EPTCS 388

Proceedings of the

13th International Workshop on

Non-Classical Models of Automata and Applications

Famagusta, North Cyprus, 18th-19th September, 2023

Edited by: Benedek Nagy and Rudolf Freund

Published: 15th September 2023
DOI: 10.4204/EPTCS.388
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

The Thirteenth International Workshop on Non-Classical Models of Automata and Applications (NCMA 2023) was held in Famagusta, North Cyprus, on September 18 and 19, 2023, organized by the Eastern Mediterranean University. The NCMA workshop series was established in 2009 as an annual event for researchers working on non-classical and classical models of automata, grammars or related devices. Such models are investigated both as theoretical models and as formal models for applications from various points of view. The goal of the NCMA workshop series is to exchange and develop novel ideas in order to gain deeper and interdisciplinary coverage of this particular area that may foster new insights and substantial progress.

The previous NCMA workshops took place in the following places: Wrocław, Poland (2009), Jena, Germany (2010), Milano, Italy (2011), Fribourg, Switzerland (2012), Umeå, Sweden (2013), Kassel, Germany (2014), Porto, Portugal (2015), Debrecen, Hungary (2016), Prague, Czech Republic (2017), Košice, Slovakia (2018), Valencia, Spain (2019). Due to the Covid-19 pandemic there was no NCMA workshop in 2020 and 2021. The Twelfth International Workshop on Non-Classical Models of Automata and Applications (NCMA 2022) was organized by the Faculty of Informatics of the University of Debrecen, Hungary. NCMA 2023, organized by the Eastern Mediterranean University, in Famagusta, North Cyprus, was co-located with the 27th International Conference on Implementation and Application of Automata (CIAA 2023, 19-22 September).

The invited lectures at NCMA 2023 have been the following:

- Friedrich Otto (Kassel, Germany): A Survey on Automata with Translucent Letters (joint invited lecture with CIAA 2023)
- György Vaszil (Debrecen, Hungary): Membrane Computing and Petri Nets

The 11 regular contributions have been selected out of 15 submissions by a total of 29 authors from 10 different countries by the following members of the Program Committee:

- Artiom Alhazov (Institute of Mathematics and Computer Science of Academy of Sciences of Moldova)
- Péter Battyányi (University of Debrecen, Hungary)
- Martin Berglund (University of Umeå, Sweden)
- Erzsébet Csuhaj-Varjú (Eötvös Loránd University, Budapest, Hungary)
- Rudolf Freund (TU Wien, Vienna, Austria), co-chair
- Zoltán Fülöp (University of Szeged, Hungary)
- Géza Horváth (University of Debrecen, Hungary)
- Szabolcs Iván (University of Szeged, Hungary)
- Sergiu Ivanov (Paris-Saclay University, France)
- Miklós Krész (Innorenew, Slovenia and University of Szeged, Hungary)
- Zbyněk Křivka (Brno University of Technology, Czech Republic)

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. ii–iii, doi:10.4204/EPTCS.388.0

- Peter Leupold (Germany)
- Ian McQuillan (University of Saskatchewan, Canada)
- Ludovic Mignot (Université de Rouen Normandie, France)
- Nelma Moreira (University of Porto, Portugal)
- Benedek Nagy (Eastern Mediterranean University, Famagusta, North Cyprus and Eszterházy Károly Catholic University, Eger, Hungary), co-chair
- Beatrice Palano (University of Milan, Italy)
- Agustín Riscos-Núñez (Universidad de Sevilla, Spain)
- Jose M. Sempere (Universitat Politècnica de València, Spain)
- Alexander Szabari (UPJŠ Košice, Slovakia)
- Bianca Truthe (University of Giessen, Germany)
- György Vaszil (University of Debrecen, Hungary)

In addition to the invited lectures and the regular submissions, NCMA 2023 also featured three short presentations to emphasize the workshop character. This volume contains the invited and regular presentations.

A special issue of the journal Acta Informatica containing extended versions of selected regular contributions to NCMA 2023 will also be edited after the workshop. The extended papers will undergo the standard refereeing process of the journal.

We are grateful to the two invited speakers, to all authors who submitted a paper to NCMA 2023, to all members of the Program Committee, their colleagues who helped evaluating the submissions, and to the members of the Eastern Mediterranean University who were involved in the local organization of NCMA 2023.

30th of August, 2023.

Rudolf Freund
Benedek Nagy

A Survey on Automata with Translucent Letters

Friedrich Otto
Universität Kassel, Germany

In this talk (see also the survey paper in the co-located CIAA proceedings: [1]), we present the various types of automata with translucent letters that have been studied in the literature. These include the finite automata and the pushdown automata with translucent letters, which are obtained as reinterpretations of certain cooperating distributed systems of a restricted type of restarting automaton, the linear automaton with translucent letters, and the visibly pushdown automaton with translucent letters. For each of these types of automata with translucent letters, it has been shown that they accept those trace languages which are obtained from the class of languages that is accepted by the corresponding type of automaton without translucent letters.

References

[1] Friedrich Otto (2023): *A Survey on Automata with Translucent Letters*. In Benedek Nagy, editor: *Proceedings of the 27th International Conference on Implementation and Application of Automata, CIAA 2023, Famagusta, North Cyprus, September 19–22, 2023*, *Lecture Notes in Computer Science* 14151, Springer, pp. 21–50, doi:10.1007/978-3-031-40247-0_2.

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 1–1, doi:10.4204/EPTCS.388.1

Membrane Computing and Petri Nets

György Vaszil
University of Debrecen, Hungary

When looking at the computations of membrane systems and the behavior of place/transition Petri nets, we might notice several features which are related to each other. Petri net transitions consume tokens from their input places and produce new tokens at their output places, so in some sense they behave similarly to membrane systems which consume, produce, and move objects around in the regions of their membrane structure. Based on these relationships, the functioning of place/transition nets can naturally be described by transformations of multisets corresponding to possible token distributions on the places of the net, while different kinds of objects and object evolution rules in different compartments of a membrane system can be represented by the places and transitions of a Petri net.

In the talk we look in more detail at these structural links between the two models which, on one hand, motivate the examination of membrane systems from the point of view of the concurrent nature of their behavior, and on the other hand, inspires the study of Petri net variants suitable for the modeling of membrane system computations.

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on
Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 2–2, doi:10.4204/EPTCS.388.2

Generating Semantic Graph Corpora with Graph Expansion Grammar

Eric Andersson Johanna Björklund* Frank Drewes† Anna Jonsson
Department of Computing Science
Umeå University, Umeå, Sweden
{dv20ean, johanna, drewes, aj}@cs.umu.se

We introduce LOVELACE, a tool for creating corpora of semantic graphs. The system uses graph expansion grammar as a representational language, thus allowing users to craft a grammar that describes a corpus with desired properties. When given such grammar as input, the system generates a set of output graphs that are well-formed according to the grammar, i.e., a graph bank. The generation process can be controlled via a number of configurable parameters that allow the user to, for example, specify a range of desired output graph sizes. Central use cases are the creation of synthetic data to augment existing corpora, and as a pedagogical tool for teaching formal language theory.

1 Introduction

Semantic representations are formalisms designed to express the meaning of natural language data in a clear and concise way, which is suitable both for manual inspection and for automated processing. A wide range of representational formats has been considered in literature. Some of the more commonly used are based on graphs, in which nodes correspond to concepts, and edges to relations between them. Prominent examples are combinatory categorial grammar [19], abstract meaning representation (AMR) [12, 3] and universal conceptual cognitive annotation [1].

It would be valuable for many applications if one could automatically translate natural language sentences into semantic graphs. However, for developing, training, and testing such approaches, corpora like the AMR corpora[1] are required. The creation of high-quality corpora is work intensive and requires both linguistic knowledge and a familiarity with the representational formalism at hand. Moreover, even skilled annotators tire, and hence the resulting translations are bound to contain errors and inconsistencies. In addition to this, hand-annotated real-world data is of limited use for conducting controlled experiments whose purpose it is to study the influence of particular structural properties of the representation on a given machine learning technique.

To address these problems, we provide the software LOVELACE[2] that generates well-formed graphs with respect to a *graph expansion grammar* (GEG) [7]. GEGs are hyperedge replacement grammars [5, 10, 8] that have been extended by a type of contextual rules inspired by [9].

Technically, a GEG is defined as a regular tree grammar that generates terms over a particular graph algebra, and these terms are then evaluated into a set of directed acyclic graphs. As usual, the evaluation of a term is done recursively. Assuming that a given subterm has already been evaluated to a graph, which will become a subgraph of the generated graph, the evaluation of an operation on top of it adds new nodes

*Supported by the Swedish Research Council under grant number 2020-03852
†Supported by the Wallenberg AI, Autonomous Systems and Software Program through the NEST project *STING*
[1] https://amr.isi.edu
[2] https://github.com/tm11ajn/lovelace/

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on
Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 3–15, doi:10.4204/EPTCS.388.3

with edges pointing to already existing nodes of the subgraph. In [7], the placement of these edges can be restricted by a formula in counting monadic second-order logic. LOVELACE does not currently make use of such a powerful mechanism, which we leave for future extensions. In another respect (to be discussed in Section 2), we generalise expansion operations slightly, which ensures that the formalism becomes more powerful than hyperedge replacement. This deviation from the original definition [7] is motivated by the fact that the focus in that work was on polynomial parsing, whereas LOVELACE is a generative tool for which the well-known NP-completeness of hyperedge replacement languages is of no relevance.

There are several semantically annotated treebanks available, including PropBank [15], FrameNet [2], and the Penn Discourse TreeBank [17, 16]. There are also tools that generate synthetic treebanks from grammars, which can, if so designed, contain semantic information. In this category of tools we have Grammatical Framework [18], a programming language specifically designed for writing string grammars, but which also provides functionality for generating corpora of parse trees with respect to a given grammar. Another example is Tiburon [13], a capable toolkit for processing weighted automata which includes an algorithm for extracting N parse trees with optimal weight from a weighted string grammar. Finally we have BETTY [6], which can extract both the N best derivation trees, but also the N best output trees with respect to a tree grammar; cf. Section 3.

Turning specifically to graph banks, Hockenmeir and Steedman propose an algorithm for translating the Penn Treebank into a corpus of CCG derivations augmented with local and long-range word–word dependencies [11]. There is also the manually created AMR bank by [4]. The present paper adds to this line of work by providing a method of creating synthetic corpora of semantic graphs from a specification given in the form of a graph expansion grammar.

The paper contains the following main sections: Section 2 recalls the graph expansion grammar formalism, Section 3 explains how to find and use the software, and Section 4 provides a summary of the work presented here together with ideas for improvement.

2 Graph Expansion Grammar

To recall the graph expansion grammar formalism [7], we first fix a few standard definitions and related notation from discrete mathematics and automata theory.

The set of natural numbers (including 0) is denoted by $\mathbb{N}$, and $[n] = \{1, \ldots, n\}$ for $n \in \mathbb{N}$. The set of all strings (that is, finite sequences) over a set S is S^*, which in particular contains the empty string ε. The subset of S^* containing only those strings which do not have repeating elements is $S^{\circledast}$. For a string w, we let $[w]$ denote the smallest set S such that $w \in S^*$. We denote the canonical extensions of a function $f\colon S \to T$ to S^* and to the powerset $\wp(S)$ of S also by f, i.e., $f(s_1 \cdots s_n) = f(s_1) \cdots f(s_n)$ for $s_1, \ldots, s_n \in S$, and $f(S') = \{f(s) \mid s \in S'\}$ for $S' \in \wp(S)$.

A *ranked alphabet* is a pair $A = (\Sigma, rk)$ consisting of a finite set of symbols Σ and a function $rk\colon \Sigma \to \mathbb{N}$ that assigns a rank to every symbol $\sigma \in \Sigma$. Writing $\sigma^{(k)}$ indicates that $rk(\sigma) = k$. If there is no danger of confusion, we keep rk implicit and identify A with Σ.

The set T_Σ of all *trees over* Σ is the smallest set of formal expressions such that $\sigma[t_1, \ldots, t_k] \in \mathrm{T}_\Sigma$ for every $\sigma^{(k)} \in \Sigma$ and all trees $t_1, \ldots, t_k \in \mathrm{T}_\Sigma$. Thus, the rank k of σ determines the number of subtrees of every occurrence of σ in a tree. If $k = 0$, then $f[] \in \mathrm{T}_\Sigma$, which we abbreviate as f, omitting the brackets.

Given a ranked alphabet Σ as above, a Σ-*algebra* is a pair $\mathscr{A} = (\mathbb{A}, (f_\mathscr{A})_{f \in \Sigma})$ consisting of a set $\mathbb{A}$, the *domain* of $\mathscr{A}$, and a function $f_\mathscr{A}\colon \mathbb{A}^k \to \mathbb{A}$ for every $f^{(k)} \in \Sigma$, the *interpretation* of f in $\mathscr{A}$. Now, if $t = f[t_1, \ldots, t_k]$ is a tree in T_Σ, evaluating t with respect to $\mathscr{A}$ yields $val_\mathscr{A}(t) \in \mathbb{A}$, defined as $val_\mathscr{A}(t) = f_\mathscr{A}(val_\mathscr{A}(t_1), \ldots, val_\mathscr{A}(t_k))$.

To generate trees over the operations of an algebra, we use *regular tree grammars*.

Definition 1 *A* regular tree grammar (over Σ) *is a tuple* $g = (N, \Sigma, P, S)$ *consisting of*

- *a ranked alphabet* N *of symbols of rank* 0*, called* nonterminals*,*
- *a ranked alphabet* Σ *of* terminals*, disjoint with* N*,*
- *a set* P *of* productions $A \to f[A_1, \ldots, A_k]$ *where* $f^{(k)} \in \Sigma$ *for some* $k \in \mathbb{N}$ *and* $A, A_1, \ldots, A_k \in N$*, and*
- *an* initial nonterminal $S \in N$*.*

The regular tree language *(rtg) generated by* g *is* $L(g) = L_S(g)$ *where* $(L_A(g))_{A \in N}$ *is the smallest family of subsets of* T_Σ *such that, for* $A \in N$*, a tree* $f[t_1, \ldots, t_k]$ *is in* $L_A(g)$ *if* $(A \to f[A_1, \ldots, A_k]) \in P$ *and* $t_i \in L_{A_i}(g)$ *for all* $i \in [k]$*. (See Figures 3 and 4 for an example regular tree grammar and a tree in its language, respectively.)*

To generate languages other than tree languages using regular tree grammars, we follow the idea of the seminal paper by Mezei and Wright [14]: the combination of a regular tree grammar g over Σ and a Σ-algebra $\mathscr{A}$ generates the subset of $\mathbb{A}$ whose elements are all $val_{\mathscr{A}}(t)$ such that $t \in L(g)$. In our case, $\mathbb{A}$ is the set of graphs (over a given set of labels). The operations are, thus, operations on graphs. However, the central operation is nondeterministic, meaning that its application to a given graph can produce several possible outputs. Formally, we model this by letting the operations work on sets of graphs instead of individual graphs.

The graphs we work with are node- and edge-labelled directed graphs, each equipped with a sequence of so-called ports. From a graph operation point of view, the sequence of ports of a graph is its "interface": its nodes are the only ones that can individually be accessed by operations to attach new edges to them. The number of ports is the *type* of the graph.

Definition 2 *Let* $\mathbb{L} = (\dot{\mathbb{L}}, \bar{\mathbb{L}})$ *be a* labelling alphabet*: a pair of finite sets of labels* $\dot{\mathbb{L}}$ *and* $\bar{\mathbb{L}}$*. A* graph *over* $\mathbb{L}$ *is a tuple* $G = (V, E, lab, port)$ *such that*

- V *is the finite set of* nodes*,*
- $E \subseteq V \times \bar{\mathbb{L}} \times V$ *is the set of* edges*,*
- $lab \colon V \to \dot{\mathbb{L}}$ *labels the nodes, and*
- $port \in V^{\circledast}$ *is the sequence of* ports *of the graph.*

The type *of* G *is* $type(G) = |port|$*. The set of all graphs of type* k *is denoted by* $\mathbb{G}_k$*.*

If the components of a graph G are not explicitly named, they are denoted by V_G, E_G, lab_G, and $port_G$, respectively.

Graph expansion grammars generate graphs using two types of graph operations: disjoint union and the more complex graph expansion operations. Disjoint union just combines two graphs into one by placing them next to each other (after making their node sets disjoint) and concatenating their port sequences. Formally, let $k, k' \in \mathbb{N}$. Then $\uplus_{kk'} \colon \mathbb{G}_k \times \mathbb{G}_{k'} \to \mathbb{G}_{k+k'}$ is defined as follows: for $G \in \mathbb{G}_k$ and $G' \in \mathbb{G}_{k'}$ with disjoint sets of nodes, $\uplus_{kk'}(G, G')$ yields the graph $(V, E, lab, port) \in \mathbb{G}_{k+k'}$ given by $V = V_G \cup V_{G'}$, $E = E_G \cup E_{G'}$, $lab = lab_G \cup lab_{G'}$, and $port = port_G port_{G'}$.[3] If $V_G \cap V_{G'} \neq \emptyset$, we silently rename nodes before we apply $\uplus_{kk'}$, because we are only interested in generating graphs up to isomorphism. Note that $\uplus_{kk'}$ is not commutative because of the concatenation of port sequences. We usually write $G \uplus_{kk'} G'$ instead of $\uplus_{kk'}(G, G')$. We extend $\uplus_{kk'}$ to $\uplus_{kk'} \colon \wp(\mathbb{G}_k) \times \wp(\mathbb{G}_{k'}) \to \wp(\mathbb{G}_{k+k'})$ by letting $\mathscr{G} \uplus_{kk'} \mathscr{G}' = \{G \uplus_{kk'} G' \mid G \in \mathscr{G},\ G' \in \mathscr{G}'\}$ for $\mathscr{G} \subseteq \mathbb{G}_k$ and $\mathscr{G}' \subseteq \mathbb{G}_{k'}$.

[3] Here, $lab_G \cup lab_{G'}$ is the usual union of binary relations.

The other type of operation, the *graph expansion*, extends an existing graph with an additional structure placed "on top" of that graph. Expansion is specified by a template graph with an additional sequence of designated nodes called *docks*. Applying an extension operation adds the template graph to the argument graph and identifies the docks with the ports of that graph. The ports of the template become the ports of the combined graph. The template also contains a number of *context nodes* that can be identified with arbitrarily chosen nodes with matching labels in the argument graph. Formally, a graph expansion operation is a unary operation given by a tuple $\Phi = (V, E, lab, port, dock)$ where $(V, E, lab, port)$, henceforth denoted by $\underline{\Phi}$, is the *underlying graph* and $dock \in V^*$ is the sequence of *docks*. Note that *dock*, in contrast to *port*, may contain repetitions. Similarly to our notation for the components of graphs, we use the notations V_Φ, E_Φ, lab_Φ, $port_\Phi$, and $dock_\Phi$ if these components are not explicitly named. Furthermore, we let $C_\Phi = V \setminus ([port] \cup [dock])$ denote the set of *context nodes* of Φ.

An expansion operation Φ as above can be applied to an argument graph $G = (V, E, lab, port) \in \mathbb{G}_\ell$ if $|dock_\Phi| = \ell$. It then yields a graph of type $|port_\Phi|$ by identifying the nodes in $dock_\Phi$ with those in *port*, and each context node with an arbitrary node in V that carries the same label. The port sequence of the resulting graph is $port_\Phi$.

Formally, let $|port_\Phi| = k$ and $|dock_\Phi| = \ell$. Then Φ is interpreted as the nondeterministic operation $\Phi\colon \mathbb{G}_\ell \to \wp(\mathbb{G}_k)$ defined as follows. For a graph $G = (V, E, lab, port) \in \mathbb{G}_\ell$, a graph $H \in \mathbb{G}_k$ is in $\Phi(G)$ if it can be obtained by the following stepewise procedure:

1. Rename the nodes of Φ to make the set of nodes of Φ disjoint with V. (As in the case of $\uplus$, we will in the following assume that this is done silently "under the hood".)
2. Add the nodes and edges of $\underline{\Phi}$ to G.
3. Identify the i-th node v of *port* with the i-th node of $dock_\Phi$ for all $i \in [\ell]$ and label the resulting node with $lab_\Phi(v)$.
4. Identify every node $u \in C_\Phi$ with any node $v \in V \setminus [port]$ for which $lab(v) = lab_\Phi(u)$.
5. Define $port_H = port_\Phi$.

Note that the process of identifying docks of Φ with ports of the argument graph G may merge ports of G if *dock* contains repetitions. The expansion operations defined here are thus more general than those in [7]. In fact, readers familiar with hyperedge replacement grammars will easily be able to see that this allows us to simulate hyperedge replacement. Together with the fact that context nodes can be used to create graphs of unbounded treewidth, this implies that graph expansion grammars, to be defined below, are strictly more powerful than hyperedge replacement grammars.

Further deviations from [7] are that the definition above does not make use of the cloning of context nodes, and that the logic formula that determines which mappings of context nodes to nodes in the argument graph are allowed has been replaced by the much simpler condition that node labels must match. The cloning ability is not needed here since we consider expansion operations rather than the special case of extension operations as in [7] (see below), which means that cloning can be implemented by repeated application of expansion. The latter has been dropped in the current paper for simplicity, and because it is not yet implemented in LOVELACE anyway.

The major result of [7] applies to a restricted form of expansion operations, the so-called extension operations. By using only extension operations, we can make sure that graphs are built bottom-up, that is, that Φ always extends the input graph by placing nodes and edges "on top", with edges being directed downwards, and that all nodes of generated graphs are reachable from the ports. For a brief explanation, let $NEW_\Phi = [port_\Phi] \setminus [dock_\Phi]$ denote the set of nodes that an application of Φ adds to the graph, i.e. those nodes of $\underline{\Phi}$ which are not identified with nodes of the argument graph when Φ is applied. Then Φ is an extension operation if it satisfies the following requirements:

(R1) $E_\Phi \subseteq NEW_\Phi \times \bar{\mathbb{L}} \times (V_\Phi \setminus NEW_\Phi)$ and
(R2) every node in $[dock_\Phi] \setminus [port_\Phi]$ has an incoming edge.

By induction, (R1) ensures that all graphs generated by a graph extension grammar (i.e., a GEG all of whose expansion operations are extension operations) are directed acyclic graphs. Likewise by induction, (R2) ensures that every node in a graph generated by a graph extension grammar is reachable from a port. While these restrictions are not employed in the current paper (since they are not needed unless one is interested in efficient parsing), they are well justified when generating semantic graphs such as AMR, because these typically consist of directed acyclic graphs in which all nodes are reachable from the roots (which would translate to ports in the graph grammar formalism). Thus, while LOVELACE does not enforce (R1) and (R2), our examples will actually obey these requirements.

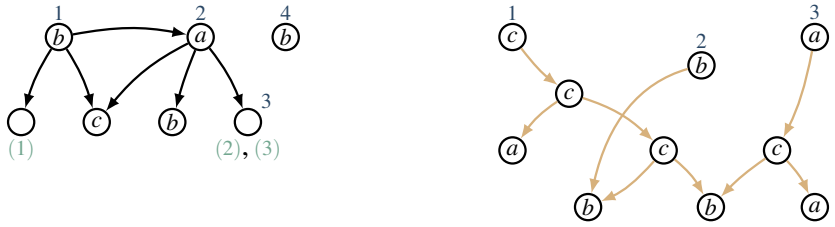


Figure 1: The figure on the left shows an expansion operation Φ with four ports (indicated with numbers above the nodes), three docks (indicated with numbers in parentheses below the nodes), and two context nodes (the ones that are neither ports nor docks). Docks 2 and 3 coincide. Applying the expansion operation identifies docks with corresponding ports of the argument graph and each context node with a non-port in the input graph that carries a matching label. The application of Φ to the graph G (on the right) yields a non-empty number of possible results because the number of ports of G coincides with the number of docks of Φ, and since there are nodes labelled b and c in G which are not ports.

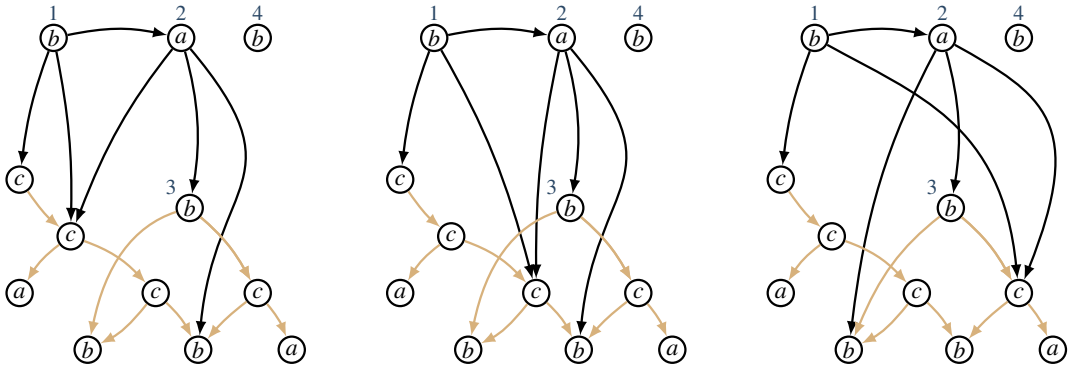


Figure 2: Three graphs in $\Phi(G)$ where Φ and G are as in Figure 1. The differences between the graphs reflect how the context nodes in Φ were chosen to be mapped to nodes in G.

Figure 1 depicts an expansion operation together with a graph to which it can be applied. Figure 2 shows three different graphs, all resulting from the application of the expansion operation to the (now argument) graph in Figure 2. The resulting graphs differ because different mappings of context nodes to nodes in the argument graph were chosen. Note that Φ fuses ports 2 and 3 of the argument graph, which become port 3 of the result, because docks 2 and 3 coincide.

A *graph expansion algebra* is a Σ-algebra $\mathscr{A} = (\wp(\mathbb{G}), (f_{\mathscr{A}})_{f\in\Sigma})$ where every symbol in Σ is interpreted as an expansion operation, a union operation, or the set $\{\phi\}$, where ϕ is the empty graph $(\emptyset,\emptyset,\emptyset,\varepsilon)$. As previously mentioned, the operations of the algebra act on sets of graphs rather than on single graphs, due to the nondeterministic nature of expansion. This also takes care of the fact that operations are only defined on graphs of matching types: we simply use the convention that the application of an operation to a graph of an inappropriate type returns the empty set.

Definition 3 *A* graph expansion grammar *is a pair* $\Gamma = (g, \mathscr{A})$ *where* $\mathscr{A}$ *is a graph expansion* Σ*-algebra for some ranked alphabet* Σ *and* g *is a regular tree grammar over* Σ*.*

$$L(\Gamma) = \bigcup_{t\in L(g)} val_{\mathscr{A}}(t)$$

is the graph language generated by Γ.

3 LOVELACE

Let us now make use of the capacity of graph expansion grammars for expressing semantic graph languages to create a semantic graph generator. We named the software tool that implements this functionality LOVELACE[4]. To use LOVELACE, one needs to have access to, or themselves define, a graph expansion grammar describing a language that contains the wanted corpora. In the rest of this section, we explain in greater detail how to combine LOVELACE with the tool BETTY[5] to generate graph corpora.

BETTY operates on weighted regular tree grammars, that is, on rtgs in which the rules are equipped with weights. In the case of BETTY, these must be taken from the tropical semiring. The resulting grammars work precisely like those in Definition 1, but assign an additional weight to every generated tree, computed as follows. The weight of a derivation is the sum of all weights of the rules applied to generate the tree. The weight of a tree in the language is the minimum of all weights of derivations that yield that tree. BETTY takes as input such a weighted grammar and some natural number N, and outputs N *best trees*, that is, N pairwise distinct trees of least weight (in the order of increasing weight). Thus, in this context, lesser weight is better. In the case of ties, BETTY gives precedence to smaller trees. In particular, assigning all rules the same weight results in picking N smallest possible trees from the generated language. It is in fact unnecessary to provide rules with an explicit weight as BETTY interprets rules without a weight as rules of weight 0. For simplicity, the example we use below in order to illustrate the generation of corpora makes use of this possibility.

To generate the semantic graph corpora, a two-step approach is used: First N best trees are extracted from the (now weighted) regular tree grammar component of the graph expansion grammar, and these are then evaluated with respect to the algebra. As there is currently no direct integration of BETTY and LOVELACE, this pipeline must be set up manually. Syntactically, the input format to BETTY is the `rtg` format of [13]; see that paper for more information. An example regular tree grammar on `rtg` format can be seen in Figure 3, and Figure 4 shows an example tree in the corresponding language. The trees that BETTY then outputs are the derivation trees that comprise the basis of the corpus.

In the next step of the generation process, the derivation trees are translated into graphs using LOVELACE. To do this, we must specify the graph expansion algebra. In other words, we must associate an operation with every terminal in the regular tree grammar and gather them in an operation file.

[4] https://github.com/tm11ajn/lovelace/
[5] https://github.com/tm11ajn/betty/

```
1 S
2 S -> op1(C)
3 C -> op2(U)
4 U -> op3(S' S)
5 S' -> op4
6 S -> op5
```

Figure 3: A regular tree grammar (on `rtg` format). The first nonterminal in the file represents the starting nonterminal.

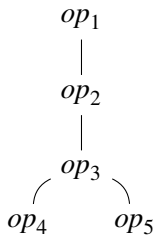


Figure 4: A visual representation of the unique tree `op1[op2[op3[op4, op5]]]` in the language generated by the regular tree grammar in Figure 3 on the left.

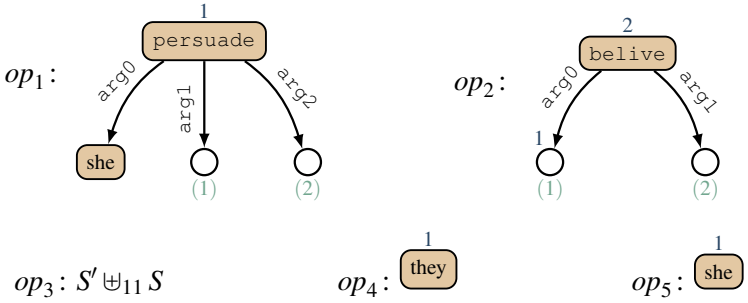


Figure 5: A definition of five graph operations. Here, op_3 is a union operation that takes two argument graphs with one port each, and the remaining operations are graph expansion operations.

Such a set of operations for the tree of Figure 4 is depicted in Figure 5. Union operations and expansion operations have similar textual formats. Both use the keyword `operation` together with the name of the operation (i.e., the corresponding terminal in the regular tree grammar) and curly brackets to enclose the operation specification. A union operation is specified – as seen in Figure 6 – using a single line of two numbers referring to the number of ports of the two input arguments. An expansion operation must necessarily specify a graph with ports and docks, which is why we found it convenient to base the representation on the `gv` digraph format used by the open-source tool Graphviz[1] (see Figures 9 and 10 for an example). In Figure 7, we provide an example expansion operation that corresponds to the operation op_1 of Figure 5. The only addition to the Graphviz format is that the user must specify which nodes are ports and docks by enumerating them using the keywords `port` and `dock`, respectively. We see that node `0` is the only port of the operation, and that nodes `2` and `3` are its docks.

Once we have both a file specifying the operations and a file of derivation trees, we can input them to LOVELACE by using the mandatory parameters `-g` and `-t`, respectively. An example usage of LOVELACE is thus given by

```
java lovelace.java -g file-of-operations.txt -t file-of-trees.txt
```

LOVELACE will then evaluate the trees into graphs (by interpreting the nodes of the trees as graph operations) and output them. The process of evaluating the tree in Figure 4 with respect to the operations

[1] https://graphviz.org/

```
operation op3 {
  1 1
}
```

Figure 6: A textual representation of the union operation `op3` that takes two graphs with one port each and turns them into a single graph with two ports.

```
operation op1 {
  0 [label="persuade", port=1]
  1 [label="she"]
  2 [dock=1]
  3 [dock=2]
  0 -> 1 [label="arg0"]
  0 -> 2 [label="arg1"]
  0 -> 3 [label="arg2"]
}
```

Figure 7: A textual representation of the expansion operation op_1 in Figure 5. The name of the operation is `op1`, which is also the label that is used in a tree grammar file to refer to this operation.

in Figure 5 is depicted in Figure 8. Each output graph is saved as a single text file in `gv` format (one such file resulting from our running example is depicted in Figure 9), which makes their visualisation by Graphviz easy. We recommend using the Graphviz online tool[6] for quick and easy graph visualisation. An example of a visualisation of the graph in Figure 9 by Graphviz is shown in Figure 10.

In addition to its basic functionality, LOVELACE allows the user to generate graphs with abstract labels, which are then replaced by concrete labels when the generated graphs are outputted. More precisely, the user can provide definitions of one-to-many label replacements, and the system will then output all possible instantiations based on these replacements. Such definitions are provided in a text file passed as an argument to LOVELACE via the `-d` option. This text file should include definitions for every label that shall be replaced by one or more labels. For example, we can generate graphs with the abstract label `sing-pronoun` which is then replaced by singular pronouns to generate various valid semantic graphs from a single result of the generation process. As a more sophisticated example, we can expand an abstract label representing the VerbNet class `conjecture-29.5-1` (which contains, for example, the verb `believe`) by any verbs in the same class. In Figure 11, we provide a definition file in which the concepts `they`, `she` and `believe` have been expanded to provide a richer variety of semantic graphs. When such a file is provided to LOVELACE, all combinations of the replacements are used to create more semantic graphs. This naturally yields a combinatorial explosion, which is why this option should be used with care. An alternative way of instantiating graphs is discussed in Section 4.

The three remaining parameters of the program are quite straight-forward: `-L` specifies the minimum number of nodes that a generated graph can have, `-H` is similar but instead provides an upper bound of nodes, and `-k` takes an operation name as an argument and forces every generation to use that particular operation at least once. In Section 4, we discuss other potential parameters for fine-tuning the output data that a user might be interested in.

To summarise the above information, we have collected the parameters implemented thus far in a cheat sheet, see Figure 12.

[6] https://dreampuf.github.io/GraphvizOnline/

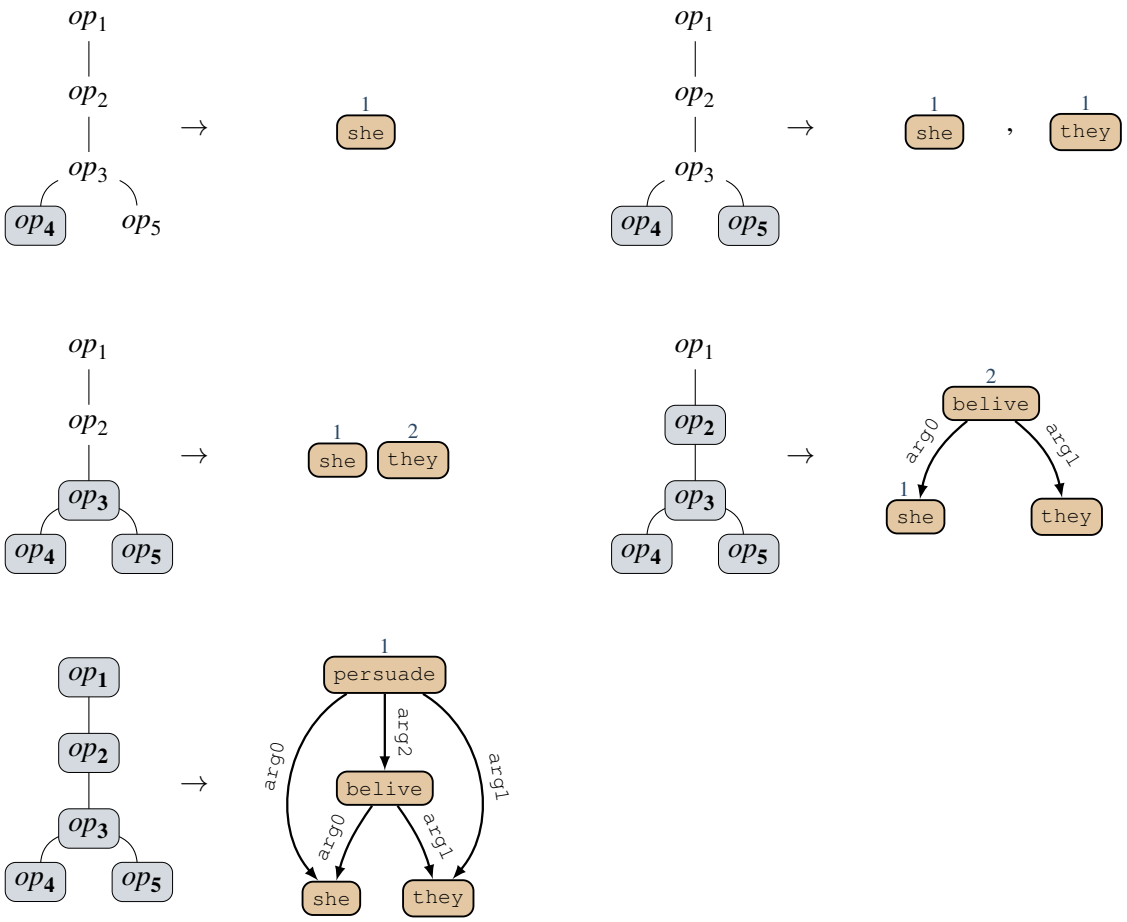


Figure 8: The bottom-up evaluation of the tree in Figure 4 produced by the regular tree grammar in Figure 3 into a graph, using the operations defined in Figure 5. When the operation corresponding to a node in the tree is applied, the node is marked to make the derivation process clearer.

4 Conclusion and Future Work

We have presented the software LOVELACE that generates corpora of semantic graphs; it is based on the formalism of graph expansion grammar. To improve the software, we would appreciate input as to what features would be useful to the natural language processing community. Below, we list some of the currently planned improvements.

As described in the previous section, BETTY and LOVELACE are currently not integrated. The user first applies the N-best extraction software BETTY to a weighted regular tree grammar and then inputs the resulting list of trees to LOVELACE, together with a file specifying graph operations and other parameters, to output a graph corpus. To make the process smoother, we plan on integrating BETTY into LOVELACE so that the transition between both steps happens automatically. This integration would require LOVELACE to take additional input parameters such as the desired size of the corpus.

The graph expansion operations used in this paper are a modified version of those used in [7]. In

```
digraph G  {
   0 [label="they"]
   1 [label="she"]
   2 [label="believe"]
   3 [label="persuade"]

   2 -> 0 [label="arg0"]
   2 -> 1 [label="arg1"]
   3 -> 1 [label="arg0"]
   3 -> 0 [label="arg1"]
   3 -> 2 [label="arg2"]
}
```

Figure 9: A file in the Graphviz format `gv` representing the graph resulting from evaluating the tree in Figure 4 with respect to the operations in Figure 5.

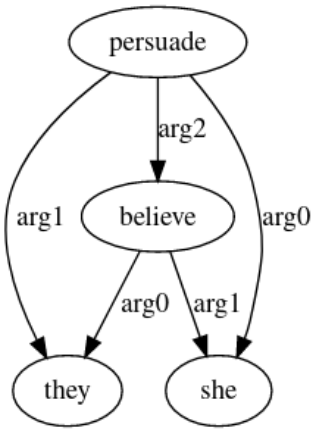


Figure 10: A visualisation of the output file depicted in Figure 9, created using Graphviz.

```
conjecture-29.5-1 = presume trust guess believe
sing-pronoun = they he she
```

Figure 11: Example definition file.

some respects they are more general, while in others they are more restricted. The major differences are the following ones:

- Graph expansion operations are allowed to contain repetitions in the sequence of docks. This ensures that graph expansion grammars can generate all hyperedge replacement languages. For a generation system such as LOVELACE, this is desirable whereas in the context of [7], which focuses on parsing, it is detrimental as it implies that NP-complete graph languages can be generated.
- In this paper, context nodes can be mapped to arbitrary nodes in the argument graph of a graph expansion operation, provided that labels match. In [7], admissible mappings are specified by counting monadic second-order logic, which is a much more powerful mechanism not yet implemented in LOVELACE.
- Finally, we do not make use of the mechanism of *cloning* context nodes. The reason is that we consider expansion operations rather than the more restricted graph extension operations (cf. the earlier discussion of requirements (R1) and (R2)). The former can implement cloning by iterated application of rules, which makes the use of this concept unnecessary. However, cloning may be added as an optional feature in the future to enable the user to make the rule set more compact.

Another planned area of improvement concerns the implementation of mapping the context nodes to nodes in the argument graph. Currently, this is done by randomly choosing a node in the argument graph with a matching label. If there is no matching candidate, then the expansion operation cannot be applied, and the program returns an error message. This is a deviation from the formal definition in two ways. On

`-t <tree file>` Mandatory parameter whose argument should be the file that lists the derivation trees.

`-g <operation file>` Mandatory parameter whose argument specifies the input expansion operations. Note that for every label of the input regular tree grammar file, there should be exactly one expansion operation specified.

`-L <min num nodes>` Sets the minimum number of nodes in the tree. For instance, if the argument is 4, then the program skips every derivation tree that has less than 4 nodes.

`-H <max num nodes>` Sets the maximum number of nodes in the tree. Its functionality is analogous to that of `L`'s, but it sets an upper bound of nodes instead of a lower bound.

`-d <definition file>` This argument allows the user to replace labels in the generated graphs to create the combination of all defined label replacements.

`-k <operation name>` Only generates the graphs whose generation process includes the specified operation. An example usage is `-k op3`.

Figure 12: Parameter cheat sheet.

the one hand, only a single graph is returned, even though there may in fact be several results due to the nondeterminism in the formal definition. Second, if there is no matching candidate at all, the tree should simply contribute zero resulting graphs to the generated corpus instead of producing an error message. (A warning message should, however, be issued as the situation may indicate a modelling error.) One possibility would be to implement an option to have all of the formally generated graphs being outputted. Of course, this may in general cause a combinatorial explosion, similarly to how instantiating nodes using a definition file may result in a combinatorial explosion.

The possibility, mentioned above, to use logical formulas to guide the mapping of context nodes is not the only way in which to improve the user control of the context node mapping. In fact, such a control mechanism may be seen as an independent module which can be implemented in whatever way suitable. In particular, we are planning to study ways of instantiating it by neural mechanisms, which would make it possible to use machine learning to learn valid context node mappings.

Finally, future work includes developing more options to more easily fine-tune the graph generation. There are plenty of ways in which we could extend the parameters that can be used to tweak the semantic graph corpora. One idea is to fine-tune the `-d` parameter: one may want the system to pick a random concept from each definition set for each instance of the concept instead of using all of the combinations. Another idea is to control more in detail what concepts should show up in the output corpus and also to what extent, that is, a type of filtering of the corpus. There are several options to achieve such filtering functionality, and future work will investigate these possibilities.

Acknowledgements

We are grateful to the anonymous reviewers for their insightful and constructive comments which helped improve the quality of this article.

References

[1] Omri Abend & Ari Rappoport (2013): *Universal Conceptual Cognitive Annotation (UCCA)*. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Association for Computational Linguistics, Sofia, Bulgaria, pp. 228–238. Available at https://aclanthology.org/P13-1023.

[2] Collin F. Baker, Charles J. Fillmore & John B. Lowe (1998): *The Berkeley FrameNet Project*. In: *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1*, Association for Computational Linguistics, Montreal, Quebec, Canada, pp. 86–90, doi:10.3115/980845.980860. Available at https://aclanthology.org/P98-1013.

[3] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer & Nathan Schneider (2013): *Abstract Meaning Representation for Sembanking*. In: *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, ACL, Sofia, Bulgaria, pp. 178–186.

[4] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer & Nathan Schneider (2014): *Abstract Meaning Representation (AMR) 1.2 Specification*. Technical Report, Information Science Institute, University of Southern California, CA, USA.

[5] Michel Bauderon & Bruno Courcelle (1987): *Graph Expressions and Graph Rewriting*. *Mathematical Systems Theory* 20, pp. 83–127, doi:10.1007/BF01692060.

[6] Johanna Björklund, Frank Drewes & Anna Jonsson (2022): *Improved N-Best Extraction with an Evaluation on Language Data*. *Computational Linguistics* 48(1), pp. 119–153, doi:10.1162/coli_a_00427. Available at https://aclanthology.org/2022.cl-1.4.

[7] Johanna Björklund, Frank Drewes & Anna Jonsson (2023): *Generation and Polynomial Parsing of Graph Languages with Non-Structural Reentrancies*. *Computational Linguistics*, pp. 1–41, doi:10.1162/coli_a_00488.

[8] Frank Drewes, Annegret Habel & Hans-Jörg Kreowski (1997): *Hyperedge Replacement Graph Grammars*. In G. Rozenberg, editor: *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*, chapter 2, World Scientific, Singapore, pp. 95–162, doi:10.1142/3303.

[9] Frank Drewes, Berthold Hoffmann, Dirk Janssens & Mark Minas (2010): *Adaptive star grammars and their languages*. *Theoretical Computer Science* 411, pp. 3090–3109, doi:10.1016/j.tcs.2010.04.038.

[10] Annegret Habel (1992): *Hyperedge Replacement: Grammars and Languages*. *Lecture Notes in Computer Science* 643, Springer, doi:10.1007/BFb0013875.

[11] Julia Hockenmaier & Mark Steedman (2007): *CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank*. *Computational Linguistics* 33(3), pp. 355–396, doi:10.1162/coli.2007.33.3.355. Available at https://aclanthology.org/J07-3004.

[12] Irene Langkilde & Kevin Knight (1998): *Generation That Exploits Corpus-based Statistical Knowledge*. In: *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics (Volume 1), Montreal, Quebec*, pp. 704–710, doi:10.3115/980845.980963.

[13] Jonathan May & Kevin Knight (2006): *Tiburon: A Weighted Tree Automata Toolkit*. In Oscar H Ibarra & Hsu-Chun Yen, editors: *Implementation and Application of Automata: 11th International Conference, CIAA 2006, Taipei, Taiwan, August 21-23, 2006. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 102–113, doi:10.1007/11812128_11.

[14] Jorge Mezei & Jesse B. Wright (1967): *Algebraic Automata and Context-Free Sets*. *Information and Control* 11, pp. 3–29, doi:10.1016/S0019-9958(67)90353-1.

[15] Martha Palmer, Daniel Gildea & Paul Kingsbury (2005): *The Proposition Bank: An Annotated Corpus of Semantic Roles*. *Computational Linguistics* 31(1), pp. 71–106, doi:10.1162/0891201053630264. Available at https://aclanthology.org/J05-1004.

[16] Rashmi Prasad, Nikhil Dinesh, Alan Lee, Eleni Miltsakaki, Livio Robaldo, Aravind Joshi & Bonnie Webber (2008): *The Penn Discourse TreeBank 2.0*. In: *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, European Language Resources Association (ELRA), Marrakech, Morocco. Available at http://www.lrec-conf.org/proceedings/lrec2008/pdf/754_paper.pdf.

[17] Rashmi Prasad, Aravind Joshi, Nikhil Dinesh, Alan Lee, Eleni Miltsakaki & Bonnie Webber (2008): *The Penn Discourse TreeBank as a resource for natural language generation*. In: *Proc. of the Corpus Linguistics Workshop on Using Corpora for Natural Language Generation*, p. 25–32.

[18] Aarne Ranta (2004): *Grammatical framework*. *Journal of Functional Programming* 14(2), pp. 145–189, doi:10.1017/S0956796803004738.

[19] Mark Steedman & Jason Baldridge (2011): *Combinatory categorial grammar*. John Wiley & Sons Inc., United States, doi:10.1002/9781444395037.ch5.

Formalizing BPE Tokenization

Martin Berglund
Department of Computing Science
Umeå University
Umeå, Sweden
mbe@cs.umu.se

Brink van der Merwe
Department of Computer Science
Stellenbosch University
Stellenbosch, South Africa
abvdm@cs.sun.ac.za

In this paper, we formalize practical byte pair encoding tokenization as it is used in large language models and other NLP systems, in particular we formally define and investigate the semantics of the SentencePiece and HuggingFace tokenizers, in particular how they relate to each other, depending on how the tokenization rules are constructed. Beyond this we consider how tokenization can be performed in an incremental fashion, as well as doing it left-to-right using an amount of memory constant in the length of the string, enabling e.g. using a finite state string-to-string transducer.

1 Introduction

Many modern NLP systems, for example large language models such as the GPT models which underpin services like ChatGPT [6], operate on a *tokenization* of text. This tokenization defines an alphabet of symbols (in the formal languages sense) which include as many common words and fragments of words as possible. For example, the OpenAI GPT-2 model has an alphabet size (a "vocabulary" in their terminology) of 50,257 tokens, which is enough to turn the sentence "taking a ride on a boat" into[1] the token sequence "taking a ride on a boat", as all words are common enough to be in the alphabet, but "partaking in a nautical excursion aboard a vessel" is tokenized as "part aking in a n autical exc ursion ab oard a ves sel", as the words are uncommon, but fragments of the words are common enough. This alphabet is then more semantically rich. For example, forming unusual plurals (turning "earths" into "earth s"), or making a noun into a "non-existing" verb (i.e. turning "verb" into "verb ing"), retaining the informative root word, but also it generally makes the model more robust to misspellings and other minor transformations of the text [7].

One common way of performing this tokenization is by byte pair encoding [7] (BPE), used by OpenAI GPT models, and e.g. the recent Swedish GPT-SW3 model [9], which uses the Google tokenizer implementation SentencePiece [4]. BPE operates similar to a compression technique, with a dictionary of token merges constructed greedily maximizing the number of tokens that get merged in a training set (see Remark 1 for a sketch of the procedure).

There are other methods for performing this type of tokenization, e.g. the unigram language model also implemented in SentencePiece [4], where tokens are individually weighted. We do not consider this case here. BPE tokenization can be contrasted to lexical analysis [5], as lexical analysis (as exhibited in e.g. the POSIX tool `lex`) differs in that the rules are typically authored by hand, and break the string into an infinite language of tokens divided among a constant set of categories. Consider for example extracting arbitrarily long identifiers and string constants when parsing programming languages. BPE tokenization can also be viewed as one case of text segmentation in natural language processing

[1] We are for the purposes of this example ignoring some details, involving whitespace and the string beginning and end.

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on
Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 16–27, doi:10.4204/EPTCS.388.4

(see e.g. [3]), which covers e.g. breaking a text into topics, sentences, or words. These text segmentation algorithms have commonly been at least partially supervised or authored, where the tokenization algorithms considered here are designed for language-agnostic unsupervised learning.

The way the tokenization procedure is defined and implemented in common tools [2, 4] is essentially *global:* the highest priority rule that can be applied to the text is, no matter where in the text this application would happen. This is not *usually* a problem, as the text is *pretokenized* by splitting it on whitespace before applying the main tokenization procedure. That is sufficient to make tokenization algorithm behavior irrelevant in general, however:

- Some natural languages simply do not use interword whitespace, such as writing systems for Thai, Chinese and Japanese.
- In many artificial (e.g. programming) languages whitespace is common but not required, for example "minified" code is very common [8], where all unnecessary whitespace is removed to reduce file sizes. Language models are commonly trained at least partially on code, e.g. the GPT-SW3 dataset contains 9.5% code in various programming languages [1].
- In some cases there is whitespace, but pretokenizing using it does not produce the best tokens. E.g. in the SQL query language "LEFT OUTER JOIN" is a single concept, and would ideally be a single token.
- Even in cases where the pretokenization normally works well, such as for English, a hypothetical system relying heavily (more so than any currently existing system) on the pretokenization creating short text fragments may then be vulnerable to a denial of service attack. Compare to e.g. [10] on such attacks for regular expression matchers.

As such, as these systems find their way into broader and more complex use, it becomes interesting to investigate robust algorithms for operating on tokenizations. We investigate the following questions.

- Can we perform a BPE tokenization online: observing a stream of text can we output a stream of tokens using only limited memory and computation?
- If we have a tokenization of a large text, and the text is modified in a localized way, can we compute a localized update of the tokenization?

The answers to all of these questions are interconnected, but first we need firm definitions of the semantics we are considering.

The outline of this paper is as follows. After introducing our notation, we define (formally) SentencePiece and HuggingFace tokenizers. Then we consider how to do tokenization in a streaming fashion. This is followed by a short section outlining our envisioned future work.

2 Notation

An alphabet Σ is a finite set of symbols. Let Σ^* denote all strings over the alphabet Σ, including the empty string ε, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. A sequence of non-empty strings is a *tokenization*, e.g. for $u_1, \ldots, u_n \in \Sigma^+$, we denote this $u_1 \wr \cdots \wr u_n$. By $\Sigma^\wr$ we denote the set of all tokenizations constructed from strings from Σ^*. We refer to those strings as the *tokens*. Let $\pi : \Sigma^\wr \to \Sigma^*$ be the concatenation of the strings in a tokenization, e.g. $\pi(u_1 \wr \cdots \wr u_n) = u_1 \cdots u_n \in \Sigma^*$. As a special case, we let π applied to the tokenization with $n = 0$, be the empty string. When $\pi(u_1 \wr \cdots \wr u_n) = w \in \Sigma^*$, we say that $u_1 \wr \cdots \wr u_n$ *is a tokenization of* w. For $\tau = u_1 \wr \cdots \wr u_n$, we denote by $|\tau|$ the integer n. Thus, $|\tau| = 0$ if and only if τ is the empty tokenization. Also, for $u \in \Sigma^*$, we let $|u|$ denote the length of u, i.e. the number of symbols from Σ in the string u. It will be clear from

the context and notation when τ denotes a tokenization with $|\tau| = 1$, i.e. a tokenization of length one instead of a string of length one, given that in this case, τ could be interpreted as either. In fact, given our notational conventions discussed below, the symbol τ will always represent a tokenization.

In addition to using $|\tau|$ and $|u|$ for the length of a tokenization τ and length of a string u respectively, we use $|S|$ to denote the cardinality of a (finite) set S.

Differentiating between strings and tokenizations becomes important as we continue, so we adopt some conventions. Let Σ denote the alphabet whenever not otherwise specified. When giving examples, we always use $\Sigma = \{a, b, c, \ldots\}$. Furthermore, we always let α, β, γ be variables denoting symbols from the alphabet, u, v, w denote strings, and τ, ϕ denote tokenizations, including all sub-/superscripted variants of each. In other words, $u, v, w \in \Sigma^*$ and $\tau, \phi, \psi \in \Sigma^{\wr}$, and for that matter $u_1, u_2 \in \Sigma^*$, $\tau' \in \Sigma^{\wr}$, etc. As such we may write e.g. $u \wr \tau \wr v = \phi$ to mean that τ is a tokenization where the first token is u, the last is v, and the intervening tokens form the tokenization τ, so $|\phi| = |\tau| + 2$.

3 Tokenizing Semantics

First, we define a byte pair dictionary, which will be used to restrict the set of possible tokenizations for a given string w. We will only use D and its sub-/superscripted variants to denote a dictionary.

Definition 1. A byte pair *dictionary* $D = [u_1 \wr v_1, \ldots, u_n \wr v_n]$ of length $|D| = n$ is a sequence of tokenizations $u_1 \wr v_1, \ldots, u_n \wr v_n$, with each tokenization $u_i \wr v_i$ being of length 2. We call each $u_i \wr v_i$ a *rule* and say that (a rule) $u_i \wr v_i$ has *higher priority* than $u_j \wr v_j$, when $i < j$.

We write $\tau \Rightarrow^D \tau'$ if $\tau = \phi \wr u \wr v \wr \phi'$ and $\tau' = \phi \wr uv \wr \phi'$, for some $u \wr v \in D$. The dictionary D will always be clear from the context, thus we omit the superscript on $\Rightarrow$. We let $\Rightarrow^+$ and $\Rightarrow^*$ denote the transitive and reflexive transitive closure of $\Rightarrow$, respectively. Also, for $w = \alpha_1 \cdots \alpha_n$, we denote by $\mathbb{T}^{\varnothing}(w)$ the tokenization $\alpha_1 \wr \cdots \wr \alpha_n$.

Next, we transfer terminology used in derivations over context-free grammars, to our setting. For $w = \alpha_1 \ldots \alpha_n$, we begin a derivation for a tokenization with $\mathbb{T}^{\varnothing}(w)$, although a more complicated pretokenizer step could certainly also be of interest, but not considered in this paper. Whereas in the case of context-free grammars, a derivation step consists of applying a grammar rule by replacing the non-terminal on the left-hand side of a rule, by its right-hand side, in our setting, a derivation step is of the form $\phi \wr u_i \wr v_i \wr \phi' \Rightarrow \phi \wr u_i v_i \wr \phi'$, for $u_i \wr v_i$ in D. A derivation terminates when no further rules from D can be applied.

The definition of a base tokenizer on D, which ignores the priority of rules in D, is as follows.

Definition 2. For $D = [u_1 \wr v_1, \ldots, u_m \wr v_m]$ and $w = \alpha_1 \cdots \alpha_n$, we obtain the *base tokenizations of w by D,* denoted as $\mathbb{T}^D_{\text{base}}(w) \subset \Sigma^{\wr}$, as follows. We have $\tau_p \in \mathbb{T}^D_{\text{base}}(w)$ if:

- $\tau_0 = \mathbb{T}^{\varnothing}(w)$,
- $\tau_0 \Rightarrow \cdots \Rightarrow \tau_p$,
- there exists no τ_{p+1} such that $\tau_p \Rightarrow \tau_{p+1}$.

That is, $\mathbb{T}^D_{\text{base}}(w)$ are the tokenizations of w which can be achieved by applying rules to an initial tokenization $\mathbb{T}^{\varnothing}(w)$ where all symbols in w are their own token, until a point where no further rules can be applied. Thus, to obtain one of the possible base tokenizations, we select non-deterministically a rule from D that can be applied to the current tokenization, until the set of rules that could be applied, is empty. Given that $|\mathbb{T}^D_{\text{base}}| \geq 1$, since there is non-deterministic choice in selecting the next applicable rule, a SentencePiece tokenizer [4] is defined to remove ambiguity from the base tokenizer. We will (in a somewhat biased way) refer to this tokenization as the correct tokenization.

Definition 3. The *SentencePiece* tokenization of w, denoted $\mathbb{T}^D(w)$, also referred to as the *correct tokenization* of w, is τ_n, with $\tau_n \in \mathbb{T}^D_{\text{base}}(w)$, where:

- $\tau_0 = \mathbb{T}^\varnothing(w)$;
- $\tau_0 \Rightarrow \cdots \Rightarrow \tau_n$, and for $0 \leq i < n$, we pick the decomposition $\tau_i = \phi \wr u \wr v \wr \phi'$, to obtain $\tau_{i+1} = \phi \wr uv \wr \phi'$, in such a way that:
  - $u \wr v$ is the highest priority rule in D for which such a decomposition exists;
  - among the remaining decompositions, we pick the unique one which minimizes $|\phi|$;
- no further rules apply to τ_n.

Observe that $\mathbb{T}^D(w)$ always exists, and is obtained, intuitively, as follows. Whenever it is possible to apply a rule from the dictionary D to merge some tokens in the interim tokenization, merge the *highest-priority* rule that occurs, and merge the left-most such pair if multiple occurrences exist. Note that this selects a unique tokenization, for each string w, from the set $\mathbb{T}^D_{base}(w)$.

Example 1. Take the dictionary $D = [a \wr b, a \wr bc, b \wr c, ab \wr c]$, then the correct tokenization of the string *abcbcab* is $abc \wr bc \wr ab$, using the following steps:

- Initially, $\tau_0 = a \wr b \wr c \wr b \wr c \wr a \wr b$, $a \wr b \in D$ applies to the leftmost $a \wr b$, producing $\tau_1 = ab \wr c \wr b \wr c \wr a \wr b$.
- The rule $a \wr b$ still applies, now to the last two tokens, producing $\tau_2 = ab \wr c \wr b \wr c \wr ab$. The rule $a \wr b$ now no longer applies anywhere.
- The next rule $a \wr bc$ does not apply, as there is no token bc, but $b \wr c$ does apply, producing $\tau_3 = ab \wr c \wr bc \wr ab$.
- The first rule that can now be applied is the rule $ab \wr c$, producing $\tau_4 = abc \wr bc \wr ab$. Now, no further rules apply, so $abc \wr bc \wr ab$ is the correct (SentencePiece) tokenization.

It is interesting to observe that the rule $a \wr bc$ has the second-highest priority in the dictionary, but was never applied. Also, when using the base tokenizer, $a \wr bc$ can certainly be applied when tokenizing the string *abcbcab*. Observe that applying $a \wr bc$ would produce the token *abc*, but tokenizing the string *abc* takes the steps $a \wr b \wr c \Rightarrow ab \wr c \Rightarrow abc$. Later on in Corollary 1 we show that a rule $u_i \wr v_i$ is *useful*, i.e. gets applied in the tokenization of some string, if and only if it gets applied when tokenizing the string $u_i v_i$. ⋄

Example 2. Take the dictionary $D = [c \wr ab, ab \wr c, a \wr b]$. Then the correct tokenization of *abcabcabcabc* is $abc \wr abc \wr abc \wr abc$. Notice that this tokenization is achieved left to right. After five steps, we have the tokenization $abc \wr abc \wr ab \wr c \wr a \wr b \wr c$. Contrast this to tokenizing the string *bcabcabcabc* (i.e. we delete the initial *a*) which tokenizes as $b \wr cab \wr cab \wr cab \wr c$, or *cabcabcabcabc* (i.e. adding an initial *c*), which leads to $cab \wr cab \wr cab \wr cab \wr c$, where we need to move left to apply $c \wr ab$, after having applied $a \wr b$. This illustrates that a small modification to the string can cause an arbitrarily large change to the resulting tokenization (to the right). ⋄

Interestingly enough, not all tokenization libraries modify the base tokenizer in the same way in order to eliminate ambiguity of tokenization. Let us consider the Python implementation of the GPT-2 tokenizer offered by HuggingFace [2], which removes ambiguity from the base tokenizer, as follows.

Definition 4. The *HuggingFace tokenization* τ_n of w, which we denote by $\mathbb{T}^D_{\text{hf}}(w) \in \mathbb{T}^D_{\text{base}}(w)$, is defined as follows, where $\tau_0 = \mathbb{T}^\varnothing(w)$. In $\tau_0 \Rightarrow^+ \cdots \Rightarrow^+ \tau_n$, for $0 \leq i < n$, we select a decomposition $\tau_i = \phi \wr u \wr v \wr \phi'$, such that $u \wr v \in D$ is the highest priority rule applicable to τ_i, and then apply $u \wr v$ from left to right, until it is no longer applicable, in order to obtain the unique τ_{i+1}.

That is, in both Definitions 3 and 4 we rewrite τ_i by picking the highest-priority applicable rule and applying it at the left-most possible position. However, the SentencePiece semantics picks the highest-priority applicable rule in *every* step, where HuggingFace picks a rule and uses it until it becomes inapplicable. This does create a formal difference in semantics, but as we will see they differ only in cases which may be considered degenerate given the way dictionaries are usually constructed.

Example 3. Take the dictionary $D = [ab \rightsquigarrow a, a \rightsquigarrow b]$ and consider the tokenization of $w = abababab$, which has $\mathbb{T}^D(w) = aba \rightsquigarrow b \rightsquigarrow aba \rightsquigarrow b$, but $\mathbb{T}^D_{\text{hf}}(w) = ab \rightsquigarrow ab \rightsquigarrow ab \rightsquigarrow ab$. ◇

Remark 1. The reason why the D in the previous example is regarded to be degenerate or improper, is that a byte pair dictionary is typically produced [7] by taking a training corpus, initially tokenizing it symbol by symbol, and then iteratively adding the most common token pair to the dictionary, tokenizing, and repeating. For example, in the training corpus $a \rightsquigarrow b \rightsquigarrow c \rightsquigarrow a \rightsquigarrow b \rightsquigarrow c \rightsquigarrow a \rightsquigarrow c \rightsquigarrow a$, the most common token pair is $c \rightsquigarrow a$, which is inserted as the first rule in the dictionary. Then, we continue using $a \rightsquigarrow b \rightsquigarrow ca \rightsquigarrow b \rightsquigarrow ca \rightsquigarrow ca$. Now, the most common pair is $b \rightsquigarrow ca$, and this rule is added to the dictionary, which now consists of the rules $[c \rightsquigarrow a, b \rightsquigarrow ca]$. The new dictionary now produces $a \rightsquigarrow bca \rightsquigarrow bca \rightsquigarrow ca$, as tokenization, and so on. Observe that when constructing a dictionary in this way, a rule $u_i \rightsquigarrow v_i$ cannot have higher priority than the rules needed to produce u_i and v_i, a property formalized in the next definition. Also, when tokenizing the training corpus with the dictionary obtained through training, using HuggingFace semantics, the tokenization obtained will be the tokenization of the training corpus at the end of the training process, and each rule in the dictionary will be used in this tokenization. ◇

Definition 5. A dictionary $D = [u_1 \rightsquigarrow v_1, \ldots u_i \rightsquigarrow v_i, \ldots, u_j \rightsquigarrow v_j, \ldots, u_n \rightsquigarrow v_n]$ is *proper* if for each j with $|u_j| > 1$, there exists $i < j$ such that $u_j = u_i v_i$, and similarly, for each j' with $|v_{j'}| > 1$, there exists $i' < j'$ such that $v_{j'} = u_{i'} v_{i'}$.

Note that a proper dictionary may still contain rules which are not useful. Consider for example the dictionary $D = [b \rightsquigarrow c, a \rightsquigarrow b, c \rightsquigarrow d, ab \rightsquigarrow cd]$. Then D is proper, but $ab \rightsquigarrow cd$ is not useful. This can be seen by noting that when $\mathbb{T}^{\varnothing}(w)$ equals $a \rightsquigarrow b \rightsquigarrow c \rightsquigarrow d$, then the first rule gets applied to produce $a \rightsquigarrow bc \rightsquigarrow d$, and thus it is not possible to apply the rules $a \rightsquigarrow b$ and $c \rightsquigarrow d$, in order so that $ab \rightsquigarrow cd$ could be applied. But note that if D is constructed from a training corpus, then D is proper and each rule in D is useful.

With the additional assumption that the dictionary is proper, the SentencePiece and HuggingFace tokenizers turn out to have equivalent semantics. This should to some extent be expected, as they are intended to achieve the same results, the HuggingFace approach effectively being a small simplification.

Lemma 1. *If D is proper, we have $\mathbb{T}^D(w) = \mathbb{T}^D_{hf}(w)$ for all w.*

Proof. By contradiction, assume that some w has $\mathbb{T}^D(w) \neq \mathbb{T}^D_{\text{hf}}(w)$. Let $\tau_0 \Rightarrow \cdots \Rightarrow \tau_n$ be the tokenization steps taken by $\mathbb{T}^D(w)$, and $\phi_0 \Rightarrow \cdots \Rightarrow \phi_m$ the tokenization steps taken by $\mathbb{T}^D_{\text{hf}}$. Let i be the smallest index such that $\tau_i \neq \phi_i$. Note, such an i must exist, otherwise, one sequence would be a subsequence of the other, which is impossible by Definition 2. Thus, one sequence cannot be a proper prefix of the other. We also have $i \geq 2$, as the semantics differ only in that Definition 4 prefers repeating the previous rule over the highest priority one, but this difference can only be exhibited when there is a previous step to repeat.

We then have $\tau_{i-2} = \phi_{i-2}$, $\tau_{i-1} = \phi_{i-1}$ and $\tau_i \neq \phi_i$. Let r_0 be the rule applied in $\tau_{i-2} \Rightarrow \tau_{i-1}$ (and $\phi_{i-2} \Rightarrow \phi_{i-1}$ as they are equal), r_1 the rule in $\tau_{i-1} \Rightarrow \tau_i$, and r_2 the rule in $\tau_{i-1} \Rightarrow \phi_i$. That is, with some abuse of notation, the following situation:

$$\cdots \Rightarrow (\tau_{i-2} = \phi_{i-2}) \overset{r_0}{\Rightarrow} (\tau_{i-1} = \phi_{i-1}) \begin{array}{l} \overset{r_1}{\Rightarrow} \tau_i \Rightarrow \cdots \\ \overset{r_2}{\Rightarrow} \phi_i \Rightarrow \ldots \end{array}$$

Then we know that r_1 has higher priority than r_2, since they must differ, and Definition 3 (SentencePiece) always picks the highest priority rule applicable. The only possible reason for them to differ is that $r_0 = r_2$, i.e. the Definition 4 (HuggingFace) semantics prioritized using the same rule as in the previous step. However, as they agree in the previous step this means that we picked r_0 in that step by virtue of Definition 3, even though r_1 has higher priority than r_0, which must mean that r_1 was *not* applicable before. This leads to a contradiction, as applying r_0 must then have created a token which made r_1 applicable, which, since r_0 is of lower priority, contradicts D being a proper dictionary. As such, our assumption was wrong and $\mathbb{T}^D_{\text{hf}}(w) = \mathbb{T}^D(w)$ by necessity. □

With this result in hand, it becomes less relevant to differentiate between the two semantics whenever considering only proper dictionaries.

Remark 2. It can be decided whether D is proper in time $\mathcal{O}(\|D\|^2)$ (assuming $|uv|$ is constant for all rules $u \wr v$ in D), where $\|D\| = \sum\{|uv| \mid u \wr v \in D\}$. For each $u \wr v \in D$, determine all $u' \wr v'$ such that uv is a substring of u' or v', and for all such rules $u' \wr v'$, verify that $u \wr v$ has lower priority than $u' \wr v'$. ◇

Next, we investigate the relationship between the tokenization of substrings of w, and the tokenization of w. First, we consider the following example. Let u, v be strings with $\mathbb{T}^D(u) = \tau_1 \wr \tau_2$ and $\mathbb{T}^D(v) = \phi_1 \wr \phi_2$. Then it is not necessarily the case that $\tau_1 \wr \phi_2 = \mathbb{T}^D(\pi(\tau_1 \wr \phi_2))$ or that $\phi_1 \wr \tau_2 = \mathbb{T}^D(\pi(\phi_1 \wr \tau_2))$. An easy counterexample is obtained by letting $D = [a \wr a, b \wr b]$, $\tau_1 = a$, $\tau_2 = b$, $\phi_1 = b$, and $\phi_2 = a$. Observe that indeed $\mathbb{T}^D(ab) = a \wr b = \tau_1 \wr \tau_2$, and $\mathbb{T}^D(ba) = b \wr a = \phi_1 \wr \phi_2$, *however* $\tau_1 \wr \phi_2 = a \wr a \neq \mathbb{T}^D(aa)$ and $\phi_1 \wr \tau_2 = b \wr b \neq \mathbb{T}^D(bb)$. This shows that tokenizations can not be decomposed and then again glued together in arbitrary ways. However, deriving the tokenization of substrings of a given string w, given the final full tokenization of w, is sometimes possible, as shown in the following lemma.

Lemma 2. *Tokenization derivations and tokenizations have the following properties:*

(i) For both SentencePiece and HuggingFace, if $\phi_1 \wr \ldots \wr \phi_k \Rightarrow^* \phi'_1 \wr \ldots \wr \phi'_k$, then if $\pi(\phi_i) = \pi(\phi'_i)$ for all i, we have that $\phi_i \Rightarrow^* \phi'_i$ for all i.

(ii) For a dictionary D and string w such that $\mathbb{T}^D(w) = \tau_1 \wr \ldots \wr \tau_k$ (or $\mathbb{T}^D_{hf}(w) = \tau_1 \wr \ldots \wr \tau_k$), it holds that $\mathbb{T}^D(\pi(\tau_i)) = \tau_i$ (respectively $\mathbb{T}^D_{hf}(\pi(\tau_i)) = \tau_i$).

Proof. For (i), let $\phi_{1,1} \wr \ldots \wr \phi_{k,1} \Rightarrow \cdots \Rightarrow \phi_{1,n} \wr \ldots \wr \phi_{k,n}$ be the steps taken by the procedure in Definition 2, such that $\phi_{i,1} = \phi_i$ and $\phi_{i,n} = \phi'_i$ for all i, and $\pi(\phi_{i,j}) = \pi(\phi_{i,j'})$ for all i, j and j'. Removing all duplicates from $\phi_{i,1}, \ldots, \phi_{i,n}$ produces the sequence of steps taken by SentencePiece or HuggingFace derivations, i.e. in each step we apply the highest-priority rule from D as left-most as possible, in the case of SentencePiece semantics, and we apply the highest-priority rule as many times as possible, in the case of HuggingFace semantics.

For (ii), take $\phi_i = \mathbb{T}^{\varnothing}(\tau_i)$ and $\phi'_i = \tau_i$ in (i). □

Remark 3. A trivial outcome of this lemma is then that one can freely truncate tokenizations. I.e. if we have tokenized a long text w and are only interested in a prefix, we can pick a suitable prefix of the tokenization (not the string) and it will be correct for the string it represents. A similar remark holds for a suffix of a tokenization.

Corollary 1. *For SentencePiece or HuggingFace semantics, a rule in a dictionary D is useful if and only if it gets applied when tokenizing the string it produces.*

Proof. The "if" part follows directly from the definition of useful, so for the converse, assume we have a derivation $\alpha_1 \wr \cdots \wr \alpha_n \Rightarrow \cdots \Rightarrow \phi \wr u \wr v \wr \phi' \Rightarrow \phi \wr uv \wr \phi'$. But then the previous lemma implies that $\mathbb{T}^{\varnothing}(uv) \Rightarrow^* u \wr v \Rightarrow uv$. □

The main purpose of Lemma 2 is that it makes it possible to do tokenizations in streaming and incremental ways. Algorithm 1 below shows how this is achieved, but in order to establish the correctness of this algorithm, we first need the following corollary, which shows how we can split and then glue tokenizations.

Corollary 2. *If $\mathbb{T}^D(v) = \tau_1 \wr u \wr \tau_2$ and $\mathbb{T}^D(w) = \phi_1 \wr u \wr \phi_2$ then we also have $\mathbb{T}^D(\pi(\tau_1 \wr u \wr \phi_2)) = \tau_1 \wr u \wr \phi_2$ and $\mathbb{T}^D(\pi(\phi_1 \wr u \wr \tau_2)) = \phi_1 \wr u \wr \tau_2$. The same result holds for the HuggingFace semantics.*

Proof. Remark 3 give us that $\mathbb{T}^D(\pi(\tau_1 \wr u)) = \tau_1 \wr u, \mathbb{T}^D(\pi(u \wr \tau_2)) = u \wr \tau_2$ and similarly $\mathbb{T}^D(\pi(\phi_1 \wr u)) = \phi_1 \wr u, \mathbb{T}^D(\pi(u \wr \phi_2)) = u \wr \phi_2$ where u, τ_1, ϕ_1, τ_1, and τ_2 are as in the corollary above. The result now follows from the observation that we can glue two tokenizations together, if the end token of the first tokenization is the same as the start token of the next tokenization. The same argument holds for HuggingFace semantics. This follows the same line of argument as Lemma 2, except instead of pruning steps from one tokenization, we interleave the steps of two, deduplicating rules applied to the overlapping token. The overlapping token ensures that one tokenization cannot "disturb" the other. □

With this in hand we can define an incremental update algorithm, which is not necessarily efficient in general (due to cases like Example 2), but will often do much less work than full retokenization, if we are in a situation where steps 4 and 5 are performed only a few times, i.e. if i is close to n and j close to 1 when the condition 'If ($v_1 = u_i$ or $i = 1$) and ($v_k = u'_j$ or $j = m$)' in step 3 holds. When showing the correctness of Algorithm 1 below, we will use the special case of the previous corollary where τ_1 and ϕ_2 are empty tokenizations, or τ_2 and ϕ_1 are empty.

Algorithm 1. Given $\mathbb{T}^D(w) = \tau$ and $\mathbb{T}^D(w') = \tau'$ we compute the tokenization $\mathbb{T}^D(ww')$ in the following way, assuming we are not in the trivial case where $w = \varepsilon$ or $w' = \varepsilon$.

1. Let $\tau = u_1 \wr \cdots \wr u_n$ and $\tau' = u'_1 \wr \cdots \wr u'_m$, initialize $i = n$ and $j = 1$.
2. Compute $\mathbb{T}^D(u_i \cdots u_n u'_1 \cdots u'_j) = v_1 \wr \cdots \wr v_k$.
3. If ($v_1 = u_i$ or $i = 1$) and ($v_k = u'_j$ or $j = m$) output $u_1 \wr \cdots \wr u_i \wr v_2 \wr \cdots \wr v_{k-1} \wr u'_j \wr \cdots \wr u'_m$ as $\mathbb{T}^D(ww')$ and halt.
4. If $u_i \neq v_1$ and $i > 1$, then $i \leftarrow i - 1$.
5. If $u'_j \neq v_k$ and $j < m$, then $j \leftarrow j + 1$.
6. Go to step 2.

Theorem 1. *Algorithm 1 is correct.*

Proof. This amounts to two applications of Corollary 2. The algorithm halts in a state where both $u_1 \wr \cdots \wr u_i$ and $v_1 \wr \cdots \wr v_k$ are correct tokenizations, the former holds by Remark 3 and the latter by construction. We also have $u_i = v_1$ (either that or $i = 1$, but that case is trivial), which allows the application of Corollary 2 to establish that $\tau_1 \wr u_i \wr v_2 \wr \cdots \wr v_k$ is a correct tokenization. To make this specific, in the terms of Corollary 2 we have $\tau_2 = \phi_1 = \varepsilon$, $u = u_i = v_1$, $\tau_1 = u_1 \wr \ldots \wr u_{i-1}$, and $\phi_2 = v_2 \wr \cdots \wr v_k$, which gives us that $\tau_1 \wr u \wr \phi_2 = u_1 \wr \cdots u_i \wr v_2 \wr \cdots \wr v_k$ is a correct tokenization. Now repeat this argument for the suffix to complete the proof. □

Remark 3 and Algorithm 1 give tools to perform arbitrary incremental updates. For example, assume we have the tokenization τ of a (long) string w, and we make a small change in w, let's say $w = v_1\alpha v_2$ and the updated string is $w' = v_1\beta v_2$. Then let τ_1 be the prefix of τ which falls entirely within v_1, let τ_2 be the suffix of τ which falls entirely inside v_2, let u be the string such that $\pi(\tau_1)u\pi(\tau_2) = w'$, then compute $\phi = \mathbb{T}^D(u)$, and obtain the tokenization of w' by applying Algorithm 1 to concatenate τ_1 to ϕ, and then that tokenization to τ_2.

Unfortunately, Algorithm 1 is not *necessarily* more efficient than retokenizing the whole string, and might be worse. Of course, an experimental bound can be used in steps 3 and 4, where once we have decreased i and increased j, more times than the specified bound, we fall back to retokenizing the complete string. In cases where there is a subset of symbols Σ' from Σ, where symbols from Σ' can only appear as the first or last symbols in uv for rules $u \wr v$, and in all input strings there is a relatively small distance between symbols in Σ' (i.e. input is not selected from all of Σ^*), we certainly have that Algorithm 1 is much more efficient than retokenizing the whole string. In general, it seems likely that steps 4 and 5 in Algorithm 1 will be performed relatively few times in most practical dictionaries, but investigating this, is left for future work. We consider the worst-case in more generality in the next section. We establish a bound on how many times i can be decremented determined solely by the dictionary (so a constant in the length of the string). Thus, we consider the worst-case when modifications happen late in a string, for example when applying appends.

4 Tokenizing Online with Finite Lookahead

In this section, we assume all dictionaries are proper (although at times we do state this explicitly, to emphasize that we are making this assumption), so by Lemma 1 $\mathbb{T}^D$ and $\mathbb{T}^D_{\text{hf}}$ are interchangeable. We investigate the following question: When we tokenize a string, in a streaming fashion, how long is the suffix that we need to tokenize again, when we resume tokenization, given we make no assumptions about the input string being tokenized. We refer to this constant as the *lookahead constant* for a dictionary D, and denote it by $l(D)$. More formally, we have the following definition.

Definition 6. Let D be proper and ϕ, τ and ψ be tokenizations. Then $l(D)$, the *lookahead constant* for D, is the smallest constant such that if $|\pi(\tau)| \geq l(D)$ and $\mathbb{T}^D(\pi(\phi)\pi(\tau)) = \phi \wr \tau$, then $\mathbb{T}^D(\pi(\phi \wr \tau)\pi(\psi)) = \phi \wr \psi'$, for some tokenization ψ'.

We show in Theorem 2, that $l(D) \leq |D| \cdot \max\{|uv| \mid u \wr v \in D\}$, where $|D|$ is the number of rules in D. After this, in Remark 5, we explore how to improve on this bound.

The next lemma will be used in Theorem 2 to perform HuggingFace tokenization in a straightforward inductive way, where for a proper dictionary $D = [u_1 \wr v_1, \ldots, u_n \wr v_n]$ we can tokenize a string by first applying the rule $u_1 \wr v_1$ as many times as possible, then the rule $u_2 \wr v_2$ as many times as possible, and so on.

Lemma 3. *Let D be proper dictionary and $\mathbb{T}^D_{hf}(w) = \tau$. Assume $r_1, \ldots, r_n$ is the sequence of rules applied to produce τ according to Definition 4 (i.e. using HuggingFace semantics). Then it must be the case that $r_1, \ldots, r_n$ are in order of decreasing priority.*

Proof. We proceed in a way similar to Lemma 1. By contradiction, assume that there exists some r_i such that r_{i+1} is of higher priority than r_i. This means that the tokenization after the first i steps contains the pair $r_{i+1} = u \wr v$, but this pair cannot have been created by the applications of r_i, as it is of lower priority than r_{i+1} and D is proper, and it also cannot have existed in the tokenization when r_i was picked as the

rule to next apply, as that contradicts how rules are picked in Definition 4. As such, our assumption was wrong, and r_i is of higher priority or equal to r_{i+1}. □

We will use the term *refinement* for the way a tokenization is developed in this way, i.e. a *tokenization τ is a refinement of τ'*, if τ' can be obtained from τ by applying π to some of the subtokenizations in τ. For example, $\phi \wr \phi' \wr \phi''$ is a refinement of $\phi \wr \pi(\phi') \wr \phi''$ and also of $\pi(\phi) \wr \pi(\phi') \wr \phi''$. We can also consider the opposite notion, i.e. a tokenization τ' is *coarser* than τ , if τ is a refinement of τ'. Thus, a (final) tokenization of a given string is obtained by using rules from D to obtain coarser and coarser tokenizations. Recall, $|D|$ denotes the number of rules in D.

Theorem 2. *Let D be proper, $|\tau| \geq |D|$ and $\mathbb{T}^D(\pi(\phi)\pi(\tau)) = \phi \wr \tau$. Then $\mathbb{T}^D(\pi(\phi \wr \tau)\pi(\psi)) = \phi \wr \psi'$, for some tokenization ψ'. That is, by Definition 6 we have $l(D) \leq |D| \cdot \max\{|uv| \mid u \wr v \in D\}$.*

Proof. This is easier to see using $\mathbb{T}^D_{\text{hf}}$, which is equivalent to $\mathbb{T}^D$ by Lemma 1. Let D be the dictionary $[u_1 \wr v_1, \ldots, u_n \wr v_n]$ and D_i be the i-length prefix of D, i.e. $D_i = [u_1 \wr v_1, \ldots, u_i \wr v_i]$ for each i. We determine $\mathbb{T}^D_{\text{hf}}(w)$ as follows: First calculate $\mathbb{T}^{D_1}_{\text{hf}}(w)$, then assuming we know $\mathbb{T}^{D_i}_{\text{hf}}$, we apply $u_{i+1} \wr v_{i+1}$ to $\mathbb{T}^{D_i}_{\text{hf}}(w)$ wherever possible (working left to right) to obtain $\mathbb{T}^{D_{i+1}}_{\text{hf}}$. This procedure is correct by Lemma 3. We refer to $\mathbb{T}^{D_i}(w)$ as a tokenization at level i, and note that $\mathbb{T}^{D_i}(w)$ is a refinement of $\mathbb{T}^D(w)$.

Let $\phi \wr \tau = w_1 \wr \ldots \wr w_k$ and $w = \pi(\phi \wr \tau)\pi(\psi)$. We show that the tokenization of w at level i is a refinement of a tokenization of the form $w_1 \wr \ldots \wr w_{k-i} \wr \psi_i$, thus that $\mathbb{T}^{D_i}_{\text{hf}}(w)$ is a refinement of a tokenization of the form $w_1 \wr \ldots \wr w_{k-i} \wr \psi_i$, for some tokenization ψ_i. More precisely, when we use the dictionary D_i, then tokenizing $\pi(\phi \wr \tau)\pi(\psi)$, instead of only $\pi(\phi \wr \tau)$, changes at most the rightmost i tokens in $\phi \wr \tau$. This implies that when $i = n$, we obtain that $\mathbb{T}^{D_n}_{\text{hf}}(w)$ is a tokenization of the form $w_1 \wr \ldots \wr w_{k-n} \wr \psi_n$, i.e. not only a refinement of a tokenization of the given form.

First, we show that $\mathbb{T}^{D_1}_{\text{hf}}(w)$ is a refinement of a tokenization of the form $w_1 \wr \ldots \wr w_{k-1} \wr \psi_1$. Note that $u_1 \wr v_1$ could potentially be applied to the substring $w_k\pi(\psi)$, when tokenizing $\pi(\phi \wr \tau)\pi(\psi)$, but $u_1 \wr v_1$ is not applied across any of the $(k-1)$ boundaries between tokens in $w_1 \wr \ldots \wr w_k$. This must be the case, otherwise $u_1 \wr v_1$ would have been applied over some of these $(k-1)$ boundaries between the tokens w_i, for $1 \leq i \leq k$, when computing the tokenization $w_1 \wr \ldots \wr w_k$ with the full dictionary D.

Moving on to the next rule in terms of priority, $u_2 \wr v_2$, we repeat the argument we used for $u_1 \wr v_1$. More precisely, $u_2 \wr v_2$ could potentially be applied, one or more times, to the substring $w_{k-1}\pi(\psi_1)$, when tokenizing $\pi(w_1 \wr \ldots \wr w_{k-1} \wr \psi_1)$, but $u_2 \wr v_2$ is not applied across any of the $(k-2)$ tokenization boundaries in $w_1 \wr \ldots \wr w_{k-1}$, otherwise it would have when $\pi(\phi \wr \tau)$ was tokenized using D. Thus, $\mathbb{T}^{D_2}_{\text{hf}}(w)$ is a refinement of a tokenization of the form $w_1 \wr \ldots \wr w_{k-2} \wr \psi_2$.

We iterate this procedure for $i = 3, \ldots, n$ to obtain the theorem. □

Example 4. In this example, we consider the bound $|\tau| \geq |D|$, in the previous corollary. Fix a positive integer n and let D be a dictionary with the following n rules:

$$a_n \wr a_{n+1},\ a_{n-1} \wr a_n a_{n+1},\ a_{n-2} \wr a_{n-1} a_n a_{n+1},\ \ldots,\ a_1 \wr a_2 \ldots a_{n+1}$$

Also, let $\phi = a_0$, $\tau = a_1 \wr a_2 \wr \ldots \wr a_n$, and let $\psi = a_{n+1}$. Then, $\mathbb{T}^D(\pi(\phi \wr \tau)\pi(\psi)) = a_0 \wr a_1 a_2 \ldots a_{n+1}$, and we can not move any prefix of the tokenization of τ to ϕ, otherwise we no longer have $\mathbb{T}^D(\pi(\phi \wr \tau)\pi(\psi)) = \phi \wr \psi'$, for some tokenization ψ'. ◇

Theorem 2 may at first appear quite abstract, but they demonstrate a fact that is very useful in practice: a finite lookahead is sufficient to tokenize a string from left to right.

Definition 7. The *sufficient lookahead* of a proper dictionary D is $|D| \cdot \max\{|uv| \mid u \wr v \in D\}$. Observe that by Theorem 2 the sufficient lookahead is greater than or equal to $l(D)$.

