by a deterministic PDA. Further, it has been demonstrated that the deterministic PDA as a model of computation is powerful enough to accept also some tree languages beyond the class of regular tree languages.

6.1 Contributions of the Dissertation thesis

In particular, the main contributions of the dissertation thesis are as follows:

- Two new algorithms for converting regular tree expressions (RTEs) to equivalent real-time height-deterministic PDAs (rHPDAs) that accept the trees in their linear notation. First is based on the Thompson’s construction algorithm for regular expressions (REs), the second is based on the Glushkov’s construction algorithm for REs.

- A new algorithm for the visibly pushdown automata (VPDAs) determinisation.

- A new algorithm for the rHPDAs determinisation.
6.2 Future work

The author of the dissertation thesis suggests exploring the following topics:

- Consider $k$-height-deterministic pushdown automata (HPDAs), where $k$ is the maximal difference of pushdown store height in different non-deterministic paths.
- Consider HPDAs with generalised normalisation (i.e. arbitrary number of pushed and popped pushdown store symbols), where $k$ is the maximal difference of pushdown store height in different non-deterministic paths.
- Consider pushdown automata (PDAs) minimisation.
- Consider possible determinisation principles for restricted classes of linear bounded automata (LBAs).
- Consider partial derivation for Brzozowski-like method of derivatives or regular tree expressions (RTEs).
Bibliography


Reviewed Publications of the Author Relevant to the Thesis


The paper has been cited in:


The paper has been cited in:

Remaining Publications of the Author Relevant to the Thesis


Remaining Publications of the Author


The paper has been cited in:
