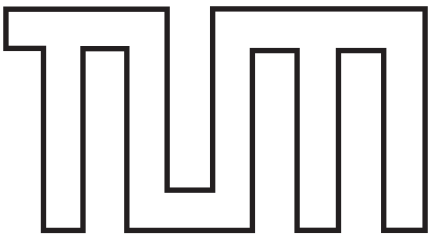


TECHNISCHE UNIVERSITÄT MÜNCHEN
DEPARTMENT OF INFORMATICS

MASTER'S THESIS

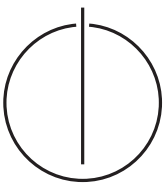
INFORMATICS

Implementation of an Efficient Equivalence Test for Sequential and Linear Tree-to-Word Transducers

Implementierung eines effizienten Äquivalenztests für sequenzielle und lineare Baum-zu-Wort Übersetzer

Author:
Benedikt Zönnchen

Submission date:
February 17, 2016



Supervisor:
Prof. Dr. Helmut Seidl
Advisor:
M. Sc. Raphaela Palenta

Zönnchen, Benedikt
(Familienname, Vorname)

München, 17. Februar, 2016
(Ort, Datum)

Eidesstattliche Erklärung

Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

_____
(Unterschrift)

Abstract

Transducers are applied to various areas in computer science. Especially, tree-to-word transducers are of great interest, since they allow concatenation in the output and they are a suitable model for general XML transformations. We study the equivalence problem for sequential and linear tree-to-word transducers and present a polynomial time algorithm. The algorithm consists of a chain of polynomial time reductions, which ends with the reduction to the morphism equivalence problem on context-free language, which is in PTIME. We discuss the theoretical background, give complexities for all involved algorithms and explain certain algorithms in detail. In our setting, the output is represented by straight-line programs i. e. the output is compressed. Therefore, we implemented a set of algorithms to work with large words especially to test equality of SLP-compressed words.

Zusammenfassung

Übersetzer finden in zahlreichen Bereichen der Informatik ihre Anwendung. Baum-zu-Wort Übersetzer sind dabei von besonderer Bedeutung, da sie die Verkettung der Ausgabe erlauben und ein passendes Modell für generelle XML-Transformationen bieten. In dieser Arbeit analysieren wir das Äquivalenzproblem sequenzieller bzw. linearer Baum-zu-Wort Übersetzer und präsentieren einen Algorithmus, welcher das Problem in polynomieller Zeit löst. Der Algorithmus besteht aus einer Kette von Reduktionen, welche alle in polynomieller Zeit durchgeführt werden können. Am Ende der Kette steht das Morphismusäquivalenzproblem auf kontextfreien Sprachen, welches in polynomieller Zeit lösbar ist. Wir diskutieren alle notwendigen theoretischen Konzepte, geben für jeden implementierten Algorithmen eine Obergrenze bezüglich der Laufzeit an und erklären wichtige Algorithmen im Detail. Wir gehen davon aus, dass die Ausgabe in Form von sogenannten straight-line programs vorliegt d.h. die Ausgabe liegt in komprimierter Form vor. Diese speziellen kontextfreien Grammatiken beschreiben Wörter exponentieller Größe. Alle implementierten Operationen auf diesen großen Wörten werden diskutiert. Besonders hervorzuheben ist der Äquivalenztest für SLP-komprimierte Wörter.

"Wisdom is not a product of schooling but of the lifelong attempt to acquire it."

– Albert Einstein

Preface

This thesis is the crowning glory of my studies, containing small pieces of the knowledge I have learned and loved. My personal, and somehow naive motivation was to understand our complicated and changing world, we know so little about. It became apparent that there is a huge ocean of ignorance and every drop of knowledge exposes another lake. Gaining deep knowledge about a subject is hard. Nevertheless, should we never stop asking questions about the world and ourselves. It is the nature of the human being, a tool for self-reflection and the source of tolerance and humanity.

This thesis wouldn't exist without the help of many special people. First of all, I would like to thank my family for their support during my difficult life. They set the course for what I'm today and taught me a beautiful set of principles and the belief in the good in the world and the people. Furthermore, I want to thank my friends, which are all masters in their own way. Together we went through ups and downs, recover from depressive episodes and enjoy the pleasure of life. They all contribute to my personal attitude and I regret not a single second of my lifetime sharing with them. Being loved and knowing how to love is the greatest gift of my life, which was given to me by these lovely people.

Without the personal and financial support from the Studienstiftung des deutschen Volkes and the Max Weber-Programm it would not have been possible for me to focus so deeply on my studies. Furthermore, these programs made it possible for me to met a lot of interesting people. In the future, I hope I can make my own contribution to support students in their studies.

Thank you all!

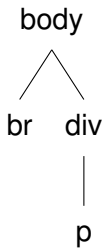
Contents

List of Figures

List of Algorithms

1. Introduction

Automata theory is a beautiful and exciting theoretical branch of computer science, established during the 20^{th} century. It deals with the logic of computation with respect to simple machines and studies their mathematical properties. In 1936, the very first theoretical device, the turing machine, was introduced by Alan Turing. Between 1940 and 1950, finite state automata were studied by W. McCulloch, W. Pitts, S. Kleene and many more. Tree automata gained interest in the 1960s. They have been designed and studied by J.R. Büchi, M.O. Rabin, Doner, Thatcher, etc. Many useful concepts and ideas came out of this program. Tree automata can be used to define XML schema languages, to analyse functional programs or even for model-checking of parallel programs [11]. Due to the popularity of XML, automata over unranked trees were developed and studied since the year 2000. The structure of common XML documents gives a natural tree structure. For example <body>
</br> </body> can be interpreted as a tree of the following form:



Transducers are the natural extensions of automata or more precise, they are automata that turns their input into output. The general case to consider is one where both inputs and outputs are time-series, and the current set of possible outputs depends not just on the current input but on the whole history of previous inputs. Different classes of transducers offer different capabilities and properties. However, a too powerful class results in undecidability of the equivalence problem of transducers of this class e. g. if we allow non-determinism, equivalence becomes already undecidable for very restricted classes of transducers [51].

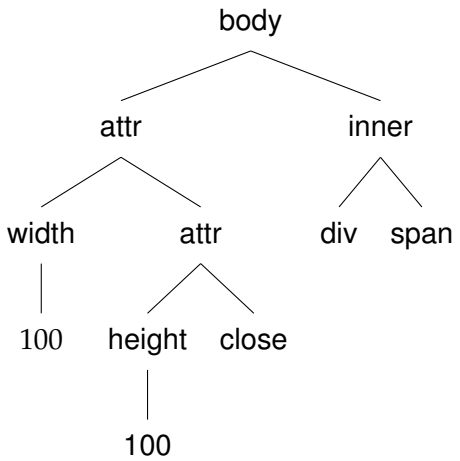
Transducers are applied to various areas in computer science, including static analysis of software [18], automatic translation of natural languages [39, 40, 42], program and data transformation [54, 43], rewriting systems [55], document processing [44, 15] and cryptography [37]. Consequently, they are of great interest in the current research.

(Deterministic) top-down tree-to-word transducers [53] and macro tree transducers [14] offers a fundamental model [51]. Macro tree transducers can be seen as a recursive first-order functional program generating trees. The choice of functions is triggered by top-down pattern matching of an input tree. They combine the feature of top-down tree transducers [48] and macro-grammars [17]. They are able to carry context informations to further processing steps which is not possible for top-down tree transducers.

For each class of automata and transducer, a very interesting question to ask is whether two machines of the same class are equivalent. In case of tree transducers it is easier to decide equivalence compared to macro tree transducers, since they output trees i. e. the output is structured (see [16]). On the other hand, the decidability of equivalence of two macro tree transducers is a well-known and long-standing open question [9]. The ability of concatenation in the output makes them so interesting.

Another very interesting class of transducers allowing concatenation in the output are tree-to-word transducers. Fortunately, the equivalence for tree-to-word transducers is decidable [51]. Restricted classes like deterministic non-copying top-down tree-to-word transducers, also called linear tree-to-word transducers (LTWs) [9], remain powerful despite some limitations, most of them caused by determinism. Similar to deterministic top-down tree automata, LTWs read trees in a top-down fashion and additionally output words before and after each node of the input tree is handled. They may re-order the output by re-ordering the child nodes of each node in the input tree. If we drop the ability of re-ordering, we are in the class of sequential tree-to-word transducers (STWs) [52].

Example 1.1. The following LTW is defined by rules on the right. q_0 is the initial state.



$$
\begin{aligned}
q_0(\text{body}) &\to \text{<body } q_1(x_2)q_2(x_1)\text{></body>} \\
q_1(\text{attr}) &\to q_2(x_1)q_1(x_2) \\
q_1(\text{close}) &\to \epsilon \\
q_2(\text{height}) &\to \text{height = } q_3(x_1) \\
q_2(\text{inner}) &\to q_4(x_2)q_5(x_1) \\
q_4(\text{div}) &\to \text{<div></div>} \\
q_5(\text{span}) &\to \text{} \\
&\dots
\end{aligned}
$$

It outputs

<body width = 100 height = 100><span . . .><div . . .> </body>

The input tree is displayed on the left. The order of the span and div tag is re-ordered.

In 1995, Wojciech Plandowski proved in [47, 35] by giving a construction for a test set for arbitrary context-free languages (CFLs), that the morphism equivalence problem on context free languages is in PTIME. Surprisingly, [52] found a connection between the morphism equivalence on CFLs and the equivalence of STWs. The authors reduce the problem of equivalence problem of STWs to the morphism equivalence problem by constructing a context-free grammar that generate all successful parallel runs of two STWs. The size of the grammar is polynomial in the size of the two STWs and therefore, the equivalence problem of STWs is in PTIME too. This result is quite surprising, since for many other classes of automata the equivalence test is strongly connected to some kind of normal form. Later, [38] introduces the so called *earliest normal form* which leads to a *Myhill-Nerode* characterization. Recently, [9] showed that we can apply the reduction presented in [52], with some preprocessing steps, to LTWs. Additionally, the authors showed that all preprocessing steps are polynomial in the size of the transducers, which implies that the equivalence problem for STWs is in PTIME.

In this thesis we implemented a polynomial time algorithm for deciding the equivalence problem on STWs and LTWs by combining results of the recent research. We solve the main problem by following a chain of reductions given by [9, 52, 47, 35], starting from an instance of the equivalence problem of LTWs, receiving an instance of the morphism equivalence problem on a CFG that generating all successful parallel runs of the LTWs (see fig. 1.1) and finally, receiving a test set containing all so called fully-compressed words we have to compare. For better understanding, and to analyse the complexity in straightforward way, we start by the word equivalence problem of fully-compressed words and end up with the LTW equivalence problem. For solving the main problem we had to work with a large variety of concepts from automata theory and formal languages including context-free languages, straight-line programs (SLPs), word automata, tree automata, nested word-to-word transducers (N2Ws), STWs and LTWs. Each output word is represented by a straight-line program such that the length of words can be exponential in the size of their representation. Since we implemented an polynomial time algorithm it is necessary to manipulate and compare these large words in polynomial time. Thus a main part of this thesis was to implement an efficient algorithm for the equivalence test of so called SLP/fully-compressed words. Solving the morphism equivalence problem on context-free languages leads to the equivalence test of words represented by straight-line programs. Since we extensively manipulate and analyse structure and the language of CFGs, the result of this thesis is not only a single algorithm for testing equivalence of STWs and LTWs, it is in fact a whole software library which supports a large variety of operations, mainly on CFGs.

We decide to structure the thesis in a bottom-up way. We start by very basic and small pieces of the complete solution, until we finally discuss the whole picture. In each chapter we give proofs for the most important statements and analyse the complexity of all implemented algorithms. The thesis is structured as follows:

- In chapter 2 we give the required theoretical background, introducing CFGs, SLPs

and all kind of required transducers and automata.

- Chapter 3 contains the description of our implementation of the most basic operations on CFGs.
- Chapter 4 is one of the main chapters. It contains the description of the recompression algorithm presented in [34] testing equality of fully-compressed words. Furthermore, we describe an extension of this technique: The fully-compressed pattern matching. We use the resulting algorithm to solve the singleton set problem for CFGs, which is also presented in this chapter.
- In chapter 5 we give insights on the generation of test sets for arbitrary context-free languages defined by CFGs and show how we can use the recompression technique to solve the morphism equivalence problem on CFGs, which also leads us to an algorithm for solving the periodicity problem for CFGs.
- Chapter 6 is the last chapter containing descriptions of algorithms. We explain each step of the reduction chain from the equivalence test for STWs to LTWs to nested word-to-word transducers to the morphism equivalence problem on CFGs. This chapter finally gives the big picture.
- In chapter 7 we give performance measures for the equivalence test of fully-compressed words and show the steps and the running time of an example for the equivalence test for LTWs.
- In the last chapter we draw our conclusion and discuss some open questions, challenges and some ideas for future work.

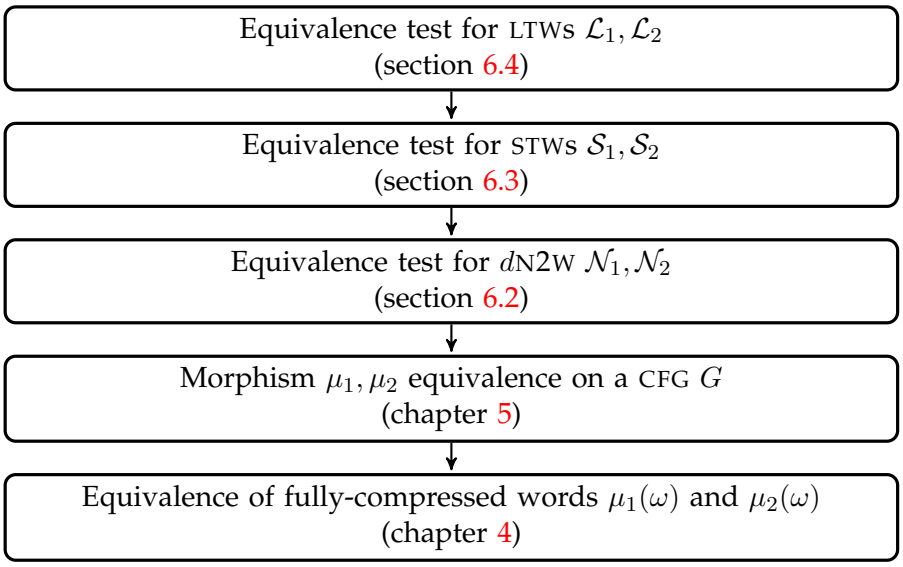


Figure 1.1.: Overview of the structure of the thesis.

2. Preliminaries

We use basic notations from formal language and automata theory. For details we refer the reader to [30, 11, 4]. The definitions of transducers are based on [52, 9]. We notate $\Delta = \{a_1, \ldots, a_n\}$ as the *finite alphabet* of a grammar or a word automata. We call elements from Δ *letters*. Any finite sequence of elements from Δ is called *word* over the alphabet Δ. The word $a_1, \ldots, a_k$ is denoted as $a_1 \ldots a_k$. We usual write ω for a word over Δ. ϵ is called the *empty word*. The set of symbols occurring in ω is $\mathrm{alph}(\omega) = \{a_1, \ldots, a_k\}$. For a natural number k we define the sets of integral numbers $[k] = \{1, \ldots, k\}$ and $[k]_0 = \{0\} \cup [k]$. If $\omega = a_1 \ldots a_k, i \in \mathbb{N}$ then the i-th letter of the word ω is denoted as

$$\omega[i] = \begin{cases} a_i & \text{if } i \in [k] \\ \epsilon & \text{otherwise,} \end{cases}$$

and a *sub word* starting with the i-th letter and ending with the j-th letter as

$$\omega[i \ldots j] = \omega[i] \ldots \omega[j].$$

We call $\omega[1 \ldots i]$ and $\omega[j \ldots |\omega|]$ a *prefix* and *suffix* of ω respectively. The size of a word is equal to the length of the sequence i. e. $|\omega| = |a_1 \ldots a_k| = k$ and $|\epsilon| = 0$. If ω is a prefix of ω' we express this by $\omega \sqsubseteq \omega'$. We introduce *concatenation* $\mathrm{concat} : \Delta^* \times \Delta^* \to \Delta^*$ defined as $\mathrm{concat}(\omega_1, \omega_2) = \omega_3$ with $\omega_3 = \omega_1[1 \ldots |\omega_1|]\omega_2[1 \ldots |\omega_2|]$. We write $\omega_1\omega_2$ instead of $\mathrm{concat}(\omega_1, \omega_2)$. It is easy to see that Δ^* with concatenation forms a *monoid*. We denote by ω^i for $i \geq 0$, the concatenation of i words of ω and define $\omega^0 = \epsilon$. We extends the concatenation to work with sets of words in a natural way. Let L_1, L_2 be two sets containing words over Δ then $L_3 = L_1 L_2$ is defined as

$$L_3 = \{uv \mid u \in L_1, v \in L_2\}$$

We note a^{-1} the inverse of a symbol a where $aa^{-1} = a^{-1}a = \epsilon$. The inverse of a word $\omega = a_1 \ldots a_k$ is $\omega^{-1} = a_k^{-1} \ldots a_1^{-1}$. Any subset $L \subseteq \Delta^*$ of words generated by Δ is called a *language* over the alphabet Δ. Δ^* is as usual defined as the Kleene star on Δ i. e.

$$\Delta^* = \{\epsilon\} \cup \bigcup_{i \in \mathbb{N}} \Delta^i.$$

We use the *random access model (RAM)* with uniform cost measure as our underlying computational model. Furthermore, we assume the reader is familiar with the *Big O notation* defined in [8].

2.1. Grammars and languages

2.1.1. Context-free grammars

A *context-free grammar* (CFG) G is a tuple of finite sets $G = (\Delta, N, S, P)$, where Δ is the set of *terminals*, N is the set of *non-terminals*, S is the start symbol or *axiom* and $P \subset N \times (\Delta \cup N)^*$ is a *set of productions*. We denote productions p as $X \to \alpha$, where $X \in N$ is its *left-hand side* ($\mathrm{lhs}(p)$) and $\alpha \in (\Sigma \cup N)^*$ its *right-hand side* ($\mathrm{rhs}(p)$). We always write non-terminals in upper- and terminals in lower-case. The size of a production $|p|$ is the length of its right-hand side i. e. $|\,\mathrm{rhs}(p)|$. The size of the grammar is

$$|G| = \sum_{p \in P} |\,\mathrm{rhs}(p)|$$

The *language* $L_X(G)$ of a non-terminal X is defined inductively as follows:

$$\frac{\begin{array}{cc} X \to \alpha_0 X_1 \dots X_n \alpha_n \in P & X_1 \dots X_n \in N \\ \omega_1 \in L_{X_1}(G), \dots, \omega_n \in L_{X_n}(G) & \alpha_0, \dots \alpha_n \in \Delta^* \end{array}}{\alpha_0 \omega_1 \dots \omega_n \alpha_n \in L_X(G)}$$

The *language of the grammar* $L(G)$ is defined as $L_S(G)$.

2.1.2. Chomsky normal form

A context-free grammar $G = (\Delta, N, S, P)$ is in *Chomsky normal form* (CNF) if each production in P is of the following form:

(i) $X \to X_1 X_2$ (binary production)

(ii) $X_a \to a$ (terminal production)

(iii) $S \to \epsilon$,

where $X_1, X_2 \in N$ and $a \in \Delta$ and S does not appear in any right-hand side of a production. If a grammar is in *weak Chomsky normal form* (wCNF), we additionally allow productions of the following form:

(iv) $X \to Y$ (unit production)

(v) $X \to \epsilon$, (epsilon production)

where $Y \in N$ and we allow that S appears at any right-hand side of a production of the grammar.

2.1.3. Straight-line programs

A *straight-line program* (SLP) over the finite alphabet Δ is a context free grammar $G = (\Delta, N, S, P)$ such that the following conditions holds:

(i) For every non-terminal $X \in N$ there exits *exactly one* production $p \in P$ such that $\mathrm{lhs}(p) = X$.

(ii) The relation $\{(X, Y) \mid p = X \to \alpha, Y \in \mathrm{alph}(\alpha)\}$ is *acyclic*.

The language generated by a SLP contains *exactly one word* and there is only one derivation tree for this word. This also holds for each language of a non-terminal i. e. $\forall X \in N : |L_X(G)| = 1$. ω_G is the word generated by G. $\mathrm{val}(x) = x$ if $x \in \Delta^*$ and $\mathrm{val}(X)$ with $X \in N$ is the word derived by starting from the production p with $\mathrm{lhs}(p) = X$ i. e. $\mathrm{val}(S) = \omega_G$. In context of SLPs we say p is the production of X if $\mathrm{lhs}(p) = X$.

2.2. Transducers and Automata

We define the size of an automata by the the sum of sizes of its rules (excluding output words), the size of its alphabets and its set of states.

2.2.1. Word Automata

For the sake of completeness we give a standard definition of word automata. A finite word automata is a tuple $\mathcal{A} = (Q, \Delta, \delta, I, F)$, consisting of

- a finite set of states Q,
- the alphabet i. e. a final set of symbols,
- a set of initial states $I \subseteq Q$,
- a set of final states $F \subseteq Q$
- a set of rules $\delta \subseteq Q \times (\Delta \cup \{\epsilon\}) \times Q$.

Let us define the ϵ-closure (ϵ-cl(q)) of a state $q \in Q$ as follows:

$$\frac{q \in Q}{q \in \epsilon\text{-cl}(q)} \qquad \frac{q, p, p' \in Q \qquad p \in \epsilon\text{-cl}(q) \qquad p \xrightarrow{\epsilon} p'}{p' \in \epsilon\text{-cl}(q)}$$

Let $\omega = a_1 a_2 \dots a_n \in \Delta^*$, we say $\omega \in L(\mathcal{A})$ or $\mathcal{A}$ recognize ω if there exists a sequence of states $q_0, \dots, q_n$ with

(i) $q_0 \in \{\epsilon\text{-cl}(q) \mid q \in I\}$

(ii) $q_{i+1} \in \epsilon\text{-cl}(q')$, where $q' \xrightarrow{a_{i+1}}, q_{i+1} \in \delta$ for each $i = 0, \ldots, n-1$ and

(iii) $q_n \in F$

$\mathcal{A}$ is *deterministic* if there are no ϵ-rules, I is a singleton set and for each $a \in \Delta$ and $q \in Q$ there is at most one rule $q \xrightarrow{a} p \in \delta$.

2.2.2. Ranked trees

We call the couple (Σ, rank), where Σ is a finite set of symbols and rank is a mapping $\mathrm{rank} : \Sigma \to \mathbb{N}_0$ that defines the *rank* ($\mathrm{rank}(f)$) of a symbol $f \in \Sigma$, a *ranked alphabet*. We often write Σ instead of (Σ, rank). We notate the set of symbols of rank i by Σ_i. The set T_Σ of *ranked trees* over Σ is the smallest set defined by:

$$\frac{a \in \Sigma_0}{a \in T_\Sigma} \qquad \frac{k \geq 1 \qquad f \in \Sigma_k \qquad \forall i \in [k] : t_i \in T_\Sigma}{f(t_1, \ldots, f_k) \in T_\Sigma}$$

The leaves of the tree are labelled with constant symbols and the internal nodes are labelled with symbols of positive rank, with out-degree equal to the rank of the label. A tree $t \in T_\Sigma$ can also be seen as a partial function $t : \mathbb{N}^* \to \Sigma$ with the *non-empty* and *prefix-closed* domain $\mathcal{P}os(t)$ satisfying the following additional properties:

$$\frac{k \in \mathcal{P}os(t) \qquad t(k) \in \Sigma_n \qquad n \geq 1}{\{j \mid kj \in \mathcal{P}os(t)\} = [n]} \qquad \frac{k \in \mathcal{P}os(t) \qquad t(k) \in \Sigma_0}{\{j \mid kj \in \mathcal{P}os(t)\} = \emptyset}$$

2.2.3. Nested words and unranked trees

Let Σ be a finite set of symbols. The set T^u_Σ is the smallest set defined by:

$$\frac{a \in \Sigma}{a \in T^u_\Sigma} \qquad \frac{k \geq 1 \qquad f \in \Sigma \qquad \forall i \in [k] : t_i \in T^u_\Sigma}{f(t_1, \ldots, f_k) \in T^u_\Sigma}$$

Also these trees can be seen as a partial function with the *non-empty* and *prefix-closed* domain $\mathcal{P}os(t)$, which satisfies the same properties as above. We call $\hat{\Sigma} = \{\mathsf{op}, \mathsf{cl}\} \times \Sigma$ a *nested word alphabet*. *Linearisation* of unranked trees are words in $\hat{\Sigma}^*$ that are *well-nested* in that all opening parenthesis are properly closed. The linearisation $\mathrm{lin} : T^u_\Sigma \to \hat{\Sigma}^*$ is defined as follows

$$\mathrm{lin}(a(t_1, \ldots, t_n)) = (\mathsf{op}, a)\,\mathrm{lin}(t_1) \ldots \mathrm{lin}(t_n)(\mathsf{cl}, a).$$

We define the set of *nested words* over Σ by $N(\Sigma) = \{\mathrm{lin}(t) \mid t \in T^u_\Sigma\}$. A more general definition can be found in [4].

2.2.4. Linear and sequential tree-to-word transducers

A *linear tree-to-word transducer* (LTW) is a tuple $\mathcal{L} = (\Sigma, \Delta, Q, Q_I, \delta)$ where

- Σ is a finite ranked alphabet,
- Δ is a finite word alphabet of output symbols,
- Q is a finite set of states,
- $Q_I \subseteq Q$ is the set of initial states,
- δ is a set of rules of the form

$$q(f(x_1, \ldots, x_n)) \to u_0 q_1(x_{\sigma(1)}) \ldots q_n(x_{\sigma(n)}) u_n$$

  where $q, q_1, \ldots q_n \in Q, f \in \Sigma_n$, $u_0, \ldots u_n \in \Delta^*$ and σ is a permutation from $[n]$ to $[n]$.

Since LTWs and STWs are *deterministic*, there is at most one rule r with $\mathrm{lhs}(r) = q(f(x_1, \ldots, x_n))$ for each pair (q, f). Furthermore, for every $f \in \Sigma$ there exists at most one state $q \in Q_I$ such that there exists a rule r with $\mathrm{lhs}(r) = q(f(x_1, \ldots, x_n))$. To shorten the notation we denote $\mathrm{rhs}(r) = q(f)$. The function $[\![\mathcal{L}]\!]_q$ of a state q is defined inductively as follows:

$$\frac{q(f) \to u_0 q_1(x_{\sigma(1)}) \ldots q_n(x_{\sigma(n)}) u_n \in \delta}{[\![\mathcal{L}]\!]_q(f(t_1, \ldots t_n)) = u_0 [\![\mathcal{L}]\!]_{q_1}(t_{\sigma(1)}) \ldots [\![\mathcal{L}]\!]_{q_n}(t_{\sigma(n)}) u_n}$$

$$\frac{q(f) \text{ is not defined in } \delta}{[\![\mathcal{L}]\!]_q(f(t_1, \ldots t_n)) = \text{undefined}}$$

The function $[\![\mathcal{L}]\!]$ of transducer $\mathcal{L}$ is defined as

$$[\![\mathcal{L}]\!](t) = \bigcup_{q \in Q_I} [\![\mathcal{L}]\!]_q(t),$$

$\mathrm{dom}(\mathcal{L})$ is the domain of $\mathcal{L}$ i. e. all trees t that are recognized by $\mathcal{L}$. We write L_q for the language over Δ, which is generated by the state q. A sequential tree-to-word transducer (STW) is a LTW where for each rule, σ is the identity. LTWs are *non-copying* and STWs additionally *order-preserving*.

2.2.5. Nested word-to-word transducers

A *nested word-to-word transducer* (N2W) $\mathcal{N}$ is a tuple of finite sets $\mathcal{N} = (\Sigma, \Delta, Q, \Gamma, R, Q_I, Q_F)$ where

- Σ is a finite unranked alphabet,
- Δ is a finite word alphabet of output symbols,
- Γ is a finite stack alphabet,
- Q is a finite set of states,
- $Q_I \subseteq$ is the set of initial states,
- $O_F \subseteq$ is the set of final states,
- $R \subseteq Q^2 \times \hat{\Sigma} \times \Delta^* \times \Gamma$ is a set of rules of the form

$$q \xrightarrow{\beta\, a/u:\gamma} q'$$

  where $q, q' \in Q$, $a \in \Sigma, u \in \Delta^*, \gamma \in \Gamma$ and $\beta \in \{\mathsf{op}, \mathsf{cl}\}$.

If $\beta = \mathsf{op}$ and $\mathcal{N}$ is in state q it can read (op, a), output u, put γ onto the stack and go in state q'. If $\beta = \mathsf{cl}$, $\mathcal{N}$ is in state q and γ is the top most element on the stack, the transducer can read (cl, a), output u, remove γ from the stack and go in state q'.

As always, the left-hand side of a rule is $\mathrm{lhs}(r) = q$ and the right-hand side $\mathrm{rhs}(r) = q'$. Furthermore, we define the output $\mathrm{out}(r) = u$, the stack symbol $\mathrm{ssy}(r) = \gamma$ and the the action $\mathrm{act}(r) = (\beta, a)$ of r. An N2W defines a relation $[\![\mathcal{N}]\!]$ based on runs, which annotate opening and closing events of nodes of unranked trees by rules. Nodes can be totally ordered using $\mathcal{P}os(t)$.

We define the set of *traversal actions* $\mathcal{A}ct(t) \subseteq \{\mathsf{op}, \mathsf{cl}\} \times \mathcal{P}os(t)$. $\mathcal{A}ct(t)$ contains all elements of a preorder traversal of t. We write $(\beta, \pi) < (\beta', \pi')$ if (β, π) is properly before (β', π'). $\mathrm{prec}((\beta, \pi))$ is the immediate *predecessor* of (β, π).

A *run* of $\mathcal{N}$ on a tree $t = f(t_1, \ldots, t_n)$ is a function $\tau : \mathcal{A}ct(t) \to R$ such that

$$\mathrm{lhs}(\tau(\mathsf{op}, \epsilon)) \in Q_I$$

and for all $(\beta, \pi) \in \mathcal{A}ct(t)$:

$$\begin{aligned}
&\mathrm{ssy}(\tau(\mathsf{op}, \pi)) = \mathrm{ssy}(\tau(\mathsf{cl}, \pi)) \wedge \\
&\mathrm{rhs}(\tau(\mathrm{prec}(\beta, \pi))) = \mathrm{lhs}(\tau(\beta, \pi)) \wedge \\
&\mathrm{act}(\tau(\beta, \pi)) = (\beta, a),
\end{aligned}$$

where a is the label of the node π in t. We call a run τ successful if $\mathrm{rhs}(\tau(\mathsf{cl}, \epsilon)) \in O_F$ i. e. the run traverses back to the root. We finally define $[\![\mathcal{N}]\!]$ by

$$[\![\mathcal{N}]\!] = \{(t, \mathrm{out}(\tau(e_1)) \dots \mathrm{out}(\tau(e_n))) \mid t \in T^u_\Sigma, \tau \text{ successful run of } \mathcal{N} \text{ on } t, \\ \text{the actions of } t \text{ are } e_1 < \dots < e_n\}.$$

A N2W is *deterministic* if Q_I is a singleton set and for each $q \in Q$ and each $(\mathsf{op}, a) \in \hat{\Sigma}$ there exists only one opening-rule of the form

$$q \xrightarrow{\mathsf{op}\, a/u:\gamma} q'$$

and for each $q \in Q$, $(\mathsf{cl}, a) \in \hat{\Sigma}$ and $\gamma \in \Gamma$ there exits only one closing-rule of the form

$$q \xrightarrow{\mathsf{cl}\, a/u:\gamma} q'.$$

We call N2W *top-down* if all closing-rules have the form

$$q \xrightarrow{\mathsf{cl}\, a/u:q'} q'.$$

3. Algorithms on CFGs and SLPs

In this chapter, we give basic algorithms on context-free grammars. We often assume that grammars are in (weak) Chomsky normal form without any useless productions. Therefore, we implemented algorithms for the deletion of useless productions and the transformation of context-free grammars into (weak) Chomsky normal form.

If we consider words derived from a context-free grammar, we risk an exponential blow up in size. Even the shortest word can be exponential in size of the grammar. To keep complexities polynomial, words of a grammar are represented by straight-line programs. Therefore, we had to implement all required operations on words dealing with SLPs e. g. the deletion of some prefix of a word. A basic, but non-trivial task for SLP-compressed words is equality checking, which we consider in the next chapter.

For solving the morphism equivalence problem on CFGs (see chapter 5) we construct a linear grammar by extracting SLPs, representing short words of a context-free language. Furthermore, we have to compute periods of quasi-periodic languages (see section 6.4). Therefore, we give algorithms for the construction of straight-line programs for the shortest, possibly non-empty word of a context-free language.

We will see that the complexity of all considered algorithms of this chapter, except the algorithm for constructing the shortest word, are linear in the size of the grammar. In case of the algorithm for the construction of the shortest we have to deal with an additional logarithmic factor.

3.1. Data structure

A context-free grammar is represented by an object containing a set of productions and a set of axioms. N is defined by all non-terminals occurring on the right- or left-hand sides and Δ is defined by all terminals occurring on the right-hand sides. We use the default hashset implementation of the Java Standard Library. In case of SLPs each non-terminal can be mapped to exactly one production. To accelerate the access to a

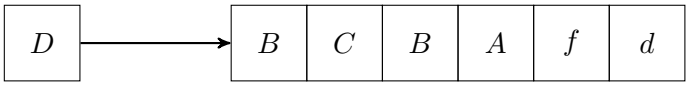


Figure 3.1.: Data structure of a production $D \to BCBAfd$. The left-hand side stored in a linked list.

specific production of a SLP we support the mapping $M : N \to P$ by replacing the set of production by a mapping. We use the default hashset implementation of the Java Standard Library with keyset N and values P. Ensuring that each symbol in $N \cup \Delta$ can be identified by a unique $id \in \mathbb{N}$ and by using an appropriate hash functions, we get the following lemma:

Lemma 3.0.1. *Let P be the set of productions, N be the set of non-terminals and $Map : N \to P$ described above. We achieve average constant running times for:*

(i) Adding a new element to P or N,

(ii) searching for an element P or N,

(iii) removing an element from P or N,

(iv) and for accessing a production p of a SLP*, where $\mathrm{lhs}(p)$ is given.*

In the memory, a production is represented by a simple Java object with members left and right representing the left- and right-hand side of a production. Non-terminals and terminals are of type GSymbol. A GSymbol can be identified as terminal or non-terminal. Left is a single GSymbol and right is a linked list of GSymbols. Searching for any GSymbol with a specific property, that we can test in $\mathcal{O}(1)$, requires $\mathcal{O}(|\mathrm{rhs}(p)|)$ if we have a pointer to the specific production. A GSymbol contains an element of generic type. Consequently, it is possible that the element of a terminal symbol is a complex object like a couple of rules of different linear tree-to-word transducers. It is even possible that a terminal symbol is a word represented by a SLP or a simple list of symbols, but it has to stand for a word over Δ. If a non-terminal represents a word, all algorithms working with or calculating the exact length of a word in $L(G)$ do no longer work. In this chapter we assume that each terminal symbol a in G is a single character, i. e. $|a| = 1$.

Lemma 3.0.2. *Suppose x^* is a pointer to the symbol $x \in (\Delta \cup N)$ occurring on the right-hand side of a production, then the following time bounds hold:*

(i) We can remove or update x in $\mathcal{O}(1)$.

(ii) We can insert k symbols after or before x in $\mathcal{O}(k)$.

(iii) We can find all symbols occurring on the right-hand side of a production p with a certain property, that we can test in $\mathcal{O}(1)$, in $\mathcal{O}(|\mathrm{rhs}(p)|)$.

(iv) We can find all symbols in G with a certain property, that we can test in $\mathcal{O}(1)$, in $\mathcal{O}(|G|)$.

Since the Standard Java Edition does not support access to nodes of the default implementation of a linked list, we implemented our own linked list. A node of the linked list contains an element of generic type. We refer to references to nodes of an element x as pointers to x i. e. x^*. Pointers has an important role when we discuss equality testing and the pattern matching for compressed words (see section 4.1 and 4.2).

3.2. Algorithms on CFGs

Definition 3.1 (Nullables)**.** Let $G = (\Delta, N, S, P)$ be a context-free grammar. A non-terminal $X \in N$ is called *nullable* for G if there exists some derivation starting from X that generates the empty word i. e. $X \Rightarrow_G^* \epsilon$. $Null_G$ is the *set of all nullables* of G. We can define $Null_G$ inductively:

$$\frac{X \to \epsilon \in P}{X \in Null_G} \qquad \frac{X \to X_1 \dots X_n \in P \qquad \forall i \in [n] : X_i \in Null_G}{X \in Null_G}$$

A production p is called *nullable* if there is a derivation of the following form $\mathrm{lhs}(p) \Rightarrow_G \mathrm{rhs}(p) \Rightarrow_G^* \epsilon$.

Remark 3.2. A nullable production contains no terminals. Clearly $\epsilon \in L(G) \iff S \in Null_G$.

Definition 3.3 (Productives)**.** Let $G = (\Delta, N, S, P)$ be a context-free grammar. A non-terminal $X \in N$ is called *productive* for G if there exists some derivation starting from X that generates a word i.e. $X \Rightarrow_G^* \omega$ for some $\omega \in \Delta^*$. $Prod_G$ is the *set of all productives* of G, which can be defined inductively:

$$\frac{X \to \omega \in P, \omega \in \Delta^*}{X \in Prod_G} \qquad \frac{X \to \alpha_0 X_1 \alpha_1 \dots \alpha_{n-1} X_n \alpha_n \in P \qquad \forall i \in [n] : X_i \in Prod_G}{X \in Prod_G}$$

A production p is *productive* if there is a derivation of the following form $\mathrm{lhs}(p) \Rightarrow_G \mathrm{rhs}(p) \Rightarrow_G^* \alpha$ with $\alpha \in \Delta^*$.

Definition 3.4 (Reachables)**.** Let $G = (\Delta, N, S, P)$ be a context-free grammar. We say $X \in N$ is *reachable* if there exists a derivation $S \Rightarrow_G^* \alpha X \beta$, for some $\alpha, \beta \in (N \cup \Delta)^*$. $Reach_G$ is the set of all reachables of G defined by the following properties:

$$\frac{}{S \in Reach_G} \qquad \frac{X \to \alpha_0 X_1 \alpha_1 \dots \alpha_{n-1} X_n \alpha_n \in P \qquad X \in Reach_G \qquad j \in [n]}{X_j \in Reach_G}$$

A production p is reachable if $\mathrm{lhs}(p) \in Reach_G$.

$Prog_G$ and $Null_G$ are inductively defined in a bottom-up fashion. Some other properties, like the minimal length of a non-terminal, can also be defined likewise. To compute those sets or properties efficiently, we use a data structure inspired by [30] (see fig. 3.2). We use a mapping Map to access the linked list for a given non-terminal X in $\mathcal{O}(1)$. The linked list for X contains a node for each occurrence of X on the right-hand sides. The node contains two pointers. One points to the counter of the production p and the other one points to the production p. In a naive implementation we could search for all occurrences of a non-terminal by going over all productions in P, whenever we add this non-terminal to the set of nullables. This would result in a quadratic running time. With the data structure above, which we can construct in $\mathcal{O}(|G|)$, we reduce this to a linear running time.

Lemma 3.4.1. *Algorithm 1 construct the described data structure in $\mathcal{O}(|G|)$.*

Proof. First we construct a linked list L_X for each terminal X in N and put these lists into a hashmap i. e. $Map[X] = L_X$. We go over the whole grammar G and add for each non-terminal X, that occurs on the right-hand side of a production p, a node to the linked list identified by X i. e. we append the node to $Map[X]$. Since we can access the linked list in average $\mathcal{O}(1)$ time, we can do this in overall constant time. While going over the right-hand side of a production, we can construct a counter and all required pointers in constant time as well. ■

Lemma 3.4.2. *Algorithm 2 runs in $\mathcal{O}(|G|)$ and returns all nullable productions.*

Proof. Line 5 in alg. 2 runs in $\mathcal{O}(|G|)$, because each operation inside the loop requires $\mathcal{O}(1)$ time. Since we add a non-terminal $X \in N$ only once to the heap H, we iterate over the linked list of X only once and therefore line 10 requires $\mathcal{O}(|G|)$. ■

Lemma 3.4.3. *Algorithm 31 runs in $\mathcal{O}(|G|)$ and returns all productive productions.*

Proof. The proof is similar to the proof of lemma 3.4.2. ■

The algorithm that returns $Reach_G$ does not use the data structure from above since we progress in a top-down fashion.

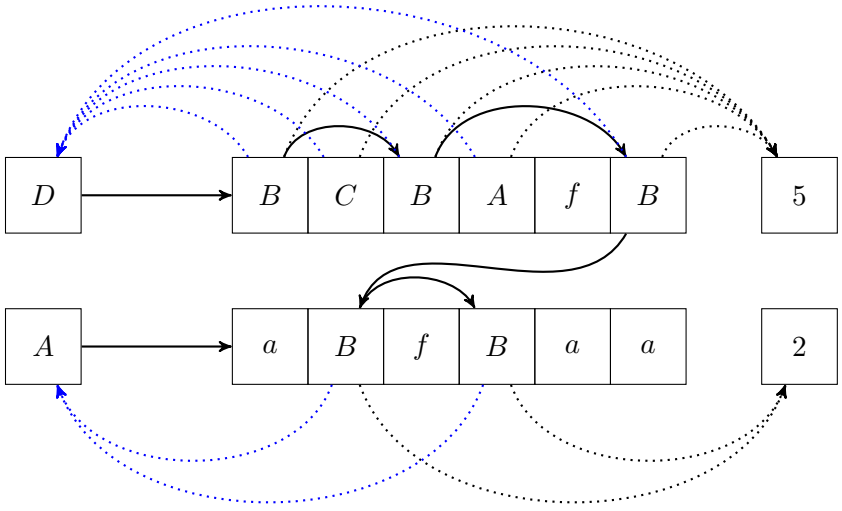


Figure 3.2.: Data structure inspired by [30] for $P = \{D \to BCBAfB, A \to aBfBaa\}$. The black arrowed lines sketch the linked list for B. The dotted black lines stand for pointers of a node and the black dotted lines for pointers to the counter of a production. The blue ones represents pointers to the production.

Algorithm 1: GETMAP: Construct the data structure displayed in fig. 3.2.

input : $G = (\Delta, N, S, P)$;
output: Map, representing the data structure described in [30];
initialize the mapping Map;
foreach $X \in N$ **do**
    $Map[X] \leftarrow$ initialize new linked list L_X;
foreach $p \in P$ **do**
    $counter \leftarrow$ number of non-terminals occurring on $\mathrm{rhs}(p)$;
    foreach $X \in \mathrm{rhs}(p)$ **do**
        $node \leftarrow (p, counter)$;
        $L_X \leftarrow Map[X]$;
        append $node$ to L_X
return Map;

Algorithm 2: GETNULLABLES: Computes the set of nullable productions.

input : $G = (\Delta, N, S, P)$;
output: $Null_G$;
$Map \leftarrow$ GETMAP(G);
$H \leftarrow \emptyset$;
$Null_G \leftarrow \emptyset$;
$NT \leftarrow \emptyset$;
foreach $p \in P$ **do**
    if $p = X \rightarrow \epsilon$ **then**
        $H \leftarrow H \cup p$;
        $Null_G \leftarrow Null_G \cup p$;
        $NT \leftarrow NT \cup \mathrm{lhs}(p)$;
while $H \neq \emptyset$ **do**
    $p \leftarrow$ first element contained in H;
    $H \leftarrow H \setminus p$;
    foreach $(q, counter) \in Map[\mathrm{lhs}(p)]$ **do**
        $counter \leftarrow (counter - 1)$;
        if $counter = 0$ **then**
            $Null_G \leftarrow Null_G \cup q$;
            if $\mathrm{lhs}(p) \notin NT$ **then**
                $H \leftarrow H \cup q$;
                $NT \leftarrow NT \cup \mathrm{lhs}(q)$;
return $Null_G$

Algorithm 3: GETREACHABLES: Computes the set of reachables

```
   input : G = (Δ, N, S, P);
   output: Reach_G;
1  Reach_G ← {S};
2  L ← {S};
3  C ← ∅;
4  while L ≠ ∅ do
5      foreach X ∈ L do
6          foreach p ∈ {q ∈ P | lhs(q) = X} do
7              foreach Y ∈ rhs(p) do
8                  if Y ∉ Reach_G then
9                      Reach_G ← Reach_G ∪ {S};
10                     C ← C ∪ {S};
11     L ← C;
12     C ← ∅;
13 return Reach_G;
```

Lemma 3.4.4. *Algorithm 3 runs in $\mathcal{O}(|G|)$ and gives us all reachable non-terminal symbols.*

Proof. We add each non-terminal to L at most once. Therefore, we have to go over a specific production at most once. ■

We can now combine algorithm 31 and 3 to construct a CFG G' that contains only **useful** productions and non-terminals.

Lemma 3.4.5. *Let $G = (\Delta, N, S, P)$ be a CFG. We can construct in $\mathcal{O}(|G|)$ a CFG $G' = (\Delta, N', S, P')$, with $P' \subseteq P, N' \subseteq N$ and $L(G) = L(G')$ such that $P = \{p \in P \mid p \text{ is useful}\}$ and $N' = \{X \in N \mid X \text{ is useful}\}$.*

Proof. From the definition of useful non-terminals and productions it follows that: If our construction of P' and N' is correct, $L(G) = L(G')$ holds. We choose the following order to construct P' and N':

1. We eliminate unproductives by using alg. 31.
2. We eliminate unreachables by using alg. 3.

We have to show that eliminating all unreachables, after eliminating all unproductives, does not generate new unproductives. Let $X \in N'$ be a non-terminal that is not productive. Since X was productive before deleting reachables and X is no longer productive after eliminating unreachables, there has to be a derivation $X \Rightarrow^*_{G_{prod}} \omega_1 Y \omega_2 \Rightarrow_{G_{prod}} \omega$, such that Y is unreachable. Since X is reachable and there is a derivation $X \Rightarrow_{G_{prod}} \omega_1 Y \omega_2$ it follows that Y is reachable, which leads to a contradiction. ■

Lemma 3.4.6. *Let $G = (\Delta, N, S, P)$ be a* CFG*, then we can construct in $\mathcal{O}(|G|)$ a grammar $G' = (\Delta, N', S, P')$, that is in **weak** Chomsky normal form. Furthermore the size of the new grammar is in $\mathcal{O}(|G|)$ and $L(G) = L(G')$.*

Proof (sketch). For each terminal $a \in \Delta$ we add a fresh non-terminal X_a and a production $X_a \to a$. We replace each terminal a in a production by X_a. After this transformation we replace a production $X \to X_1 \dots X_n$ by $X \to X_1 A_1$, $A_1 \to X_2 A_2 \dots A_{n-2} \to X_{n-1} X_n$, where $A_1, \dots A_{n-2}$ are fresh non-terminals. The constructed grammar is in weak Chomsky normal form. Furthermore, all steps can be done in $\mathcal{O}(|G|)$ and the size of the grammar only increases by a constant factor. For details we refer to [30]. ■

Algorithm 4: GETUSEFUL: Deletes all useless productions from a CFG.

input : $G = (\Delta, N, S, P)$;
output: G without useless productions;
1 $G' \leftarrow$ GETPRODUCTIVES(G);
2 $G' \leftarrow$ GETREACHABLES(G');
3 **return** G'

Lemma 3.4.7. *Let $G = (\Delta, N, S, P)$ be a context-free language in wCNF, then we can construct a grammar G' without nullable productions p with $\mathrm{lhs}(p) \neq S$ in $\mathcal{O}(|G|)$. Additionally, the size of the new grammar is in $\mathcal{O}(|G|)$ and $L(G) = L(G')$.*

Proof (sketch). We first compute all nullable productions and non-terminals following lemma 3.4.2 in $\mathcal{O}(|G|)$. Let $G' = (\Delta, (N \backslash Null_G) \cup \{S\}, S, P')$. P' consists of the following productions:

$$\frac{X \to \alpha \in P \qquad \alpha \in (N \cup \Delta)}{X \to \alpha \in P'} \qquad \frac{S \in Null_G}{S \to \epsilon \in P'}$$

$$\frac{X \to X_1 X_2 \in P \qquad i, j \in \{1, 2\} \ \wedge j \neq i \qquad X_i \in Null_G}{\begin{array}{c} X \to X_1 X_2 \in P' \\ X \to X_j \in P' \end{array}}$$

The grammar size increases at most by $\mathcal{O}(|G|)$, since we replace each production by at most 3 new productions. We can prove by induction over the derivation that $L(G) = L(G')$. ■

Algorithm 5: TOWEAKCNF: Transforms a CFG into weak CNF.

input : $G = (\Delta, N, S, P)$;
output: G transformed into wCNF;
1 $G' \leftarrow$ GETUSEFUL(G);
2 $G' \leftarrow$ REPLACETERMINALS(G');
3 $G' \leftarrow$ TOBINARY(G');
4 **return** G'

Definition 3.5 (Shortest word and shortest non-empty word)**.** Let $G = (\Delta, N, S, P)$ be a context-free grammar. We call $\omega \in L(G)$ a *shortest word* of $L(G)$ if $\forall \omega' \in L(G) : |\omega'| \geq |\omega|$. ω is a *shortest non-empty word* of $L(G)$ if $\forall \omega' \in L(G) \setminus \{\epsilon\} : |\omega'| \geq |\omega|$ and $\omega \neq \epsilon$.

Definition 3.6 (Minimal length)**.** We define the minimal length $\text{len}_P(p)$ of a production p inductively. Let $p = X \to \omega \in \Delta^*$, then $\text{len}(p) = |\omega|$. Let $p = X \to (N \cup \Delta)^*$ and $X_1, \ldots, X_k$ non-terminals of the right-hand side of p, then

$$\text{len}_P(p) := \sum_{i=1}^{k} \min_{\substack{q \in P \\ \text{lhs}(q) = X_i}} (\text{len}_P(q)) + \text{count}_t(p),$$

where $\text{count}_t(p)$ is the number of terminals at the right-hand side of p. The minimal length of a non-terminal X is defined as

$$\text{len}_N(X) := \min_{\substack{p \in P \\ \text{lhs}(p) = X}} (\text{len}_P(p)).$$

Lemma 3.6.1. *Let $G = (\Delta, N, S, P)$ be a non-empty context-free grammar containing only useful productions and let $G' = (\Delta, N, S, P')$ be a* SLP *where P' contains for each $X \in N$ exactly one production p with $\text{len}_P(p) = \text{len}_X(p)$ then $L(G')$ is a singleton set containing the shortest word of G.*

To construct a shortest word we compute the shortest length of each production. We can do this by using the algorithm for computing all productives but we replace the unsorted heap by a sorted one. We sort the productions according to len_P in ascending order.

Lemma 3.6.2. *Let G be a non-empty language, then Algorithm 6 computes a set of productions that represents a* SLP *G that generates a single shortest word of G.*

Proof. We have to prove that $\forall X \in N : len^*_N(X) = \text{len}_N(X)$. Let $p_1, \ldots, p_n$ be the order in which the productions became productive i. e. the order in which we compute $len^*_X(p)$. If the following holds

(i) $len^*_P(p) = \text{len}_P(p)$

(ii) $\text{len}_P(p_1) \leq \ldots \leq \text{len}_P(p_n)$,

then it follows that if we pop a production p from the sorted heap H, $len^*_N(\text{lhs}(p)) = \text{len}_N(p)$. Therefore, we extract only the shortest productive productions for the construction of the SLP. Since $L(G)$ is not empty, S is productive, consequently, the constructed SLP generates a shortest word of G.

(i): We first prove that whenever we access $len^*_N(p)$ then $len^*_N(p) = \text{len}_N(p)$. First of all note that $\forall p \in P$, $len^*_N(p)$ is only defined once for each $p \in P$, namely right before p is pushed onto the heap. Let $p_1, \ldots, p_{|P|}$ be the order in which the productions became productive, i. e. the order in which we define len^*_N in line 8 and 23. Then for $n = 1$ we

get $len^*_N(p_n) = \text{len}_N(p_n)$ since the first productions for which we define len^*_N (see line 8) contain only terminals. Let's assume that $len^*_N(p_i) = \text{len}_N(p_i)$ holds for all $i \leq n'$. The reason for defining $len^*_N(p_{n'+1})$ is that the last remaining non-terminal of the right-hand side of $p_{n'+1}$ becomes productive. By the definition of algorithm 6 $len^*_N(p_{n'+1})$ is defined as

$$len^*_N(p_{n'+1}) := \left(\sum_{i=1}^{k} len^*_N(p_j)\right) + \text{count}_t(p_{n'+1}).$$

By induction hypothesis we have $\forall i \leq n' : len^*_N(p_i) = \text{len}_N(p_i)$ and therefore we get

$$len^*_N(p_{n'+1}) = \left(\sum_{i=1}^{k} \text{len}_N(p_j)\right) + \text{count}_t(p_{n'+1}) = \text{len}_N(p_{n'+1}).$$

(ii): We finally have to prove that $\text{len}_P(p_1) \leq \ldots \leq \text{len}_P(p_n)$. For $n = 1$ this is true, since the first production that we pop, is the shortest one containing only terminals. Assume the statement holds for n' and we pop the production $p_{n'+1}$. Since the heap is sorted all productions that are still contained in the heap have a length greater or equal than $\text{len}_P(p_{n'+1})$. The reason for pushing $p_{n'+1}$ onto the heap is that the last non-terminal Y of the right-hand side became productive i. e. p_j with $\text{lhs}(p_j) = Y$ was popped from the heap. By induction hypothesis this is the largest production p_j i. e. $\text{len}_P(p_l) \leq \text{len}_P(p_j)\ \forall l < j$. Therefore, all popped productions, that were popped before $p_{n'+1}$ was pushed onto the heap, are smaller than $\text{len}_P(p_{n'+1})$, since $\text{len}_P(p_{n'+1}) \geq \text{len}_P(p_j)$ and from the induction hypothesis we get $\forall l \leq j : \text{len}_P(p_l) \leq \text{len}_P(p_j)$. Since the heap is sorted, the statement follows. ■

Lemma 3.6.3. *Let $G = (\Delta, N, S, P)$ be a context-free grammar. If $L(G) \neq \emptyset$, we can construct a* SLP *G' that generates the shortest word, using algorithm 6 in $\mathcal{O}(|G| + |P| \cdot \log(|P|))$. Furthermore, if $L(G) \cap \{\epsilon\} \neq \emptyset$ we can construct a* SLP *G' that represents a shortest non-empty word of $L(G)$ in $\mathcal{O}(|G| + |P| \cdot \log(|P|))$.*

Proof. Since the heap is sorted by len^*_P line 11 and 25 requires $\mathcal{O}(\log(n))$ where n is the number of elements contained in the heap. The heap contains at most $|P|$ elements. Furthermore, after $|P|$ steps the loop in line 10 ends. Consequently, alg. 6 requires $\mathcal{O}(|G| + |P| \cdot \log(|P|))$. The correctness of the set of productions follows from lemma 3.6.2.

Suppose now $L(G) \cap \{\epsilon\} \neq \emptyset$. We can test in $\mathcal{O}(|G|)$ if $\epsilon \in L(G)$. If $\epsilon \notin L(G)$ we compute the shortest word using alg. 6. Otherwise we transform G into a new grammar G' that is in weak Chomsky normal form with $L(G') = L(G)$. We can do this in $\mathcal{O}(|G|)$. After that we transform G' into a grammar G'' that does not contain nullable production with $L(G'') = (L(G') \setminus \{\epsilon\})$. We especially delete the production $S \to \epsilon$. This requires $\mathcal{O}(|G|)$ and $\mathcal{O}(|G''|) = \mathcal{O}(|G|)$ holds. Alg. 6 on G'' computes us the shortest word of $L(G'')$ and therefore the shortest non-empty word of $L(G)$. ■

3.3. Algorithms on SLPs

In every step of the final algorithm that solves the equivalence problem of linear and sequential tree-to-word transducers we have to deal with straight-line programs. In this section, we describe basic algorithms to be able to check certain properties of SLPs. More precisely, let $G_1, \ldots, G_n$ be straight line programs and $k \in \mathbb{N}$, then we require the following operations:

(a) CONCATENATE i. e. compute $\omega_{G_1} \ldots \omega_{G_n}$,

(b) SPLIT i. e. compute ω_l and ω_r with $\omega_{G_1} = \omega_l \omega_r$,

(c) SHIFT i. e. compute $\omega_{G'} = u^{-1} \omega_{G_1} u$ where u is a prefix of $\omega_{G'}$,

(d) EVAL i. e. computation of $\omega_{G'}[k]$ where $k \in [|\operatorname{val}(\omega_{G'})|]$,

(e) LENGTH i. e. computation of $|\omega_{G_1}|$,

(f) MORPH i. e. let $M : \Delta^* \to \Gamma^*$ be a morphism, compute $M(\omega_{G_1})$,

(g) DELETE i. e. let $k_1, k_2 \in \mathbb{N}, k_1 < k_2, k_2 \leq |\omega_{G_1}|$ compute $\omega_{G_1}[1 \ldots k_1]\omega_{G_1}[k_2 \ldots |\omega_{G_1}|]$.

(h) EQUALS i. e. does $\omega_{G_1} = \omega_{G_2}$ hold (see section 4.1)

(i) MATCHING i. e. get the positions of the k^{th} occurrences of ω_{G_2} in ω_{G_1} (see section 4.2)

Before we discuss these operations on SLPs we show that we can check for a given context-free grammar G if the grammar is a well-defined SLP. Since a SLP can be seen as an acyclic graph where each node of the graph represents a non-terminal of the grammar, we can define a topological order relation $\preceq$. If this relation is well-defined on a context-free grammar and there exists only one production with the same left-hand side in the set of productions, then the grammar is a well-defined SLP. We will see that the order relation can be used to simplify the computation of some properties of a non-terminal of the SLP.

Definition 3.7 (Topological order $\preceq$)**.** Let $G = (\Delta, N, S, P)$ be a SLP, then $\forall X, Y \in N$: $X \preceq Y \iff$ there is no occurrence of Y in any derivation starting from X.

Clearly $\forall X \in N : X \preceq S$. To test whether a CFG is a SLP, we first check if $\forall X \in N$: $|\{p \in P \mid \operatorname{lhs}(p) = X\}| \leq 1$ holds, which can be easily done in $\mathcal{O}(|P|)$. Then we have to check if G is acyclic or in other words that the order relation is defined on all useful non-terminals of G.

Lemma 3.7.1. *Let $G = (\Delta, N, S, P)$ be a context-free grammar. We can test if G is a* SLP *in $\mathcal{O}(|G|)$. Furthermore, if the topological order is defined on G, we can compute the relation $\preceq$ in $\mathcal{O}(|G|)$ time.*

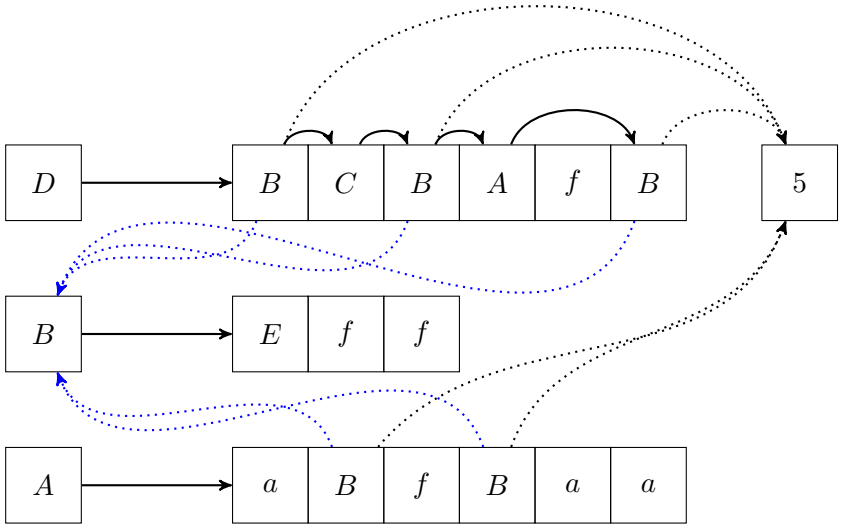


Figure 3.3.: Data structure inspired by [30] with $P = \{D \to BCBAfB, B \to Eff, A \to aBfBaa\}$. The black arrowed lines sketch the linked list for D. The dotted lines represents pointers of a node. The black dotted lines represents pointers to the counter of the non-terminal B, the blue ones represents pointers to the production.

Proof. We construct a similar data structure, we use for the computation of $Null_G$ (see section 3.2) but this time a node, containing X, points to the production p with $\mathrm{lhs}(p) = X$ and to a counter. The counter is initialized with the number of occurrences of a non-terminal X on the right-hand sides (see fig. 3.3). If the counter of X becomes zero we add the production with $\mathrm{lhs}(p) = X$ to the end of a list L. Furthermore, for each non-terminal in $\mathrm{rhs}(p)$ we decrease the corresponding counter by 1. Initially, we can add all non-terminals to L and H, that do not occur on a right-hand side of any production. In case of a well-defined SLP this would be at least the axiom. After this progress terminates the order relation is defined on G if all useful non-terminals of G are contained in the list. Note that we may delete all useless productions beforehand. Clearly, if X comes before Y in the list then $X \succeq Y$. ■

Definition 3.8 (First and last terminal)**.** Let $G = (\Delta, N, S, P)$ be a SLP representing a word $\omega \neq \epsilon$ and $X \in N$, then we define $\mathrm{first}(X) = \mathrm{val}(X)[1]$ and $\mathrm{last}(X) = \mathrm{val}(X)[|\mathrm{val}(X)|]$. We can define $\mathrm{first}(X)$ and $\mathrm{last}(X)$ inductively by the following properties:

$$\frac{p = X \to s_1 \alpha s_2 \qquad s_1 \in \Delta}{\mathrm{first}(X) = s_1} \qquad \frac{p = X \to s_1 \alpha s_2 \qquad s_2 \in \Delta}{\mathrm{last}(X) = s_2}$$

$$\frac{p = X \to s_1 \alpha s_2 \qquad s_1 \in N}{\mathrm{first}(X) = \mathrm{first}(s_1)} \qquad \frac{p = X \to s_1 \alpha s_2 \qquad s_2 \in N}{\mathrm{last}(X) = \mathrm{last}(s_2)}$$

Lemma 3.8.1. *Let $G = (\Delta, N, S, P)$ be a* SLP*. Then we can compute* $\mathrm{last}(X)$ *and* $\mathrm{first}(X)$ *for all productive non-terminals $X \in N$ in $\mathcal{O}(|G|)$ time.*

Proof. We first compute the topological order $\preceq$ on G in $\mathcal{O}(|G|)$. We go over all non-terminal in ascending topological order. Note that we assume that we can access a production p for a given left-hand side in $\mathcal{O}(1)$. Therefore, we start with productions $p = X \to \omega$ where $\omega \in \Delta^*$. Thus $\mathrm{first}(X) = \omega[1]$ and $\mathrm{last}(X) = \omega[|\omega|]$. Since the right-hand side of a production is stored as a linked list we can access the head and the tail of the list in $\mathcal{O}(1)$. We process with the next non-terminals and whenever the head or the tail of the right-hand side of a production is a non-terminal X we can lookup $\mathrm{last}(X)$ and $\mathrm{first}(X)$ (see alg. 8). Since we process in topological order, it is not possible that a non-terminal, for which we compute the last and first value, occurs in any production of any non-terminal, for which we have computed the first and last value beforehand. Therefore, we compute $\mathrm{first}(X)$ and $\mathrm{last}(X)$ for each productive non-terminal. ■

Remark 3.9. If we already computed the topological order we can compute compute $\mathrm{last}(X)$ and $\mathrm{first}(X)$ for all productive non-terminals $X \in N$ in $\mathcal{O}(|N|)$ time. This is useful if we manipulate the grammar without changing the topological order (see chapter 4).

Definition 3.10 (Shifts of a word)**.** Let u be a word, possibly represented by a SLP. Let v be a prefix of u and w be a suffix of u. $v^{-1}uv$ is a *left to right shift* of u by v and wuw^{-1} is a *right to left shift* of u by w.

Definition 3.11. Let $G = (\Delta, N, S, P)$ be a SLP, then we define $\mathrm{len}(s)$ by

$$\frac{s \in \Delta^*}{\mathrm{len}(s) = |s|} \qquad \frac{s \in N}{\mathrm{len}(s) = \mathrm{len}_N(s)} \qquad \frac{s = \omega_0 X_1 \dots X_n \omega_n \in (\Delta \cup N)^*}{\mathrm{len}(s) = \sum_{i=0}^{n} \mathrm{len}(\omega_i) + \sum_{j=1}^{n} \mathrm{len}(X_j)}$$

Algorithm 9 computes a SLP G' where $\omega_{G'}$ is a shifted version of ω_G. If the variable $leftToRight$ is true alg. 9 computes the left to right shift, otherwise it computes the right to left shift.

Lemma 3.11.1. *Let $\omega_G, \omega_{G'}$ be words represented by straight-line programs, $k \in [|val(S)|]$ be a natural number and $\mu : \Delta^* \to \Sigma^*$ be a morphism, then the following holds:*

(i) We can compute $|\omega_G|$ in $\mathcal{O}(|G|)$ time.

(ii) We can compute $\omega_G[k]$ in $\mathcal{O}(|G|)$.

(iii) We can compute $\omega_{G''} = \mu(\omega_G)$ in $\mathcal{O}(|G| \cdot \max\{|\mu(a)| \mid a \in \Delta\})$ time.

(iv) We can compute $\omega_{G''} = \omega_G[k \dots |\omega_G|]$ in $\mathcal{O}(|G|)$ time.

(v) We can compute $\omega_{G''} = \omega_G[1 \dots k]$ in $\mathcal{O}(|G|)$ time.

(vi) We can compute $\omega_{G''}$ such that $\omega_{G''}$ is a shifted version of ω_G, shifted by k in $\mathcal{O}(|G|)$.

(vii) we can compute $\omega_{G''} = \omega_G \omega_{G'}$ in $\mathcal{O}(\max\{|G|, |G'|\})$.

(viii) *We can compute ω_l and ω_r such that $\omega_l = \omega_G[1 \dots k]$ and $\omega_r = \omega_G[k \dots |val(S)|]$ in $\mathcal{O}(|G|)$.*

Proof. We proof each statement one by one.

(i) For the computing the lengths of a production of the SLP we can use alg. 6. Since there is only one production for each non-terminal we replace the sorted heap by an unsorted heap. The complexity of $\mathcal{O}(|G|)$ follows.

(ii) In the first step we calculate $|\operatorname{val}(X)|\ \forall X \in N$ in $\mathcal{O}(|G|)$: We walk down the derivation tree of $\operatorname{val}(S)$. We store in a number r how many non-terminals we have to read until we reach position k. At start $r = k$. So suppose we read the production $X \to_G X_1X_2$. If $|\operatorname{val}(X_1)| < r$ we can skip X_1 and set r to $r - |\operatorname{val}(A)|$. We proceed with the rule $X_2 \to \alpha$. Otherwise we can skip X_2 and proceed with the rule $X_1 \to \beta$. At some point we eventually reach a rule $N_a \to a$ and r becomes 0. Therefore $\operatorname{val}(S)[k] = a$.

(iii) We iterate over all symbols of all productions and replace each terminal a by $\mu(a)$. The resulting grammar can not be larger than $|G| \cdot \max\{|\mu(a)| \mid a \in \Delta\}$.

(iv) We compute $\operatorname{val}(S)[k]$ but instead of skipping symbols we delete these symbols from G.

(v) Symmetric to (iv).

(vi) We compute $\operatorname{val}(S)[|u|]$ but instead of skipping symbols we add them to lists L, R. Whenever the generated word of a symbol is fully part of u we add these symbol to L and whenever a symbol is fully not part of u we add this symbol to R. After we have reached $\operatorname{val}(S)[|u|]$ we construct a new axiom $S^* \to RL$. For details see algorithm 9. We go through $\mathcal{O}(|G|)$ symbols so the size of L plus the size of R is in $\mathcal{O}(|G|)$. We may transform the resulting grammar into wCNF and delete all useless productions. The size of the resulting grammar is in $\mathcal{O}(|G|)$.

(vii) First assume that $N \cap N' = \emptyset$. Let $\Delta'' = \Delta \cup \Delta', N'' = N \cup N'$ and $P = P' \cup P'' \cup \{axiom\}$ with $axiom := S'' \to_{G''} SS'$, then $G'' = (\Delta'', N'', S'', P'')$ generates $\omega_G\omega_{G'}$. We can construct the $axiom$ in constant time and we can merge P into P' in $\mathcal{O}(\min\{|P|, |P'|\})$. If $N \cap N' \neq \emptyset$, we can rename all non-terminals of G or G' beforehand. This requires $\mathcal{O}(\max\{|G|, |G'|\})$. Overall, the operations require $\mathcal{O}(\max\{|G|, |G'|\} + \min\{|P|, |P'|\}) = \mathcal{O}(\max\{|G|, |G'|\})$.

(viii) We first copy G. Then we use (iv) on G to construct ω_l and (v) on the copy to construct ω_r.

■

Lemma 3.11.2. *Let $G = (\Delta, N, S, P)$ be a SLP. Then we can compute in $\mathcal{O}(|G|)$ a SLP G' in Chomsky normal form such that $L(G) = L(G')$.*

Proof (sketch). The transformation is similar to the transformation of a context-free grammar into Chomsky normal form but since there is only one production p with $\mathrm{lhs}(p) = X$ for any $X \in N$ the elimination of nullable productions and unit productions is easy. We divide the construction into 5 steps:

1. We introduce for each terminal $a \in N$ a new fresh non-terminal N_a and replace each occurrence of a in any production by N_a, furthermore we add $N_a \to_{G'} a$ to P'.
2. We split right-hand sides of length at least 3 into right-hand sides of length 2 by introducing new non-terminals, see lemma 3.4.6.
3. We eliminate all ϵ-productions (except for the axiom): If S is nullable we can replace all productions by $S \to \epsilon$, otherwise we can delete all nullable productions.
4. Let U be the set of non-terminals that are the left-hand sides of a unit production. We go over the set U in topological order and thereby redefine $\mathrm{rhs}(A) := \mathrm{rhs}(\mathrm{rhs}(A))$ for every $A \in U$. After that we can delete all unit productions. This requires $\mathcal{O}(|P|)$ time.
5. We delete all useless productions in $\mathcal{O}(|G|)$ time using lemma 3.4.5. The resulting SLP is reduced and in Chomsky normal form.

The proof for the correctness of the construction can be found in [30]. ■

Algorithm 6: GETSHORTESTWORD: Generates a SLP representing the shortest word of a CFG.

input : $G = (\Delta, N, S, P)$ containing only useful productions;
output: SLP $G = (\Delta, N, S, P')$ the shortest word, $\mathrm{len}_N : N \to \mathbb{N}$;

```
1  Map ← GETMAP(G);
2  P' ← ∅;
3  len*_P ← ∅;
4  len*_N ← ∅;
5  H ← ∅ ;                                              // sorted heap
6  foreach p ∈ P do
7      if p = X → α ∧ α ∈ Δ* then
8          len*_P(p) ← |α|;
9          H ← H ∪ p;
10 while H ≠ ∅ do
11     p ← first element contained in H;
12     H ← H \ p;
13     if len*_N(lhs(p)) is undefined then
14         len*_N(lhs(p)) ← len*_P(p);
15         P' ← P' ∪ p;
16         foreach (q, counter) ∈ Map[lhs(p)] do
17             counter ← (counter − 1);
18             if counter = 0 then
19                 l ← 0;
20                 foreach X ∈ rhs(q) do
21                     l ← len*_N(X) + l
22                 l ← l+ number of terminals on rhs(q);
23                 len*_P(q) ← l;
24                 if len*_N(lhs(q)) is undefined then
25                     H ← H ∪ q;
26 G' ← (Δ, N, S, P');
27 return (G', len*_N)
```

Algorithm 7: TOBINARY: Transforms a CFG into a binary CFG.

input : $G = (\Delta, N, S, P)$;
output: G without right-hand sides containing more than two non-terminals;

$P' \leftarrow \emptyset$;
foreach $p = X \rightarrow \omega_1 X_1 \dots X_k \omega_{k+1} \in P$ **do**
  if $k > 2$ **then**
    $p \leftarrow (X \rightarrow \omega_1 X_1 A_{p,1})$;
    $P' \leftarrow P' \cup p$;
    for $i = 1 \dots k-3$ **do**
      $p \leftarrow (A_{p,i} \rightarrow A_{p,i+1} \omega_{i+1} X_{i+1})$;
      $P' \leftarrow P' \cup p$;
    $p \leftarrow (A_{p,k-2} \rightarrow \omega_{p,k-1} X_{k-1} \omega_k X_k \omega_{k+1})$;
    $P' \leftarrow P' \cup p$;
    $N' \leftarrow N \bigcup_{i=1}^{k-2} A_{p,i}$
  else
    $P' \leftarrow P' \cup p$;
$G' \leftarrow (\Delta, N', S, P')$;
return G';

Algorithm 8: GETFIRSTLAST: Computes $\forall X \in N(\text{first}(X), \text{last}(X))$.

input : A reduced SLP $G = (\Delta, N, S, P)$;
output: Computes $\forall X \in N\ (\text{first}(X), \text{last}(X))$;

$L \rightarrow$ GETTOPOLOGICALORDER(G);
for $i = 1 \dots |L|$ **do**
  $X \leftarrow L[i]$;
  $p \leftarrow (X \rightarrow \alpha \in P)$;
  if $\alpha[1] \in \Delta$ **then**
    $first \leftarrow \alpha[1]$;
  else
    $first \leftarrow \text{first}(\alpha[1])$;
  if $\alpha[|\alpha|] \in \Delta$ **then**
    $last \leftarrow \alpha[|\alpha|]$;
  else
    $last \leftarrow \text{last}([\alpha[|\alpha|])$;
  $(\text{first}(X), \text{last}(X)) \leftarrow (first, last)$;
return the mapping $X \rightarrow (\text{first}(X), \text{last}(X))$;

Algorithm 9: SHIFT: Computes a shifted version of ω_G.

input : $G = (\Delta, N, S, P)$ with $\mathrm{val}(S) = \omega_G, k \in \mathbb{N}, leftToRight \in \{true, false\}$;
output: SLP G' representing the shifted word;

```
1  if k = 0 ∨ k mod |val(S)| = 0 then
2      return G;
3  else
4      k ← k mod |val(S)|;
5  if ¬leftToRight then
6      k ← |val(S)| − k;
7  let p be the production with lhs(p) = S;
8  initialize linked lists L, R, R';
9  len ← 0;
10 while k < len do
11     finished ← false;
12     foreach x ∈ rhs(p) do
13         if len + len(x) < k then
14             len ← len + len(s);
15             append x to L;
16         else if ¬finished then
17             finished ← true;
               // x is a non-terminal at this point.
18             let q be the production with lhs(p) = x;
19         else
20             prepend x to R;
21     append R to R' in reverse order;
22     remove all elements form R;
23     p ← q;
24 p_S' ← (S' → R'L);
25 return G' = (Δ, N, S', P ∪ {p_S'})
```

Algorithm 10: GETTOPOLOGICALLORDER: Compute the topological order on an acyclic CFG.

input : $G = (\Delta, N, S, P)$;
output: L that contains all non-terminal in topological order;

initialize the mapping $Count$;
$L \leftarrow \emptyset$;
$H \leftarrow \emptyset$;
$G' \leftarrow \text{GETREACHABLES}(G)$;
foreach $p \in P'$ **do**
    for $X \in \text{rhs}(p)$ **do**
        $Count[X] \leftarrow Count[X] + 1$;
foreach $X \in N'$ **do**
    if $Count[X] = 0$ **then**
        $H \leftarrow H \cup X$;
        append X to L;
while $H \neq \emptyset$ **do**
    $X \leftarrow$ first element contained in H;
    $H \leftarrow H \setminus X$;
    foreach $p \in \{q \in P \mid \text{lhs}(q) = X\}$ **do**
        foreach $X' \in \text{rhs}(p)$ **do**
            $Count[X'] \leftarrow Count[X'] - 1$;
            if $Count[X'] = 0$ **then**
                $H \leftarrow H \cup X'$;
                append X to L;
return L;

4. Advanced operations on compressed words

Deciding whether $L(G_1) = L(G_2)$ holds is a well-known undecidable problem for context-free grammars in general.

Theorem 4.1 ([30]). *The following problem is undecidable:*

Input: *Given* CFGs G_1 *and* G_2

Output: $L(G_1) = L(G_2)$*?*

If we restrict ourselves to acyclic context-free grammars, the problem becomes decidable. However, [31] showed that the problem is NEXPTIME-COMPLETE and therefore, there exists no efficient algorithm for solving the problem.

Theorem 4.2 ([31]). *The following problem is* NEXPTIME-COMPLETE*:*

Input: *Given acyclic* CFGs G_1 *and* G_2

Output: $L(G_1) = L(G_2)$*?*

If we add further conditions to the grammars such that the grammars are SLPs the problem can be solved efficiently. [47], [28] and [45] showed independently that there exists a polynomial time algorithm for the equality test of two SLP-compressed words.

Theorem 4.3 ([47, 28, 45]). *The following problem is in* PTIME*:*

Input: *Given two* SLPs G_1 *and* G_2

Output: $L(G_1) = L(G_2)$*?*

[47] and [28] used combinatorial properties of words, in particular the periodicity lemma of [46] and achieved a $\mathcal{O}(n^4)$ running time, where $n = |G_1| + |G_2|$. Both use the equality test of SLP-compressed words as a tool to solve another problem. [47] used the result to develop a polynomial time algorithm for solving the morphism equality problem for context-free languages which we will discuss in chapter 5. Two promising approaches for solving the equivalence problem for SLP-compressed words were published recently. Both contributions suggest to computes a single natural number which represents a whole word.

The first approach, described in [45], contains an efficient data structure for maintaining dynamically a family of strings. The following operations are supported:

(a) ADD(s): One can add a string s with $|s| = 1$ to the data structure in $\mathcal{O}(\log(m))$, where m is the number of the operation.

(b) EQUALS(s_1, s_2): For two given strings s_1, s_2, that are contained in the data structure, one can check if $s_1 = s_2$ in $\mathcal{O}(1)$.

(c) CONCATENATE(s_1, s_2): For two given strings s_1, s_2, that are contained in the data structure, one can add a concatenated string $s_3 = s_1 s_2$ to the data structure without destroying s_1 and s_2 in $\mathcal{O}(\log(n) \cdot (\log(m) \log^*(m) + \log(n)))$ time, where m is the number of the operation and $n = |s_3|$.

(d) SPLIT(s_3, k): For a given string that is contained in the data structure, one can split the string into two parts without destroying the original string in $\mathcal{O}(\log(n) \cdot (\log(m) \log^*(m) + \log(n)))$ time, where m is the number of the operation and $n = |s_3|$.

Since the data structure supports the CONCATENATE and SPLIT operation, the transition to SLPs is straightforward. With the m^{th} operation, we can generate a string of size at most 2^m, therefore $\log(n) \leq m$. This leads to a cubic running time algorithm for checking equality of SLP-compressed strings [41]. The approach was improved by [3]. They achieve a time complexity of $\mathcal{O}(n^2 \log^*(n))$, where n is the size of the involved grammars i. e. $n = |G_1| + |G_2|$ but the transition to SLPs is not straightforward [34]. The idea behind the approach presented in [45, 3], is to compute for each string s a signature $\sigma(s)$, which is a small unique number. Instead of comparing strings s_1, s_2, they compare signatures $\sigma(s_1), \sigma(s_2)$. The signature is computed by breaking the string s into blocks and replace each block by a single natural number, which is computed by a pairing function. This progress goes on until a single number $\sigma(s)$, which represents s, is computed. They ensure that the boundaries of each block are determined by a small neighbourhood of the block [45, 3]. Suppose we concatenate s_1 and s_2 to get $s_3 = s_1 s_2$. If s_1 and s_2 are already contained in the data structure, we only have to compute a small subset of the natural numbers that are involved in the computation of $\sigma(s_3)$. This advantage arises from the fact that most of the numbers are already computed, since we already computed $\sigma(s_1)$ and $\sigma(s_2)$. Roughly spoken, at each stage of the computation we have to consider only a small neighbourhood around $s_3[|s_1|]$. In some sense we undo the compression locally.

We decide to solve the problem by implement the second approach that was recently published in [34]. This approach uses a technique of local recompression: In each iteration of the algorithm we locally decompress parts of the grammars and compress the whole grammar afterwards. We ensure that the compression of equal sub words results in equal signatures. This relates to [45, 3] in the sense that we also generate some sort of unique fully-compressed representation of a whole SLP-compressed word. However, in contrast to all other contributions, we do not consider any combinatorial properties of the encoded words. Instead, we iteratively analyse and change the way in which words are decried by the SLPs [34]. We wanted to work directly with SLPs without using a complicated data structure. Additionally, the approach presented in [45] requires that numbers of size $\mathcal{O}(|\omega_{G_1}|)$ can be manipulated in constant time, even bit operations are required [34]. The improvement in [3] solves this problem but introduce randomised hashing. The derandomisation approach suffer from a logarithmic factor

[34]. Therefore we decide to implement the recompression algorithm presented in [34], which is the best known algorithm for the SLP-compressed equality test and the fully-compressed pattern matching if $\mathcal{O}(|\omega_{G_1}|)$ fits in one machine word. If this condition does not hold the running time for the equality test is slightly worse then the running time achieved by [3].

Theorem 4.4 ([34]). *Let G_1 and G_2 be two* SLPs *and $n = |G_1| + |G_2|$. We can decide $L(G_1) = L(G_2)$ in $\mathcal{O}(n^2 \log(n))$ time. If $\mathcal{O}(|\omega_{G_1}|)$ fits in one machine word the running time decreases to $\mathcal{O}(n^2)$.*

We additionally implemented the pattern matching for SLP-compressed words using the approach presented in [34], which gives us the ability to test for two fully-compressed words ω, ω' whether ω is a prefix of ω'. In the setting of pattern matching, G_1 represents the text and G_2 the pattern and we ask for the k^{th} occurrences of ω_{G_2} in ω_{G_1}. There is no known algorithm that outperforms the recompression algorithm presented in [34] in the case of fully-compressed pattern matching. In [34] they proved the following:

Theorem 4.5 ([34]). *Assuming that numbers of size M can be manipulated in constant time, we can return an $\mathcal{O}(n+m)$ representation of all pattern occurrences, where n (m) is the size of the* SLP*-compressed text (pattern, respectively) and M is the size of the decompressed pattern. It runs in $\mathcal{O}((n+m)\log(M))$ time. If only numbers of size $n+m$ can be manipulated in constant time, the running time and the representation size increase by a multiplicative $\mathcal{O}(\log(n+m))$ factor. This representation allows calculation of the number of pattern occurrences and, if N fits in $\mathcal{O}(1)$ codewords, also the position of the first/last pattern. Under the same assumption, the position of an occurrence of an arbitrary rank can be given in $\mathcal{O}(n+m)$ time.*

In this chapter we will study two implemented algorithms FCEQUALS and FCPMATCHING which both achieve, an upper bound of $\mathcal{O}(n^2 \log(n))$ with the restriction that numbers of size M can be manipulated in constant time. This is due to the fact that we don't implemented the special representation of large blocks described in [34]. We decided to go ahead and focus on the solution of equivalence problem of STWs and LTWs. However, our approach can be easily extended to achieve the exact same time bounds.

4.1. Fully-compressed equality checking

We will explain each step of our implementation in detail and we will give complexities for all involved algorithms. As already mentioned the recompression algorithm changes the grammars during each iteration. To avoid confusion let $G_t = (\Delta, S_t, N_t, P_t)$ be the changing grammar of the first word and $G_p = (\Delta, S_p, N_p, P_p)$ the changing grammar of the second word or, in case of pattern matching, the changing grammar of the pattern. W. l. o. g. let $N_t \cap N_p = \emptyset$. Let $G = (\Delta, \{S_t, S_p\}, N_t \cup N_p, P_t \cup P_p)$ be the changing grammar representing the set of words i. e. $L(G) = \{\omega_{G_t}, \omega_{G_p}\}$. Note that Δ changes during the algorithm as well. Furthermore, let n and m be the size of original grammars i. e. $|G_t|$ and $|G_p|$ before the first iteration of the algorithm.

Similar to [45, 3] the goal is to compute a signature $\sigma(G_t)$ and $\sigma(G_p)$ and then decide whether $\sigma(G_t) = \sigma(G_p)$ holds. In each *phase* of the computation, the grammars G_t and G_p change such that ω_{G_t} and ω_{G_p} shorten by constant factor. In some sense we carefully refactor the grammar while storing all required informations to answer the final question. Clearly, in case of equality checking

$$\sigma(G_t) = \sigma(G_p) \iff \omega_{G_t} = \omega_{G_p}$$

has to hold in each step of the algorithm.

Definition 4.6 (Pair)**.** Let ω be a word over an alphabet Δ. We call ab with $a, b \in \Delta$ a *pair* of ω if $i \in [|\omega| - 1]$ and $\omega[i] = a \ \wedge \ \omega[i+1] = b$. We call a pair a *proper* pair if $a \neq b$. We say ab *occurs* at position $i \in [|\omega|]$ in ω if $\omega[i] = a \ \wedge \ \omega[i+1] = b$.

Definition 4.7 (Block)**.** Let $l > 1$ and $a \in \Delta$. We call a^l a *block* of length l. Let ω be a word over an alphabet Δ. We say a block a^l *occurs* at position i in ω if $i \in [|\omega| - l]$ and $\omega[i] = \ldots = \omega[i+l] = a \ \wedge \ \omega[i-1] \neq a \ \wedge \ \omega[i+l+1] \neq a$.

Remark 4.8. ab or a^l is a pair or block respectively, of ω, if and only if ab or a^l occurs at some position in ω.

4.1.1. The replacement schema

The replacement schema of [34] is not the same as the schema described in [45, 3]. We compress proper pairs ab and blocks a^l, but without breaking the word into blocks. We will shortly discuss the major disadvantage that arises form skipping this particular step in section 4.3. We apply the following two basic compressions:

- Pair compression of a pair ab: We replace each occurrence of a proper pair ab in ω_G by the same fresh letter c.
- Block compression of a block a^l: We replace each (maximal) occurrence of a block of length $l > 1$ in ω_G by a fresh letter a_l.

From now on we write a_l for the replacement of a^l. It is essential that if we replace a pair or a block, we replace all occurrences in ω_G in a row. Otherwise the following trivial lemma would not hold:

Lemma 4.8.1. *Let $\omega_{G_t'}$ and $\omega_{G_p'}$, obtained from ω_{G_t} and ω_{G_p} respectively, by compressing a global pair ab or a global block a^l in the words then*

$$\omega_{G_t'} = \omega_{G_p'} \iff \omega_{G_t} = \omega_{G_p}.$$

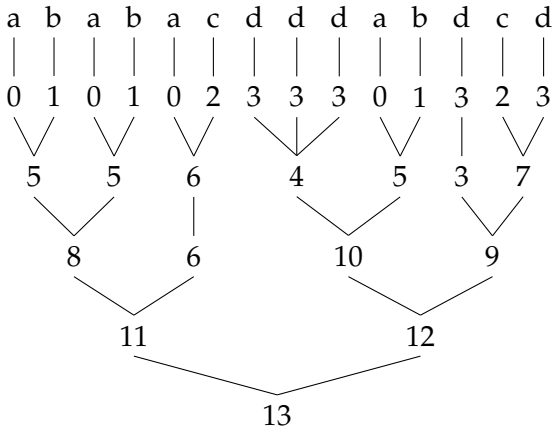
Example 4.9 (Pair compression)**.** Let $\omega_{G_t} = \omega_{G_p} = ababcdabbb$. After replacing ab by e we get $\omega_{G_t'} = \omega_{G_p'} = eecdebb$. The compression of ab destroys pairs $\{bc, da\}$ i. e. pairs $b\Delta$ and Δa. Therefore, we are not allowed to compress bc or da in between the compression of ab. The same is true for b-blocks. Suppose we do not compress all occurrences in a row. Let us compress ab in ω_{G_t} and da afterwards. In ω_{G_p} we compress da first and ab afterwards. We would get $eecdebb = \omega_{G_t'} \neq \omega_{G_p'} = eecfbbb$.

To realise the compression in-order, the recompression algorithm sorts pairs and blocks with RADIXSORT.

Lemma 4.9.1 (Radix sort)**.** *Using* RADIXSORT *we can sort* $\mathcal{O}(n+m)$ *numbers of size* $\mathcal{O}((n+m)^c)$ *for some constant* c *in* $\mathcal{O}(c(n+m))$ *time.*

The recompression algorithm iteratively compresses all blocks and pairs. In each iteration a letter a is compressed at most once. We call one iteration a *phase* of the recompression. A phase starts with gathering pointers to all blocks and pairs in ω_{G_t} and ω_{G_p}. We sort these pointers according to (a, b) (in case the pointer points to a pair ab) or according to (a, l) (in case the pointer points to a block a^l). Afterwards, we replace each occurrence of pairs and blocks in ω_{G_t} and ω_{G_p} by fresh letters (see alg. 11). Finally, we rename letters that were not replaced by any fresh letter of the current phase.

Example 4.10 (uncompressed recompression)**.** The following example shows how the recompression algorithm works for uncompressed words without a renaming step for each phase. Let $\omega_{G_t} = \omega_{G_p} = ababacdddabdcd$ two uncompressed words. Alg. 11 computes in 4 phases $\sigma(G_t) = \sigma(G_p) = 13$:



Luckily in each phase the symbols form an integral interval. Occurrences of pairs are sorted according to the first component, and then to the second one. We compress them in-order.

We use the same data structure described in chapter 3. A word is a linked list of GSymbols. For the recompression algorithm we have to store some additional informations. Therefore, we extend the GSymbol to JezSymbol.

Definition 4.11 (JezSymbol)**.** A JezSymbol is a quadruple $symbol = (id, phase, len, weight) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$. The parameters have the following meaning:

- id: The unique number of a symbol of a phase
- $phase$: The phase in which this symbol was created

Algorithm 11: FCEQUALS: Checks whether two words are equal.

input : G with $L(G) = \{\omega_{G_t}, \omega_{G_p}\}$;
output: true if $\omega_{G_t} = \omega_{G_p}$, otherwise false;

```
1 if |ω_{G_p}| ≠ |ω_{G_t}| then
2     return false;
3 FIRSTRENAME(G);
4 while |ω_{G_p}| > 1 do
5     COMPRESSBLOCKS(G);
6     Pairs ← GETPAIRS(G);
7     COMPRESSPAIRS(Pairs, true);
8     RENAME(G);
9 return ω_{G_t} = ω_{G_p};
```

- len: The length of the symbol a, e. g. if the length of a symbol a is greater than 1, the symbol $a = (id_a, phase_a, len_a, weight_a)$ represents a block of symbols $b = (id_a, phase_a, 1, weight_a)$
- $weight$: The length of the fully uncompressed version of the symbol i. e. the weight of all symbols at the beginning of the algorithm is equal to 1. Weights became important if we calculate the position of the occurrence of a pattern in ω_{G_t} (see section 4.2).

Remark 4.12 (Grammar position). Let $\preceq$ be a topological order on G. We can ensure that such a order exists and we can compute in $\mathcal{O}(n + m)$ time a list L, containing non-terminals in topological order by using algorithm 10.

Definition 4.13 (Grammar position)**.** Let's say we have $X_1 \preceq \ldots \preceq X_{|N|}$. We notate $G[i]$ as the occurring symbol at $(\mathrm{rhs}(X_1) \ldots \mathrm{rhs}(X_{|N|}))[i]$.

We assume that all letters are numbered from an input alphabet $\Delta = [|G|^c]_0$ for some constant c. Before we start the recompression we rename all letters in G such that each letter is in $[|G|]_0$.

Theorem 4.14. *Let ω_{G_t} and ω_{G_p} be two uncompressed words. We can decide in $\mathcal{O}(n + m)$, whether $\omega_{G_t} = \omega_{G_p}$ holds by using the recompression technique.*

Proof. Note that in this case $\mathcal{O}(n + m) = \mathcal{O}(|\omega_{G_t}| + |\omega_{G_p}|)$ at the beginning of the algorithm. First of all, we replace each letter in ω_{G_t} and ω_{G_p} by an appropriated JezSymbol (see alg. 12). Since all ids of symbols are in $[|G|^c] = [(n + m)^c]$ for some constant c, we can gather and sort all pointers in $\mathcal{O}(n + m)$ by going over the words. Since we rename all occurrences of letters in-order we require no dictionary operations. After line 3 in algorithm 11, each id of any symbol is in $[n + m]_0$.

In the same way we gathered all pointers to all letters in ω_{G_t} and ω_{G_p}, we can gather pointers to all occurrences of blocks and pairs in ω_{G_t} and ω_{G_p}. Note that pointers always

Algorithm 12: FIRSTRENAME: Renames all letters in G.

```
input : G with L(G) = {ω_{G_t}, ω_{G_p}};
initialize the list Pointers;
for i = 1 ... |G| do
    if G[i] is a terminal then
        create new pointer x* to G[i];
        append x* to Pointers;
Pointers ← RADIXSORT(Pointers);
count ← −1;
for i = 1 ... |Pointers| do
    if i = 1 ∨ Pointers[i − 1] ≠ Pointers[i] then
        count ← count + 1;
    replace the letter at Pointer[i] by a (count, 0, 1, 1);
```

point to the left most element of a pair or block, respectively. A pair is represented by a triple (a, b, x^*), where a, b are JezSymbols and x^* points to a. A block is represented by a triple (a, l, x^*), where a is a JezSymbol, l is the length of the block and x^* points to the left most symbol of the block. We sort these triples with respect to their first two components using RADIXSORT. We can do this in $\mathcal{O}(n + m)$ time. We can now assign a fresh id to all equal pair or block occurrences in parallel by iterating over the list of sorted pointers without any dictionary operations by starting with $id = 0$. Furthermore, we can compute the new $phase, length$ and $weight$ of the JezSymbol as follows:

- compression of a pair ab by c: $c = (id, phase_a + 1, 1, weight_a + weight_b)$
- compression of a block a^l by a_l: $a_l = (id, phase_a + 1, 1, weight_a \cdot l)$

We can do this for all pointers in overall $\mathcal{O}(n + m)$. It may happen that a pointer x^* points no longer to a pair, since some part of the pair was already compressed. We can check this for a pair by comparing the symbol at x^* with a and the symbol at $x^* + 1$ with b in constant time. If the pair disappeared, we do nothing. Since we compress blocks before pairs, blocks will never disappear before compressing them.

After the block and pair compression, we rename all symbols of $phase$ equal to the number of the last phase i. e. all uncompressed symbols. We gather all pointers to those symbols in $\mathcal{O}(n + m)$ time by going over ω_{G_t} and ω_{G_p} again. We sort all pointers in $\mathcal{O}(n + m)$ time by the id of the symbol they are pointing at, using RADIXSORT. We apply algorithm 12 but we start with $count$ equal to the largest id of fresh symbols. After the renaming, each symbol can be identified by an $id \in [n + m]_0$.

We now have to prove that we shorten the words by a constant factor: Let us look at the beginning of an arbitrary phase of algorithm 11 and a sub word ω of ω_{G_t} with $|\omega| = 2$. Clearly, if ω is part of a block, it will be replaced by a single letter. Let's assume ω is a

proper pair ab. Suppose a won't be compressed by the current phase, then b has to be compressed at the end of the phase, otherwise the pair ab occurs in ω_{G_t} after the phase, which is, by the definition of algorithm 11, not possible. In any case, one of the letter of ω is compressed after the phase.

Let us assume a letter u is not compressed after the phase. Then by the argument above, the two letters to the right of u are compressed (if u is not the last letter of ω_{G_t}). Therefore, for each uncompressed letter u (expect for the last one letter of ω_{G_t}) we get two compressed letters to the right i. e.

$$m_u - 1 \leq 2 \cdot m_c,$$

where m_u is the number of uncompressed and m_c is the number of compressed letters of ω_{G_t}. From this we get

$$m_u - 1 \leq 2 \cdot m_c \iff m_u + m_c - 1 \leq 3m_c \iff |\omega_{G_t}| - 1 \leq 3m_c \iff \frac{|\omega_{G_t}| - 1}{3} \leq m_c.$$

At least each compressed two letter of ω_{G_t} are replaced by one single fresh letter and thus the length $|\omega_{G_{t'}}|$ of the new text $\omega_{G_{t'}}$ can be bounded by

$$|\omega_{G_{t'}}| \leq |\omega_{G_t}| - \frac{|\omega_{G_t}| - 1}{3} \iff |\omega_{G_{t'}}| \leq \frac{2|\omega_{G_t}| + 1}{3}.$$

Therefore, ω_{G_t} shorten by a constant factor - the proof for ω_{G_p} is similar. Thus, algorithm 11 terminates after a constant number of iterations if ω_{G_t} and ω_{G_p} are uncompressed. Since the running time of a single phase is linear, the theorem follows. ■

Example 4.15. The same situation as described in example 4.10 but with a renaming step in each phase. We guarantee that after each phase the set of ids is in $[|\Delta|]_0$.

4.1.2. The local decompression

In the setting of compressed words, ω_{G_t} and ω_{G_p} are represented by SLPs G_t, G_p and therefore, $|G_t| \neq |\omega_{G_t}|$ possibly holds. Gathering and compressing pairs or blocks seems difficult, since they can be distributed over different productions of the grammar.

Example 4.16 (Distributed pairs and blocks)**.** Let $G_t = (\Delta, N, S, P)$ with $P = \{S \to aX_1X_1bX_3, X_1 \to ba, X_3 \to bbbX_4, X_4 \to bbab\}$. We have $\omega_{G_t} = abababbbbbab$, pairs $\{ab, ba\}$ and one block $\{bbbbb\}$. Some pairs are distributed over two productions. The block is distributed over three productions.

We introduce the second important property of the recompression algorithm: local modification of the grammars G_t and G_p such that we can apply the compression of words and blocks. In particular, we will decompress the grammar locally [34].

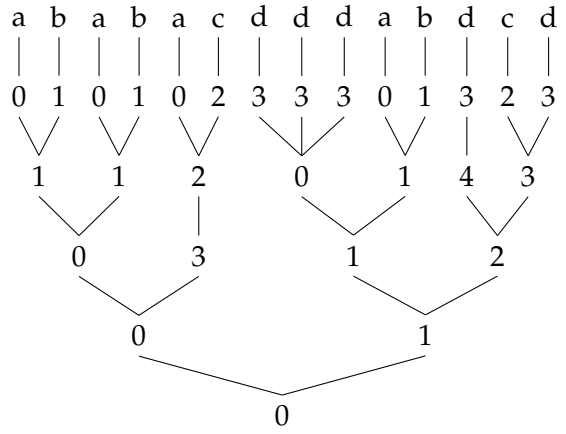


Figure 4.1.: Equality checking by using the recompression technique. Each number represents the id of a corresponding JezSymbol. Pairs ab are sorted according to the first component a and then to the second one.

Without loss of generality we assume that the grammar G is in Chomsky normal form without any useless or nullable non-terminals. We can ensure this by rename all non-terminals beforehand and apply lemma 3.11.2 to transform the grammars into CNF in linear time. As already mentioned in remark 4.12 let $X_i \preceq X_j \iff i \leq j$.

Definition 4.17 ((Non-)crossing pairs [34])**.** Consider a pair ab and its fixed occurrence in $\mathrm{val}(X_i)$, where the production for X_i is $X_i \to \alpha_1 X_j \alpha_2 X_k \alpha_3$ (or $X_i \to \alpha_1 X_j \alpha_2$ or $X_i \to \alpha_1$). We say that this *occurrence* is

1. *explicit* for X_i if this ab comes from α_1, α_2 or α_3
2. *implicit* for X_i if this occurrence comes from $\mathrm{val}(X_j)$ or $\mathrm{val}(X_k)$ and
3. *crossing* for X_i otherwise.

A pair ab is *crossing* if it has a *crossing occurrence* for **any** X_i, otherwise it is non-crossing.

Remark 4.18. If ab occurs implicit for X_i it has to be crossing or explicit for some other non-terminal.

Lemma 4.18.1. *A pair ab is crossing if and only if one of the following conditions hold:*

(i) aX_i *occurs in some production and* $\mathrm{first}(X_i) = b$

(ii) $X_i b$ *occurs in some production and* $\mathrm{last}(X_i) = a$

(iii) $X_i X_j$ *occurs in some production* $\mathrm{last}(X_i) = a$ *and* $\mathrm{first}(X_j) = b$

Example 4.19 (Explicit, implicit and crossing occurrences)**.** If we look back to example 4.16 at the production $S \to aX_1X_1bX_3$ we can identify ab as a crossing pair since there is a crossing occurrence for S: $\mathrm{last}(a) = a$ and $\mathrm{first}(X_1) = b$. Furthermore, there is an explicit occurrence of ab for X_4 and an implicit occurrence of ab for X_3.

Definition 4.20 ((Non-)crossing blocks [34])**.** Consider a letter a and an occurrence a^l in $\mathrm{val}(X_i)$, where the production for X_i is $X_i \to \alpha_1 X_j \alpha_2 X_k \alpha_3$ (or $X_i \to \alpha_1 X_j \alpha_2$ or $X_i \to \alpha_1$). We say that this occurrence is

1. *explicit* for X_i if this a^l comes from α_1, α_2 or α_3
2. *implicit* for X_i if this occurrence comes from $\mathrm{val}(X_j)$ or $\mathrm{val}(X_k)$ and
3. *crossing* for X_i otherwise.

A letter a has a *crossing* block of length l if a^l has a *crossing occurrence* for **any** X_i, otherwise it has no crossing blocks.

Lemma 4.20.1 ([34])**.** *Let ab (a^l) be a non-crossing pair (a non-crossing block). Let $G_{t'}, G_{p'}$ be the grammars constructed by replacing each explicit occurrence ab (a^l) by a fresh letter c (a_l), in G_t and G_p receptively, then*

$$\omega_{G_t} = \omega_{G_p} \iff \omega_{G_{t'}} = \omega_{G_{p'}}$$

holds.

4.1.3. Compression of non-crossing pairs

Lemma 4.20.2. *We can compress all non-crossing pairs in $\mathcal{O}(|G|)$ time by applying algorithm 13 and 14 i. e.* CompressPairs(GetPairs(G), $false$).

Proof. We go over all right-hand sides of G. whenever we spot an explicit pair ab, we append $(a, b, crossing, x^*)$ to the list $Pairs$ of pairs. $crossing$ is a flag that indicates that the entry represents an occurrence of a non-crossing ($crossing = 1$) or an occurrence of a crossing ($crossing = 0$) pair. Lemma 4.18.1 gives us an algorithm for computing all crossing pairs:

We can compute $\mathrm{first}(X)$ and $\mathrm{last}(X)$ for all non-terminals in G in $\mathcal{O}(n+m)$ time in a bottom up fashion by using alg. 8. By using $\mathrm{last}(X), \mathrm{first}(X)$ while scanning G for non-crossing pairs, we can simultaneously spot all crossing pairs see algorithm 13 e. g. if we spot aX at some position we add the pair $(a, \mathrm{first}(X), 0, x^*)$ to the list of pairs if $\mathrm{first}(X) \neq a$.

For all crossing occurrences of pairs we set $crossing$ to 0. Then we sort all spotted pairs using RadixSort with respect to the first 3 components. By lemma 4.26.3 the size of Δ is in $\mathcal{O}((n+m)^3)$, and RadixSort sorts the entries in $Pairs$ in $\mathcal{O}(|G| + n + m) = \mathcal{O}(|G|)$ time. Since $Pairs$ is sorted according to $crossing$, we can easily identify crossing and non-crossing pairs. There are only $\mathcal{O}(|G|)$ entries in $Pairs$. We can introduce a fresh letter for each non-crossing pair in constant time. Whenever we introduce a fresh letter we will never require to lookup any letters introduced before, since the entries are sorted according to ab.

We can replace all explicit pairs ab in constant time since we have pointers x^* to their occurrence (see alg 14). Thus, each iteration step requires $\mathcal{O}(1)$ time and algorithm 14 compresses all non-crossing pairs in $\mathcal{O}(|G|)$ time. ■

Algorithm 13: GETPAIRS: Returns a unsorted list of all pairs in G.

input : compressed represented by $G = (\Delta, N, \{S_t, S_p\}, P)$;
output: pointers to all proper crossing and non-crossing pairs of G;

```
1  compute ∀X (last(X), first(X)) using GETFIRSTLAST(G);
2  initialize the list Pairs;
3  foreach q ∈ P do
4      r ← rhs(q);
5      for i = 1 ... |rhs(q) − 1| do
6          x* ← undefined;
7          crossing ← 0;
8          if r[i] ∈ Δ ∧ r[i + 1] ∈ Δ then
9              x* ← pointer to r[i];
10             a ← r[i];
11             b ← r[i + 1];
12             crossing ← 1;
13         else if r[i] ∈ Δ ∧ r[i + 1] ∈ N then
14             a ← r[i];
15             b ← first(r[i + 1]);
16         else if r[i] ∈ N ∧ r[i + 1] ∈ Δ then
17             a ← first(r[i]);
18             b ← r[i + 1];
19         else if r[i] ∈ N ∧ r[i + 1] ∈ N then
20             a ← last(r[i]);
21             b ← first(r[i + 1]);
22         if a ≠ b then
23             append (a, b, crossing, x*) to Pairs
24 return Pairs;
```

4.1.4. Compression of crossing pairs

From lemma 4.20.1 we derive a strategy, for reducing the problem of the compression of crossing pairs and blocks to the uncrossed case. If we want to compress a crossing pair ab or a crossing block a^l we first uncross the pair or the block, respectively, and compress it afterwards. Uncrossing increases the size of the grammar, since we introduce new symbols to a production and can therefore be seen as a decompression of the

SLP-compressed word. We will see that the size of the grammar does not increase too much and that we decrease the size of the fully uncompressed words at the end of a phase due to the recompression.

Algorithm 14: CompressPairs: Compresses uncrossed or all pairs in $Pairs$.

```
   input : pairs Pairs, all ∈ {true, false};
1  Pairs ← RadixSort(Pairs);
2  initial list CrossingPairs
3  while Pairs ≠ ∅ do
4      entry ← (a, b, crossing, x*) ∈ Pairs;
5      remove entry from Pairs;
6      if ≠ crossing then
7          a' ← a;
8          b' ← b;
9          c ← fresh symbol;
10         foreach entry ← (a', b', crossing, x*) ∈ Pairs with a = a' ∧ b = b' do
11             if ab is still at x* then
12                 replace ab by c at x*;
13             remove entry from Pairs;
14     else
15         foreach entry ← (a', b', crossing, x*) ∈ Pairs with a = a' ∧ b = b' do
16             remove entry from Pairs;
17         append entry to CrossingPairs;
18 if all then
19     CompressAllCrossingPairs(CrossingPairs);
```

Let ab an occurrence of a crossing pair. We know ab is crossing due to (i), (ii) or (iii) of lemma 4.18.1. In case (i) we can uncross a pair by replacing all occurrences of aX_i by abX_i, then we delete b from $\mathrm{val}(X_i)$. We call this a *left-pop* of b [34]. (ii) is symmetric, i. e. we *right-pop* a. In case (iii) we replace all occurrences of X_iX_j by X_iabX_j and delete the last letter (a) from $\mathrm{val}(X_i)$ and the first letter (b) from $\mathrm{val}(X_j)$ i. e. left-pop a and right-pop b. In fact it is not necessary for left-popping b that $\mathrm{val}(X_i)$ starts with a. For the pop operation, we have to get access to all occurrences of a non-terminal in constant time. The following lemma gives us this ability.

Lemma 4.20.3. *We can construct a mapping Occ that maps each non-terminal $X \in N$ to a list of pointers L_X such that: For every occurrence of X on right-hand sides, there is exactly one pointer in L_X, pointing to that occurrence of X. We can construct Occ in in $\mathcal{O}(n+m)$ time.*

Proof. We go over all right-hand sides and whenever we spot a non-terminal X we add a pointer to the list L_X. We can identify this list, create the pointer and append

the pointer to the list in constant time. We have to do this only once before we start the algorithm. Therefore, we can create all lists in overall $\mathcal{O}(n+m)$ time. ■

Lemma 4.20.4. *Let Δ be the actual terminal alphabet and $\Delta_l, \Delta_r \subseteq \Delta$ with $\Delta_r \cap \Delta_l = \emptyset$. After we left-pop all $b \in \Delta_r$ and right-pop $a \in \Delta_l$ simultaneously, no pair $ab \in \Delta_l\Delta_r$, is crossing. Furthermore, ω_{G_t} and ω_{G_p} have not changed.*

Proof (sketch). Let us look at non-terminal X with a crossing occurrence for ab before we left-pop b and right-pop a. This is due to one of the three cases in lemma 4.18.1 i. e. there is an occurrence of aY where $\mathrm{first}(Y) = b$ or Yb where $\mathrm{last}(Y) = a$ or YZ where $\mathrm{last}(Y) = a, \mathrm{first}(Z) = b$. After applying left-pop b and right-pop a this occurrence is surely uncrossed. Since $a \notin \Delta_r$ and $b \notin \Delta_l$ we will never right-pop b or left-pop a. Thus only way the occurrence ab became crossed again is that, at some point, we right-pop a or left-pop b. However, the condition is that $\mathrm{first}(X) = b$ or $\mathrm{last}(X) = a$ which is not possible since a occurs before b. ■

Algorithm 15: POP: Right-pop Δ_l and left-pop Δ_r.

```
   input : Δ_l, Δ_r, P_1, ... P_i;
 1 foreach X ∈ N \ {S_t, S_p} in topological order do
 2     let X ← α ∈ P and α[1] = b;
 3     if b ∈ Δ_r then
 4         remove leading b from α;
 5         replace X in productions of G by bX;
 6         if we replaced cX by cbX with c ≠ b and cb is contains no new fresh
           letter then
 7             let i be the index of cb;
 8             append (c, b, false, x*) to P_i, where x* points to c;
 9         let a be the last letter of α;
10         if a ∈ Δ_l then
11             remove ending a from α;
12             replace X in productions of G by Xa;
13             if we replaced Xc by Xac with c ≠ b and ac is contains no new fresh
               letter then
14                 let i be the index of ac;
15                 append (a, c, false, x*) to P_i, where x* points to a;
16     if α = ε then
17         remove X from t or p;
18 return Pairs;
```

Lemma 4.20.5. *Algorithm 15 implements the left-pop of Δ_r and the right-pop of Δ_l in $\mathcal{O}(n+m)$. It introduces at most $4(n+m)$ fresh letters to the grammars. Furthermore, it returns a list of pointers pointing to all uncrossed pairs.*

Remark 4.21. If we consecutively apply POP($\Delta_{l,1}, \Delta_{r,1}$) and POP($\Delta_{l,2}, \Delta_{r,2}$) without compressing blocks in $\Delta_{l,1}\Delta_{r,1}$ in between, we may create a crossing block $ab \in \Delta_{l,1}\Delta_{r,1}$ during the second POP. Therefore, we have to compress blocks in $\Delta_{l,1}\Delta_{r,1}$ right after calling POP($\Delta_{l,1}, \Delta_{r,1}$).

Naively, we could apply POP($\{a\}, \{b\}$) for each crossing pair ab separately. However, this would result in a running time of $\mathcal{O}((n+m)^2)$ since there are $\mathcal{O}(n+m)$ crossing pairs in G. In [34] two strategies are presented:

- Partition Δ into $\mathcal{O}(\log(n+m))$ partitions such that after applying POP and COMPRESSPAIRS $\mathcal{O}(\log(n+m))$ times, **all** crossing pairs are compressed.
- Partition Δ into $\mathcal{O}(1)$ partitions, containing those crossing pairs with high occurrence such that after applying POP and COMPRESSPAIRS $\mathcal{O}(1)$ times, **enough** crossing pairs are compressed.

Compression of all crossing pairs

After each phase of the recompression $\Delta = [|\Delta|]_0$ due to the renaming. For $a \neq b, a, b \in \Delta$ their binary representation differ at some position k where $k \in [1; \log(\lceil|\Delta|\rceil)$. Let us build for $i = 1, \ldots, \lceil\log(\Delta)\rceil$ partitions

- $\Delta_l^{2i-1} = \Delta_r^{2i}$ consisting of elements of Δ that have a 0 at the i-th position in the binary notation and
- $\Delta_r^{2i-1} = \Delta_l^{2i}$ consisting of elements of Δ that have a 1 at the i-th position in the binary notation [34].

Clearly

$$\Delta = \bigcup_{j=1}^{\lceil\log(|\Delta|)\rceil} \Delta_l^j = \bigcup_{j=1}^{\lceil\log(|\Delta|)\rceil} \Delta_r^j \ \wedge\ \forall j \in [\lceil\log(|\Delta|)\rceil] : \Delta_l^j \cap \Delta_r^j = \emptyset$$

holds. Furthermore, we can use standard bit operations to calculate the first position on which a and b differ in constant time and therefore the index j of their partition.

Lemma 4.21.1. *We can uncross all crossing pairs using $\mathcal{O}(\log(|\Delta|))$ consecutive calls of* POP *and* COMPRESSPAIRS *by partition the alphabet Δ of a phase into $\log(|\Delta|)$ different partitions (see alg, 16).*

Proof. The statement follows directly from lemma 4.20.2, 4.20.4 and 4.20.5 and the correctness of the construction of the partitions Δ_l, Δ_r. ■

Lemma 4.21.2. *We can compress all pairs by using* GETPAIRS *(alg. 13),* COMPRESSPAIRS *(alg. 14) and* COMPRESSALLCROSSINGPAIRS *(algorithm 16) by calling*

$$\text{COMPRESSPAIRS}(\text{GETPAIRS}(G), true).$$

The running time of algorithm is in $\mathcal{O}((n+m)\cdot\log(n+m) + |G|)$.

Algorithm 16: CompressAllCrossingPairs: Compresses all crossing pairs.

```
   input : Pairs crossing pairs;
1  partition Pairs into groups P_1, P_2, ..., P_i with i ∈ [⌈log(Δ)⌉];
2  for j ← 1 ... 2i do
3      Pop(Δ_l^j, Δ_r^j, P_1, ..., P_i);
4      CompressPairs(P_j, false);
```

Proof. Note that we compress crossing pairs after non-compressing pairs. By lemma 4.20.2 the compression of non-crossing pairs takes $\mathcal{O}(|G|)$.

After that, $Pairs$ contains an entry for each explicit occurrence of a crossing pair ab. We sort entries $(a, b, crossing, x^*)$ in $Pairs$ by using RadixSort with respect to the first three components, where $crossing = 1$ if x^* points to the occurrence of an explicit pair, otherwise, $crossing$ is equal to 0. Note that after the compression of non-crossing pairs no more non-crossing pairs are contained in $Pairs$. Furthermore, $Pairs$ contains exactly one entry for each crossing pair that occurs **only** crossing.

CompressCrossingPairs partitions $Pairs$ into $P_1, \ldots, P_i$. If we uncross any pair ab during Pop(P_j) and both, a and b are letters from the last phase, we put ab into the partition defined by the binary representation of a and b (see line 8 and 15). We can do this in $\mathcal{O}(1)$ time since we have pointers to the pair and we can compute the correct partition number in $\mathcal{O}(1)$ time. Note that by our renaming schema and by the definition of a JezSymbol we can test whether a letter is a letter introduced during the current phase $\mathcal{O}(1)$ time.

Since we add pairs to P_j during Pop(P_i) where i has not to be equals j, P_j becomes unsorted. Note that $i \geq j$ since all other pairs are already uncrossed and compressed. Therefore, we have to sort P_j before the compression starts. By the same arguments stated in lemma 4.20.2 both, the sorting and the compression, can be done in $\mathcal{O}(|P_j| + n + m)$ time. So the overall running time of the compression of crossed pairs is

$$\sum_{j=1}^{2\lceil\log(|\Delta|)\rceil} c\left(|P_j| + n + m\right) = 2c\left(n+m\right)\lceil\log(|\Delta|)\rceil + c\sum_{j=1}^{2\lceil\log(|\Delta|)\rceil}|P_j|$$
$$\overset{*}{=} \mathcal{O}((n+m)\log(n+m)) + c\sum_{j=1}^{2\lceil\log(|\Delta|)\rceil}|P_j|,$$

(*) follows from lemma 4.26.3. We now have to bound the size of P_j. Note that it may even happen that we have two entries in P_j pointing to the same pair occurrence.

There are at most $\mathcal{O}(|G|)$ explicit occurrences of crossing pairs and $4(n+m)$ crossing occurrences of crossing pairs before the loop in alg. 16 (each non-terminal can only be

part of at most 4 crossing pairs and at each position of a right-hand side there can only begin one explicit pair). Therefore, we get

$$\sum_{j=1}^{2\lceil \log(|\Delta|)\rceil} |P_j| \leq 4(n+m) + |G|$$

before the loop. However, each POP can introduce new explicit pairs, but at most 2 for each non-terminal in G. Therefore, we can bound the sum of all partitions by

$$\sum_{j=1}^{2\lceil \log(|\Delta|)\rceil} |P_j| \leq 4(n+m) + |G| + 2\lceil \log(|\Delta|)\rceil \overset{*}{=} \mathcal{O}(|G| + (n+m)\log(n+m)),$$

(*) follows from lemma 4.26.3. Therefore, the total running time is in $\mathcal{O}((n+m)\log(n+m)+|G|)$. Furthermore, overall $\mathcal{O}(\log(n+m))$ pairs were introduced into a production since for each POP, $\mathcal{O}(1)$ pairs were introduced into a production. ■

Compression of enough crossing pairs

Suppose we divide Δ into Δ_l, Δ_r randomly, where each letter a in Δ is assigned with probability $1/2$ to Δ_l or Δ_r. Then, for a fixed pair ab, $ab \in \Delta_l\Delta_r \cup \Delta_r\Delta_l$ with probability of $1/2$. In [34] a deterministic construction of a partition Δ_l, Δ_r, where $\Delta_l\Delta_r \cup \Delta_r\Delta_l$ covers at least half of the occurrences of crossing pairs, is presented by using a simple derandomisation approach. Since we construct a partition Δ_l, Δ_r such that $\Delta_l\Delta_r \cup \Delta_r\Delta_l$ covers one half of all occurrences of crossing pairs, we get by the pigeon-hole principle that $\Delta_l\Delta_r$ or $\Delta_r\Delta_l$ covers $1/4$ of all occurrences of crossing pairs. To construct such a partition we count the number of occurrences of a crossing pair ab in ω_{G_t} and ω_{G_p} and assign a or b to Δ_l and Δ_r respectively or vice versa, based on the number of occurrences such that at least $1/2$ of all crossing pairs are covered by $\Delta_l\Delta_r \cup \Delta_r\Delta_l$. Since there may exists exponential many occurrences of a fixed pair ab in the words generated by G, it is necessary, for complexity reasons, to manipulate integral numbers in $[|\omega_{G_t}| + |\omega_{G_p}|]_0$, by using basic operations, in constant time. Therefore, $|\omega_{G_t}| + |\omega_{G_p}|$ should fit in $\mathcal{O}(1)$ machine word. Note that the number of occurrences of a pair is in $\mathcal{O}(|\omega_{G_t}| + |\omega_{G_p}|)$.

Definition 4.22 (Occurrence of non-terminals)**.** Let $G = (\Delta, S, N, P)$ be a SLP. We call the occurrence $\mathrm{occ}(X)$ of a non-terminal $X \in N$ the amount of subwords $s = \mathrm{val}(X)$ it generates in $\mathrm{val}(S)$.

Lemma 4.22.1. *Let $G = (\Delta, S, N, P)$ be a SLP in weak Chomsky normal form. We can compute the mapping $\mathrm{occ} : N \to \mathbb{N}$ in $\mathcal{O}(|G|)$.*

Proof. First of all, we go over all productions q in P. We create a linked list L_X for each non-terminal in N. Whenever we spot a non-terminal X on the right-hand side of q, we add $\mathrm{lhs}(q)$ to the list of X. Clearly, this gives us for each non-terminal $X \in N$ a list of non-terminals that has a production in which X occurs. Note that there maybe

duplicated non-terminals in the list e. g. if X occurs in the rule of Y multiple times. This construction takes $\mathcal{O}(|G|)$ time. We can define $\mathrm{occ}(X)$ inductively by

$$\mathrm{occ}(X) := \sum_{X_i \in N} \mathrm{occ}(X_i) \cdot \text{ number of times } X \text{ occurs in the } q \in P \text{ with } \mathrm{lhs}(q) = X_i.$$

Therefore, we compute $\mathrm{occ}(X)$ by going over all $X \in N$ in descending topological order. We start by $\mathrm{occ}(S_t) = \mathrm{occ}(S_p) = 1$. Suppose now we have calculated $\mathrm{occ}(X_i)$ for all $i \in [n]$ in descending topological order i. e. $\forall X_i, i \in [n]$ with $X_1 \succeq \ldots \succeq X_n$. Suppose we compute $\mathrm{occ}(X_{n+1})$ with $X_n \succeq X_{n+1}$. None of the subwords $\mathrm{val}(X_i)$ occurs in $\mathrm{val}(X_{n+1})$. Therefore, we can compute $\mathrm{occ}(X_{n+1})$ since all values $\mathrm{occ}(X_i)$ for $i < n+1$ are already defined. With the constructed list $L_{X_{n+1}}$, this can be done in constant time since the grammar is in weak Chomsky normal form and therefore, L_X contains at most 2 elements. ■

Remark 4.23. During the recompression we may delete non-terminals from the grammar, since they become nullable (see alg. 15). This does not effect the computation of $\mathrm{occ}(X)$, since we will never require $\mathrm{occ}(X)$ after deleting X. Additionally, if we delete X, all X_i, that generate a sub word in $\mathrm{val}(X)$, will be deleted as well.

Lemma 4.23.1. *We can compute the number of occurrences of all crossing pairs in $\mathcal{O}(|G|)$ time.*

Proof. By lemma 4.22.1 we can compute $\mathrm{occ}(X)$ $\forall X \in N$ in $\mathcal{O}(|G|)$ time. We have already shown that we can compute a list of pointers $Pairs$ that contains for each occurrence of a crossing pair ab an entry. At the time we define this entry, we can access the left-hand side X of the production in which the pair of the entry occurs. Therefore, we can add the additional information $\mathrm{occ}(X)$ to each entry in the list without changing the running time. Note that $\mathrm{occ}(X)$ of an entry is equal to the number of occurrences in $\mathrm{val}(S_t)$ ($\mathrm{val}(S_p)$) of this pair ab occurring in G. The computation and the sorting of this list (according to the pairs) can be done in $\mathcal{O}(|G|)$ time. In the next step we go over the whole list and sum up the number of occurrences occ_{ab} of a crossing pair ab in $\mathcal{O}(|G|)$ time. ■

Lemma 4.23.2. *Algorithm 17 computes a partition Δ_l, Δ_r of Δ such that the occurrence of at least $1/4$ of all crossing pairs in ω_{G_t} and ω_{G_p} is covered by $\Delta_l\Delta_r$.*

Proof. We prove the following. After each iteration step $\Delta_l\Delta_r \cup \Delta_r\Delta_l$ covers $1/2$ of $(\Delta_l \cup \Delta_r)^2$. At the end of the algorithm $\Delta_l \cup \Delta_r = L$ and each crossing pair is covered by L^2. By the pigeon-hole principle $\Delta_l\Delta_r$ or $\Delta_r\Delta_l$ covers at least $1/4$ of the pairs.

Let n be the number of the iteration. For $n = 0$ the statement holds since $\Delta_l = \Delta_r = \emptyset$. Let's assume the statement hold for n^{th} iteration. Therefore, $\Delta_l\Delta_r \cup \Delta_r\Delta_l$ covers $1/2$ of $(\Delta_l \cup \Delta_r)^2$ at the beginning of iteration $n+1$. We get

$$\begin{aligned}(\Delta_l \cup \Delta_r \cup \{a\})^2 &= (\Delta_l \cup \Delta_r)^2 \cup (\Delta_r \times \{a\}) \cup (\Delta_l \times \{a\}) \cup (\{a\} \times \Delta_r) \cup (\{a\} \times \Delta_l) \\ &= (\Delta_l \cup \Delta_r)^2 \cup T_l \cup T_r \text{ with } T_l \cap T_r = \emptyset \ \wedge \ (\Delta_l \cup \Delta_r)^2 \cap (T_l \cup T_r) = \emptyset\end{aligned}$$

Algorithm 17: GREEDYPAIRS: Computes a partition of crossing pairs that covers at least $1/4$ of all occurrences of all crossing pairs.

input : $Pairs$;
output: Δ_l, Δ_r;
1 $L \leftarrow$ set of letters that are components of a crossing pair in $Pairs$;
2 $\Delta_l \leftarrow \Delta_r \leftarrow \emptyset$;
3 **for** $a \in L$ **do**
4 Let $T_{all} \leftarrow (\Delta_l \cup \Delta_r \cup \{a\})^2$;
5 Let $T_l \leftarrow (\{a\} \times \Delta_r) \cup (\Delta_r \times \{a\})$;
6 Let $T_r \leftarrow (\Delta_l \times \{a\}) \cup (\{a\} \times \Delta_l)$;
7 **if** T_l covers more pairs from T_{all} **then**
8 $\Delta_l \leftarrow \Delta_l \cup \{a\}$;
9 **else**
10 $\Delta_r \leftarrow \Delta_r \cup \{a\}$;
11 **if** $(\Delta_l \Delta_r)$ covers more crossing pairs than $\Delta_r \Delta_l$ **then**
12 **return** (Δ_l, Δ_r);
13 **else**
14 **return** (Δ_r, Δ_l);

By induction hypothesis we already cover at least $1/2$ of all occurrences of pairs in $(\Delta_l \cup \Delta_r)^2$. We have to cover $1/2$ of the pairs in $T_l \cup T_r$. Algorithm 17 picks the set that covers the most pairs (see line 7). Since their are only two of them it follows (by the pigeon-hole principle), that we cover at least $1/2$ of the pairs in $(\Delta_l \cup \Delta_r \cup \{a\})^2$. ■

Algorithm 17 gives the idea how the partition is created but we require an effective implementation. Our implementation is similar to the described implementation in [34]. Instead of deciding which set covers more pairs we decide which choice destroys more covers, then we choose the better one. The result is identical.

We create a table $right$ such that $right(a) = \{(b, k_{ab}) \mid ab \text{ is a crossing pair}\}$ and $left(a) = \{(b, k_{ba}) \mid ba \text{ is a crossing pair}\}$, where k_{ab} is the number of occurrence of ab. Note that we ensure that $\Delta = [|\Delta|]$ and therefore, we can use a simple array of size $|\Delta|$ to realize the tables. Furthermore, we can orginize Δ_l and Δ_r as BitSets. By lemma 4.23.1 we can compute all tuples in $\mathcal{O}(|G|)$ time and therefore, we can construct the tables in $\mathcal{O}(|G|)$ time. Another ingredient is the array $count_l$ and $count_r$. $count_l[a]$ gives us the number of occurrences of pairs we do not cover if we add a to Δ_l. The meaning of $count_r$ is similar. At the beginning of the algorithm $\forall a \in \Delta count_l[a] = count_r[a] = 0$.

Suppose at some iteration of algorithm 17 we have to deal with a. We replace line 7 in alg. 17 by a comparison of $count_l[a]$ and $count_r[a]$. We assign a to Δ_l if $count_l[a] \leq count_r[a]$ otherwise we assign a to Δ_r. Suppose we add a to Δ_l. If we look at an arbitrary letter b $count_r[b]$ does not change since if we add b to Δ_r we would cover ab

and ba. However, $count_l[b]$ changes, since a will for sure not be contained in Δ_r. If we assign b to Δ_l we 'lose' all pairs ab and ba. Therefore, we increase $count_l[b]$ by $k_{ab} + k_{ba}$. We can do this by using the tables. While we update $count_l[b]$ we can delete all table entries containing tubles (x, k_{ab}) or (x, k_{ba}) since we will never require them again. We can access one table entry in $\mathcal{O}(1)$ time and there are no more than $\mathcal{O}(|G|)$ table entries, since there are no more crossing pairs. Since we use BitSets we can assume that we can check $a \in \Delta_l$ and we can compute $\Delta_l \cup \{a\}$ in $\mathcal{O}(1)$ time. Line 11 in algorithm 7 requires $\mathcal{O}(|G|)$ time. We just go over all the number of all occurrences of all pairs ab and sum up all k_{ab} with $ab \in \Delta_l\Delta_r$ and k_{ab} $ab \in \Delta_r\Delta_l$. Therefore, we require that $|\omega_{G_t}| + |\omega_{G_p}|$ fits in $\mathcal{O}(1)$ machine word.

Lemma 4.23.3. *In $\mathcal{O}(|G| + n + m)$ time we can find a partition of Δ into Δ_l, Δ_r such that $1/4$ of all occurrences of crossing pairs in G_t and G_p are covered by this partition.*

Proof. We refer to the construction above and to [34]. ■

Algorithm 18: CompressGreedyCrossingPairs: Compresses enough crossing pairs.

input : $Pairs$ crossing pairs;
1 $(\Delta_l, \Delta_r) \leftarrow$ GreedyPairs($Pairs$) ; `// considering occurrences in` $\omega_{G_t}, \omega_{G_p}$
2 $(\Delta'_l, \Delta'_r) \leftarrow$ GreedyPairs($Pairs$) ; `// considering occurrences in` G
3 Pop($\Delta_l, \Delta_r, Pairs$);
4 CompressPairs($P_j, false$);
5 Pop($\Delta'_l, \Delta'_r, Pairs$);
6 CompressPairs($P_j, false$);

Lemma 4.23.4 ([34])**.** *By using the modified version of compressing pairs, the recompression algorithm keeps the size of the grammars $\mathcal{O}(n+m)$ and has still only $\mathcal{O}(\log(M))$ phases, where $M = \min\{|\omega_{G_t}|, |\omega_{G_p}|\}$*

Proof. The recompression algorithm works if we replace CompressAllCrossingPairs by CompressGreedyCrossingPairs additionally, we have to adapt the way we add pair entries to different lists. Instead of choosing a specific partition (see line 8 and 15 in alg. 15) we just add each entry to the same list i. e. to $Pairs$.

Let's prove the the upper bound of the number of phases. Let us look at a pair ab. If $a = b$ the pair will be compressed, since we do not change the block compression. If $a \neq b$ and ab is not crossing we can use the analysis of theorem 4.14. So let's assume $a \neq b$ and ab is a crossing pair. If we look at all occurrences of all crossing pairs, we guarantee by lemma 4.23.2 that $1/4$ of these occurrences are compressed. By using the same argument that we use in theorem 4.14, $1/8$ of letters of these occurrences crossing pairs are compressed and consequently, the words shorten by at least $1/16$ i. e. by a constant factor. It follows that the recompression algorithm, still has only $\mathcal{O}(\log(M))$ phases, however with a large constant factor. ■

Remark 4.24. We showed that we can find a partition Σ_l, Σ_r and still shorten the fully decompressed words by a constant factor. However, this does not mean that we also shorten G enough and we want to keep the size of the grammar as small as possible (see lemma 4.26.3). Therefore, we construct Σ'_l, Σ'_r in the same way but based on occurrences of pairs in G i. e. all $k_i = 1$. After compressing pairs $\Sigma_l\Sigma_r$, we compress $\Sigma'_l\Sigma'_r$ in addition (see algorithm 18).

4.1.5. Block compression

In case of pairs we have to deal with the fact that if we left-pop a, we may create a new crossing pair ab.

Example 4.25. Let $G = (\Delta, N, S, P)$ with $P = \{S \to cXc, X \to abb\}$. If we left-pop a we get $P' = \{S \to caXc \to bb\}$, aX occurs in S and $\mathrm{val}(X)[1] = b$, thus we create a new crossing pair ab.

This changes if we uncross blocks. Instead of popping a single letter we pop a maximal a-prefix or b-suffix respectively. Suppose a^lX occurs in a production and $\mathrm{val}(X)$ starts with a^r i. e. there is a crossing a-block. If we left-pop a^r by changing a^lX to a^la^rX and removing a^r from $\mathrm{val}(X)$ it is impossible to create a new crossing block. Maybe the block is still crossing since there could be productions $Y \to a^la^rX$ and $Z \to \alpha a^kY$ but if we successively pop the maximal prefix and suffix from all non-terminals (except for the axioms) in topological order, we can ensure that there are no more crossing blocks.

Remark 4.26. A crossing block can be of exponential size and therefore, we cannot replace a^lX by a^la^rX explicitly. Instead we create a single JezSymbols for a^l and a^r such that $a^r = (id_a, phase_a, r, weight_a)$. After UncrossBlocks we replace each maximal sequences $a^{l_1}a^{l_2}\dots a^{l_k}$ by a fresh JezSymbol

$$a_d = (id_a, phase_a + 1, 1, weight_a \cdot d),$$

with

$$d = \sum_{i=1}^{k} l_i.$$

Lemma 4.26.1. *After applying algorithm 19, there are no more crossing blocks in G. Furthermore, the algorithm runs in $\mathcal{O}(|G|)$ time.*

Proof. The runtime is easy to see. Computing the length of the a-prefix (suffix) requires to read at most the whole production. Insertion of a_l can be done in $\mathcal{O}(1)$, since we have pointers to each non-terminal. Since we go over each production only once this can be done in $\mathcal{O}(|G|)$ time.

After popping the prefix and suffix from a production q with $\mathrm{lhs}(q) = X$, X is not part of a crossing block, since we replace any occurrence $\alpha_1X\alpha_2$ by $\alpha_1a^lXb^l\alpha_2$ and

Algorithm 19: UNCROSSBLOCKS: Uncrosses all crossing blocks.

```
foreach X ∈ N \ {S_t, S_p} in topological order do
    let X ← α ∈ P and α[1] = a and a is not a fresh letter;
    remove a-prefix a^l from α;
    replace X in productions of G by a^l X;
    if α ≠ ϵ then
        let b be the last letter of α and b is not a fresh letter;
        remove b-suffix b^r from α;
        replace X in productions of G by X a^r;
    if α = ϵ then
        remove X from G;
```

$\text{first}(X)[1] \neq a \wedge \text{last}(X) \neq b$ (and we do this in topological order for each non-terminal). Since we progress in topological order it is not possible that we pop a's or b's into X and therefore, $\text{first}(X)[1] \neq a$ and $\text{last}(X) \neq b$ holds for the for remaining iterations. ■

Lemma 4.26.2. *Under the assumption that $|\omega_{G_t}|$ fits in $\mathcal{O}(1)$ machine word, algorithm 21 compresses all blocks in G. The compression is performed in $\mathcal{O}(|G| + (n+m)\log(n+m))$.*

Proof. We can merge all sequences $a^{l_1} \dots a^{l_k}$ into $a^{l_1+\dots+l_k}$ by going over the whole grammar, since $|\omega_{G_t}|$ fits in $\mathcal{O}(1)$ machine word we can do this in $\mathcal{O}(|G|)$ time. While doing this we can create a pointer to the merged entry. Additionally, we can decide if the block was generated by more than one non-terminal e. g. there was a rule $X \to aaaX_1bbb$ and a rule $X_1 \to aaaa\alpha$. Only these blocks can be exponential in the size of $(n+m)$. Therefore, we treat these blocks separately and put them in another list (see alg. 20). We can sort the list of short blocks in $\mathcal{O}(|G| + n + m)$ time using RADIXSORT, since there are no more than $\mathcal{O}(|G|)$ blocks and $|\Delta| \in \mathcal{O}((n+m)^3)$ (see lemma 4.26.3). We can sort all other blocks using any modern sorting technique. Since there are at most $n+m$ non-terminals there are $\mathcal{O}(n+m)$ large blocks. We can sort the list of large blocks B_l in $\mathcal{O}((n+m)\log(n+m))$ time. We can join the two sorted lists in $\mathcal{O}(|G| + (n+m))$ time together. Each compression of an occurrence of a block takes $\mathcal{O}(1)$ and therefore, we can compress all blocks in the merged list in $\mathcal{O}(|G| + n + m)$ time. Thus the total running time is in $\mathcal{O}(|G| + (n+m)\log(n+m))$ ■

4.1.6. Alphabet and grammar size

In the last section we often showed that the running time is in $\mathcal{O}(|G|)$. But G changes during the algorithm and therefore we have to bound $|G|$. Note that even if ω_{G_t} and ω_{G_p} shorten, the size of G may increase. Furthermore, we have to ensure that Δ does not become too large, since we want to use RADIXSORT for sorting the letters in Δ and therefore, Δ has to be bounded by $\mathcal{O}((n+m)^c)$, where c is a constant.

Algorithm 20: GETBLOCKS: Computes a list of pointers to all proper non-crossing blocks.

input : G;
output: Two lists of pointers, pointing to proper blocks;

initialize list B_l;
initialize list B_s;
foreach $q \in (P_t \cup P_p)$ **do**
  $r \leftarrow \mathrm{rhs}(q)$;
  for $i = 1 \ldots |\mathrm{rhs}(q) - 1|$ **do**
    if there is a sequence $a^{l_1} \ldots a^{l_k}, k > 1, \exists l_j > 1$ at position i **then**
      let x^* be the pointer, pointing to a^{l_1};
      append $(a, \sum_{i=1}^{k} l_i, x^*)$ to B_l;
    else if $r[i] = a^l, l > 1$ **then**
      let x^* be the pointer, pointing to a^l;
      append (a, l, x^*) to B_l;
    else if there is a sequence $a \ldots a$ of size $s > 1$ at position i **then**
      let x^* be the pointer, pointing to the first a of the sequence;
      append (a, s, x^*) to B_s;
return (B_l, B_s);

Lemma 4.26.3. *The following bounds hold for different pair compression strategies:*

Algorithm	$\lvert G\rvert$	$\lvert\Delta\rvert$
*Compress **all** pairs (alg. 16)*	$\mathcal{O}((n+m)\log(n+m))$	$\mathcal{O}((n+m)\log(n+m))$
*Compress **enough** pairs (alg. 18)*	$\mathcal{O}(n+m)$	$\mathcal{O}(n+m)$

Proof. The argument of theorem 4.14, that the length of the uncompressed word shorten by a constant factor, holds also for the case of compressed words if we use COMPRESSALLCROSSINGPAIRS to compress all crossing pairs. This is easy to see since each compressed pair may replace several pairs in ω_{G_t} or ω_{G_p}. If we use COMPRESSGREEDYCROSSINGPAIRS we refer to lemma 4.23.4. So we can assume that there are only constant numbers of phases with respect to $\omega_{G_t}, \omega_{G_p}$ and therefore $\mathcal{O}(\log(\min\{|\omega_{G_t}|, |\omega_{G_p}|\}))$ number of phases. A production p in P increases due to compression of crossing pairs by at most $\mathcal{O}(\log(n+m))$ and due to block compression by at most 4.

It follows that if we use COMPRESSALLCROSSINGPAIRS we increase the size of G by $\mathcal{O}((n+m)\log(n+m))$ in total. If we use COMPRESSGREEDYCROSSINGPAIRS instead we would increase the size of G by $\mathcal{O}(n+m)$ in total, since there is only two calls of POP in each phase.

Let's assume we use COMPRESSALLCROSSINGPAIRS. First considering only the increase due to decompression. The size of G increase by $\mathcal{O}((n+m)\log(n+m))$ in total

Algorithm 21: CompressBlocks: Compresses blocks.

```
   input : G;
1  UncrossBlocks; (B_l, B_s) ← GetBlocks(G);
2  sort B_l with respect to the letter and the length, using QuickSort;
3  sort B_s with respect to the letter and the length, using RadixSort;
4  B ← (merge B_l and B_s);
5  k ← −1
6  foreach (a, l, x*) ∈ B do
7      if k = −1 ∨ k ≠ l then
8          a_l ← fresh letter;
9          k ← l;
10     replace any a sequence at x* by (a_l, phase_a + 1, 1, weight_a · l)
```

(during a phase), since there are $\mathcal{O}(\log(n+m))$ calls of Pop and $\mathcal{O}(n+m)$ productions in G. Second consider the decrease of G due to recompression. We can use the argument in theorem 4.14 to work also for productions of G. The explicit word of a production at the **beginning** of a phase, shorten by a constant factor. Note that we only look at 'old' symbols of the phase. Since the grammar is in weak CNF the sum of lengths of explicit words in all productions is $|G| - 2(n+m)$. There maybe 3 positions in $\mathrm{rhs}(X)$ without an successor in Δ. Therefore, a production p is shorten by

$$\frac{\mathrm{rhs}(p) - 3}{3}$$

and the whole grammar is shorten by

$$\frac{|G| - 2(n+m) - 3(n+m)}{3} = \frac{|G| - 5(n+m)}{3}.$$

Let G_i be the grammar after the i^{th} phase. If we consider the increase of the grammar we get

$$\begin{aligned} |G_{i+1}| &= |G_i| - \frac{|G_i| - 5(n+m)}{3} + c_1 \cdot (n+m)\log(n+m) \\ &= \frac{2}{3}|G_i| + 5(n+m) + 3c_1(n+m)\log(n+m) \\ &\leq \frac{2}{3}|G_i| + c_2(n+m)\log(n+m) \end{aligned}$$

with $r = c_2(n+m)\log(n+m)$ and $G_0 = G$ we get

$$\begin{aligned}
|G_i| &= \left(\frac{2}{3}\right)^i |G_0| + r + \left(\frac{2}{3}\right) r + \ldots + \left(\frac{2}{3}\right)^{i-1} r \\
&= \left(\frac{2}{3}\right)^i |G_0| + r \cdot \sum_{k=0}^{i-1} \left(\frac{2}{3}\right)^k \\
&= \left(\frac{2}{3}\right)^i |G_0| + r \cdot c_3 = \frac{2}{3}^i |G_0| + \mathcal{O}(r) \quad \text{with } c_3 < 3 \\
&= \left(\frac{2}{3}\right)^i |G_0| + \mathcal{O}((n+m)\log(n+m))
\end{aligned}$$

and therefore $|G|$ is in $\mathcal{O}((n+m)\log(n+m))$ for each phase. Note however that $c_2 \cdot c_3$ can be large.

Assume now we use CompressGreedyCrossingPairs for the compression of crossing pairs. Again we consider first the increase due to local decompression. Clearly, this time $|G|$ increases in each phase by at most $\mathcal{O}(n+m)$, since we have at most 2 calls of Pop to uncross pairs. The question is how much the size of G decreases due to recompression? The crucial observation is that $\Delta'_l\Delta'_r$ covers $1/4$ of occurrences of all pairs in G. Therefore, we compress $1/8$ of letters of those occurring pairs, thus G will be shorten by at least $1/16$ (using the analysis from theorem 4.14). We derive

$$\begin{aligned}
|G_{i+1}| &= |G_i| - \frac{|G_i|}{16} + \mathcal{O}(n+m) \\
&= \frac{15}{16}|G_i| + \mathcal{O}(n+m).
\end{aligned}$$

Using the same calculations as above gives us

$$|G_i| = \left(\frac{15}{16}\right)^i |G_0| + \mathcal{O}(n+m).$$

Note that the constant factor is even larger but we showed that the grammar size is in $\mathcal{O}(n+m)$ for each phase if we use CompressGreedyCrossingPairs.

We guarantee by the renaming scheme that $\Delta = [|\Delta|]_0$. We construct Δ in each phase by starting with 0 and increase the counter for $|\Delta|$ by 1 for each fresh letter. Each newly introduced letter is contained in some right-hand side of G. Uncompressed letters will be renamed by Rename (see line line 8 in algorithm 11). All fresh letters created by compression or by the renaming are in some left-hand side of G. Since there are no more then $|G|$ different letters in G it follows that $|\Delta| = \mathcal{O}(|G|)$. Note that this differs from [34], however we may introduce overall $\mathcal{O}(|G|\log(\min\{|\omega_{G_t}|, |\omega_{G_p}|\}))$ different pairs $(a,i) \in \Delta \times \mathbb{N}$, where i is the number of the phase. ■

Theorem 4.27 (FCEquals)**.** *Under the assumption that $|\omega_{G_t}|$ fits into $\mathcal{O}(1)$ machine word, equality testing using* FCEQUALS *requires $\mathcal{O}((|G|+(n+m)\log(n+m))\log(|\omega_t|)) = \mathcal{O}(((n+m)\log(n+m)) \cdot \log(\min\{|\omega_{G_t}|, |\omega_{G_p}|\})) = \mathcal{O}((n+m)^2 \log(n+m))$ using one of the pair compression techniques.*

Proof. Since ω_{G_t} and ω_{G_p} shorten by a constant factor the algorithm terminates after $\log(\min\{|\omega_{G_t}|, |\omega_{G_p}|\}) = \mathcal{O}(n+m)$ phases. We have the following running times for each phase (by using lemma 4.26.3):

- renaming: $\mathcal{O}(|G|)$
- compression of uncrossed pairs: $\mathcal{O}(|G|)$
- compression of crossing pairs: $\mathcal{O}(|G|)$
- block compression: $\mathcal{O}(|G| + (n+m)\log(n+m))$, the factor $\log(n+m)$ is required because we sort large blocks without using RADIXSORT

$\mathcal{O}(|G|) = \mathcal{O}((n+m)\log(n+m))$, if we compress all pairs or $\mathcal{O}(|G|) = \mathcal{O}((n+m))$, if we compress enough pairs. Therefore, we achieve a total running time of $\mathcal{O}((n+m)^2 \log(n+m))$. ■

Remark 4.28. There is no theoretical effect of replacing COMPRESSALLCROSSINGPAIRS by COMPRESSGREEDYCROSSINGPAIRS but it effects the practical running time for some instances i. e. if $\log(\max\{\omega_{G_t}, \omega_{G_p}\})$ fits in one machine word. Note that we require the condition, that $\mathcal{O}(|\omega_{G_t}|)$ fits in one machine word for the block compression of large blocks. We should note that the author of [34] introduced a technique to overcome this condition. The basic idea is to represent large block length by the sum of two number, a large and a small one. For future work, it should be easy to adapt our implementation in such a way that we can use this technique. By using this this technique we can get rid of the $\log(n+m)$ factor for sorting large blocks and could achieve a better theoretical running time.

4.2. Fully-compressed pattern matching

In this section we will shortly discuss the extension of the recompression algorithm to pattern matching. We use the pattern matching to test if two words share a common prefix or suffix. String pattern matching is one of the most fundamental operations on strings. Searching for particular pattern in DNA sequences is one of many exciting applications. In case of uncompressed words one can solve the problem by using the finite word automata, the Rabin-Karp algorithm or the Knuth-Morris-Pratt approach [12]. In our setting the pattern and the text are SLP-compressed words. Many researchers studied the problem of compressed pattern matching for various compression methods for decades (see table 4.1). Note that there is a connection between the SLP- and the LZ-compression shown by the following lemma.

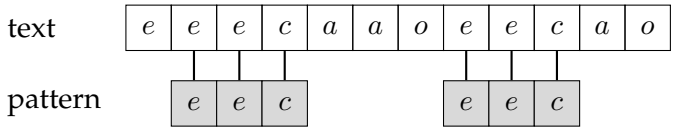


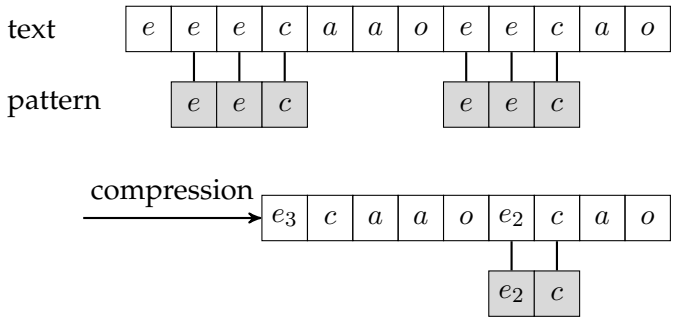
Figure 4.2.: An example of the string-pattern matching problem, where we want to find all occurrences of the pattern eec in the text $eeecaaoeecao$. The pattern occurs at position 2 and 8.

Lemma 4.28.1 ([10, 33, 49, 50]). *Let N be the size of a decompressed text and n be the size of his LZ-compression. The text can be converted into an equivalent* SLP *of size $\mathcal{O}(n \log(N/n))$ in $\mathcal{O}(n \log(N/n))$ time.*

Lemma 4.28.2 ([10, 49]). *Let G be the grammar of an* SLP*. G can be converted to an equivalent LZ of size $\mathcal{O}(|G|)$ in linear time.*

We will explain the main concepts of the fully-compressed patter matching presented in [34]. In general we can apply the same replacement scheme but we have to protect the pattern inside the text. We may destroy an occurrence of the pattern if we replace a sequence of letters that is only partly part of the pattern.

Example 4.29. In the following example we compress e-blocks. We replace the sequence eee by e_3. The compression destroys the first occurrence of the pattern.



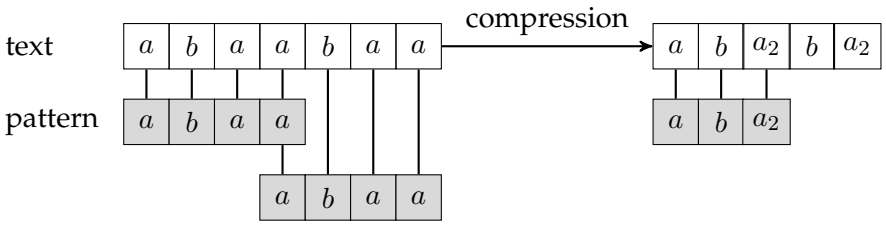
The problem in example 4.29 is that only the two last letters of the sequence eee are part of the pattern. We compress only parts of the pattern, consequently, we destroy

Compression method	Compression pattern matching algorithm
Run-length	[13]
Run-length (two dim.)	[5, 6, 7]
Lempel-Ziv (LZ) methods	[22, 24, 25, 23, 20, 21, 19]
SLPs	[47, 27, 3]

Table 4.1.: Compression methods and pattern matching algorithms concerning these methods.

this occurrence of the pattern. A even more complicated situation occurs if the pattern begins and ends with the same letter. In that case it could be that a block of letters is part of the beginning and the ending of an occurrence of the pattern (see example 4.30).

Example 4.30. In the following example we compress a-blocks. We replace the sequence aa by a_2. The compression destroys the second occurrence of the pattern.



To avoid the destruction of pattern occurrences we *fix the end and the beginning of the pattern* right before applying the compression of pairs and blocks. Observe that in case of pattern matching we do not care whether $\omega_{G_t} = \omega_{G_p}$ holds or not. We only have to protect the information about the occurrences of the pattern in the text. We have to distinguish between four sub cases [34]:

1. $\text{first}(S_p) \neq \text{last}(S_p)$
   (a) ω_{G_p} does not begin/end with a proper block
   (b) ω_{G_p} begins/ends with a proper block
2. $\text{first}(S_p) = \text{last}(S_p)$
   (a) ω_{G_p} does not begin/end with a proper block
   (b) ω_{G_p} begins/ends with a proper block

4.2.1. Fixing different ends

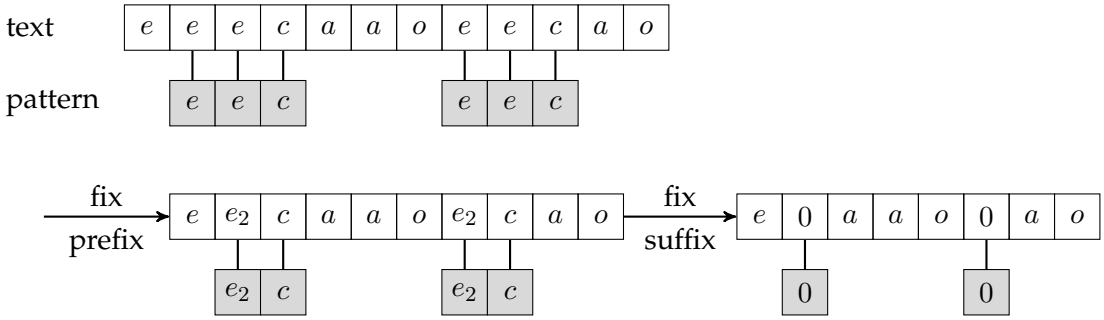
If the pattern does not end and begin with the same letter, i. e. $\text{first}(S_p) \neq \text{last}(S_p)$, we can fix the prefix and the suffix of the pattern separately.

Let's first assume the pattern does not begin with a proper block. In that case we just compress the pair $\omega_{G_p}[1]\omega_{G_p}[2]$ and we are done. Since we introduce a fresh letter, which will never be compressed during the phase (except if we fix the suffix), it is impossible that we compress a pair or a block in ω_{G_t} that is only partly part of the pattern. As already mentioned, fixing the ending of the pattern is symmetric i. e. instead of compressing the pair $\omega_{G_p}[1]\omega_{G_p}[2]$ we compress the pair $\omega_{G_p}[|\omega_{G_p}| - 1]\omega_{G_p}[|\omega_{G_p}|]$.

Let's assume the pattern begins with a proper block i. e. $\omega_{G_p} = a^l\alpha$ with $l > 1, \alpha[1] \neq a$. In that case we have to replace each maximal proper a-block of length $k = l + r$ with $r \geq 0$ in G_t and G_p by $a_r a_l$ (instead of a_k). Furthermore, we compress all other a-blocks

i. e. all a-blocks of length $k < l$. After the replacement, we can delete a_l at the end of G_t, since a pattern can not begin at this position. Again fixing the ending of the pattern is symmetric i. e. instead of replacing a^k by $a_r a_l$ we replace a^k by $a_l a_r$.

Clearly, blocks of length $< l$ cannot be part of the beginning of the pattern and therefore we can compress them without further adjustments. The result of example 4.29 after the fixing would be the following:



4.2.2. Fixing same ends

In that case we have to fix the prefix and the suffix of the pattern together. First of all we get rid of a special but easy case where $\omega_{G_p} = a^k$ for some $a \in \Delta, k > 0$. In this special case we just compress all a-blocks. We mark each fresh letter a_k as being a compression of an a-block (we require this information for the computation of the position of an occurrence of the pattern). After this step, the recompression terminates since $|\omega_{G_p}| = 1$.

Let's move on to the difficult cases. In example 4.30 we would destroy the first occurrence of the pattern if we fix the prefix and compress ab. Following the description in [34], we introduce two markers a_L and a_R that mark the possible beginning and ending of a pattern in ω_{G_t}. So let l be the length of the maximal block at $\omega_{G_p}[1]$ and r be the length of the maximal block at the end of the pattern, possibly $l = r = 1$. a_L represents the prefix-block of the pattern and a_R the suffix-block of the pattern. Since we are not interested in the position of the ending of the pattern we set $weight_{a_R} = 0$ and $weight_{a_L} = weight_a \cdot l$. Let's look at some sequence a^m in ω_{G_t}. If $m > r, l$ then an occurrence of the pattern may begin with the letters a^{m-l} and/or end with the letters a^{m-r}. Therefore, we replace a^m by $a_R a_{m-l} a_L$. If $m < l, r$ the pattern cannot begin/end with this sequence and therefore, we replace a^m by a_m. If $\min\{l, r\} \leq m \leq \max\{l, r\}$, the replacement rely on the relation between l and r:

- $l = r$: We replace each a^m by $a_R a_L$, since a pattern can start and end with $a^m = a^l = a^r$.

Algorithm 22: FCPMATCHING: Recompression of ω_{G_p} and ω_{G_t} without destroying any pattern occurrence.

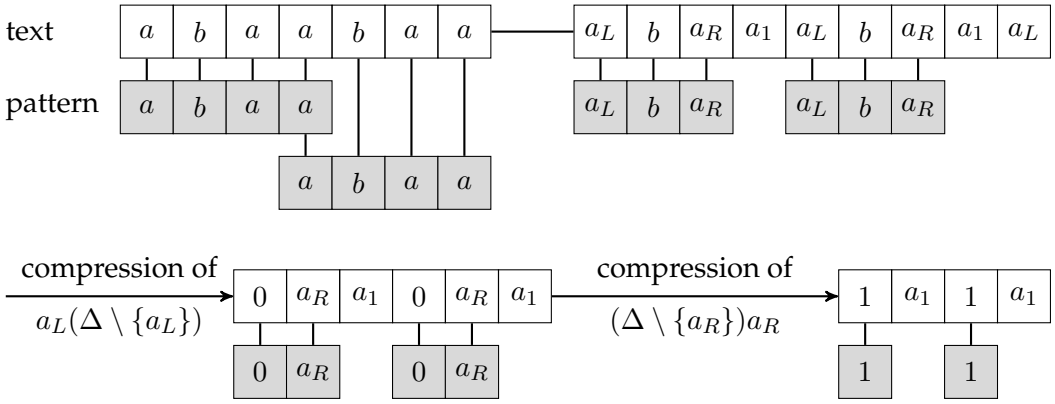
input : G with $L(G) = \{\omega_{G_t}, \omega_{G_p}\}$;
output: Recompressed grammars ω_{G_p} and ω_{G_t};

```
1 FIRSTRENAME(G);
2 fix the beginning and the end;
3 while |ω_{G_p}| > 1 do
4     COMPRESSBLOCKS(G);
5     Pairs ← GETPAIRS(G);
6     COMPRESSPAIRS(Pairs, true);
7     RENAME(G);
8 return G_t, G_p;
```

- $l < r$: We replace each a^l by a_L, each a^r by $a_R a_{r-l} a_L$ and each $a^m, l < m < r$ by $a_{m-l} a_L$. Clearly, since $l < r$ a pattern can not end with a subsequence of l a's. If $m = r$ the pattern could begin after $r - l$ a's. If $m < r$ we know that the pattern can not end with this sequence of a's.
- $l > r$: We replace each a^l by $a_R a_L$ and each $a^m, r \leq m < l$ by $a_R a_m$. If $m < l$ the pattern can not start with this sequence.

Remember that $weight_{a_R} = 0$ and $weight_{a_L} = weight_a \cdot l$ therefore, if we sum up the weights of the letter that replace any a^m, we get $weight_a \cdot m$. After these replacements, the pattern is well protected and we can start the phase. Note that a_R and a_L won't be compressed since they are fresh letters of the phase! Before starting the phase we can delete the occurrence of a_L at the end of ω_{G_t} since a pattern can not start there. The same is true for an occurrence of a_R at the $\omega[1]$. To reduce the size of the grammar and the fully-compressed words by a constant factors we additionally compress all pairs $a_L(\Delta \setminus \{a_L\})$ and $(\Delta \setminus \{a_R\})a_R$. We are in case (a) and therefore, we can do this without destroying any occurrence of the pattern in G_t.

In the last step we compress $a(\Delta \setminus \{a\})$ if $r = 1 < l$. At that time the pattern can not end with a. Suppose there is an occurrence of ab and the pattern starts at that position with b. Before we start the fixing process the pattern had a prefix a^l with $l \geq 1$. Therefore, there was a block aa^l at this position. By the replacement scheme, described above, we would had replaced this sequence by $a_R a_{l+1-1} a_L = a_R a_1 a_L$ (since $r < l < m$). But the letter a_1 is a fresh introduced letter, i. e. $a_1 \neq a$. This is somehow a special case since a_1 and a represents the same fully uncompressed word, but they are not equal. This leads to a contradiction and therefore the pattern can not cannot start with b. We can conclude that this last step can't destroy an occurrence of the pattern in G_t. The following shows the fixing same ends for example 4.30. Observe that $weight_0 = weight_1 = weight_{a_1} + weight_b = 2$, hence the second occurrence of the pattern starts at position $weight_1 + weight_{a_1} = 3$.



Lemma 4.30.1 ([34]). *Fixing different or same ends can be performed in* $\mathcal{O}(|G|+(n+m)\log(n+m))$ *time. It introduces* $\mathcal{O}(n+m)$ *new letters.*

All compressions of pairs or blocks can be done in $\mathcal{O}(n+m)$ since we only require constant number of calls of POP.

Lemma 4.30.2 ([34]). *A phase of* FCPMATCHING *shortens* G *and* ω_{G_p} *by a constant factor.*

Definition 4.31 (Weight of non-terminals)**.** Let $X \in N$ be non-terminal of G. we define the weight of $weight_X$ inductively by:

$$\frac{X \to \alpha \qquad alpha \in \Delta^*}{weight_X = \sum_{a \in \text{alph}(\alpha)} weight_a} \qquad \frac{X \to \alpha \qquad \alpha = \omega_0 X_1 \dots X_n \omega_n}{weight_X = \left(\sum_{i=0}^{n} \sum_{a \in \text{alph}(\omega_i)} weight_a \right) + \sum_{i=1}^{n} weight_{X_i}}$$

Since the recompression algorithm for pattern matching is equal to the equality testing, with the additional fixing before each phase, and we still shorten the shorten G and ω_{G_p} by constant factor we get the following theorem.

Theorem 4.32. *Under the assumption that* $|\omega_{G_t}|$ *fits in* $\mathcal{O}(1)$ *machine word after applying* FCPMATCHING *on* G_t *(text) and* G_p *(pattern), which requires* $\mathcal{O}((n+m)^2 \log(n+m))$ *time, we can ask for the position of the* k^{th} *occurrence of the pattern in* $\mathcal{O}(n+m)$ *time.*

Proof. The time bound for applying FCMATCHING is a result derived from the fact that we only add the additional fixing part to each phase in combination of lemma 4.30.1.

Since $\mathcal{O}(|\omega_{G_p}|)$ does fit in $\mathcal{O}(1)$ machine word we can calculate the weight $weight_a$ of a JezSymbol as suggested in definition 4.11 and for each non-terminal X using definition 4.31. For non-terminals we can do this in topological order in overall $\mathcal{O}(|G|)$ time and for JezSymbols we require no additional time. After the last phase there are two cases:

(a) $c = a_l$ is the result of a block compression of a letter a: We go over G_t and whenever we spot same a_k (for $k \geq l$) we can add $k-l+1$ to the number of occurrences of the pattern.

(b) $c = ab$ is the result of a pair compression: We go over G_t and whenever we spot c we increase the number of occurrences of the pattern by 1.

For each X we compute and store how many occurrences of pattern are contained in $\mathrm{val}(X)$. We can do this, as usual, in topological order using (a) or (b). If we ask for the k^{th} occurrence of ω_{G_p} in ω_{G_t} we read the derivation **path** to this explicit occurrence. For a non-terminal X on the path, we can decide whether we should search for the pattern in its right-hand side i. e. X is part of the path and the position of the occurrence is in $\mathrm{val}(X)$ or we should skip it. To compute the position of the occurrence, we add all weights of visited symbols together (except for non-terminals that are part of the path). Computing the weight for all non-terminals requires $\mathcal{O}(|G_t|)$ time. The traversal can be done in $\mathcal{O}(|G_t|)$ time. By lemma 4.30.1 in combination of lemma 4.26.3 $|G_t| = \mathcal{O}(m)$ and the statement follows. ■

4.3. Fully-compressed LCP and LCS

In [3], they extend the set of operations presented in [45] by the longest common prefix LCPREFIX and longest common suffix LCSUFFIX. They show that their data structure supports these operations in $\mathcal{O}(\log(n))$ time if the the strings were contained in the data structure, where n is the length of the involved strings. Unfortunately, the algorithm presented in [34] does not support these operations.

Example 4.33. Let $\omega_{G_t} = edcbhz, \omega_{G_p} = edcbaz$, i. e. the words differ only at position 5. The first level of the tree shows the word before the FCEQUALS starts. The second level shows the words after FIRSTRENAME and each following level shows the situation after a phase. After the first phase ω_{G_t} and ω_{G_p} differ at each position.

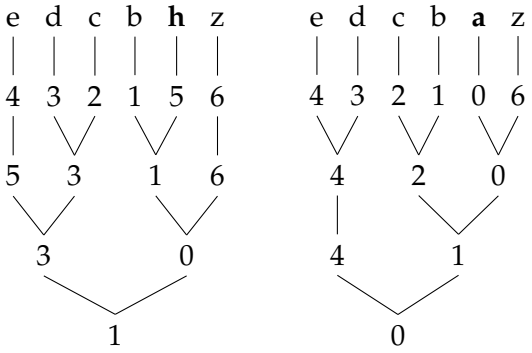


Figure 4.3.: The progress of FCEQUALS for $\omega_{G_t} = edcbhz$ and $\omega_{G_p} = edcbaz$. We assume that $a < b$ if a comes before b in lexicographical order.

The main problem is the missing phase of breaking words into blocks. If we break a word ω into n blocks such that $\omega = \omega_1 \dots \omega_n$ and we compress these blocks equally by

computing some signature σ we can be sure that if $\sigma(\omega_i) = \sigma(\omega_j)$ holds, $\omega_i = \omega_j$ follows. In case of the longest common prefix the challenge would be to find at each stage the block containing the first different letter (from left to right). Since all blocks left from this block are equally compressed, we have to decompress only one block for each phase of the compression (see [3]). The same principle works for CONCATENATE and SPLIT. In both cases we have to deal with blocks around the position of the concatenation or split, respectively.

However, the recompression algorithm treats the whole word as one single block. So even if two words differ only at one position, they may share not a single fresh introduced letter after a phase of the algorithm (see example 4.33). We had the same problem in the setting of pattern matching where we protect the pattern such that we never compress a sequence that is only partly part of the pattern. For example 4.33, this would mean that we have to avoid the compression of a sequence that is only partly part of the longest common prefix. Unfortunately, we already had to know $\mathrm{lcp}(\{\omega_{G_t}, \omega_{G_p}\})$ to do this. The only way to overcome this problem is to break the word into blocks before the compression starts which seems impossible since we somehow would have to go through the whole word and mark some letters that should not be compressed. Furthermore, we have to mark the exact some letters in the both fully-compressed words to compress them equally.

Remark 4.34. In each phase of recompression algorithm we know very little about the fully decompressed version of a letter. If $a = b$ it follows that the fully decompressed words ω_a and ω_b are equal. If $a \neq b$ and they have the same length, $\omega_a \neq \omega_b$ follows. But if $\omega_a \neq \omega_b$ and $|\omega_a| \neq |\omega_b|$ we can not compare them at all. ω_a could be a prefix of ω_b and vice versa.

4.4. The singleton set problem

We are now ready to solve the first problem that relies on the fully-compressed equality test presented in section 4.1.

Theorem 4.35 (The singleton set problem for CFGs)**.** *The following problem can solved in polynomial time.*

Input: *Given a* CFG G

Output: $|L(G)| = 1$*?*

Definition 4.36 (Singleton productions and non-terminals)**.** Let $G = (\Delta, N, S, P)$ be a reduced context-free grammar in weak Chomsky normal form without ϵ-productions (except for the axiom). We call a production $p \in P$ a *singleton production* if for all derivations $\mathrm{lhs}(p) \Rightarrow \mathrm{rhs}(p) \Rightarrow^* \alpha$ we generate the same word in Δ^*. We call a non-terminal $X \in N$ a *singleton non-terminal* of G if for all derivations starting from X we generate

the same word in Δ^*. We can define the set of all singleton non-terminals $Singleton_G$ inductively:

$$\frac{\forall q \in \{p \in P \mid \text{lhs}(p) = X\} : q = X \to \omega \qquad \omega \in \Delta^*}{X \in Singleton_G}$$

$$\frac{\forall p \in P : (\text{lhs}(p) = X, \forall Y \in \text{rhs}(p) : Y \in Singleton_G)}{X \in Singleton_G}$$

Theorem 4.37. *Let $G = (\Delta, N, S, P)$ be a context-free grammar and $n = |G|$. Algorithm 23 tests in $\mathcal{O}(n^3 \log(n))$ time if $L(G)$ is a singleton set.*

Proof (sketch). The condition that G is in weak Chomsky normal form is important. Otherwise, the definition above does not work. To simplify the computation it is easier to eliminate ϵ-productions beforehand. From the inductive definition of $Singleton_G$ we can derive an algorithm for solving the decision problem. If $L(G)$ is a singleton set it has to be acyclic (since we eliminate ϵ-productions), otherwise we could easily create infinite many words. Therefore, $\preceq$ is defined on G. To check whether $L(G)$ is a singleton set we first check for all non-terminals X in

$$\{Y \in N \mid \forall p \in P : (\text{lhs}(p) = Y \Rightarrow \text{rhs}(p) \in \Delta^*)\},$$

if

$$|\{\text{rhs}(q) \mid q \in P \ \wedge \ \text{lhs}(q) = X\}| = 1$$

holds. If this is not the case, $L(G)$ is not a singleton set. Otherwise we can add all these non-terminal to $Singleton_G$. Suppose we have a non-terminal X with some productions in G. Let's assume two of them are $p = X \to X_1 \dots X_k$ and $q = X \to X'_1 \dots X'_{k'}$ and $\forall i \in [k] : X_i \in Singleton_G$ and $\forall j \in [k'] : X_j \in Singleton_G$. We check if the word derived by starting from p is equals to the word derived by starting from q. Since all non-terminals at the right-hand sides are singletons, we derive exactly one word ω_q from q and one word ω_p from p. Furthermore, we can pick for each non-terminal X_i, X'_j exactly one production p_i, q_j with $\text{lhs}(p_i) = X_i, \text{lhs}(q_j) = X_j$ to compute the derivations (this also hold inductively for all productions we add). Since G is acyclic we just construct two SLPs representing ω_q and ω_p. We apply FCEQUALS for these SLPs. Let $p_1, \dots p_k$ be all productions with $\forall i \in [k] : \text{lhs}(p_i) = X$. Then we first check the equality of words derived from p_1 and p_2 afterwards we equality for words derived by p_2, p_3 and so on. Whenever two words aren't equal we can be sure that $L(G)$ is not a singleton set. If on the other hand all words are equal we can add X to $Singleton_G$ and proceed until all $X \in N$ are contained in $Singleton_G$. After this process terminates without reporting that some words aren't equal, we can be sure that $L(G)$ is a singleton set. Since $\preceq$ is defined on G we can test productions in topological order of their left-hand sides and eventually decide $X \in Singleton$ or $L(G)$ is not a singleton set.

By theorem 4.27 we can decide for two given SLPs G_1, G_2 if

$$L(G_1) = L(G_2)$$

in $\mathcal{O}(n^2 \log(n))$ time, where $n = |G_1| + |G_2|$. For constructing the SLPs we use for generating a shortest word, but replacing the sorted heap by an unsorted one. Since there are are only $\mathcal{O}(|P|)$ productions and the construction of an SLP requires $\mathcal{O}(|G|)$ time, the statement follows. ■

Remark 4.38. We can even drop the condition that the grammar is reduced and in wCNF without ϵ-productions, since we can transform any grammar into wCNF without ϵ-productions in linear time. Furthermore, the size of the constructed grammar is linear in the size of G.

Algorithm 23: IsSingleton: Checks whether $L(G)$ is a singleton set.

```
   input : G = (Δ, N, S, P);
   output: true if G is a singleton set, otherwise false;
 1 transform G into wCNF without ε-productions;
 2 L ← GetOrder(G);
 3 if ∃X ∈ N : X ∉ L then
 4     return false;
 5 foreach X ∈ N in topological order do
 6     P_X ← {q ∈ P | lhs(q) = X};
 7     G_1 ← undefined;
 8     G_2 ← undefined;
 9     foreach p ∈ P_X do
10         P' ← ((P \ P_X) ∪ {p});
11         if G_2 is defined then
12             if ¬IsEquals(G_1, G_2) then
13                 return false;
14         G' ← (Δ, N, X, P');
15         G_1 ← GetWord(G');
16         G_2 ← G_1;
17 return true;
```

5. The morphism equivalence problem for context-free languages

In the following chapter we will discuss the morphism equivalence problem for context-free languages. [35, 47] showed the following

Theorem 5.1 ([35, 47]). *The following problem is in* PTIME.

Input: *Given a* CFG $G = (\Delta, N, S, P)$ *and two morphisms* $\mu_1, \mu_2 : \Delta^* \to \Sigma^*$

Output: *Does* $\forall \omega \in L(G) : \mu_1(\omega) = \mu_2(\omega)$ *hold?*

The problem was original formulated by [32] and is strongly connected to problems from automata theory and formal languages. Recently [52, 9] found a connection between this problem and equivalence problem of N2Ws, STWs and LTWs. [35, 47] improved the upper bound of the test set size from double exponential [1] and single exponential [36] to polynomial, or more precisely to $\mathcal{O}(m^6)$, where m is the number of productions of the grammar. Furthermore, they give a lower bound of $\Omega(m^3)$.

Theorem 5.2 ([32]). *The following problem is undecidable.*

Input: *Given a deterministic context-sensitive language* L $G = (\Delta, N, S, P)$ *and two morphisms* $\mu_1 : \Delta^* \to \Sigma^*, \mu_2 : \Delta^* \to \Sigma^*$

Output: *Does* $\forall \omega \in L : \mu_1(\omega) = \mu_2(\omega)$ *hold?*

The huge gap of undecidable and decidable in polynomial time between deterministic context-sensitive and context-free languages is quite surprising.

5.1. Test sets

By the Ehrenfeucht conjecture, which was proved to be true, we can reduce the problem of the morphism equivalence problem for context-free languages to the problem of testing the agreement of morphisms μ_1, μ_2 on a finite test set of words $T \subseteq L(G)$.

Theorem 5.3 (Ehrenfeucht conjecture [2, 26]). *For each language* $L \subset \Delta^*$ *over a finite alphabet* Δ *there exists a* ***finite*** *subset* $T \subseteq L$ *such that for any two morphisms* μ_1, μ_2 *on* Δ^*

$$\forall \omega \in L : \mu_1(\omega) = \mu_2(\omega) \iff \forall \omega \in T : \mu_1(\omega) = \mu_2(\omega).$$

Definition 5.4 (Test set [47]). Let L be a language and Δ, Σ be alphabets. We say $T \subseteq L$ is a test set for L if and only if:

(i) $T \subseteq L$

(ii) for any two morphisms $\mu_1, \mu_2 : \Delta^* \to \Sigma^*$

$$(\forall \omega \in T : \mu_1(\omega) = \mu_2(\omega)) \Rightarrow (\forall \omega \in L : \mu_1(\omega) = \mu_2(\omega)) .$$

The main task is to find and generate a test set that is as small as possible. We use the same strategy introduced in [47] to generate such a set. A word in the test set is represented by a SLP. To test morphism equality on a word ω_G we first apply the morphisms on the SLP (see lemma 3.11.1), which gives us two SLPs G_{μ_1} and G_{μ_2} such that $\omega_{G_{\mu_1}} = \mu_1(\omega_G)$ and $\omega_{G_{\mu_2}} = \mu_2(\omega_G)$. Then we apply FCEQUALS on G_{μ_1} and G_{μ_2}. If $\omega_{G_{\mu_1}}$ is equal to $\omega_{G_{\mu_2}}$, we test the next word in the test set, otherwise we can report that μ_1, μ_2 don't agree on G. The rest of the chapter describes the construction of the test set of G that contains short words of $L(G)$. We skip proofs that are not relevant for the construction itself, you can find all these proofs in the original material.

5.2. Test sets for context-free languages

Lemma 5.4.1. *Let L be a language T_1 a test set for L and $T_2 \subseteq T_1$ a test set for T_1, then T_2 is a test set for L.*

This follows directly from the definition of a test set. By this fact and the key lemma 5.5.1, [35, 47] proved that the linearisation L_{lin} of a context-free language L is a test set for L. The proof of the key lemma is very long and technical. Since the proof doesn't play a role in the construction of the test set, we refer to [35, 47] for details.

Definition 5.5. We define $G_4 = (\Delta, N, S, P_4)$ by the following productions

$$\begin{aligned} S &\to b_4 X_3 \hat{b}_4, & S &\to a_4 X_3 \hat{a}_4, \\ X_3 &\to b_3 X_2 \hat{b}_3, & X_3 &\to a_3 X_2 \hat{a}_3 \\ X_2 &\to b_2 X_1 \hat{b}_2, & X_2 &\to a_2 X_1 \hat{a}_2 \\ X_1 &\to b_1 \hat{b}_1, & X_1 &\to a_1 \hat{a}_1. \end{aligned}$$

Furthermore, we define $L_4 = L(G_4)$ and $T_4 = L_4 \setminus \{b_4 b_3 b_2 b_1 \hat{b}_1 \hat{b}_2 \hat{b}_3 \hat{b}_4\}$.

Lemma 5.5.1 (Key lemma [35, 47]). *T_4 is a test set for L_4.*

Lemma 5.5.2 ([35, 47]). *Let $u, w, z \in \Delta^*$. If two morphisms μ_1, μ_2 agree on words uw, zw, uy they agree on zy.*

Proof. We have

$$\begin{aligned}
\mu_1(uw) &= \mu_2(uw) = a_1 \dots a_k, \\
\mu_1(zw) &= b_1 \dots b_{j_1} \underbrace{a_{j_2} \dots a_k}_{\mu_1(w)} = b_1 \dots b_{j_3} \underbrace{a_{j_4} \dots a_k}_{\mu_2(w)} = \mu_2(zw), \\
\mu_1(uy) &= \underbrace{a_1 \dots a_{i_1}}_{\mu_1(u)} c_{i_2} \dots c_{k'} = \underbrace{a_1 \dots a_{i_3}}_{\mu_2(u)} c_{i_4} \dots c_{k'} = \mu_2(uy).
\end{aligned}$$

Furthermore, we get

$$\begin{aligned}
\mu_1(zwuy) &= b_1 \dots b_{j_1} \underbrace{a_{i_1+1} \dots a_k a_1 \dots a_{i_1}}_{\mu_1(wu)} c_{i_2} \dots c_{k'} \\
&= b_1 \dots b_{j_3} \underbrace{a_{i_3+1} \dots a_k a_1 \dots a_{i_3}}_{\mu_2(wu)} c_{i_4} \dots c_{k'} = \mu_2(zwuy)
\end{aligned}$$

If we delete $\mu_1(wu)$ and $\mu_2(wu)$ from the middle part of $\mu_1(zwuy)$, and $\mu_2(zwuy)$ respectively, the resulting words are equal if and only if they were equal before the deletion. Note that $\mu_1(wu)$ has to be equal to a shifted version of $\mu_2(wu)$, therefore we get

$$\begin{aligned}
\mu_1(zy) &= \mu_1(z)\mu_1(wu)\mu_1(wu)^{-1}\mu_1(y) \\
&= b_1 \dots b_{j_1} c_{i_2} \dots c_{k'} = b_1 \dots b_{j_3} c_{i_4} \dots c_{k'} \\
&= \mu_2(z)\mu_2(wu)\mu_2(wu)^{-1}\mu_2(y) = \mu_2(zy)
\end{aligned}$$

■

Definition 5.6 (Non-terminal word)**.** Let $G = (\Delta, N, S, P)$ be a reduced context-free grammar. Then for each $X \in N$ the word ω_X is a shortest word derived by a derivation starting from X.

Definition 5.7 (Linearisation of a CFG)**.** Let $G = (\Delta, N, S, P)$ be a context-free grammar in weak Chomsky normal form without useless productions and non-terminals. We call $G_{lin} = (\Delta, N, S, P_{lin})$ a linearisation of G where P_{lin} is defined as follows:

$$\frac{p : X \to X_1 X_2}{\begin{array}{c} X \to X_1 \omega_{X_2} \in P_{lin} \\ X \to \omega_{X_1} X_2 \in P_{lin} \\ X \to \omega_{X_1} \omega_{X_2} \in P_{lin} \end{array}} \qquad \frac{p : X \to Y}{\begin{array}{c} X \to \omega_Y \in P_{lin} \\ X \to Y \in P_{lin} \end{array}} \qquad \frac{p : X \to \alpha \qquad \alpha \in \Delta^*}{X \to \alpha \in P_{lin}}$$

We interpret all ω_X as words in Δ^*.

Definition 5.8 (Grammar graph)**.** Let $G_{lin} = (\Delta, N, S, P_{lin})$ be a linear grammar. We call the directed multigraph $\mathcal{G} = (V, E)$ grammar graph of grammar G_{lin} with

$$\begin{aligned}
V =& N \cup \{T\}, \\
E =& \{(u, v) \in V \times V \mid \exists p \in P_{lin} : \mathrm{lhs}(p) = u \ \wedge \ v \in (\mathrm{alph}(\mathrm{rhs}(p)) \cap N)\} \cup \\
& \{(X, T) \in V \times T \mid \exists (X \to \alpha) \in P_{lin}, \alpha \in \Delta^*\}.
\end{aligned}$$

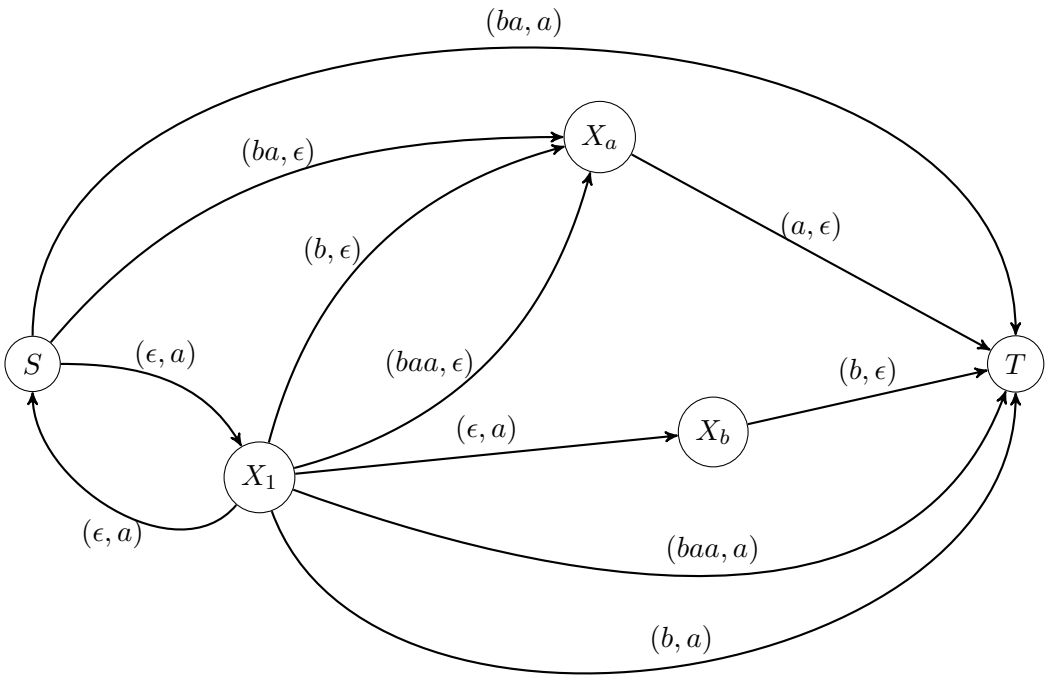


Figure 5.1.: $\mathcal{G}$ of the linear grammar of example 5.11. Note that instead of the explicit word, e. g. baa, we construct a SLP generating the corresponding word e. g. ω_S.

Furthermore, we define a function $\mathrm{w} : E \to \Delta^* \times \Delta^*$ such that $\mathrm{w}((u,v)) = (\omega_{X_1}, \omega_{X_2})$ where $\omega_{X_1}, \omega_{X_2}$ are non-terminal words in the production of the corresponding edge. If $v \neq T$, ω_{X_1} or ω_{X_2} is equal to the empty word ϵ (see fig. 5.1).

Definition 5.9 (Max word length)**.** Let L be a language. Then $\max_i(L)$ is the set of words in L of length at most i.

Definition 5.10 (Partly linear language)**.** Let $G = (\Delta, N, S, P)$ be a reduced context-free grammar in Chomsky normal form and $G_{lin} = (\Delta, N, S, P_{lin})$ its linearisation. Then a word in $L_d(G)$ is generated by derivation trees such that productions corresponding to nodes of height at most d are from P_{lin} and all other productions are from P.

Example 5.11 (Linearisation of a CFG)**.** Let $G = (\Delta, N, S, P)$ with productions

$$\begin{aligned} S &\to X_1 X_a, \\ X_1 &\to S X_a, \; X_1 \to X_b X_a, \\ X_a &\to a, \\ X_b &\to b \end{aligned}$$

then P_{lin} contains the following productions

$$\begin{aligned} S &\to X_1 a, \quad S \to baX_a \quad S \to baa, \\ X_1 &\to Sa, \quad X_1 \to baaX_a, \quad X_1 \to baaa, \\ X_1 &\to X_b a, \quad X_1 \to bX_a, \quad X_1 \to ba, \\ X_a &\to a, \\ X_b &\to b. \end{aligned}$$

The grammar graph $\mathcal{G}$ of G_{lin} is displayed in fig. 5.1.

Lemma 5.11.1. *Let $G = (\Delta, N, S, P)$ be a reduced context-free grammar in Chomsky normal form. Then we can construct G_{lin} in $\mathcal{O}(|N| \cdot (|G| + |P| \log(|P|)))$ and $\mathcal{G}$ in $\mathcal{O}(|G|)$ time. The size of the linear grammar G_{lin} is in $\mathcal{O}(|G| \cdot |N|)$ and the size of $\mathcal{G}$ is in $\mathcal{O}(|G|)$.*

Proof. We have to construct all productions for the SLPs and at most 3 productions for each production in G. We can construct for each $X \in G$ a SLP that represents the shortest word of the grammar $G_X = (\Delta, N_X, X, P_X)$ in $\mathcal{O}(|G| + |P| \log(|P|))$ time using algorithm 6. The grammar of the SLP is of size $\mathcal{O}(|G|)$ and we require $\mathcal{O}(|N|)$ SLPs. Therefore the size of G_{lin} is at most $3 \cdot |G| + |N| \cdot |G|$ which is in $\mathcal{O}(|G| \cdot |N|)$. The construction of all SLPs requires overall $\mathcal{O}(|N| \cdot (|G| + |P| \log(|P|)))$ time and the construction of the remaining productions require $\mathcal{O}(|G|)$ time. Overall it takes $\mathcal{O}(|N| \cdot (|G| + |P| \log(|P|)))$ time to construct G_{lin}. An edge $(\omega_{X_1}, \omega_{X_2})$ of $\mathcal{G}$ is represented by the couple (X_1, X_2), therefore the construction of $\mathcal{G}$ takes $\mathcal{O}(|G|)$ time. ∎

Remark 5.12. Note that for the construction of G_{lin} we have to construct SLPs with distinct non-terminals. Otherwise it could happen that a SLP is not well-defined.

Theorem 5.13 ([35, 47] Test sets for context-free languages)**.** *Let G be a context-free grammar, then $L(G_{lin})$ is a test set for $L(G)$.*

Proof. We prove the following

(i) $\max_i(L_{d+1})$ is a test set for $\max_i(L_d)$ for $i \geq d+1 \geq 1$

(ii) $\max_i(L_{lin})$ is a test set for $\max_i(L)$ for $i \geq 1$

(iii) $L(G_{lin})$ is a test set for $L(G)$

(i) Let's assume that morphisms μ_1, μ_2 agree on $\max_i(L_{d+1})$. Let ω be a word in $\max_i(L_d) \setminus \max_i(L_{d+1})$. ω is of the form $uvxw$ and since we use productions from P_{lin} for derivations on the level $\leq d$ in the derivation tree, there is a non-terminal X at height d and a derivation steps $S \Rightarrow_G^* \alpha \Rightarrow uXw$. For the next derivation steps we use productions from P to derive $uvxw$ and since the grammar is in Chomsky normal form, the next derivation step would be $uXw \Rightarrow_G uX_1X_2w$. However, we know

$$u\omega_{X_1}X_2w, uX_1\omega_{X_2}w, u\omega_{X_1}\omega_{X_2}w \in L_{d+1}.$$

Using lemma 5.5.2 and the fact that μ_1, μ_2 agree on these words, we can follow that they agree on $uvxw$.

(ii) From (i) we get $\max_d(L_{d+1})$ is a test set for $\max_d(L_d)$ and $\max_d(L_{d+2})$ is a test set for $\max_d(L_{d+1})$ therefore $\max_d(L_{d+2})$ is a test set for $\max_d(L_d)$ and so on. We can derive that $\max_d(L_d)$ is a test set for $\max_d(L_0) = \max_d(L)$. Since G is in Chomsky normal form, every derivation tree generating a word of length at most d contains only internal nodes of height at most d. Hence, $\max_d(L_d) = \max_d(L_{lin})$ and the result follows.

(iii) Since $\max_i(L_{lin})$ is a test set for $\max_i(L)$ for $i \geq 1$, we can construct the following test set:

$$\bigcup_{i=1}^{\infty} \max_i(L_{lin}) = L(G_{lin})$$

which is a test set for

$$\bigcup_{i=1}^{\infty} \max_i(L) = L.$$

■

Remark 5.14. In the proof of theorem 5.13 we assume that the grammar G is in Chomsky normal form. For the following construction of the test set it is sufficient that G is in weak Chomsky normal form.

5.3. Test sets for linear grammars

If we would have a construction for test sets for an arbitrary linear grammar, we get by theorem 5.13, that we can construct test sets for arbitrary context-free grammars.

Definition 5.15 ([35, 47] Path)**.** Let $p = (v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$ be a path of length n of the grammar graph of G_{lin} and assume $\mathrm{w}((v_i, v_{i+1})) = (\omega_i^l, \omega_i^r)$. Then we extend the definition of w to paths in a natural way i. e.

$$\mathrm{w}(p) = \omega_0^l \ldots \omega_{n-1}^l \omega_{n-1}^r \ldots \omega_0^r \in \Delta^*$$

is the word generated by the path p.

We get the following connection between G_{lin} and its graph:

Lemma 5.15.1. *Let G_{lin} be a linear grammar then,*

$$L(G_{lin}) = \{\mathrm{w}(p) \mid p \text{ is a path of the graph of } G_{lin} \text{ and } p = (S, v_1) \ldots (v_{n-1}, T).\}.$$

Therefore we can contract all words in $L(G_{lin})$ by traversing through the graph of G_{lin} starting in S and ending in T. While traversing the graph, we can concatenate all words of visited edges.

Definition 5.16 (Non-terminal trees)**.** Let $G_{lin} = (\Delta, N, S, P_{lin})$ be a linear grammar. We define for each $X \in N$ a unique tree of X by picking one tree out of the set of trees. We denote this tree by $\mathrm{tree}(X)$. The tree contains all nodes of the grammar graph that are reachable from X and X is the root of the tree.

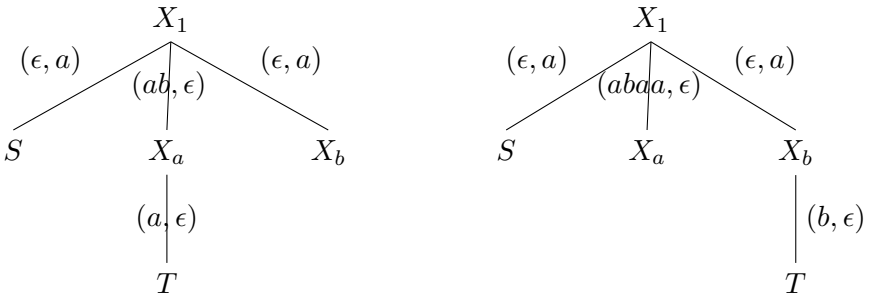


Figure 5.2.: Two possible trees for X_1 of the grammar graph displayed in fig. 5.1.

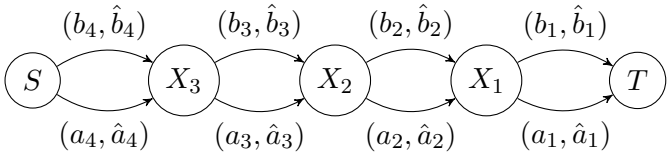
Remark 5.17 (Non-terminal trees). For a non-terminal X there exit several trees (see fig. 5.2). For the following construction it is sufficient to pick only one of these trees.

Definition 5.18 (Plandowski path)**.** Let $G_{lin} = (\Delta, N, S, P_{lin})$ be a linear grammar, $\mathcal{G}$ the grammar graph of G_{lin} and $\mathcal{T} = \{\mathrm{tree}(X) \mid X \in N\}$. We call the following paths $\lambda = (u_1, v_1)(u_2, v_2) \ldots (u_n, v_n)$, Plandowski paths of a linear grammar. We start in $\mathrm{tree}(S)$ traverse to u_1 and traverse the edge (u_1, v_1) in $\mathcal{G}$. We go in $\mathrm{tree}(v_1)$ traverse to u_2 and take the edge (u_2, v_2) and so on. In the last step we traverse the edge (u_n, v_n) in $\mathcal{G}$ go in $\mathrm{tree}(v_n)$ and traverse to T. We call an edge of a Plandowski path, Plandowski edge.

Remark 5.19. For a Plandowski path $\lambda = (u_1, v_1)(u_2, v_2)\dots(u_n, v_n)$ it is not necessary that $v_i = u_{i+1}$ but u_{i+1} has to be reachable from v_i. Therefore, for some sequences the Plandowski path might be undefined. For each sequence of Plandowski edges, at most one path is in $\mathcal{G}$ associated.

Lemma 5.19.1. *Let $G_{lin} = (\Delta, N, S, P_{lin})$ be a linear grammar, $\mathcal{G}$ the grammar graph of G_{lin}. Furthermore, let $\mathrm{path}(\lambda)$ be the path of edges in $\mathcal{G}$ associated with the Plandowski path λ and $F_k(G_{lin}) = \{\mathrm{w}(\mathrm{path}(\lambda)) \mid \lambda$ has at most k edges.$\}$, then $F_6(G_{lin})$ is a test set for $L(G_{lin})$.*

Proof. First note that G_4 in definition 5.5 is a linear grammar that has the following grammar graph:



Furthermore, we know that if μ_1, μ_2 agree on words in $L(G_4) \setminus \{\omega\}$ with $\omega \in L(G_4)$ they agree on $L(G_4)$. Let $k \geq 6$ and suppose μ_1, μ_2 agree on $F_k(G)$. Let

$$\lambda = (u_1, v_1), \dots, (u_{k+1}, v_{k+1})$$

such that $\mathrm{w}(\mathrm{path}(\lambda)) \in F_{k+1}(G)$ i. e. λ is a Plandowski path with $k+1$ edges. Let π_1 be the path from S to u_2 in $\mathrm{tree}(S)$, π_2 be the path from v_2 to u_4 in $\mathrm{tree}(v_2)$, π_3 be the path from v_4 to u_6 in $\mathrm{tree}(v_4)$ and π_4 be the path from u_6 to t in $\mathrm{tree}(u_6)$ (see fig. 5.3). If we

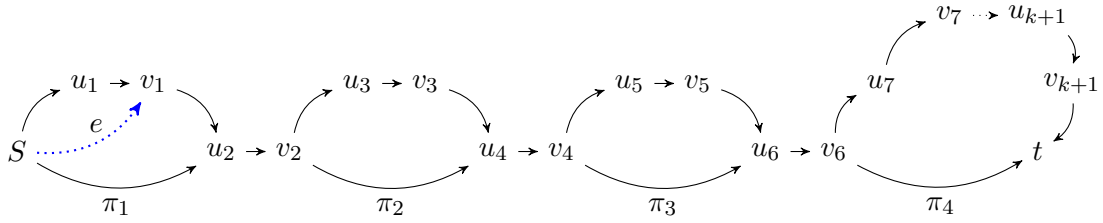


Figure 5.3.: Situation in the proof of lemma 5.19.1. All path except the top most one belongs to $F_k(G)$.

remove at least one of the Plandowski edges $(u_1, v_1), (u_3, v_3), (u_5, v_5)$ or the sequence of Plandowski edges $(u_7, v_7), \dots, (u_{k+1}, v_{k+1})$ from λ and use π_1, π_2, π_3 or π_4 instead, the new Plandowski path has k Plandowski edges. Since μ_1, μ_2 agree on $F_k(G)$ we can follow by using the key lemma 5.5.1 that μ_1, μ_2 agree on $F_{k+1}(G)$. Using lemma 5.4.1 and the fact that for each word $\omega \in L(G)$ there exists a n such that $\omega \in F_n(G)$ we can conclude that $F_6(G_{lin})$ is a test set for $L(G_{lin})$. ■

Remark 5.20. The reason why the proof above isn't applicable for $k < 6$ is that we don't find paths $\pi_1, \dots, \pi_4$. It is necessary to get a valid Plandowksi path if we replace a path

by some π, π has to end in a tree node of λ. Furthermore, the path should contain a Plandowski edge. By these conditions it is not possible to find all these paths for F_6. The path e in fig. 5.3, for example, can't be used to build π_1 since, if we use e, we would traverse through a tree two times in a row. Therefore the path would not be contained in F_5. In addition we can't use the path from S to u_1, since in that case we would not replace a Plandowski edge.

Algorithm 24: MORPHSIMEQUALITY: Tests if μ_1, μ_2 agree on a CFG G.

input : G, μ_1, μ_2;
output: true if μ_1, μ_2 agree on a CFG G;
1 construct the grammar graph $\mathcal{G}$ of G;
2 construct all trees $\mathcal{T}$;
3 **return** AGREEONPATH($[], \mu_1, \mu_2, \mathcal{G}, \mathcal{T}$)

Theorem 5.21. *Let $G = (\Delta, N, S, P)$ be a context-free grammar. We can check whether μ_1, μ_2 agree on G using algorithm 24 in $\mathcal{O}(m^6 \cdot (|G| \cdot |N| \cdot r)^2 \log(|G| \cdot |N| \cdot r))$ time, where m is the number of productions in G and $r = \max\{\max\{|\mu_1(a)|, |\mu_2(a)|\} \mid a \in \Delta\}$.*

Proof. By lemma 5.11.1 we construct G_{lin} and $\mathcal{G}$ in $\mathcal{O}(|N| \cdot (|G| + |P| \log(|P|)))$ time . We organize the SLPs in a mapping, such that we can access a SLP G_X for a non-terminal $X \in N$ in constant time. Furthermore, we construct for each non-terminal $X \in N$ a tree $\mathrm{tree}(X)$ of size $\mathcal{O}(|G|)$ in $\mathcal{O}(|G|)$ time using a breath first search on $\mathcal{G}$. We traverse through each tree and each Plandowski edge, starting with the tree $\mathrm{tree}(S)$ in a depth first fashion (see algorithm 25). We use two stacks, one for the Plandowski edges and one for the tree paths, therefore we can add and remove a tree path and a Plandowski edge in $\mathcal{O}(1)$ time.

Suppose we generate a path $(X_1, Y_1) \dots (X_n, Y_n)$ where $X_i, Y_i \in N$. then we construct the word $G' = (\Delta, N', S', P')$ where N', P' contain all non-terminals and productions of G_{X_i}, G_{Y_i} for $1 \leq i \leq n$. We add the production

$$S' \to X_1 \dots X_n Y_n \dots Y_1$$

to P'. Then we apply the morphism μ_1, μ_2 on G' and transform both grammars into Chomsky normal form to get grammars G_1, G_2. Afterwards we call FCEQUALS(G_1, G_2).

Clearly, each Plandowski path is unique and there are

$$\sum_{i=1}^{6} 3m^i = \mathcal{O}(m^6)$$

such paths, where m is the number of productions of G. For each such graph we traverse at most 6 trees of size at most $\mathcal{O}(|N|)$. Therefore, the construction of one path can be done in $\mathcal{O}(|N|)$ and the construction of all paths in $\mathcal{O}(m^6 \cdot |N|)$ time. The

size of one path is in $\mathcal{O}(|N|)$ therefore, the size of the SLP, representing the word defined by the path, is in $\mathcal{O}(|G| \cdot |N|)$. By lemma 3.11.1 and 3.11.2 the size of the final grammars is in $\mathcal{O}(|G| \cdot |N| \cdot \max\{\max\{|\mu_1(a)|, |\mu_2(a)|\} \mid a \in \Delta\})$ and they can be constructed in $\mathcal{O}(|G| \cdot |N| \cdot \max\{\max\{|\mu_1(a)|, |\mu_2(a)|\} \mid a \in \Delta\}))$ time. By theorem 4.27 the equality test of all words require $\mathcal{O}(m^6 \cdot (|G| \cdot |N| \cdot r)^2 \log(|G| \cdot |N| \cdot r))$ where $r = \max\{\max\{|\mu_1(a)|, |\mu_2(a)|\} \mid a \in \Delta\}$. Therefore the overall running time is $\mathcal{O}(m^6 \cdot (|G| \cdot |N| \cdot r)^2 \log(|G| \cdot |N| \cdot r) + m^6 \cdot |G|) = \mathcal{O}(m^6 \cdot (|G| \cdot |N| \cdot r)^2 \log(|G| \cdot |N| \cdot r))$. ■

Performance tuning

Suppose we want to solve the morphism equivalence problem on a singleton set i. e. $L(G) = \{\omega\}$. If we apply algorithm 24 without further improvements we would test the equality of $\mu_1(\omega)$ and $\mu_2(\omega)$ multiple times. All Plandowski paths would represent the same word, since each word represented by a Plandowski path is in the language of the grammar. To reduce the running time we first check which non-terminals of G are in $Singleton_G$ in $\mathcal{O}(|G|^3 \log(|G|))$ time using algorithm 23. We avoid the construction of nodes in $\mathcal{G}$ for non-terminals $X \in Singleton_G$, i. e. if there exists a production $X \to X_1X_2$ in P and $X_1 \in Singleton_G$ we do not add the production $X \to X_1\omega_{X_2}$ to P_{lin}. Since we construct trees based on $\mathcal{G}$, trees also contain no singleton non-terminal.

The second improvement can be made due to parallelization. The agreement test of μ_1, μ_2 on each of the $\mathcal{O}(m^6)$ words can be done in parallel. The problem is highly parallelisable in an easy way. Imagine we would had m processors. Each processor could check the agreement for Plandowski paths for a fixed first Plandowski edge. Therefore we could reduce the running time to $\mathcal{O}(m^5 \cdot (|G| \cdot |N| \cdot r)^2 \log(|G| \cdot |N| \cdot r))$.

5.4. The periodicity problem on context-free languages

Definition 5.22 (Periodic words languages [9])**.** A word w is said to be periodic of period u if u is the smallest word such that $w \in u^*$. A language L is said to be periodic of period u if u is the smallest word such that $L \subseteq u^*$.

Theorem 5.23 ([9])**.** *The following problem is in* PTIME.

Input: *Given a* CFG $G = (\Delta, N, S, P)$ *with* $L(G) \neq \emptyset$

Output: *Is* $L(G)$ *a periodic language?*

The algorithm for testing equality of two LTWs includes the test whether a non-empty language is period.

Lemma 5.23.1 (Periodicity lemma)**.** *Let* u, w *words over* Δ^* *then* $uw = wu$ *if and only if* $u = \epsilon \vee w = \epsilon$ *or* u, w *are of the same period.*

Proof. First assume $uw = wu$: We prove by induction over $|u| + |w|$ that $u = \epsilon \vee w = \epsilon$ or u, w are of the same period. Suppose $|u| + |w| = 0 \Rightarrow u = w = \epsilon$ and the statement follows. Suppose now $|u| + |w| = n > 0$. If $|u| = |w|$ it follows that u is a prefix of w and w a prefix of u and therefore $u = w$ and we are done. Otherwise, assume w. l. o. g. $|w| > |u|$. Clearly, u is a prefix of w and therefore we have

$$w = ut \tag{5.1}$$

for some $t \neq \epsilon$. By this we get

$$uw = wu \iff uut = utu \iff ut = tu.$$

However, $|u| + |t| < |u| + |w|$. Since $u, t \neq \epsilon$ it follows, by the induction hypothesis, that u, t are of the same period z i. e. $u = z^i, t = z^j, i, j \geq 1$ and by using eq. (5.1) we get

$$w = ut = z^i z^j = z^{i+j}$$

which proves that u, w are of the same period.

Suppose $u = \epsilon \vee w = \epsilon$ or u, w are of the same period: If u or w is the empty word, it is trivial that $uw = wu$ holds. Suppose u and w share the same period z i. e. $u = z^i, w = z^j, i, j \geq 1$. We get

$$z^i z^j = uw = z^{i+j} = z^j z^i = wu$$

■

Theorem 5.24. *Let $G = (\Delta, N, S, P)$ be a non-empty context-free language. We can check whether $L(G)$ is periodic in $\mathcal{O}(m^6 \cdot (|G| \cdot |N|)^2 \log(|G| \cdot |N|))$ time, where m is the numbers of productions in G.*

Proof. Let $G = (\Delta, N, S, P)$ be a context-free grammar. By using lemma 3.6.2, we can construct a SLP $G' = (\Delta', N', S', P')$ for a shortest non-empty word of $L(G)$ in $\mathcal{O}(|G| + |P| \log(|P|))$ time. Furthermore, we can copy G' to construct a $G'' = (\Delta'', N'', S'', P'')$ with $\omega_{G'} = \omega_{G''}$ in $\mathcal{O}(|G'|)$ time. Δ' and Δ'' are copies of Δ. We construct morphisms μ' and μ'' such that μ' maps every $a \in \Delta''$ to ϵ and each $a \in \Delta'$ to the corresponding $a \in \Delta$. μ'' on the other hand maps $a \in \Delta'$ to ϵ and $a \in \Delta''$ to the corresponding $a \in \Delta$. Furthermore, we construct $G_0 = (\Delta, N \cup N' \cup N'', S_0, P_0)$ with $P_0 = P \cup S_0 \to S'SS''$. G is of period $\omega_{G'}$ if and only if

$$\forall u \in L(G) : \omega_{G'} u = u \omega_{G''} \iff \forall w \in L(G_0) : \mu_1(w) = \mu_2(w)$$

holds [9]. Clearly, the size of G'' is in $\mathcal{O}(|G|)$ and the morphism increases the size of the grammar and the resulting word by a constant factor. By applying theorem 5.21 we get the desired running time. ■

Remark 5.25. We realise μ' by replacing each non-terminal $X'' \in N''$ on any right-hand sides of a generated word by ϵ. Likewise we realize μ''.

Algorithm 25: AGREEONPATH: Tests if μ_1, μ_2 agree on a Plandowksi containing at most 6 Plandwoski edges.

input : $path = (X_1, X_2) \dots (X_{n-1}, X_n), \mu_1, \mu_2, \mathcal{G}, \mathcal{T}$;
output: true if μ_1, μ_2 agree on Plandowski paths containing at most 6 edges;

```
if path contains at most 5 Plandowski edges then
    if path is empty then
        tree ← T(S);
    else
        tree ← T(X_n);
    for treepath ∈ tree do
        let treepath = (Y_1, Y_2) ... (Y_{k-1}, Y_k);
        append treepath to path;
        let E be the set of outgoing edges of Y_k in G;
        for e ∈ E do
            let e = (Z_1, Z_2);
            append the Plandowski edge e to path;
            if Z_2 ≠ T then
                append (Z_2, T) to path;
            G_1 ← CONCATENATE(μ_1(w(path)));
            G_2 ← CONCATENATE(μ_2(w(path)));
            isEquals ← FCEQUALS(G_1, G_2);
            if Z_2 ≠ T then
                remove (Z_2, T) from path;
            if ¬isEquals then
                return false;
            isEquals ← AGREEONPATH(path, μ_1, μ_2, G, T)
            remove e from the end of path;
        remove treepath from the end of path;
```

6. Equivalence test for STWs and LTWs

6.1. Equivalence test for finite top-down tree automata

The domain of a sequential or linear tree to word transducer is defined by its deterministic top-down tree automata i. e. a transducer without any output. The domain is the tree language which is recognized by the tree automata. We reduce the problem of testing equivalence of deterministic top-down tree automata $\mathcal{M}_1, \mathcal{M}_2$ to the equivalence problem of deterministic word automata (NFAs).

Definition 6.1 (Paths)**.** Let Σ be a ranked alphabet and $t = f(t_1, \ldots, t_i) \in T_\Sigma$ be a tree, then the paths $\mathrm{paths}(t)$ of the tree is a *subset of words* $(\Sigma \cup \mathbb{N})^*$ defined by:

$$\frac{t = a \in \Sigma \qquad \mathrm{rank}(a) = 0}{a \in \mathrm{paths}(t)} \qquad \frac{t = f(t_1, \ldots, t_k) \in T_\Sigma \qquad 1 \leq i \leq k \qquad \omega \in \mathrm{paths}(t_i)}{fi\omega \in \mathrm{paths}(t)}$$

The *path closure* $\text{paths-cl}(L)$ of a tree language $L \subset T_\Sigma$ is the set of all trees that contain only paths of trees in L i. e.

$$\text{paths-cl}(L) := \{t \mid \mathrm{paths}(t) \subseteq \mathrm{paths}(L)\}$$

We call L *path-closed* if $L = \text{paths-cl}(L)$.

Lemma 6.1.1. *Given a finite tree automata $\mathcal{M}$, we can construct in $\mathcal{O}(|\mathcal{M}|)$ a finite word automata $\mathcal{A}$ of size $\mathcal{O}(|\mathcal{M}|)$ over a finite subset of $\Sigma \times (\mathbb{N} \cup \{0\})$ such that $L(\mathcal{A}) = \mathrm{paths}(L(\mathcal{M}))$.*

Proof. Let $\mathcal{M} = (\Sigma, Q_\mathcal{M}, Q_F, \delta_\mathcal{M})$ be a finite tree automata. Then we construct $\mathcal{A} = (\Sigma', Q_\mathcal{A}, Q'_F, Q'_I, \delta_\mathcal{A})$ with $\Sigma' \subset (\Sigma \times (\mathbb{N} \cup \{0\}))$, $Q'_I = Q_F, Q_\mathcal{A} = Q_\mathcal{M}, Q'_F = \Sigma_0$ and the transition relation $\delta_\mathcal{A}$ defined by:

$$\frac{a \to q \in \delta_\mathcal{M}}{q \xrightarrow{(a,0)} a \in \delta_\mathcal{A}} \qquad \frac{f(q_1, \ldots, q_k) \to q \in \delta_\mathcal{M} \qquad 1 \leq i \leq k}{q \xrightarrow{(f,i)} q_i \in \delta_\mathcal{A}} \qquad \frac{q \to p \in \delta_\mathcal{M}}{q \xrightarrow{\epsilon} p \in \delta_\mathcal{A}}$$

It follows from the construction that $L(\mathcal{A}) = \mathrm{paths}(L(\mathcal{M}))$. For each rule in $\delta_\mathcal{M}$ we construct a constant number of rules for the NFA and we can construct these rules by iterating over all rules in $\delta_\mathcal{M}$. Therefore, we can construct $\mathcal{A}$ in $\mathcal{O}(|\mathcal{M}|)$ time. ■

If the tree automata $\mathcal{M}$ is top-down deterministic, the constructed word automata is deterministic, too. One can show that the language recognized by a deterministic top-down tree automata is path-closed [11].

Theorem 6.2. *Given two finite top-down tree automata $\mathcal{M}_1, \mathcal{M}_2$, we can decide $L(\mathcal{M}_1) = L(\mathcal{M}_2)$ in almost $\mathcal{O}(|\mathcal{M}_1| + |\mathcal{M}_2|)$ time.*

Proof. By lemma 6.1.1, we can construct in $\mathcal{O}(|\mathcal{M}_1|+|\mathcal{M}_2|)$ two NFAs $\mathcal{A}_1, \mathcal{A}_2$ such that $L(\mathcal{A}_1) = \text{paths}(L(\mathcal{M}_1))$ and $L(\mathcal{A}_2) = \text{paths}(L(\mathcal{M}_2))$. Since $\mathcal{M}_1$ and $\mathcal{M}_2$ are top-down deterministic $L(\mathcal{M}_1)$ and $L(\mathcal{M}_2)$ are path-closed i. e. $L(\mathcal{M}_1) = \text{paths-cl}(L(\mathcal{M}_1))$ and therefore

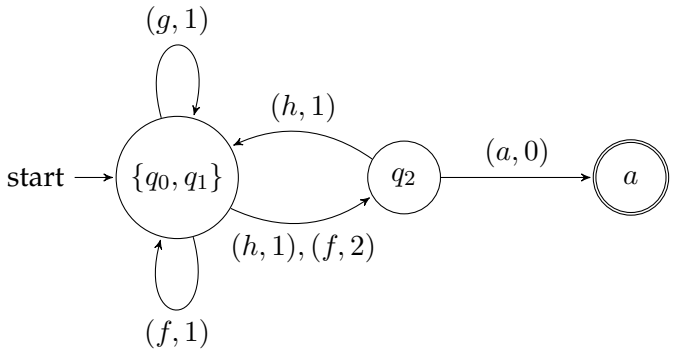
$$L(\mathcal{M}_1) = L(\mathcal{M}_2) \iff \text{paths}(L(\mathcal{M}_1)) = \text{paths}(L(\mathcal{M}_2)).$$

Since $\mathcal{M}_j$ is deterministic, there are no ϵ-rules in $\mathcal{A}_j$ and $\delta_{\mathcal{A}_j}(q, (f, i))$ is a function. Consequently, we can merge all initial states (we may introduce more than one) into one initial state. It follows that $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic. We apply the algorithm presented in [29], to solve $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ in almost $\mathcal{O}(|\Sigma'| \cdot (|Q_{\mathcal{A}_1}| + |Q_{\mathcal{A}_2}|)) = \mathcal{O}(|\mathcal{A}_1| + |\mathcal{A}_2|) = \mathcal{O}(|\mathcal{M}_1| + |\mathcal{M}_2|)$ time. ■

Remark 6.3. In the construction above we use the notation of bottom-up tree automata. For receiving top-down tree automata one has to reverse the arrows in the rules and swap the set of final states to the set of initial states.

Example 6.4. The above deterministic word automata on the right is the result of the construction from above by converting the underlying tree automaton of the STW $\mathcal{M}$ defined by the rules on the left into the corresponding word automata. q_0, q_1 are initial states of the STW and therefore of the underlying top-down tree automata.

$$
\begin{aligned}
q_0(f) &\to u_0 q_1 u_1 q_2 u_2,\\
q_1(g) &\to v_0 q_1 v_1,\\
q_1(h) &\to w_0 q_2 w_1,\\
q_2(h) &\to z_0 q_0 z_1,\\
q_2(a) &\to y_0
\end{aligned}
$$



6.2. Equivalence test for dN2Ws

In the following section we will discuss the equivalence problem for deterministic nested word-to-word transducers (dN2Ws). Solving this problem is an intermediate step for solving equivalence problem for STWs and LTWs. In section 6.3, we will see how we can transform STWs into dN2Ws in linear time. [52] showed the following

Theorem 6.5 ([52]). *The following problem is in* PTIME

Input: *Given two deterministic N2Ws $\mathcal{N}_1, \mathcal{N}_2$ with the same input Σ and output alphabet.*

Output: *Does $[\![\mathcal{N}_1]\!] = [\![\mathcal{N}_2]\!]$ hold?*

We skip the equivalence test of the domain of $\mathcal{N}_1, \mathcal{N}_2$ since we check the domains of the LTWs and STWs. Therefore, we can assume that the domains of the constructed N2Ws are equal. Nevertheless, if the N2Ws are deterministic, we could test their domains in polynomial time (see [4]). We will require the notion of co-reachable states of two N2Ws.

Definition 6.6 (Co-reachable states). Let $\mathcal{N}_1 = (\Sigma, \Delta_1, Q_1, \Gamma_1, R_1, Q_{1,I}, Q_{1,F}), \mathcal{N}_2 = (\Sigma, \Delta_2, Q_2, \Gamma_2, R_2, Q_{2,I}, Q_{2,F})$ be two N2Ws. We define the relation $\mathcal{C}o(\mathcal{N}_1, \mathcal{N}_2) \subseteq Q_1 \times Q_2$ by the following properties:

$$\frac{q_1 \in Q_{1,I} \qquad q_2 \in Q_{2,I}}{(q_1, q_2) \in \mathcal{C}o(\mathcal{N}_1, \mathcal{N}_2)}$$

$$\frac{(q_1, q_2) \in \mathcal{C}o(\mathcal{N}_1, \mathcal{N}_2) \qquad q_1 \xrightarrow{\mathsf{op}\, a/u_1:\gamma_1} p_1 \in R_1 \qquad q_2 \xrightarrow{\mathsf{op}\, a/u_2:\gamma_2} p_2 \in R_2}{(p_1, p_2) \in \mathcal{C}o(\mathcal{N}_1, \mathcal{N}_2)}$$

$$\frac{(q_1, q_2) \in \mathcal{C}o(\mathcal{N}_1, \mathcal{N}_2) \qquad q_1 \xrightarrow{\mathsf{cl}\, a/u_1:\gamma_1} p_1 \in R_1 \qquad q_2 \xrightarrow{\mathsf{cl}\, a/u_2:\gamma_2} p_2 \in R_2}{(p_1, p_2) \in \mathcal{C}o(\mathcal{N}_1, \mathcal{N}_2)}$$

In [52] the authors describe how we can reduce the problem of the equivalence problem of N2Ws to the morphism equivalence problem on CFGs. They give a construction for a context-free grammar containing all successful parallel runs of $\mathcal{N}_1$ and $\mathcal{N}_2$.

Definition 6.7 (Successful parallel runs [52]). Let t be a tree and $\mathcal{A}ct(t) = \{e_1, \ldots, e_n\}$ its traversal actions with $e_1 < \ldots < \ldots e_n$. A *successful parallel run* of $\mathcal{N}_1$ and $\mathcal{N}_2$ on the tree t is a *word* s over the alphabet $\mathcal{R} = R_1 \times R_2$ such that there exist two successful runs τ_1, τ_2 of $\mathcal{N}_1$ and $\mathcal{N}_2$ on t with

$$s = (\tau_1(e_1), \tau_2(e_1)) \ldots (\tau_1(e_n), \tau_2(e_n)).$$

Note that a successful run starting from (q_1, q_2) and ending in (p_1, p_2) can only exists if (q_1, p_1) and (q_2, p_2) are contained in $Co(\mathcal{N}_1, \mathcal{N}_2)$. Remember that $\tau_1(e) = r$, where r is a rule of $\mathcal{N}_1$. Suppose we have a grammar generating all successful parallel runs of $\mathcal{N}_1$ and $\mathcal{N}_2$ and let $\mu_1, \mu_2 : \mathcal{R} \to \Delta^*$ defined by

$$\mu_i((r_1, r_2)) = u, \text{ if } r_i = q \xrightarrow{\beta\, a/u:\gamma} q',$$

then $[\![\mathcal{N}_1]\!] = [\![\mathcal{N}_2]\!]$ holds if and only if μ_1, μ_2 agree on the grammar of successful parallel runs [52]. The construction of the grammar is given by the proof of the following lemma.

Lemma 6.7.1 ([52]). *For any two* N2Ws $\mathcal{N}_1, \mathcal{N}_2$ *with the same alphabets there exists a* CFG G *such that* $L(G)$ *is the set of all successful parallel runs of* $\mathcal{N}_1$ *and* $\mathcal{N}_2$*. The grammar can be constructed in* $\mathcal{O}(|\mathcal{N}_1|^2 \cdot |\mathcal{N}_2|^2 + |Q_{\mathcal{N}_1}|^3 \cdot |Q_{\mathcal{N}_2}|^3)$ *time. It contains at most* $\mathcal{O}(Co(\mathcal{N}_1, \mathcal{N}_2)^3) = \mathcal{O}(|Q_{\mathcal{M}_1}|^3 \cdot |Q_{\mathcal{M}_2}|^3)$ *productions.*

Proof (sketch). They construct the grammar $G = (\mathcal{R}, N, S, P)$ as follows. The set of non-terminals is $N = \{S\} \cup Q_{\mathcal{N}_1}^2 \times Q_{\mathcal{N}_2}^2$. A non-terminal $((p_1, q_1), (p_2, q_2))$ is supposed to produce a parallel run of $\mathcal{N}_1$ from p_1 to q_1 and of $\mathcal{N}_2$ from p_2 to q_2 (reading the same input). There is only one start symbol S and productions in P are defined as follows:

$$\frac{\begin{array}{lllll} r_1, r_1' \in R_{\mathcal{N}_1} & r_1 = p_1 \xrightarrow{\mathsf{op}\ a/u_1:\gamma_1} q_1 & r_1' = p_1' \xrightarrow{\mathsf{cl}\ a/u_1':\gamma_1} q_1' & p_1 \in Q_{I,\mathcal{N}_1} & q_1' \in Q_{F,\mathcal{N}_2} \\ r_2, r_2' \in R_{\mathcal{N}_2} & r_2 = p_2 \xrightarrow{\mathsf{op}\ a/u_2:\gamma_2} q_2 & r_2' = p_2' \xrightarrow{\mathsf{cl}\ a/u_2':\gamma_2} q_2' & p_2 \in Q_{I,\mathcal{N}_2} & q_2' \in Q_{F,\mathcal{N}_2} \end{array}}{S \to (r_1, r_2)((q_1, p_1'), (q_2, p_2'))(r_1', r_2')}$$

$$\frac{\begin{array}{lll} r_1, r_1' \in R_{\mathcal{N}_1} & r_1 = p_1 \xrightarrow{\mathsf{op}\ a/u_1:\gamma_1} q_1 & r_1' = p_1' \xrightarrow{\mathsf{cl}\ a/u_1':\gamma_1} q_1' \\ r_2, r_2' \in R_{\mathcal{N}_2} & r_2 = p_2 \xrightarrow{\mathsf{op}\ a/u_2:\gamma_2} q_2 & r_2' = p_2' \xrightarrow{\mathsf{cl}\ a/u_2':\gamma_2} q_2' \end{array}}{((p_1, q_1'), (p_2, q_2')) \to (r_1, r_2)((q_1, p_1'), (q_2, p_2'))(r_1', r_2')}$$

$$\frac{p_1, p_1', q_1 \in Q_{\mathcal{N}_1} \qquad p_2, p_2', q_2 \in Q_{\mathcal{N}_2}}{((p_1, q_1), (p_2, q_2)) \to ((p_1, p_1'), (p_2, p_2'))((p_1', q_1), (p_2', q_2))} \qquad \frac{q_1 \in Q_{\mathcal{N}_1} \qquad q_2 \in Q_{\mathcal{N}_2}}{((q_1, q_1), (q_2, q_2)) \to \epsilon}$$

■

The first set of productions suppose that both N2Ws start their run using some initial rules r_1, r_2 i. e. the right-hand side of r_1, r_2 is an initial state. Furthermore, we ensure that they read the same input (op, a) and put some stack symbol γ_1, γ_2 on the stack. They produce some output u_1 and u_2, respectively. After this step $\mathcal{N}_1$ is in state q_1 and $\mathcal{N}_2$ is in state q_2. If the parallel run ever terminates, $\mathcal{N}_1$ has to use a rule that removes γ_1 from the stack and reads (cl, a) from the input. $\mathcal{N}_2$ has to read the same input and removes γ_2 from the stack. The destination states q_1', q_2' of the rules r_1', r_2' have to be final states. One production of the second and first set of productions represents the parallel outgoing (using (r_1, r_2)) and incoming (using (r_1', r_2')) traversal at a node $f \in \Sigma$ in the tree t. In between, there has to be a parallel run, that first handles the left most

child and last handles the right most child of f. Without the third set of productions we could only generate successful runs accepting paths, displayed above.

f —— g —— g —— a

This changes due to the third set of productions. We may derive a sequence of non-terminals by starting from the non-terminal $((p_1, q_1), (p_2, q_2))$. At some point in the derivation, non-terminals disappear due to the set of productions introduced lastly.

Remark 6.8. We only add any grammar production, in the construction above, if for each non-terminal $((p_1, p_1'), (p_2, p_2'))$, p_1' is reachable from p_1 in $\mathcal{N}_1$, p_2' is reachable from p_2 in $\mathcal{N}_2$ and $(p_1, p_2), (p_1', p_2') \in Co(\mathcal{N}_1, \mathcal{N}_2)$. This reduces the runtime and the memory consumption of the construction for many instances. In the last step we delete all useless non-terminals and productions.

6.3. Equivalence test for STWs

Next we show how we can transform a STW $\mathcal{S}$ into a top-down dN2W $\mathcal{N}$ such that $[\![\mathcal{S}]\!] = [\![\mathcal{N}]\!]$. Since unranked trees comprise a subset of ranked trees, the reduction is straightforward. Basically, the transformation changes the traversal of the tree from top-down to pre-order.

Lemma 6.8.1 ([52]). *Let $\mathcal{S}$ be a* STW. *We can convert $\mathcal{S}$ to a d*N2W *$\mathcal{N}$ with $[\![\mathcal{S}]\!] = [\![\mathcal{N}]\!]$ in $\mathcal{O}(|\mathcal{S}| \cdot n)$ time, where $n = \max\{\text{rank}(f) \mid f \in \Sigma\}$.*

Proof (sketch). Let $\mathcal{S} = (\Sigma, \Delta, Q_{\mathcal{S}}, Q_{I,\mathcal{S}}, \delta_{\mathcal{S}})$ be the STW. First we extend the rank function to $\delta_{\mathcal{S}}$ in the following way: $\text{rank}(q(f) \to \alpha) = \text{rank}(f)$. Let $\mathcal{N} = (\Sigma, \Delta, Q_{\mathcal{N}}, \Gamma_{\mathcal{N}}, R_{\mathcal{N}}, Q_{I,\mathcal{N}}, Q_{F,\mathcal{S}})$ be the constructed dN2W, such that

$$Q_{\mathcal{N}} = \Gamma_{\mathcal{S}} = \{(r, j) \mid r \in \delta_{\mathcal{S}}, 0 \leq j \leq \text{rank}(r)\} \cup \{\mathsf{o}, \mathsf{f}\},$$

$Q_{I,\mathcal{N}} = \{\mathsf{o}\}, Q_{F,\mathcal{N}} = \{\mathsf{f}\}$ and $R_{\mathcal{N}}$ consists of the following rules:

$$\frac{q_0 \in Q_{I,\mathcal{N}} \qquad r = q_0(g) \to u_0 q_1 \dots q_n u_n}{\mathsf{o} \xrightarrow{\mathsf{op}\, g/u_0:\mathsf{f}} (r, 0) \qquad (r, n) \xrightarrow{\mathsf{cl}\, g/\epsilon:\mathsf{f}} \mathsf{f}}$$

$$\frac{r = q(g) \to u_0 q_1 \dots q_j u_j \dots q_n u_n \qquad r' = q_j(b) \to v_0 p_1 \dots p_m v_m \quad 1 \leq j \leq n}{(r, j-1) \xrightarrow{\mathsf{op}\, b/v_0:(r,j)} (r', 0) \qquad (r', m) \xrightarrow{\mathsf{cl}\, b/u_j:(r,j)} (r, j)}$$

■

We have $|Q_{\mathcal{N}}| = \mathcal{O}(|\delta_{\mathcal{S}}| \cdot n) = \mathcal{O}(|\mathcal{S}|)$ states and $|R_{\mathcal{N}}| = \mathcal{O}(|\mathcal{S}|n)$ rules contained in the constructed dN2W.

Example 6.9. Let $\delta_{\mathcal{S}} = \{r_1 = h(q_0) \to u_0q_1u_1q_0u_2, r_2 = g(q_1) \to v_0q_2v_1, r_3 = a(q_2) \to w_0, r_4 = a(q_0) \to y_0\}$, with $O_{I,\mathcal{S}} = \{q_0\}$, then the rules $R_{\mathcal{N}}$ are the followings:

$$
\begin{aligned}
r'_1 = \mathsf{o} &\xrightarrow{\mathsf{op}\, h/u_0:\mathsf{f}} (r_1, 0), & r'_2 = (r_1, 2) &\xrightarrow{\mathsf{cl}\, h/\epsilon:\mathsf{f}} \mathsf{f},\\
r'_3 = (r_1, 0) &\xrightarrow{\mathsf{op}\, g/v_0:(r_1,1)} (r_2, 0), & r'_4 = (r_2, 1) &\xrightarrow{\mathsf{cl}\, g/u_1:(r_1,1)} (r_1, 1),\\
r'_5 = r(r_1, 1) &\xrightarrow{\mathsf{op}\, h/u_0:(r_1,2)} (r_1, 0), & r'_6 = (r_1, 2) &\xrightarrow{\mathsf{cl}\, h/u_2:(r_1,2)} (r_1, 2),\\
r'_7 = (r_1, 1) &\xrightarrow{\mathsf{op}\, a/y_0:(r_1,2)} (r_4, 0), & r'_8 = (r_4, 0) &\xrightarrow{\mathsf{cl}\, a/u_2:(r_1,2)} (r_1, 2),\\
r'_9 = (r_2, 0) &\xrightarrow{\mathsf{op}\, a/w_0:(r_2,1)} (r_3, 0), & r'_{10}(r_3, 0) &\xrightarrow{\mathsf{cl}\, a/v_1:(r_2,1)} (r_2, 1),
\end{aligned}
$$

Suppose we read the tree $t = h(g(a), h(g(a), a)))$. We get

$$[\![\mathcal{S}]\!](t) = u_0v_0w_0v_1u_1u_0v_0w_0v_1u_1y_0u_2u_2$$

and the corresponding successful run of $\mathcal{N}$ on t is

$$s = r'_1r'_3r'_9r'_{10}r'_4r'_5r'_3r'_9r'_{10}r'_4r'_7r'_8r'_6r'_2.$$

Theorem 6.10. *Let $\mathcal{S}_1 = (\Sigma, \Delta, Q_{\mathcal{S}_1}, Q_{I,\mathcal{S}_1}, \delta_{\mathcal{S}_1})$ and $\mathcal{S}_2 = (\Sigma, \Delta, Q_{\mathcal{S}_2}, Q_{I,\mathcal{S}_2}, \delta_{\mathcal{S}_2})$ be two* STWs. *We can decide whether $\mathcal{S}_1, \mathcal{S}_2$ are equivalent in*

$$\mathcal{O}(n^6 \cdot (n \cdot |G_L|)^2 \log(n \cdot |G_L|))$$

time, where $n = |\mathcal{S}_1|^3 \cdot |\mathcal{S}_2|^3$ and G_L is the largest SLP *representing an output word that is contained in the rules of $\mathcal{S}_1, \mathcal{S}_2$.*

Proof. We first check in $\mathcal{O}(|\mathcal{S}_1| + |\mathcal{S}_1|)$ time, if the domains are the same by using theorem 6.2. However, $\mathcal{O}(|\mathcal{S}_1| + |\mathcal{S}_1|) \cdot \Sigma) = \mathcal{O}(n)$, since $\mathcal{S}_1, \mathcal{S}_2$ are deterministic. If the domains coincide, we construct two dN2Ws $\mathcal{N}_1, \mathcal{N}_2$ by using the transformation in lemma 6.8.1 in $\mathcal{O}((|\mathcal{S}_1| + |\mathcal{S}_2|) \cdot \max\{\mathrm{rank}(f) \mid f \in \Sigma\})$ time, otherwise we can return false.

In the next step we construct the grammar of successful parallel runs given by lemma 6.7.1 in $\mathcal{O}(|Q_{\mathcal{N}_1}| \cdot |Q_{\mathcal{N}_2}| \cdot \Sigma + |Q_{\mathcal{N}_1}|^3 \cdot |Q_{\mathcal{N}_2}|^3) = \mathcal{O}(n)$ time. Since the dN2Ws are deterministic, we can replace $|\mathcal{N}_1|^2 \cdot |\mathcal{N}_1|^2$ in the equation of lemma 6.7.1 by $|Q_{\mathcal{N}_1}| \cdot |Q_{\mathcal{N}_2}| \cdot \Sigma$. Note that for each q and $a \in \Sigma$ there is at most one rule r with reading (op, a) in both dN2Ws and one reading rule (cl, a) which fits to the the stack symbol $\mathrm{ssy}(r)$. The resulting grammar G has $\mathcal{O}(n)$ productions.

Furthermore, we construct μ_1, μ_2 defined in 6.7. The size of a grammar of a word generated by a Plandowski path is in $\mathcal{O}(n \cdot |G_L|)$, where G_L is the largest SLP representing a output word contained in a rule of $\delta_{\mathcal{S}_1}, \delta_{\mathcal{S}_2}$. Since G has $\mathcal{O}(n)$ productions the statement follows by using theorem 5.21. ■

Remark 6.11 (Combinatorial explosion). We don't know whether the set of non-terminals of SLPs representing output words are disjoint or not. Therefore, we may have to rename all non-terminals in all SLPs beforehand. However, this can easily be done in $\mathcal{O}(|\mathcal{S}_1| + |\mathcal{S}_2|)$. The runtime for the equivalence check of STWs is clearly polynomial but the polynomial has a large exponent. Large problem instances that has a large variety of parallel runs requires a very long running time. However, if $[\![\mathcal{S}_1]\!] = [\![\mathcal{S}_2]\!]$ does not hold, we might get a quick answer.

6.4. Equivalence test for LTWs

Finally, we have all tools in place to implement an algorithm for solving the equivalence problem for linear tree-to-word transducers.

Theorem 6.12 ([9]). *The following problem is in* PTIME.

Input: *Given two* LTWs $\mathcal{L}_1, \mathcal{L}_2$ *with the same input* Σ *and output* Δ *alphabet.*

Output: *Does* $[\![\mathcal{L}_1]\!] = [\![\mathcal{L}_2]\!]$ *hold?*

LTWs are from a more general class than STWs. They are non-copying but not necessarily order-preserving i. e. the output might be re-ordered. From the periodicity lemma (see lemma 5.23.1) we derive that if two LTWs are equivalent, the difference in the order can't be too large.

6.4.1. Same-ordered LTWs

Let's first look at a special case of $\mathcal{L}_1, \mathcal{L}_2$ being same-ordered (see definition 6.14). One can think of same-ordered in the sense that whenever $\mathcal{L}_1$ re-orders the output, $\mathcal{L}_2$ does it in the same way. Intuitively, if two LTWs generate the same output at each node of an input tree, and additionally re-order their output in the same way, they should be equal. Fortunately, this was proven to be true by [9]. Therefore, it suffice to ignore the re-ordering for the equivalence test of same-ordered LTWs i. e. we interpret both LTWs as STWs. We show that we can test whether two LTWs $\mathcal{L}_1, \mathcal{L}_2$ are same-ordered in $\mathcal{O}(|\mathcal{L}_1| \cdot |\mathcal{L}_2|)$ time (see lemma 6.14.1). Consequently, the complexity of solving the equivalence problem for same-ordered LTWs is equal to the complexity of solving the equivalence problem for two STWs containing the same rules.

Definition 6.13 (Co-reachable states). Let $\mathcal{L}_1 = (\Sigma, \Delta, Q_1, Q_{1,I}, \delta_1)$, $\mathcal{L}_2 = (\Sigma, \Delta, Q_2, Q_{2,I}, \delta_2)$ be two LTWs. We define the relation $\mathcal{C}o(\mathcal{L}_1, \mathcal{L}_2) \subseteq Q_1 \times Q_2$ by the following properties:

$$\frac{q_1 \in Q_{1,I} \qquad q_2 \in Q_{2,I}}{(q_1, q_2) \in \mathcal{C}o(\mathcal{L}_1, \mathcal{L}_2)}$$

$$\frac{(q_1, q_2) \in \mathcal{C}o(\mathcal{L}_1, \mathcal{L}_2) \qquad \begin{matrix} f(q_1) \to u_0 q_1'(x_{\sigma_1(1)}) \dots q_n'(x_{\sigma_1(n)}) u_n \in \delta_1 \\ f(q_2) \to v_0 q_1''(x_{\sigma_2(1)}) \dots q_n''(x_{\sigma_1(n)}) v_n \in \delta_2 \end{matrix} \qquad \sigma_1(i) = \sigma_2(j)}{(q_i', q_j'') \in \mathcal{C}o(\mathcal{L}_1, \mathcal{L}_2)}$$

Lemma 6.13.1. *Let $\mathcal{L}_1 = (\Sigma, \Delta, Q_1, Q_{1,I}, \delta_1), \mathcal{L}_2 = (\Sigma, \Delta, Q_2, Q_{2,I}, \delta_2)$ be two* LTWs*. Algorithm 26 Computes $\mathcal{C}o(\mathcal{L}_1, \mathcal{L}_2)$ in $\mathcal{O}(|Q_1| \cdot |Q_2| \cdot \Sigma \cdot \max\{\mathrm{rank}(f) \mid f \in \Sigma\}) = \mathcal{O}(|\mathcal{L}_1| \cdot |\mathcal{L}_2|)$ time.*

Proof. For each LTW we keep rules in an appropriate data structure and therefore, supports $\forall i \in \{1, 2\}$ a mapping $\Sigma \times Q_i \to \delta_i$ to get access to a rule with a specific left-hand side in constant time. In each round of the loop, in line 7, we remove one pair from the heap and put new elements into the heap. However, no pair (q_1, q_2) will be added twice. There are at most $|Q_1| \cdot |Q_2|$ pairs. By using the data structure, one iteration of the outer loop requires $\mathcal{O}(\Sigma \cdot \max\{\mathrm{rank}(f) \mid f \in \Sigma\})$ time. Which proves the running time. The correctness follows by the constructive definition of co-reachable states, since GETCOREACHABLES realises exactly this recursive construction. ■

Algorithm 26: GETCOREACHABLES: Computes the relation $\mathcal{C}o(\mathcal{L}_1, \mathcal{L}_2)$.

input : STWs $\mathcal{L}_1 = (\Sigma, \Delta, Q_1, Q_{1,I}, \delta_1)$, $\mathcal{L}_2 = (\Sigma, \Delta, Q_2, Q_{2,I}, \delta_{\mathcal{L}_2})$;
output: $\mathcal{C}o(\mathcal{L}_1, \mathcal{L}_2)$;

```
1  H ← ∅;
2  Co ← ∅;
3  for q_1 ∈ Q_{1,I} do
4      for q_2 ∈ Q_{2,I} do
5          Co ← Co ∪ (q_1, q_2);
6          H ← H ∪ (q_1, q_2);
7  while H ≠ ∅ do
8      remove (q_1, q_2) from the heap H;
9      foreach rule of the form f(q_1) → u_1 q_1'(x_{σ_1(1)}) ... q_n'(x_{σ_1(n)}) u_n do
10         foreach rule of the form f(q_2) → v_1 q_1''(x_{σ_2(1)}) ... q_n''(x_{σ_2(n)}) v_n do
11             foreach i ∈ [n] do
12                 if (q'_{σ_1(i)}, q''_{σ_2(i)}) ∉ Co then
13                     Co ← Co ∪ (q'_{σ_1(i)}, q''_{σ_2(i)});
14                     H ← H ∪ (q'_{σ_1(i)}, q''_{σ_2(i)});
15 return Co;
```

Definition 6.14 (Same-ordered [9])**.** Two LTWs $\mathcal{L}_1, \mathcal{L}_2$ are same-ordered if for each pair of co-reachable states $(q_1, q_2) \in \mathcal{C}o(\mathcal{L}_1, \mathcal{L}_2)$ and for each symbol $f \in \Sigma$ neither q_1 nor q_2 have a rule for f, or the permutation for these two rules is identical i. e. $\sigma_1 = \sigma_2$.

Lemma 6.14.1. *Let $\mathcal{L}_1, \mathcal{L}_2$ be two* LTWs*. We can decide whether these* LTWs *are same-ordered in $\mathcal{O}(|\mathcal{L}_1| \cdot |\mathcal{L}_2|)$ time.*

Proof. In the first step we compute $\mathcal{C}o(\mathcal{L}_1, \mathcal{L}_2)$ in $\mathcal{O}(|\mathcal{L}_1| \cdot |\mathcal{L}_2|)$ time. By the assumption that we can get access to a rule by its left-hand side in $\mathcal{O}(1)$, we can check for each pair $(q_1, q_2) \in \mathcal{C}o(\mathcal{L}_1, \mathcal{L}_2)$, in $\mathcal{O}(|\Sigma| \cdot \max_{f \in \Sigma} \operatorname{rank}(f))$ time, whether the same-ordered condition holds for these pairs or not. If at some point the condition does not hold, we can report that the LTWs aren't same-ordered. We just have to check whether for each symbol the permutations of the two (due to determinism) corresponding rules are the same i. e. $\sigma_1 = \sigma_2$. Consequently, the total running time is $\mathcal{O}(|\mathcal{L}_1| \cdot |\mathcal{L}_2| + |\mathcal{C}o(\mathcal{L}_1, \mathcal{L}_2)| \cdot |\Sigma| \cdot \max\{\operatorname{rank}(f) \mid f \in \Sigma\}) = \mathcal{O}(|\mathcal{L}_1| \cdot |\mathcal{L}_2|)$. ■

Lemma 6.14.2. *Let $\mathcal{L}_1 = (\Sigma, \Delta, Q_1, Q_{I,1}, \delta_1)$ and $\mathcal{L}_2 = (\Sigma, \Delta, Q_2, Q_{I,2}, \delta_2)$ be two same-ordered* LTWs*. We can decide whether $\mathcal{L}_1, \mathcal{L}_2$ are equivalent in*

$$\mathcal{O}(n^6 \cdot (n \cdot |G|)^2 \log(n \cdot |G_L|))$$

time, where $n = (|\mathcal{L}_1|^3 \cdot |\mathcal{L}_2|^3)$ and G_L is the largest SLP *representing an output word that is contained in the rules of $\mathcal{L}_1, \mathcal{L}_2$.*

Proof. Follows directly from the fact that we can interpret these LTWs as STWs combined with lemma 6.14.1 and theorem 6.10. ■

6.4.2. Earliest LTWs

The earliest form of STWs was introduced in [38] and extended to LTWs by [9]. Transforming STWs into their earliest form and minimizing them gives us a canonical form which leads to the Myhill-Nerode theorem for the class of transformations definable with STWs. However, the transformations may costs us an exponential blow-up in the size of the output word of a rule or the numbers of states which we have to avoid. Furthermore, we will see that the earliest form does lead directly to a normal form for LTWs. However, with a little additional work we can overcome this problem.

Definition 6.15 (Earliest STW and LTW [38])**.** A linear or sequential tree-to-word transducer is *earliest* if and only if the following two conditions are satisfied:

(i) for every state q that is not an initial state we have $\operatorname{lcp}(L_q) = \operatorname{lcs}(L_q) = \epsilon$,

(ii) for every rule $f(q) \to u_0 q_1 \dots q_n u_n \in \delta$ and $1 \leq i \leq n$ we have $\operatorname{lcp}(L_{q_i} u_i) = \epsilon$.

Intuitively, (i) ensures that no factor of the output can be pushed up and (ii) ensures that no factor can be pushed left in the traversal [38]. Suppose $\operatorname{lcp}(L_q) \neq \epsilon$. This means that there exists a word that is the prefix of each word generated by the state q recognizing any tree $t \in \operatorname{dom}(q)$ i. e. $w \sqsubseteq [\![\mathcal{L}]\!](t)$. Consequently, w does not depend on the input tree t and therefore, we could replace each occurrence of q in any right-hand side of $\mathcal{L}$ by wq and change $\mathcal{L}$ in such a way that $\operatorname{lcp}(L_q) = \epsilon$. The following key lemma shows that

for two equivalent earliest LTWs the order of rules, directly accessible from equivalent states, can't differ too much.

Lemma 6.15.1 (Key lemma [9]). *Let $\mathcal{L}$ and $\mathcal{L}'$ be two earliest* LTWs. *Let q be a state of $\mathcal{L}$ and q' be a state of $\mathcal{L}'$ such that $[\![\mathcal{L}]\!]_q = [\![\mathcal{L}']\!]_{q'}$. Let*

$$q(f) \to u_0 q_1(x_{\sigma(1)}) \dots q_n(x_{\sigma(n)}) u_n$$
$$q'(f) \to v_0 q'_1(x_{\sigma'(1)}) \dots q'_n(x_{\sigma'(n)}) v_n$$

be two rules in $\mathcal{L}$ and $\mathcal{L}'$, receptively, i the first index such that $\sigma(i) \neq \sigma'(i)$, and j such that $\sigma(j) = \sigma'(i)$. Then for k such that $i \leq k \leq j$, all L_{q_k} are periodic of the same period and for k' such that $i \leq k' < j$, $u_{k'} = \epsilon$. Moreover, we have $[\![\mathcal{L}']\!]_{q'_i} = [\![\mathcal{L}]\!]_{q_j}$

Proof. Since the LTWs are earliest, we get the following situation

$$L_{q_i} u_i \dots L_{q_j} u_j \dots L_{q_n} u_n = L_{q'_i} v_i \dots L_{q'_{j'}} v_{j'} \dots L_{q'_n} v_n \tag{6.1}$$

such that $\sigma(i) = \sigma'(j')$ and $\sigma(j) = \sigma'(i)$. Suppose $L_{q'_i} = \{\epsilon\}$, then we could change the order of the states such that $\sigma(i) = \sigma(j)$. So suppose there is a word $\omega \in L_{q'_i}$ with $\omega \neq \epsilon$. Since $\sigma(i) \neq \sigma'(i)$, the languages are generated by accepting different subtrees of a input tree and therefore, it follows that

$$\omega \sqsubseteq L_{q_i} \text{ or } L_{q_i} \sqsubseteq \omega. \tag{6.2}$$

Hence, $\exists \omega' \in L_{q'_i}$ that is the prefix of each word in $L_{q'_i}$. By the earliest condition we have $\mathrm{lcp}(L_{q'_i}) = \epsilon$, thus $\epsilon \in L_{q'_i}$. Since $\epsilon \in L_{q_i}$, we get that

$$\omega \sqsubseteq u_i \text{ or } u_i \sqsubseteq \omega. \tag{6.3}$$

Assume $u_i \neq \epsilon$. By using eq. (6.3) and the fact that $\mathrm{lcp}(L_{q_i} \setminus \{\epsilon\}) \neq \epsilon$ (see eq. (6.2)) we would get $\mathrm{lcp}(L_{q_i} u_i) \neq \epsilon$ which can not be the case since $\mathcal{L}$ is in earliest normal form. Assume now for all l' with $i \leq l' \leq l < j, \epsilon \in L_{q'_l}$. Let's look at $L_{q_{l+1}}$. By the reordering we get that the languages $L_{q_i} \dots L_{q_{l+1}}$ are independent from $L_{q'_i}$. Furthermore, we know that $\forall i \leq l' \leq l \; u_{l'} = \epsilon$ holds. The situation is the following

$$\epsilon\, \epsilon \dots L_{q_{l+1}} u_{l+1} \dots L_{q_j} u_j \dots L_{q_n} u_n = L_{q'_i} v_i \dots L_{q'_{j'}} v_{j'} \dots L_{q'_n} v_n \iff$$
$$L_{q_{l+1}} u_{l+1} \dots L_{q_j} u_j \dots L_{q_n} u_n = L_{q'_i} v_i \dots L_{q'_{j'}} v_{j'} \dots L_{q'_n} v_n$$

Which is the same situation as shown in eq. (6.1). Which implies that $\epsilon \in L_{q_{l+1}}$ and $u_{l+1} = \epsilon$. If we consider the languages $L_{q'_l}$ for $i \leq l \leq j'$, the argument is symmetric. Overall we get

$$\begin{aligned} \forall l, i \leq l \leq j : \epsilon \in L_{q_l} \;\wedge\; u_l = \epsilon \\ \forall l, i \leq l \leq j' : \epsilon \in L_{q'_l} \;\wedge\; v_l = \epsilon. \end{aligned} \tag{6.4}$$

Now we fix an input tree $t = f(t_1, \dots, t_n)$ and we look at the part of the word generated by the rules i. e.

$$[\![\mathcal{L}]\!]_{q_i}(t_{\sigma(i)}) u_i [\![\mathcal{L}]\!]_{q_{i+1}}(t_{\sigma(i+1)}) \dots [\![\mathcal{L}]\!]_{q_i}(t_{\sigma(n)}) u_n =$$
$$[\![\mathcal{L}']\!]_{q'_i}(t_{\sigma'(i)}) v_i [\![\mathcal{L}']\!]_{q'_{i+1}}(t_{\sigma'(i+1)}) \dots [\![\mathcal{L}']\!]_{q'_{j'}}(t_{\sigma'(j')}) v_{j'} \dots [\![\mathcal{L}']\!]_{q'_i}(t_{\sigma'(n)}) v_n$$

We can replace the subtree $t_{\sigma(i)} = t_{\sigma'(j')}$ by any other tree $t' \in \mathrm{dom}(q_i) = \mathrm{dom}(q'_{j'})$ and get for all these subtrees

$$[\![\mathcal{L}]\!]_{q_i}(t')u_i[\![\mathcal{L}]\!]_{q_{i+1}}(t_{\sigma(i+1)}) \dots [\![\mathcal{L}]\!]_{q_i}(t_{\sigma(n)})u_n =$$
$$[\![\mathcal{L}']\!]_{q'_i}(t_{\sigma'(i)})v_i[\![\mathcal{L}']\!]_{q'_{i+1}}(t_{\sigma'(i+1)}) \dots [\![\mathcal{L}']\!]_{q'_{j'}}(t')v_{j'} \dots [\![\mathcal{L}']\!]_{q'_i}(t_{\sigma'(n)})v_n$$

which implies that for all these trees t' we have

$$|[\![\mathcal{L}]\!]_{q_i}(t')| = |[\![\mathcal{L}']\!]_{q'_{j'}}(t')|.$$

By eq. (6.4), there are trees generating the following languages:

$$u_0L_{q_1} \dots L_{q_{i-1}}u_{i-1}L_{q_i} \dots L_{q_n}u_n =$$
$$v_0L_{q'_1} \dots L_{q'_{i-1}}v'_{i-1}L_{q'_{j'}} \dots L_{q_n}u_n$$

i. e. we choose for $t_l \in \mathrm{dom}(q'_l)$ such that $[\![\mathcal{L}']\!]_{q'_l}(t_l) = \epsilon$ for all $i \leq l < j'$. Furthermore, we have $v_k = u_k$ and $L_{q_{k+1}} = L_{q'_{k+1}}$ for all $0 \leq k < i$. If we look at one word we get

$$wz = wz' \tag{6.5}$$

with $z \in L_{q_i}$ and $z' \in L_{q'_{j'}}$. Since $|z| = |z'|$, $z = z'$ follows and therefore, we get $\forall t \in \mathrm{dom}(q_i) = \mathrm{dom}(q'_{j'}) : [\![\mathcal{L}]\!]_{q_i}(t) = [\![\mathcal{L}']\!]_{q'_{j'}}(t)$ which is equal to

$$[\![\mathcal{L}]\!]_{q_i} = [\![\mathcal{L}']\!]_{q'_{j'}}.$$

By symmetry we get

$$[\![\mathcal{L}]\!]_{q_j} = [\![\mathcal{L}']\!]_{q'_i}.$$

Again we fix for all l with $i \leq l \leq j$ and for all l' with $i \leq l' \leq j', t_l \in \mathrm{dom}(q_l), t_{l'} \in \mathrm{dom}(q'_{l'})$ such that $[\![\mathcal{L}]\!]_{q_l}(t_l) = \epsilon$ and $[\![\mathcal{L}']\!]_{q'_{l'}}(t_{l'}) = \epsilon$. This allows us to isolate certain languages. We can deduce that for all $w \in L_{q_i} = L_{q'_{j'}}$ and $w' \in L_{q_j} = L_{q'_i}$

$$ww' = w'w$$

which implies by using lemma 5.23.1 that L_{q_i} and L_{q_j} are periodic with the same period. We can further follow that for all l with $i \leq l \leq j$, L_l are periodic with the same period. ■

The lemma says that, due to the periodicity of the languages, we can change the order of states causing difference in the order of a rule for a state q, q' with $[\![\mathcal{L}]\!]_q = [\![\mathcal{L}]\!]'_q$ without changing the output language. Furthermore, only periodic languages can cause order difference in equivalent earliest LTWs. This means we can assume that the rules of equivalent states in two earliest LTWs are same-ordered [9]. By using the earliest property we even get the following theorem.

Theorem 6.16 ([9]). *Let $\mathcal{L}$ and $\mathcal{L}'$ be two equivalent earliest* LTWs*, q be a state of $\mathcal{L}$, q' be a state of $\mathcal{L}'$ and $(q, q') \in Co(\mathcal{L}, \mathcal{L}')$, then $[\![\mathcal{L}]\!]_q = [\![\mathcal{L}']\!]_{q'}$.*

If two equivalent LTWs are in earliest normal form and they are not same-ordered, their order can only differ by periodic or erasing states i. e. states q with $L_q = \{\epsilon\}$. We could solve the equivalence problem for LTWs by the following steps:

1. We transform both LTWs into earliest normal from.
2. We re-order periodic and erasing states by applying lemma 6.15.1.
3. We test if both LTWs are same-ordered. If this does not hold, we can report that they are not equals. Otherwise, we use lemma 6.14.1 to test if they are equivalent.

However, the transformation into earliest normal form may cost us an exponential blow-up in the size of the LTWs. Fortunately, in a non-earliest LTW the corresponding states might not be periodic but they have to have a somehow weaker property called quasi-periodicity.

Definition 6.17 (Quasi-periodic languages and states [9]). A language L is *quasi-periodic on the left* (respectively *on the right*) of handle w and period u if $L \subseteq w(u)^*$ (respectively $L \subseteq (u)^* w$). A language is *quasi-periodic*, if it is quasi-periodic on the left or on the right. L is *proper* quasi-periodic, if it is quasi-periodic and not periodic. A state q is *quasi-periodic*, if and only if L_q is quasi-periodic. Furthermore, qu with $u \in \Delta^*$ is quasi-periodic if and only if $L_q u$ is quasi-periodic.

[9] showed that it is sufficient to replace step 1 by ensuring that any proper quasi-periodic state is earliest. Furthermore, they showed that this transformation is in PTIME by presenting the construction steps. We followed their construction and implemented each step by combining all algorithms we already discussed.

6.4.3. LTWs in partial normal form

In [9] the authors defined the so called partial normal form. Intuitively, a LTW is in partial normal form after applying the three steps above. However, one have to define the exact re-ordering such that both LTWs are same-ordered, if they are equivalent.

Definition 6.18 (Partial normal form [9]). We say that a LTW is in partial normal form, if

(i) it contains no proper quasi-periodic states, (i. e. proper quasi-periodic states will be transformed to being earliest by pushing the handle up)

(ii) for each $f(q) \to u_0 q_1(x_{\sigma(1)}) \dots q_n(x_{\sigma(n)}) u_n$ if $L_{q_i} = \{\epsilon\}$ then for all j with $i \leq j \leq n$ we have $L_{q_j} = \{\epsilon\}$ and $u_j = \epsilon$. (erasing states are at the rightmost position of a rule and they are ordered)

(iii) if $q_i(x_{\sigma(i)})u_iq_{i+1}(x_{\sigma(i+1)})$ is part of a rule and $L_{q_i}u_iL_{q_{i+1}}$ is quasi-periodic then we have $\sigma(i) < \sigma(i+1)$ (intuitively, we push the handle to the left).

Lemma 6.18.1 ([9]). *Let $\mathcal{L}$ and $\mathcal{L}'$ be two* LTWs. *If $\mathcal{L}$ and $\mathcal{L}'$ are equivalent and in partial normal form then they are same-ordered.*

To solve the equivalence problem for two arbitrary LTWs, we reduce the problem to the case of same-ordered LTWs by transforming both LTWs into LTWs in partial normal form. By lemma 6.18.1 the resulting LTWs are same-ordered and we can apply 6.14.2. It is essential that we can test, if a state is quasi-periodic. Surprisingly, the quasi-periodic test and the algorithm for replacing quasi-periodic states are almost the same. In the remaining part of this chapter we describe the construction of the partial normal form.

6.4.4. Transforming LTWs into partial normal form

Let $\rho_n(u) = \rho_n^l(u)$ realises the right to left shift of a word v that is a prefix of u defined by definition 3.10 i. e. $\rho_n^l(u) = v^{-1}uv$ with $|v| = n$. We can extend this operation to work with $n > |u|$ by $\rho'_n(u) = \rho_{(n \mod |u|)}(u)$. For the rest of the chapter we write ρ_n instead of ρ'_n. Note that the left to right shift of a suffix of size n is equal to $\rho_{|u|-n}(u) = \rho_n^r[u]$. We already describe in algorithm 9 how we can perform a shift for a fully-compressed word in $\mathcal{O}(|G|)$ where G is the grammar of the SLP representing the word.

Definition 6.19 (Shift of a period [9]). If two states q_1, q_2 are of period $u = u_1u_2$ and $u' = u_2u_1$, respectively, then we note the *shift in their period* by $\text{shift}(q_1, q_2) = |u_1|$.

Example 6.20. Let us look to a context-free grammar G which is the grammar produced by some LTW defined by the productions on the left. In the following table you can find the languages, the period, the handle and the shift with respect to S for all non-terminals contained in the grammar.

$S \to abXabcZb$, $Z \to aV$,
$X \to cabX$, $X \to c$,
$V \to Vbca$, $V \to bca$

Language	Period	Handle	$\text{shift}(S, x)$
$L_S(G) = ab(cab)^*$	cab	ab	0
$L_X(G) = (cab)^*c$	abc	c	1
$L_Z(G) = a(bca)^+$	bca	a	2
$L_V(G) = (bca)^+$	bca	ϵ	2

If we only have the information of S being quasi-periodic with period cab, then we can follow, just by looking at the production $S \to abXabcZb$, that all words in $L_X(G)$ have to end with c and all words in $L_Z(G)$ have to start and end with a. Otherwise, S would not be of period cab.

The example 6.20 above gives the intuition of the following two lemmas stated and proved in [9]. Together they gave us an algorithm to transform a quasi-periodic state into a state that is earliest.

Lemma 6.20.1 ([9]). *Let q be a state that is quasi-periodic on the left with period u and let $q(f) \to u_0 q_1(x_{\sigma(1)}) \dots q_n(x_{\sigma(n)}) u_n$ be a rule for q, then each q_i is either quasi-periodic of period ϵ or $\rho_{\mathrm{shift}(q,q_i)}[u]$, where $\mathrm{shift}(q, q_i) \equiv |u_i w_i \dots w_n u_n| \pmod{|u|}$ and w_i is the smallest word of L_{q_i}.*

Remark 6.21. If q is *quasi-periodic on the right*

$$\mathrm{shift}(q, q_i) \equiv |u| - |u_0 w_0 \dots w_{i-1} u_{i-1}| \pmod{|u|}.$$

Note, that we defined the shift operation $\rho_n(u)$ for $n > |u|$ using the modulo operation. Thus, we can forget about the modulo in the calculation of the shift. However, it might be useful to keep numbers small.

Lemma 6.21.1 ([9]). *If q is a quasi-periodic state with period u of a* LTW *$\mathcal{L}$ and q' is accessible from q in $\mathcal{L}$, then q' is quasi-periodic with period ϵ or a shift of u. Moreover, we can calculate the $\mathrm{shift}(q, q')$ in polynomial time.*

The intuition should be clear by looking at example 6.20. For a complete proof we refer to [9]. Let us assume q is quasi-periodic on the left i. e. $L_q \subseteq wu^*$. Lemma 6.20.1 and 6.21.1 tells us how we can transform q into a state q^e that is periodic. We remove the handle from L_q. We do this by pushing all $\mathrm{lcp}(q_i)$ of states q_i in a rule of q' accessible from q up and to the left and remove the prefix $\mathrm{lcp}(q')$ from the pushed factor. Remember, q is quasi-periodic and q' is quasi-periodic with period ϵ or a shifted version of u. Consequently, the generated word is $\mathrm{lcp}(q')$ or $\mathrm{lcp}(q')w$ where w is a shifted version of u. Since we can calculate the shift, we can transform w into u. All these transformations do not change $[\![\mathcal{S}]\!]$ if q is quasi-periodic on the left.

Example 6.22. Let us look back to example 6.20 to the rule $X \to cabX$. Let's assume we transform S such that we replace it by some non-terminals that are earliest. Therefore, we would apply the following transformation on the rule $X \to cabX$:

$$\begin{aligned}
&X \to X\rho^r_{\mathrm{shift}(S,X)}[cab\,\mathrm{lcp}(L_X(G))\,\mathrm{lcp}(L_X(G))^{-1}] && \iff\\
&X \to X\rho^r_1[cabcc^{-1}] && \iff\\
&X \to X\rho^r_1[abc] && \iff\\
&X \to Xcab.
\end{aligned}$$

The transformation for the rule $V \to Vbca$ would be the following

$$\begin{aligned}
&V \to \rho^l_{\mathrm{shift}(S,V)}[\mathrm{lcp}(L_V(G))^{-1} bca\,\mathrm{lcp}(L_V(G))]V && \iff\\
&V \to \rho^l_1[(bca)^{-1} bcabca]V && \iff\\
&V \to \rho^l_1[bca]V && \iff\\
&V \to cabV.
\end{aligned}$$

If q is not quasi-periodic, $\mathrm{shift}(q, q')$ is not defined. However, for the quasi-periodic test we require a functions $\mathrm{shift}_l(q, q'), \mathrm{shift}_r(q, q')$ that are defined for all pairs of states,

computable without knowing that q is quasi-periodic and coincident with $\mathrm{shift}(q, q')$ such that $\mathrm{shift}_l(q, q') = \mathrm{shift}(q, q')$, if q is quasi-periodic on the left and $\mathrm{shift}_r(q, q') = \mathrm{shift}(q, q')$ if q is quasi-periodic on the right. In [9] the authors introduce the so called mock-shifts $\mathrm{shift}_l, \mathrm{shift}_r$. We use a slightly different definition than [9], replacing $\mathrm{lcp}(q_i)$ by the smallest word of L_{q_i}.

Definition 6.23 (Mock shift)**.** Let $\mathcal{L} = (\Sigma, \Delta, Q, Q_I, \delta)$ be a LTW. We define $\mathrm{shift}_l, \mathrm{shift}_r : Q \times Q \to \mathbb{N}$ as the *left to right* and the *right to left mock-shift*, respectively. So let us denote w_i as the *smallest word* of L_{q_i}. The following defines the mock-shifts:

$$\frac{q \in Q}{0 \in \mathrm{shift}'_l(q, q), 0 \in \mathrm{shift}'_r(q, q)} \qquad \frac{f(q) \to u_0 q_1(x_{\sigma(1)}) \dots q_n(x_{\sigma(n)}) u_n}{\begin{array}{c}|u_i w_i \dots w_n u_n| \in \mathrm{shift}'_l(q, q_i), \\ |u_0 w_0 \dots u_{i-1}| \in \mathrm{shift}'_r(q, q_i)\end{array}}$$

$$\frac{n \in \mathrm{shift}'_l(q_1, q_2) \qquad m \in \mathrm{shift}'_l(q_2, q_3)}{\mathrm{shift}'_l(q_1, q_3) = n + m} \qquad \frac{n \in \mathrm{shift}'_r(q_1, q_2) \qquad m \in \mathrm{shift}'_r(q_2, q_3)}{\mathrm{shift}'_r(q_1, q_3) = n + m}$$

$\mathrm{shift}_l(q, q') = \min(\mathrm{shift}'_l(q, q'))$ and $\mathrm{shift}_r(q, q') = \min(\mathrm{shift}'_r(q, q'))$.

Remark 6.24. If q is quasi-periodic, the smallest word in $\mathrm{lcp}(L_q) = \mathrm{lcs}(L_q) = w_q$ where w_q is the smallest word in L_q.

Lemma 6.24.1. *Let $\mathcal{L}$ be a* LTW *and q be a state of the* LTW*. We can compute $\mathrm{shift}_l(q, q')$ and $\mathrm{shift}_r(q, q')$ for all q' accessible from q in $\mathcal{O}(|\mathcal{L}|)$ time.*

Proof. For a rule $r = f(q) \to u_0 q_1 \dots q_n u_n \in \delta$ we can compute $\mathrm{shift}'_l(q, q_i)$ in $\mathcal{O}(|\mathrm{rhs}(r)|)$ by using algorithm 27. Therefore, we can compute this shift for each rule in $\mathcal{O}(|\mathcal{L}|)$ time. Starting from q we can compute all reachable, similar to the calculation of reachables for a context-free grammar. During this calculation we can inductively compute $\mathrm{shift}_l(q, q')$ by using the pre-calculated mock shifts shift'_l. If $\mathrm{shift}_l(q, q')$ is already defined we take the minimum. Similar to context-free grammars, the computation of all reachables takes at most $\mathcal{O}(|\mathcal{L}|)$ time. Consequently the total running time is $\mathcal{O}(|\mathcal{L}|)$. The proof for computation of the right to left shifts i. e. $\mathrm{shift}_r(q, q')$ is symmetric. ■

In the following we describe an algorithm presented in [9] that tests whether a state q is quasi-periodic on the left. Furthermore, if q is quasi-periodic on the left the algorithm transforms the LTW into an equivalent LTW containing a replacement q^e such that q^e is periodic. Furthermore all states q'^e accessible from q^e are periodic as well i. e. the new LTW has one less quasi-periodic state.

Lemma 6.24.2 ([9])**.** *Let q be a state in a* LTW *$\mathcal{L}$ and $\mathcal{T}^q$ be constructed by algorithm 28. Then q is quasi-periodic on the left if and only if $[\![\mathcal{L}]\!]_q = [\![\mathcal{T}^q]\!]$ and L_{q^e} is periodic i. e. if the test in line 15 does not fail.*

Algorithm 27: MOCKSHIFTOFRULE: Computes all shifts for a rule.

input : $f(q) \to u_0 q_1 \dots q_n u_n \in \delta$;
output: $\mathrm{shift}_l'(q, q_i)\ \forall i \in [n]$;
1 **for** $i = n \dots 1$ **do**
2 **if** $i \neq n$ **then**
3 $shift \leftarrow shift + |w_i|$;
4 $shift \leftarrow shift + |u_i|$;
5 $\mathrm{shift}_l'(q, q_i) \leftarrow shift$;
6 **return** $\mathrm{shift}_l'(q, q_i)$;

The first direction of the proof ensures that algorithm 28 replaces the state q by periodic states if it is quasi-periodic without changing $[\![\mathcal{L}]\!]$. The other direction is easy to see. Let us assume $[\![\mathcal{L}]\!]_q = [\![\mathcal{T}^q]\!] = w_q [\![\mathcal{T}^{q^e}]\!]$. If q^e is periodic then $[\![\mathcal{L}]\!]_q$ is quasi-periodic.

Remark 6.25. We can use algorithm 28 to test whether $L_q u$ is quasi-periodic on the left. We replace in line 10 $\mathrm{shift}_l(q, q')$ by $\mathrm{shift}_l(q, q') + |u|$. In line 13 we replace q by $w_q u$. Furthermore, we replace qu in any right-hand side in $r \in \delta$ by $w_q u q^e$ (see line 16). Note that we delete q only if all occurrences of q were replaced i. e. q becomes useless. Additionally, we can use algorithm 28 to test whether L_q is quasi-periodic on the right (see algorithm 34). Before shifting, we remove the suffix v of length $|w'|$ instead of the prefix from $u_0 w_1 \dots w_n u_n$. In line 10 we replace $\mathrm{shift}_l(q, q')$ by $\mathrm{shift}_r(q, q')$ and we prepend u at the end of the rule. In line 13 we append $u_{q'} w_q$ at the end of the rule. Furthermore, we replace q in any right-hand side in $r \in \delta$ by $q^e w_q$ (see algorithm 34).

Lemma 6.25.1. *Let $\mathcal{L} = (\Sigma, \Delta, Q, Q_I, \delta)$ be a* LTW. *By using algorithm 28 we can construct a* LTW *$\mathcal{L}'$ containing no proper quasi-periodic states with $[\![\mathcal{L}]\!] = [\![\mathcal{L}']\!]$ in*

$$\mathcal{O}(|Q| \cdot n^6 \cdot (n \cdot |G_L|)^2 \log(n \cdot |G_L|))$$

time, where $n = |\mathcal{L}|^6$ and G_L is the largest grammar of the set of grammars representing fully-compressed output words in $\mathcal{L}$. $\mathcal{L}'$ is of size $\mathcal{O}(|Q| \cdot \mathcal{L})$.

Proof. First note that n is an upper bound for the size of the constructed context-free grammar generating the language of all parallel successful runs of $\mathcal{L}_q, \mathcal{T}^q$. The size of $\mathcal{L}_q, \mathcal{T}^q$ is clearly in $\mathcal{O}(|\mathcal{L}|)$. Furthermore, they are same-ordered. Consequently, algorithm 28 can test whether a state of $\mathcal{L}$ is quasi-periodic in $\mathcal{O}(n^6 \cdot (n \cdot |G_L|)^2 \log(n \cdot |G_L|))$ time by applying lemma 6.14.1. Testing whether L_{q^e} is periodic requires less time, or more precise it requires $\mathcal{O}(m^6 \cdot (|\mathcal{L}| \cdot |Q|))$ time, where m is the numbers of rules in $\mathcal{L}$ applying theorem 5.24. After $|Q|$ application of the algorithm, there are no more quasi-periodic states in $\mathcal{L}$, since for each call we remove one quasi-periodic state or we mark an unmarked state as being not quasi-periodic. Furthermore, all added states are periodic [9]. Note that we change the output, mostly due to concatenation, thus the size of G_L may increase. However, the size of the grammar of the fully-compressed words on

Algorithm 28: IsQPonTheLeft: Tests whether the state q is quasi-periodic on the left and transforms q into a periodic state q^e, if this is true.

input : LTW $\mathcal{L} = (\Sigma, \Delta, Q, Q_I, \delta)$, $q \in Q$;
output: the result of the quasi-periodic test;

1 $\delta_q \leftarrow \emptyset$;
2 create a fresh state q_I;
3 $w_q \leftarrow$ the smallest word of L_q;
4 $Q_{acc} \leftarrow$ states accessible from q;
5 **foreach** $q' \in Q_{acc}$ **do**
6 add a copy q'^e to Q^q;
7 **foreach** $r = q'(f) \to u_0 q_1(x_{\sigma(1)}) \dots q_n(x_{\sigma(n)}) u_n \ \wedge \ q' \in Q_{acc}$ **do**
8 $w' \leftarrow$ the smallest word of $L_{q'}$;
9 $v \leftarrow$ the prefix of $u_0 w_1 \dots w_n u_n$ of size $|w'|$;
10 $u \leftarrow \rho_{\text{shift}_l(q,q')}[v^{-1} u_0 w_1 \dots w_n u_n]$;
11 $\delta_q \leftarrow \delta_q \cup q'^e(f) \to u q_1^e(x_{\sigma(1)}) \dots q_n^e(x_{\sigma(n)})$;
12 **if** $q' = q$ **then**
13 $\delta_q \leftarrow \delta_q \cup q_I(f) \to w_q u q_1^e(x_{\sigma(1)}) \dots q_n^e(x_{\sigma(n)})$;
14 $\mathcal{T}^q \leftarrow (\Sigma, \Delta, Q^q, \{q_I\}, \delta_q)$;
15 **if** L_{q^e} is periodic $\wedge \ [\![\mathcal{L}]\!]_q = [\![\mathcal{T}^q]\!]$ **then**
16 replace q in any right hand-side in $r \in \delta$ by $w_q q^e$;
17 $\delta \leftarrow \delta \cup \{r \in \delta_q \mid \text{lhs}(q) \neq q_I\}$;
18 $Q \leftarrow Q \cup Q^q$;
19 **return** true;
20 **else**
21 **return** false;

which we apply FCEquals do not increase, since these words were created due to the same concatenation. We just concatenate earlier and for many terminals (r_1, r_2) of the grammar of successful runs we have $\mu_i(r_1, r_2) = \epsilon$. The proof for the correctness can be found in [9]. We use almost the same algorithm presented in [9]. ■

Remark 6.26. We use the exact same algorithm described in [9] with the exception, that we have no axiom. Note that we use a slightly different definition of STWs and LTWs.

We now give a very informal description of the algorithm for transforming an arbitrary LTW $\mathcal{L}$ into its partial normal form.

Remark 6.27 (Algorithm 29). We replace all quasi-periodic states in $\mathcal{L}$ (see line 1) by applying algorithm 28. If q was identified as being quasi-periodic on the left we can proceed with the next state. Otherwise, we apply the adapted version of algorithm 28 to test quasi-periodicity on the right (and replace the state if this is true). In line 3 we push erasing states to the right such that if $L_{q_i} = \{\epsilon\}$ then $L_{q_{i+1}} = \{\epsilon\}$ and $u_i = \epsilon$.

Algorithm 29: ToPartialNormalForm: Transform an LTW into its partial normal form.

input : LTW $\mathcal{L} = (\Sigma, \Delta, Q, Q_I, \delta)$;
1 replace all quasi-periodic states in $\mathcal{L}$;
2 **foreach** rule in the $\mathcal{L}$ **do**
3 push all erasing states to the right and order them such that for all positions $i < j \Rightarrow \sigma(i) < \sigma(j)$;
4 $\delta' \leftarrow \delta$;
5 **foreach** $r = f(q) \to u_0 q_1(x_{\sigma(1)}) \dots q_n(x_{\sigma(n)}) u_n \in \delta'$ **do**
6 **foreach** $i \leftarrow n-1 \dots 1$ **do**
7 replace all $q_i u_i$ with $L_{q_i} u_i$ is quasi-periodic **on the left** by periodic states following remark 6.25;
8 **foreach** $f(q) \to u_0 q_1(x_{\sigma(1)}) \dots q_n(x_{\sigma(n)}) u_n \in \delta$ **do**
9 search for the largest k such that $L_{q_i}, \dots, L_{q_{i+k}}$ are period with the same period and $u_i = \dots = u_{i+k} = \epsilon$;
10 re-order $q_i(x_{\sigma(i)}), \dots, q_{i+k}(x_{\sigma(i+k)})$ such that $\sigma(i) < \dots < \sigma(i+k)$;

We then reorder these states according to definition 6.18 (ii). To realise line 7 we use an adapted version of algorithm 28 following remark 6.25. Note that we only consider occurrences of $q_i u_i$ if $L_{q_i} u_i$ is quasi-periodic on the left.

By using the following lemma we can improve the performance of algorithm 29.

Lemma 6.27.1. *Let $\mathcal{L} = (\Sigma, \Delta, Q, Q_I, \delta)$ be a* LTW*. Let $q \in Q$ be a state with L_q is not proper quasi-periodic. $L_q v$ is quasi-periodic on the left, if and only if L_q is periodic of period u and v is a prefix of u or $v = u^k w$ such that w is a prefix of u.*

Whenever the periodicity test for a state q is positive we store the pair (q^e, v) where v is the period of L_{q^e}. For testing whether qu is periodic, we first check if there is a pair (q, v). If there is such a pair, we can apply lemma 6.27.1 instead of the expensive test in line 15 in algorithm 28.

Lemma 6.27.2. *Let $\mathcal{L} = (\Sigma, \Delta, Q, Q_I, \delta)$ be a* LTW *and $q \in Q$. If we know that L_q is periodic with period u we can test in*

$$\mathcal{O}((|G_v + G_u|)^2 \log(|G_v + G_u|))$$

if $L_q v$ is quasi-periodic on the left, where G_v is the grammar of the fully-compressed word v and G_u is the grammar of the fully-compressed word of u.

Proof. First, we delete the suffix w of length $(|v| \mod |u|)$ from v receiving v' i. e. $v = v'w$. Then, we test by using FCEquals if $v'u = uv'$ i. e. v' and u are periodic with the same period. If this is true, we use FCPMatching to test if the pattern w occurs at position 1 in u. If this is the case, we know that $v = u^k w$ for some $k \geq 0$ and w is a prefix

of u, thus $L_q v$ is quasi-periodic on the left. In any other case, we know that $L_q v$ is not quasi-periodic on the left. The manipulations on the words can be done in linear time (see chapter 3). Testing equality of fully-compressed words and the pattern matching can be done in $\mathcal{O}((|G_v + G_u|)^2 \log(|G_v + G_u|))$ by using theorem 4.27 and 4.32. ■

Lemma 6.27.3. *Let $\mathcal{L} = (\Sigma, \Delta, Q, Q_I, \delta)$ be a* LTW*. By using algorithm 29 we can transform $\mathcal{L}$ into $\mathcal{L}'$ that is in partial normal form with $[\![\mathcal{L}]\!] = [\![\mathcal{L}']\!]$ in*

$$\mathcal{O}((|Q| + p) \cdot n^6 \cdot (n \cdot |G_L|)^2 \log(n \cdot |G_L|)) \text{ where } p = \sum_{\substack{q(f) \to \alpha \in \delta \\ \operatorname{rank}(f) > 1}} (\operatorname{rank}(f) - 1)\,,$$

$n = |\mathcal{L}|^6$ and G_L is the largest grammar of the set of grammars representing fully-compressed output words in $\mathcal{L}$. The size of the constructed transducer is in $\mathcal{O}(|\mathcal{L}|^2)$.

Proof. For the correctness proof we refer to [9]. To avoid confusion let δ' be the set of rules of $\mathcal{L}'$. At the beginning of the algorithm $\delta' = \delta$. We gave an upper bound for the elimination of all proper quasi-periodic states (see lemma 6.25.1). All operations inside the while-loop except the test for quasi-periodicity can be ignored if we look at the runtime, since these operation requires at most $\mathcal{O}(|\mathcal{L}|)$ time. The test for quasi-periodicity on the left for $L_q u$ is cheap, if we know that L_q is periodic (see lemma 6.27.2). Whenever we add new rules to δ', they contain only states that are periodic. Therefore, we have to use the expensive test for quasi-periodicity i. e. line 15 in algorithm 28, at most for each occurrence of a state in a rule of $\mathcal{L}$ containing at least two states (see line 6) i. e.

$$p = \sum_{\substack{q(f) \to \alpha \in \delta \\ \operatorname{rank}(f) > 1}} (\operatorname{rank}(f) - 1)$$

and twice for each state in Q (see line 1). Whenever we delete an occurrence of qu we add at most $|\delta|$ rules to δ'. All these rules do not contain any qu with $u \neq \epsilon$ (by construction of these rules in algorithm 28). Thus, we don't have to consider these rules again and after at most p applications of algorithm 28 there are no more positions i such that $q_i u_i$ is quasi-periodic on the left.

During the algorithm we add at most $(|Q| + p)$ times the rules contained in δ to $\mathcal{L}'$ thus, $|\mathcal{L}'| = \mathcal{O}((p + |Q|) \cdot |\mathcal{L}|) = \mathcal{O}(|\mathcal{L}|^2)$. ■

Theorem 6.28. *Let $\mathcal{L}_1, \mathcal{L}_2$ be two* LTWs*. By using algorithm 30 we can decide whether $[\![\mathcal{L}_1]\!] = [\![\mathcal{L}_2]\!]$ holds in*

$$\mathcal{O}(n^6 \cdot (n \cdot |G_L|)^2 \log(n \cdot |G_L|))$$

time, where $n = |\mathcal{L}_1|^6 \cdot |\mathcal{L}_2|^6$ and G_L is the largest grammar representing an output word.

Proof. Using lemma 6.27.3 we can construct for $\mathcal{L}_i$ an equivalent LTWs $\mathcal{L}_i'$ that is in partial normal form. The size of $\mathcal{L}_i'$ is quadratic in the size of $\mathcal{L}_i$. We can do this construction in $\mathcal{O}(|\mathcal{L}_i| \cdot (m^6 \cdot (m \cdot |G_L|)^2 \log(m \cdot |G_L|)) = \mathcal{O}(n^6 \cdot (n \cdot |G_L|)^2 \log(n \cdot |G_L|))$, where $m = |\mathcal{L}_i|^6$. We can test whether $\mathcal{L}_1'$ and $\mathcal{L}_2'$ are same-ordered. If this does not hold, we

Algorithm 30: LTWEQUALS: Test whether two LTWs are equivalent.

input : LTWs $\mathcal{L}_1, \mathcal{L}_2$;
output: $true$ if $\mathcal{L}_1, \mathcal{L}_2$ are equivalent, otherwise $false$;

```
1  if dom(L1) ≠ dom(L2) then
2      return false;
3  else if IsSameOrdered(L1, L2) then
4      return STWEquals(L1, L2);
5  else
6      ToPartialNormalForm(L1);
7      ToPartialNormalForm(L2);
8      if IsSameOrdered(L1, L2) then
9          return STWEquals(L1, L2);
10     else
11         return false;
```

can report that $[\![\mathcal{L}_1]\!] \neq [\![\mathcal{L}_2]\!]$. Otherwise, we test for equivalence of same-ordered LTWs using lemma 6.14.2. By the lemma this requires

$$\mathcal{O}(n^6 \cdot (n \cdot |G_L|)^2 \log(n \cdot |G_L|))$$

time. Since $[\![\mathcal{L}'_1]\!], [\![\mathcal{L}'_2]\!]$ are same-ordered we have

$$[\![\mathcal{L}'_1]\!] = [\![\mathcal{L}'_2]\!] \iff [\![\mathcal{L}_1]\!] = [\![\mathcal{L}_2]\!]$$

which finishes the proof. ■

7. Results and Examples

In this section we give running times for our implementation and explain the algorithm of the last section for a complete example. We used a MacBook Pro with 3.1 Ghz Intel Core i7 (processor) and 16 GB 1867 MHz DDR (memory).

7.1. Performance of the recompression algorithm

Since we extensively use the recompression algorithm, we test it for large instances separately. We construct the following SLP $G_k = (\Delta, N, X_1, P)$ defined by the following productions:

$$\begin{aligned} X_1 &\to aX_2bX_2c \\ X_2 &\to aX_3bX_3c \\ &\dots \\ X_{k-1} &\to aX_kbX_kc \\ X_k &\to d. \end{aligned}$$

$\Delta = \{a, b, c, d\}$, $|P| = |N| = k$, the size of the grammar is in $\mathcal{O}(k)$ and the size of the words ω_{G_k} in $\Omega(2^k)$. We compute the running time of the recompression i. e. of FCEQUALS for $G_t = G_p$.

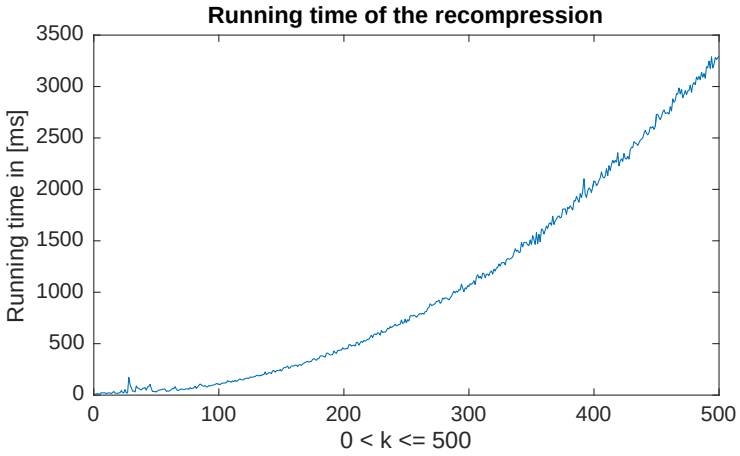


Figure 7.1.: Running time in milliseconds of the recompression applied on two grammars equals G_k for various $k \in [500]$.

7.2. LTW equality test example

We will now give a short example of an equality test for LTWs inspired by the example presented in [9]. Consider a LTW $\mathcal{L} = (\Sigma, \Delta, Q, Q_I, \delta)$ where $Q_I = \{q_0\}, \Sigma = \{f, g, h\}$ and f, g, h have the following rules:

$$
\begin{aligned}
&q_0(f) \to h\ q_1(x_3)\ a\ q_2(x_2)\ b\ q_3(x_1) && \\
&q_1(g) \to ell\ q_4(x_1) && \\
&q_2(g) \to q_2(x_1)\ bca && q_2(h) \to bca \\
&q_3(g) \to c\ q_5(x_1) abend && \\
&q_4(g) \to q_4(x_1) && q_4(h) \to o \\
&q_5(g) \to abc\ q_5(x_1) && q_5(h) \to \epsilon
\end{aligned}
$$

Let the second LTW $\mathcal{L}' = (\Sigma, \Delta, Q', Q'_I, \delta')$ where $Q_I = \{q_0\}$ be defined by the following rules:

$$
\begin{aligned}
&q'_0(f) \to helloa\ q'_1(x_2)\ q'_2(x_1)\ cabend\ q'_3(x_3) && \\
&q'_1(g) \to bca\ q'_1(x_1) && q'_1(h) \to b \\
&q'_2(g) \to cab\ q'_4(x_1) && \\
&q'_3(g) \to q'_5(x_1) && \\
&q'_4(g) \to cab\ q'_4(x_1) && q'_4(h) \to \epsilon \\
&q'_5(g) \to q'_5(x_1) && q'_5(h) \to \epsilon
\end{aligned}
$$

In the first step we test if $\mathrm{dom}(\mathcal{L}) = \mathrm{dom}(\mathcal{L}')$, which is the case. We furthermore test if they are same-ordered. In that case we could apply the equality test for STWs. However this does not hold since $(q_0, q'_0) \in \mathcal{C}o(\mathcal{L}, \mathcal{L}')$ but their rules are not same-ordered. Consequently, we transform both LTW into LTWs in partial normal form. We identify states $q_1, q_2, q_3, q_4, q_5, q'_1, q'_2, q'_3, q'_4, q'_5$ as quasi-periodic. Let's look for example at state q_3. $L_{q_3} = (cab)^+ end \subseteq (cab)^* end$ i. e. quasi-periodic on the right. Therefore the period is $u = cab$. We replace q_3 by introducing the following rules:

$$
\begin{aligned}
&q^e_3(g) \to q^e_5(x_1)\ \rho^r_{\mathrm{shift}_r(q_3,q_3)}[cabend\,\mathrm{lcs}(L_{q_3})^{-1}] \to q^e_5(x_1)\ \rho^r_0[cabend(cabend)^{-1}] \to q^e_5(x_1) \\
&q^e_5(g) \to q^e_5(x_1)\ \rho^r_{\mathrm{shift}_r(q_3,q_5)}[abc\,\mathrm{lcs}(L_{q_5})^{-1}] \to q^e_5(x_1)\ \rho^r_1[abc(\epsilon)^{-1}] \to q^e_5(x_1)\ cab \\
&q^e_5(h) \to \epsilon.
\end{aligned}
$$

Then we can replace q_3 by $(q^e_3\,\mathrm{lcp}(L_{q_3})) = (q_3\ cabend)$ and we can delete q_5 since it became unreachable. In the next step the algorithm may consider q_1. $L_{q_1} = \{ello\}$ thus q_1 is periodic and we push the factor to the left and creates the following rules:

$$
\begin{aligned}
&q^e_1(g) \to \rho^l_{\mathrm{shift}_l(q_1,q_1)}[\mathrm{lcp}(L_{q_1})^{-1} ello]\ q^e_4(x_1) \to \rho^l_0[(ello)^{-1} ello]\ q^e_4(x_1) \to q^e_4(x_1) \\
&q^e_4(g) \to q^e_4(x_1) \\
&q^e_4(h) \to \rho^l_{\mathrm{shift}_l(q_1,q_4)}[\mathrm{lcp}(L_{q_4})^{-1} o] \to \rho^l_1[(o)^{-1} o] \to \epsilon.
\end{aligned}
$$

Then we can replace q_1 by $(q_1^e \operatorname{lcp}(L_{q_1})) = (q_1^e\ ello)$ and we can delete q_4 since it became unreachable. In the last step we transform q_2 into periodic states. $L_{q_2} = (bca)^+$ thus q_2 is periodic and we introduce the following rules such that q_2 becomes earliest:

$$q_2^e(g) \to \rho^l_{\text{shift}_l(q_2,q_2)}[\operatorname{lcp}(L_{q_2})^{-1}bcabca]\ q_2^e(x_1) \to \rho_0^l[(bca)^{-1}bcabca]\ q_2^e(x_1) \to bca\ q_2^e(x_1)$$
$$q_2^e(h) \to \rho^l_{\text{shift}_l(q_2,q_2)}[\operatorname{lcp}(L_{q_2})^{-1}bca]) \to \rho_0^l[(bca)^{-1}bca] \to \epsilon.$$

Then we replace q_2 by $q_2^e\ \operatorname{lcp}(L_{q_2}) = q_2^e\ bca$. Note that we could had also identify q_2 as quasi-periodic on the right. After this first step of the algorithm there are no proper quasi-periodic states in $\mathcal{L}$. The rules of the transducer are

$$\begin{aligned}
&q_0(f) \to hello\ q_1^e(x_3)\ a\ q_2^e(x_2)\ bcab\ q_3^e(x_1)\ cabend && \\
&q_3^e(g) \to q_5^e(x_1) && \\
&q_5^e(g) \to q_5^e(x_1)\ cab && q_5^e(h) \to \epsilon \\
&q_1^e(g) \to q_4^e(x_1) && \\
&q_4^e(g) \to q_4^e(x_1) && q_4^e(h) \to \epsilon \\
&q_2^e(g) \to bca\ q_2^e(x_1) && q_2^e(h) \to \epsilon
\end{aligned}$$

We do the whole replacement of quasi-periodic states for $\mathcal{L}'$ too and receive the following rules:

$$\begin{aligned}
&q_0^{'e}(f) \to helloa\ q_1^{'e}(x_2)bcab\ q_2^{'e}(x_1)\ cabend\ q_3^{'e}(x_3) && \\
&q_1^{'e}(g) \to q_1^{'e}(x_1)\ bca && q_1^{'e}(h) \to \epsilon \\
&q_2^{'e}(g) \to q_4^{'e}(x_1) && \\
&q_3^{'e}(g) \to q_5^{'e}(x_1) && \\
&q_4^{'e}(g) \to cab\ q_4^{'e}(x_1) && q_4^{'e}(h) \to \epsilon \\
&q_5^{'e}(g) \to q_5^{'e}(x_1) && q_5^{'e}(h) \to \epsilon
\end{aligned}$$

In the second step we re-order states generating the empty language i. e. $\{q_1, q_4\}$ in $\mathcal{L}$ and $\{q_3^{'e}, q_5^{'e}\}$ in $\mathcal{L}'$. Therefore we only change the rule $q_0(f)$ as follows:

$$q_0(f) \to helloa\ q_2^e(x_2)\ bcab\ q_3^e(x_1)\ cabend\ q_1^e(x_3)$$

In the next step we identify and eliminate qu if $L_q u$ is quasi-periodic on the left but not for the right most state of a rule. In our example we only have to look at $q_0(f), q_0'(f)$. To avoid confusion we call the new introduced states q^r. We identify $L_{q_2}bcab = (bca)^*bcab \subseteq b(cab)^*$ and therefore q_2bcab is quasi-periodic on the left. The same holds for $L_{q_1^{'e}}bcab = (bca)^*bcab \subseteq b(cab)^*$. For $\mathcal{L}$ we introduce new rules:

$$\begin{aligned}
q_2^r(g) &\to \rho^l_{\text{shift}_l(q_2^e,q_2^e)+|bcab|}[\operatorname{lcp}(L_{q_2^e})^{-1}bca]\ q_2^r(x_1) \to \rho_4^l \mod 3[\epsilon^{-1}bca]\ q_2^r(x_1) \\
&\to \rho_1^l[bca]\ q_2^r(x_1) \to cab\ q_2^r(x_1) \\
q_2^r(h) &\to \epsilon
\end{aligned}$$

and we replace $q_2^e\ bcab$ by $\mathrm{lcp}(L_{q_2^e}bcab)\ q_2^r = bcab\ q_2^r$. Furthermore, we introduce for $\mathcal{L}'$ the rules

$$\begin{aligned} q_1^{'r}(g) &\to \rho^l_{\mathrm{shift}_l(q_1^{'e},q^{'e})+|bcab|}[\mathrm{lcp}(L_{q_2^{'e}})^{-1}bca]\ q_1^{'r}(x_1) \to \rho_4^l \mod 3 [\epsilon^{-1}bca]\ q_1^{'r}(x_1) \\ &\to \rho_1[bca]\ q_1^{'r}(x_1) \to cab\ q_1^{'r}(x_1) \\ q_1^{'r}(h) &\to \epsilon \end{aligned}$$

and we replace $q_1^{'e}\ bcab$ by $\mathrm{lcp}(L_{q_1^{'e}}bcab)\ q_1^{'r} = bcab\ q_1^{'r}$. The constructed $\mathcal{L}$ is defined by the following rules:

$$\begin{aligned} &q_0(f) \to helloabcab\ q_2^r(x_2)\ q_3^e(x_1)\ cabend\ q_1^e(x_3) && \\ &q_1^e(g) \to q_4^e(x_1) && \\ &q_2^r(g) \to cab\ q_2^r(x_1) && q_2^r(g) \to \epsilon \\ &q_3^e(g) \to q_5^e(x_1) && \\ &q_5^e(g) \to q_5^e(x_1)\ cab && q_5^e(h) \to \epsilon \\ &q_4^e(g) \to q_4^e(x_1) && q_4^e(h) \to \epsilon \end{aligned}$$

The constructed $\mathcal{L}'$ is defined by the following rules:

$$\begin{aligned} &q_0^{'e}(f) \to helloabcab\ q_1^{'r}(x_2)\ q_2^{'e}(x_1)\ cabend\ q_3^{'e}(x_3) && \\ &q_3^{'e}(g) \to q_5^{'e}(x_1) && \\ &q_1^{'r}(g) \to cab\ q_1^{'r}(x_1) && q_1^{'r}(h) \to \epsilon \\ &q_2^{'e}(g) \to q_4^{'e}(x_1) && \\ &q_4^{'e}(g) \to cab\ q_4^{'e}(x_1) && q_4^{'e}(h) \to \epsilon \\ &q_5^{'e}(g) \to q_5^{'e}(x_1) && q_5^{'e}(h) \to \epsilon \end{aligned}$$

In the last step we swap $q_2^r(x_2)\ q_3^e(x_1)$ and $q_1^{'r}(x_2)\ q_2^{'e}(x_1)$ since they are of the same period. Note that both transducers $\mathcal{L}, \mathcal{L}'$ contain the exact same rules with the exception of different state names. Duo to the complexity of the problem, our implementation solves the problem on the mentioned machine in 4.3 hours.

8. Discussion

The aim of the master thesis was to implement an efficient algorithm to solve the equivalence problem for STWs and LTWs. We studied the theoretical background and all required ingredients. We gave all implementation details and an upper bound of our implementation. We proved that our implementation solves the problem in polynomial time. However, we also mentioned that the running time, even for small problem instances, can be huge.

One possible improvement is to replace the recompression algorithm, described in chapter 4 by the approach presented in [3]. If we look back to the creation of words of the test set in chapter 5, we should mention that most of consecutive created Plandowski paths differ only in one edge. Suppose we have two Plandowski edges

$$\lambda = (u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4), (u_5, v_5), (u_6, v_6)$$

If we use the data structure presented in [3], we could add each non-terminal word and each concatenation we already produced by traversing the grammar graph and the trees. We could add all words of all incomplete Plandowski paths to the data structure as well. Therefore, the word of the incomplete path

$$\lambda' = (u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4), (u_5, v_5)$$

would be contained in the data structure. To test whether

$$\mu_1(w(\lambda)) = \mu_2(w(\lambda))$$

holds, we have to add $\mu_1(w(\lambda)), \mu_2(w(\lambda))$ to the data structure under the condition that $\mu_1(w(\lambda')), \mu_2(w(\lambda'))$ was already added to the data structure. The running time would be $\mathcal{O}(\log(n+m))$ where n and m is the size of the grammar of the SLP representing $\mu_1(w(\lambda'))$ and the size of of the grammar of the SLP representing $\mu_2(w(\lambda'))$, respectively. This outperforms the recompression algorithm, since we have many words which share a common prefix. A second source for improvement is the upper bound of $\mathcal{O}(m^6)$ of the size of the test set, where m is the number of productions of the grammar. In the future one might find a construction for a smaller test set.

Nevertheless, this is the first implementation that solves the equivalence problem for STWs and LTWs in polynomial time. Furthermore, an object-oriented modular and generic designed Java library, based on modern design patterns, has been created. The library supports a large variety of operations to create, manipulate and work with CFGs, SLPs, STWs and LTWs. The application area of fully compressed pattern matching and

equivalence test (see chapter 4) is large and we offer a library that supports efficient solvers for these problems.

For future work we suggest to focus on extending the library to support further equality tests for other transducers like tree-to-tree transducers and to support the data structure presented in [3]. Note that the equivalence of tree-to-tree transducers can be reduced to equivalence of tree-to-word transducers that linearise output trees [52]. Consequently, we can easily extend our implementation to obtain a polynomial time algorithm for deciding the equivalence of deterministic top-down and bottom-up tree-to-tree transducers that are non-copying. In many settings it could make sense to use a heuristic approach to reduce the running time of the algorithm. One might pick only a subset of the test set described in chapter 5. We created the base for further adjustments and improvements, which goes beyond the scope of this thesis.

Appendices

A. Algorithms

Here we list some algorithm we mentioned but not list in the main content of the thesis.

Algorithm 31: GETPRODUCTIVES: Computes the set of productives.

input : $G = (\Delta, N, S, P)$;
output: $Prod_G$;
$Map \leftarrow$ GETMAP(G);
$H \leftarrow \emptyset$;
$Prod_G \leftarrow \emptyset$;
$NT \leftarrow \emptyset$;
foreach $p \in P$ **do**
    if $p = X \rightarrow \alpha \ \wedge \ \alpha \in \Delta^*$ **then**
        $H \leftarrow H \cup p$;
        $Prod_G \leftarrow Prod_G \cup p$;
        $NT \leftarrow NT \cup \text{lhs}(p)$;
while $H \neq \emptyset$ **do**
    $p \leftarrow$ first element contained in H;
    $H \leftarrow H \setminus p$;
    foreach $node \in Map[\text{lhs}(p)]$ **do**
        $node.counter \leftarrow (node.counter - 1)$;
        if $node.counter = 0$ **then**
            $Prod_G \leftarrow Prod_G \cup node.p$;
            if $\text{lhs}(node.p) \notin NT$ **then**
                $H.push(node.p)$;
                $NT \leftarrow NT \cup \text{lhs}(node.p)$;
return $Prod_G$

Algorithm 32: GETWORD: Generates an SLP representing a word in $L(G)$.

input : $G = (\Delta, N, S, P)$ reduced grammar, with $L(G) \neq \emptyset$;
output: SLP $G = (\Delta, N, S, P')$ a word contained in $L(G)$;

```
1  Map ← GETMAP(G);
2  P' ← ∅;
3  len*_P ← ∅;
4  len*_N ← ∅;
5  H ← ∅ ;                                          // unsorted heap
6  foreach p ∈ P do
7      if p = X → α ∧ α ∈ Δ* then
8          len*_P(p) ← |α|;
9          H ← H ∪ p;
10 while H ≠ ∅ do
11     p ← first element contained in H;
12     H ← H \ p;
13     if len*_N(lhs(p)) is undefined then
14         len*_N(lhs(p)) ← len*_P(p);
15         P' ← P' ∪ p;
16         foreach (q, counter) ∈ Map[lhs(p)] do
17             counter ← (counter − 1);
18             if counter = 0 then
19                 l ← 0;
20                 foreach X ∈ rhs(q) do
21                     l ← len*_N(X) + l
22                 l ← l+ number of terminals on rhs(q);
23                 len*_P(q) ← l;
24                 if len*_N(lhs(q)) is undefined then
25                     H ← H ∪ q;
26 return G' = (Δ, N, S, P')
```

Algorithm 33: GETSHORTESTNONEMPTYWORD: Generates an SLP representing the shortest non-empty word of a CFG.

input : $G = (\Delta, N, S, P)$;
output: The shortest non-empty word of $L(G)$;

$G' \leftarrow$ TOBINARY(G);
$G' \leftarrow$ TOEPSILONFREE(G');
$G' \leftarrow$ GETUSEFUL(G');
if $p = S \to \epsilon \in P'$ **then**
    $P' \leftarrow P' \setminus \{p\}$;
if $P' = \emptyset$ **then**
    return undefined;
else
    return GETSHORTESTWORD(G')

Algorithm 34: ISQPONTHERIGHT: Tests whether the state q is quasi-periodic on the right and transforms q into a periodic state q^e, if this is true.

input : LTW $\mathcal{L} = (\Sigma, \Delta, Q, Q_I, \delta)$, $q \in Q$;
output: the result of the quasi-periodic test;

$\delta_q \leftarrow \emptyset$;
create a fresh state q_I;
$w_q \leftarrow$ the smallest word of L_q;
$Q_{acc} \leftarrow$ states accessible from q;
foreach $q' \in Q_{acc}$ **do**
    add a copy q'^e to Q^q;
foreach $r = q'(f) \to u_0 q_1(x_{\sigma(1)}) \dots q_n(x_{\sigma(n)}) u_n \ \wedge \ q' \in Q_{acc}$ **do**
    $w' \leftarrow$ the smallest word of $L_{q'}$;
    $v \leftarrow$ the suffix of $u_0 w_1 \dots w_n u_n$ of size $|w'|$;
    $u \leftarrow \rho_{\text{shift}_r(q,q')}[u_0 w_1 \dots w_n u_n v^{-1}]$;
    $\delta_q \leftarrow \delta_q \cup q'^e(f) \to q_1^e(x_{\sigma(1)}) \dots q_n^e(x_{\sigma(n)}) u$;
    if $q' = q$ **then**
        $\delta_q \leftarrow \delta_q \cup q_I(f) \to q_1^e(x_{\sigma(1)}) \dots q_n^e(x_{\sigma(n)}) u w_q$;
$\mathcal{T}^q \leftarrow (\Sigma, \Delta, Q^q, \{q_I\}, \delta_q)$;
if L_{q^e} is periodic $\wedge \ [\![\mathcal{L}]\!]_q = [\![\mathcal{T}^q]\!]$ **then**
    replace q in any right hand-side in $r \in \delta$ by $q^e w_q$;
    $\delta \leftarrow \delta \cup \{r \in \delta_q \mid \text{lhs}(q) \neq q_I\}$;
    $Q \leftarrow Q \cup Q^q$;
    return true;
else
    return false;

Algorithm 35: STWEQUALS: Test whether two STWs are equivalent.

input : STWs $\mathcal{S}_1, \mathcal{S}_2$;
output: $true$ if $\mathcal{S}_1, \mathcal{S}_2$ are equivalent, otherwise $false$;

```
if dom(S_1) ≠ dom(S_2) then
    return false;
else
    transform S_1 into an equivalent dN2W N_1;
    transform S_2 into an equivalent dN2W N_2;
    construct the CFG G generating all successful parallel runs for N_1, N_2;
    construct μ_1, μ_2 defined in section 6.2;
    return MORPHSIMEQUALITY(G, μ_1, μ_2);
```

Bibliography

[1] J. Albert, K. Culik, and J. Karhumaeki. Test sets for context free languages and algebraic systems of equations over a free monoid. *Information and Control*, 52(2): 172 – 186, 1982. ISSN 0019-9958. doi: http://dx.doi.org/10.1016/S0019-9958(82)80028-4. URL http://www.sciencedirect.com/science/article/pii/S0019995882800284. 5

[2] M.H. Albert and J. Lawrence. A proof of ehrenfeucht's conjecture. *Theoretical Computer Science*, 41:121 – 123, 1985. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(85)90066-0. URL http://www.sciencedirect.com/science/article/pii/0304397585900660. 5.3

[3] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 819–828. Society for Industrial and Applied Mathematics, 2000. URL http://dl.acm.org/citation.cfm?id=338219.338645. 4, 4.1, 4.1.1, 4.2, 4.3, 4.3, 8

[4] Rajeev Alur. Marrying words and trees. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, pages 233–242, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-685-1. doi: 10.1145/1265530.1265564. URL http://doi.acm.org/10.1145/1265530.1265564. 2, 2.2.3, 6.2

[5] A. Amir and C. Benson. Efficient two-dimensional compressed matching. In *Data Compression Conference, 1992. DCC '92.*, pages 279–288, March 1992. doi: 10.1109/DCC.1992.227453. 4.2

[6] Amihood Amir and Gary Benson. Two-dimensional periodicity and its applications. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 440–452, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics. ISBN 0-89791-466-X. URL http://dl.acm.org/citation.cfm?id=139404.139489. 4.2

[7] Amihood Amir, Gary Benson, and Martin Farach. Let sleeping files lie: Pattern matching in z-compressed files. *Journal of Computer and System Sciences*, 52(2):299 – 307, 1996. ISSN 0022-0000. doi: http://dx.doi.org/10.1006/jcss.1996.0023. URL http://www.sciencedirect.com/science/article/pii/S0022000096900239. 4.2

[8] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009. ISBN 0521424267, 9780521424264. 2

[9] Adrien Boiret and Raphaela Palenta. Deciding equivalence of linear tree-to-word transducers in polynomial time. *Unpublished*, 2016. 1, 1, 2, 5, 5.22, 5.23, 5.4, 6.12, 6.4.1, 6.14, 6.4.2, 6.15.1, 6.4.2, 6.16, 6.17, 6.4.2, 6.4.3, 6.18, 6.18.1, 6.19, 6.4.4, 6.20.1, 6.21.1, 6.4.4, 6.4.4, 6, 6.24.2, 21, 6.26, 10, 7.2

[10] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *Information Theory, IEEE Transactions on*, 51(7):2554–2576, July 2005. ISSN 0018-9448. doi: 10.1109/TIT.2005.850116. 4.28.1, 4.28.2

[11] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007. 1, 2, 6.1

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848. 4.2

[13] Tali Eilam-Tzoreff and Uzi Vishkin. Matching patterns in strings subject to multi-linear transformations. *Theoretical Computer Science*, 60(3):231 – 254, 1988. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(88)90112-0. URL http://www.sciencedirect.com/science/article/pii/0304397588901120. 4.2

[14] Joost Engelfriet. Some open questions and recent results on tree transducers and tree languages. In *Perspectives and Open Problems*, Formal Language Theory, pages 241–286. Academic Press, 1980. 1

[15] Joost Engelfriet and Sebastian Maneth. A comparison of pebble tree transducers with macro tree transducers. *Acta Informatica*, 39(9):613–698. ISSN 1432-0525. doi: 10.1007/s00236-003-0120-0. URL http://dx.doi.org/10.1007/s00236-003-0120-0. 1

[16] Zoltán Ésik. Decidability results concerning tree transducers i. *Acta Cybernetica*, 5 (1):1–20, 1980. 1

[17] Michael J. Fischer. *Grammars with Macro-like Productions*. PhD thesis, Harvard University, Massachusetts, 1968. 1

[18] Zoltan Fueloep and Heiko Vogler. *Syntax-directed semantics : formal models based on tree transducers*. Monographs in theoretical computer science. Springer, Berlin, New York, 1998. ISBN 3-540-64607-8. URL http://opac.inria.fr/record=b1094476. 1

[19] L. Gasieniec and W. Rytter. Almost-optimal fully lzw-compressed pattern matching. In *Data Compression Conference, 1999. Proceedings. DCC '99*, pages 316–325, Mar 1999. doi: 10.1109/DCC.1999.755681. 4.2

[20] Leszek Gasieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for lempel-ziv encoding. In Rolf Karlsson and Andrzej Lingas, editors, *Algorithm Theory - SWAT'96*, volume 1097 of *Lecture Notes in Computer Science*, pages 392–403. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61422-7. doi: 10.1007/3-540-61422-2_148. URL http://dx.doi.org/10.1007/3-540-61422-2_148. 4.2

[21] Leszek Gasieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Randomized efficient algorithms for compressed strings: the finger-print approach. In Dan Hirschberg and Gene Myers, editors, *Combinatorial Pattern Matching*, volume 1075 of *Lecture Notes in Computer Science*, pages 39–49. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61258-2. doi: 10.1007/3-540-61258-0_3. URL http://dx.doi.org/10.1007/3-540-61258-0_3. 4.2

[22] Pawel Gawrychowski. Pattern matching in lempel-ziv compressed strings: Fast, simple, and deterministic. In Camil Demetrescu and MagnusM. Halldorsson, editors, *Algorithms - ESA 2011*, volume 6942 of *Lecture Notes in Computer Science*, pages 421 – 432. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-23718-8. doi: 10.1007/978-3-642-23719-5_36. URL http://dx.doi.org/10.1007/978-3-642-23719-5_36. 4.2

[23] PawełGawrychowski. Optimal pattern matching in lzw compressed strings. In *Proceedings of the Twenty-second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, pages 362–372. SIAM, 2011. URL http://dl.acm.org/citation.cfm?id=2133036.2133065. 4.2

[24] Pawel Gawrychowski. Simple and efficient lzw-compressed multiple pattern matching. In Juha Kaerkkaeinen and Jens Stoye, editors, *Combinatorial Pattern Matching*, volume 7354 of *Lecture Notes in Computer Science*, pages 232–242. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31264-9. doi: 10.1007/978-3-642-31265-6_19. URL http://dx.doi.org/10.1007/978-3-642-31265-6_19. 4.2

[25] Pawel Gawrychowski. Tying up the loose ends in fully lzw-compressed pattern matching. In Christoph Dürr and Thomas Wilke, editors, *29th International Symposium on Theoretical Aspects of Computer Science (STACS 2012)*, volume 14 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 624–635, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-35-4. doi: http://dx.doi.org/10.4230/LIPIcs.STACS.2012.624. URL http://drops.dagstuhl.de/opus/volltexte/2012/3397. 4.2

[26] V. S. Guba. Equivalence of infinite systems of equations in free groups and semigroups to finite subsystems. *Mathematical notes of the Academy of Sciences of the*

USSR, 40(3):688–690, 1986. ISSN 1573-8876. doi: 10.1007/BF01142470. URL http://dx.doi.org/10.1007/BF01142470. 5.3

[27] M. Hirao, A. Shinohara, M. Takeda, and S. Arikawa. Fully compressed pattern matching algorithm for balanced straight-line programs. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 132–138, 2000. doi: 10.1109/SPIRE.2000.878188. 4.2

[28] Yoram Hirshfeld, Mark Jerrum, and Faron Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theor. Comput. Sci.*, 158(1-2):143–159, May 1996. ISSN 0304-3975. doi: 10.1016/0304-3975(95)00064-X. URL http://dx.doi.org/10.1016/0304-3975(95)00064-X. 4, 4.3, 4

[29] J. Hopcroft and R. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 0, Dept. of Computer Science, Cornell U, December 1971. 6.1

[30] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363. (document), 2, 3.2, 3.2, 1, 3.4.6, 3.3, 3.11.2, 4.1

[31] Harry B. Hunt, Daniel J. Rosenkrantz, and Thomas G. Szymanski. On the equivalence, containment, and covering problems for the regular and context-free languages. *Journal of Computer and System Sciences*, 12(2):222 – 268, 1976. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/S0022-0000(76)80038-4. URL http://www.sciencedirect.com/science/article/pii/S0022000076800384. 4, 4.2

[32] Karel Culik II and Arto Salomaa. On the decidability of homomorphism equivalence for languages. *J. Comput. Syst. Sci.*, 17(2):163–175, 1978. doi: 10.1016/0022-0000(78)90002-8. URL http://dx.doi.org/10.1016/0022-0000(78)90002-8. 5, 5.2

[33] Artur Jez. Approximation of grammar-based compression via recompression. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching*, volume 7922 of *Lecture Notes in Computer Science*, pages 165–176. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38904-7. doi: 10.1007/978-3-642-38905-4_17. URL http://dx.doi.org/10.1007/978-3-642-38905-4_17. 4.28.1

[34] Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Trans. Algorithms*, 11(3):20:1–20:43, 2015. doi: 10.1145/2631920. 1, 4, 4.4, 4, 4.5, 4, 4.1.1, 4.1.2, 4.17, 4.20, 4.20.1, 19, 18, 18, 4, 14, 14, 4.23.4, 4.1.6, 4.28, 4.2, 4.2, 4.2.2, 4.30.1, 4.30.2, 4.3

[35] J. Karhumaki, W. Plandowski, and W. Rytter. Polynomial size test sets for context-free languages. *Journal of Computer and System Sciences*, 50(1):11 –

19, 1995. doi: http://dx.doi.org/10.1006/jcss.1995.1002. URL http://www.sciencedirect.com/science/article/pii/S0022000085710021. 1, 5, 5.1, 5, 5.2, 5.5.1, 5.5.2, 5.13, 5.15

[36] Juhani Karhumaki, Wojciech Rytter, and Stefan Jarominek. Efficient constructions of test sets for regular and context-free languages. *Theoretical Computer Science*, 116 (2):305 – 316, 1993. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(93)90325-N. URL http://www.sciencedirect.com/science/article/pii/030439759390325N. 5

[37] Ralf Kuesters and Thomas Wilke. Transducer-based analysis of cryptographic protocols. *Information and Computation*, 205(12):1741–1776, 2007. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/j.ic.2007.08.003. URL http://www.sciencedirect.com/science/article/pii/S0890540107000946. 1

[38] Grégoire Laurence, Aurélien Lemay, Joachim Niehren, Slawek Staworko, and Marc Tommasi. Normalization of sequential top-downtree-to-word transducers. In Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications*, volume 6638, pages 354–365. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-21254-3_28. 1, 6.4.2, 6.15, 6.4.2

[39] Yang Liu, Qun Liu, and Shouxun Lin. Tree-to-string alignment template for statistical machine translation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the Association for Computational Linguistics*, ACL-44, pages 609–616, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics. doi: 10.3115/1220175.1220252. URL http://dx.doi.org/10.3115/1220175.1220252. 1

[40] Yang Liu, Yun Huang, Qun Liu, and Shouxun Lin. Forest-to-string statistical translation rules. In *In 45th Annual Meeting of the Association for Computational Linguistics*, 2007. 1

[41] Markus Lohrey. Algorithmics on slp-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012. 4

[42] Andreas Maletti. The power of extended top-down tree transducers. *SIAM J. COMPUT*, 2008. 1

[43] S. Maneth and et al. Xml type checking with macro tree transducers, 2005. 1

[44] Sebastian Maneth, Thomas Perst, and Helmut Seidl. *Database Theory – ICDT 2007: 11th International Conference, Barcelona, Spain, January 10-12, 2007. Proceedings*, chapter Exact XML Type Checking in Polynomial Time, pages 254–268. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-69270-6. doi: 10.1007/11965893_18. URL http://dx.doi.org/10.1007/11965893_18. 1

[45] K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. ISSN 0178-4617. doi: 10.1007/BF02522825. 4, 4.3, 4, 4, 4.1, 4.1.1, 4.3

[46] H. S. Wilf N. J. Fine. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. ISSN 00029939, 10886826. URL http://www.jstor.org/stable/2034009. 4

[47] Wojciech Plandowski. *The Complexity of the Morphism Equivalence Problem for Context-Free Languages*. PhD thesis, Department of Mathematics, Informatics and Mechanics Warsaw University, 1995. 1, 4, 4.3, 4, 4.2, 5, 5.1, 5, 5.4, 5.1, 5.2, 5.5.1, 5.5.2, 5.13, 5.15

[48] William C. Rounds. Mappings and grammars on trees. *Mathematical systems theory*, 4(3):257–287. ISSN 1433-0490. doi: 10.1007/BF01695769. URL http://dx.doi.org/10.1007/BF01695769. 1

[49] Wojciech Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211 – 222, 2003. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(02)00777-6. URL http://www.sciencedirect.com/science/article/pii/S0304397502007776. 4.28.1, 4.28.2

[50] Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(2-4):416 – 430, 2005. ISSN 1570-8667. doi: http://dx.doi.org/10.1016/j.jda.2004.08.016. URL http://www.sciencedirect.com/science/article/pii/S1570866704000632. Combinatorial Pattern Matching (CPM) Special IssueThe 14th annual Symposium on combinatorial Pattern Matching. 4.28.1

[51] Helmut Seidl, Sebastian Maneth, and Gregor Kemper. Equivalence of deterministic top-down tree-to-string transducers is decidable. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 943–962, 2015. doi: 10.1109/FOCS.2015.62. URL http://dx.doi.org/10.1109/FOCS.2015.62. 1

[52] Slawomir Staworko, Grégoire Laurence, Aurélien Lemay, and Joachim Niehren. Equivalence of deterministic nested word to word transducers. In Miroslaw Kutylowski, Witold Charatonik, and Maciej Gebala, editors, *Fundamentals of Computation Theory*, volume 5699 of *Lecture Notes in Computer Science*, pages 310–322. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03408-4. doi: 10.1007/978-3-642-03409-1_28. 1, 1, 2, 5, 6.2, 6.5, 6.2, 6.7, 6.2, 6.7.1, 6.8.1, 8

[53] James W. Thatcher. Generalized2 sequential machine maps. *Journal of Computer and System Sciences*, 4(4):339 – 367, 1970. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/S0022-0000(70)80017-4. URL http://www.sciencedirect.com/science/article/pii/S0022000070800174. 1

[54] JANIS VOIGTLAENDER and ARMIN KUEHNEMANN. Composition of functions with accumulating parameters. *Journal ofFunctional Programming*, 14:317–363, 2004. ISSN 1469-7653. doi: 10.1017/S0956796803004933. URL http://journals.cambridge.org/article_S0956796803004933. 1

[55] David J. Weir. Linear context-free rewriting systems and deterministic tree-walking transducers. In *Proceedings of the 30th Annual Meeting on Association for Computational Linguistics*, ACL '92, pages 136–143, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics. doi: 10.3115/981967.981985. URL http://dx.doi.org/10.3115/981967.981985. 1