That is, for a proper dictionary D and a string w, if we know that ϕ is a prefix of $\mathbb{T}^D(w)$ (beginning the process by taking $|\phi| = 0$) we can compute a prefix $\phi \wr u$ by inspecting only the sufficient lookahead many next symbols. Observe that this lookahead length does not depend on $|w|$. This has potential to improve tokenization performance by cache locality (where e.g. SentencePiece [4] and HuggingFace [2] access the string contents with random access), but also enables doing streaming tokenization using a constant amount of memory, for when the entire string is not available or impractical to hold in memory. One way to express this finite state tokenization approach is as a deterministic string-to-string transducer. First, to avoid special cases for the end of the string, let us define a simple normal form.

Definition 8. For a dictionary D over the alphabet Σ, let k be the sufficient lookahead for D, assume $z \notin \Sigma$, then a string $v \in (\Sigma \cup \{z\})^*$ is the *end-padding of* w if it is of the form $wzz\cdots z$ where there are a total of k trailing zs.

Remark 4. Observe that if v is w end-padded, then we have $\mathbb{T}^D(v) = \mathbb{T}^D(w) \wr \phi$ where $\phi = z \wr \cdots \wr z$, since D contains no rules involving z. This means that tokenizing v from the left to right we can end the procedure the moment the lookahead consists only of zs, as that is the padding which will always tokenize to $\phi = z \wr \cdots \wr z$. ◇

This allows us to state a straightforward left-to-right tokenization algorithm without having special cases for when the

Algorithm 2. Let D be a proper dictionary over the alphabet Σ and k its sufficient lookahead. Assume that $z \notin \Sigma$.

Precompute $f : (\Sigma \cup \{z\})^k \to \Sigma^\wr$ such that for all $w \in \Sigma \cup \{z\}$ we have $f(w) = u$ where $\mathbb{T}^D(w) = u \wr \tau$ for some τ. Observe that then $\mathbb{T}^D(ww') = u \wr \tau'$ for all w' as well, by Theorem 2.

Then for any string w let v be its end-padding, we can then compute $\mathbb{T}^D(v)$ using the following steps.

1. Split $v = v'v''$ such that $|v'| = k$.
2. If $v' = z\cdots z$, halt.
3. Lookup $f(v') = u$, output u.
4. Split $v = uu'$ (observe that u must be a prefix of v' and in turn v).
5. Update v to be u', then go to 1.

Theorem 3. *For any fixed proper D and any string w, Algorithm 2 outputs $\mathbb{T}^D(w)$ in time $\mathcal{O}(|w|)$ and using $\mathcal{O}(1)$ taking D to be fixed and the input string to be read only.*

Proof. Correctness follows from Theorem 2, with the algorithm picking tokens based on a long enough prefix of the string (guaranteed to exceed $l(D)$) that it is guaranteed that any suffixes will still have the correct tokenization produce that same token.

The time and space bounds are trivial noting that each iteration step uses up part of the string, and suitably implemented each step uses an amount of time and space bounded in $l(D)$, which is $\mathcal{O}(1)$. Reusing space, the indicated bounds are reached. □

A perhaps more natural presentation of this algorithm would be constructing a string-to-string transducer. This is not very complicated, the transducer would read $l(D)$ symbols and then, in much the same way as the precomputed f in Algorithm 2, output a token accordingly. However, the bound here provided is quite large, and a more sophisticated construction is likely needed.

The bound established by Theorem 2 is clearly quite loose for most realistic dictionaries. Basically, it assumes that the rules in the dictionary create a chain, where each successive rule can "interfere" with the application of the next lower priority rule.

Remark 5. We have from Theorem 2 that $l(D) \leq |D| \cdot \max\{|uv| \mid u \curlyvee v \in D\}$, where $l(D)$ is the lookahead constant of D, since $|\pi(\tau)| \geq |D| \cdot \max\{|uv| \mid u \curlyvee v \in D\}$, implies $|\tau| \geq |D|$. Next, we consider (informally) how to improve the bound $|\tau| \geq |D|$, which will then improve the bound on the lookahead constant. Consider dictionaries $D = [a \curlyvee b, c \curlyvee d, ab \curlyvee cd]$ and $D' = [c \curlyvee d, a \curlyvee b, ab \curlyvee cd]$, i.e. D and D' have the same rules, but we switched the order of the first two rules in these dictionaries. Now, note that the tokenizations of any string w will be the same, independent of if we use D or D'. We say that dictionaries D and D' are *equivalent* if $\mathbb{T}^D(w) = \mathbb{T}^{D'}(w)$ for all strings $w \in \Sigma^*$. The complexity of deciding if two dictionaries are equivalent, and if equivalence is even decidable, is left for future research.

Next, we define the *chain length* of a dictionary D, denoted as $c(D)$. For a dictionary D, we let $c(D)$ be the maximum value of n such that we have a sequence of rules of decreasing priority, $r_1, \ldots, r_n$, in all dictionaries equivalent to D. Thus, we certainly have that $c(D) \leq |D|$. Again, the complexity of computing $c(D)$ will be considered in a future publication, but once we have some, but necessary all dictionaries equivalent to D, we can obtain an upper bound for $c(D)$, most likely better than $|D|$. In particular, if $r = u \curlyvee v$ and $r' = u' \curlyvee v'$ are neighbouring rules in D, such that a non-empty suffix of uv is not a prefix of $u'v'$, a non-empty prefix of uv is not a suffix of $u'v'$, and uv is not a substring of $u'v'$, and we have similar conditions when swapping r and r', then certainly we can switch r and r' in D and this will not change the tokenization of any string. For example, if $D = [a \curlyvee b, c \curlyvee d, ab \curlyvee cd]$, then $c(D) \leq 2$, since $[a \curlyvee b, c \curlyvee d, ab \curlyvee cd]$ and $[c \curlyvee d, a \curlyvee b, ab \curlyvee cd]$ are equivalent dictionaries.

Finally, note that with these concepts in hand, the proof of Theorem 2 actually shows a stronger result: the theorem holds if we replace the bound $|\tau| \leq |D|$ by $|\tau| \leq c(D)$. This can be seen by noting that the proof of Theorem 2 implies that there is a sequence of rules $r_1, \ldots, r_n$, such that if we apply these rules in order, as in HuggingFace semantics, then we obtain a refinement of $w_1 \curlyvee \ldots \curlyvee w_{k-i} \curlyvee \psi_i$ after having applied r_i. But, independently of which dictionary equivalent to D is used, r_1 is only potentially applied over the tokenization boundary between w_k and ψ, and not over any of the $(n-1)$ tokenization boundaries between any of the w_i. Similarly, none of the r_i, with $i \geq 2$, is applied over any of the $(k-i-1)$ tokenization boundaries in $w_1 \curlyvee \ldots \curlyvee w_{k-i}$. This is the case, since with equivalent dictionaries, by definition, we obtain the same tokenization, and as shown in the proof of Theorem 2, in order to cross the next tokenization boundary from the right, we need a lower priority rule. In summary, each r_i from $r_1 \ldots r_n$, will, in order, cross at most one more tokenization boundary, from the right, in $w_1 \curlyvee \ldots \curlyvee w_k \curlyvee \psi_i$, and each r_i has lower priority than r_{i-1}, independent of which dictionary equivalent to D is considered. We thus have that $l(D) \leq c(D) \cdot \max\{|uv| \mid u \curlyvee v \in D\}$.

It is necessary to obtain $c(D)$ through a more efficient procedure than pure enumeration. Note, the enumeration involved in constructing f in Algorithm 2 inefficiently "finds" the true $l(D)$ anyway. ⋄

5 Conclusions and Future Work

In some ways, the main contribution of this paper is the more formal definition of the tokenization semantics, allowing them to be studied in closer details. We leveraged this to establish some interesting properties of tokenizations, and established algorithms for both incrementally modifying a tokenization and for doing tokenizations left-to-right using space constant in the length of the string.

Much future work remains, including the following.

- Experimental studies should be performed, for example, testing how the incremental algorithm behaves in random cases. It seems likely to be extremely efficient in practice, as long chains of changes, or infinite ones such as in Example 2, do not seem to be very common or realistic. We do

not offer implementation details in this paper, as in *general* the bounds implied by the constructions are high enough to be impractical.

- Determining better upper bounds for the lookahead constant. Some more aspects that will be considered, are listed throughout the paper, and in particular in Remark 5. Once such bounds are established practical implementation details can be considered.
- Beyond improving the bounds of Theorem 2 there is also its converse, determining how much a tokenization may change from appending strings on the *left*. Example 2 already demonstrates that this is not finite for all D, but from random testing it seems to *often* be finite. It should be investigated whether the dictionaries exhibiting these infinite ripples of changes do so due to some easily decidable property, and whether the "lookbehind" (compare Definition 6) is efficiently computable when finite.

References

[1] Ariel Ekgren, Amaru Cuba Gyllensten, Felix Stollenwerk, Joey Öhman, Tim Isbister, Evangelia Gogoulou, Fredrik Carlsson, Alice Heiman, Judit Casademont & Magnus Sahlgren (2023): *GPT-SW3: An Autoregressive Language Model for the Nordic Languages*. arXiv:2305.12987.

[2] Hugging Face (2023): *Transformers*. `https://github.com/huggingface/transformers/blob/v4.28.1/src/transformers/models/gpt2/tokenization_gpt2.py`.

[3] Marti A Hearst (1997): *Text tiling: Segmenting text into multi-paragraph subtopic passages*. *Computational linguistics* 23(1), pp. 33–64.

[4] Taku Kudo & John Richardson (2018): *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*. *CoRR* abs/1808.06226. arXiv:1808.06226.

[5] Monica Lam, Ravi Sethi, Jeffrey D Ullman & Alfred Aho (2006): *Compilers: principles, techniques, and tools*. Pearson Education.

[6] OpenAI (2022): *ChatGPT: Optimizing language models for dialogue*. Available at `https://openai.com/blog/chatgpt/`.

[7] Rico Sennrich, Barry Haddow & Alexandra Birch (2016): *Neural Machine Translation of Rare Words with Subword Units*. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Association for Computational Linguistics, Berlin, Germany, pp. 1715–1725, doi:10.18653/v1/P16-1162. Available at `https://aclanthology.org/P16-1162`.

[8] Philippe Skolka, Cristian-Alexandru Staicu & Michael Pradel (2019): *Anything to Hide? Studying Minified and Obfuscated Code in the Web*. In: *The World Wide Web Conference*, WWW '19, Association for Computing Machinery, New York, NY, USA, p. 1735–1746, doi:10.1145/3308558.3313752.

[9] Felix Stollenwerk (2023): *Training and Evaluation of a Multilingual Tokenizer for GPT-SW3*. arXiv:2304.14780.

[10] Nicolaas Weideman, Brink van der Merwe, Martin Berglund & Bruce W. Watson (2016): *Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA*. In Yo-Sub Han & Kai Salomaa, editors: *Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings*, *Lecture Notes in Computer Science* 9705, Springer, pp. 322–334, doi:10.1007/978-3-319-40946-7_27.

On Languages Generated by Signed Grammars

Ömer Eğecioğlu
Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
omer@cs.ucsb.edu

Benedek Nagy
Department of Mathematics, Eastern Mediterranean University
99628 Famagusta, North Cyprus, Mersin-10, Turkey
Department of Computer Science, Institute of Mathematics and Informatics,
Eszterházy Károly Catholic University, Eger, Hungary
nbenedek.inf@gmail.com

We consider languages defined by signed grammars which are similar to context-free grammars except productions with signs associated to them are allowed. As a consequence, the words generated also have signs. We use the structure of the formal series of yields of all derivation trees over such a grammar as a method of specifying a formal language and study properties of the resulting family of languages.

1 Introduction

We consider properties of signed grammars, which are grammars obtained from context-free grammars (CFGs) by allowing right hand sides of productions to have negative signs in front. The concept of generation for such grammars is somewhat different from that of context-free grammars. A signed grammar is said to generate a language $\mathscr{L}$ if the formal sum of the yields over all derivation trees over the grammar corresponds to the list of words in $\mathscr{L}$. For a signed grammar, the yields of derivation trees may have negative signs attached to them, but the requirement is that when the arithmetic operations are carried out in the formal sum, the only remaining words are those of $\mathscr{L}$, each appearing with multiplicity one.

The structure of context-free languages (CFLs) under a full commutation relation defined on the terminal alphabet is the central principle behind Parikh's theorem [24]. In partial commutation, the order of letters of some pairs of the terminal alphabet is immaterial, that is, if they appear consecutively, the word obtained by swapping their order is equivalent to the original one. These equivalence classes are also called traces and studied intensively in connection to parallel processes [18, 12, 21, 4]. Our motivation for this work is languages obtained by picking representatives of the equivalence classes in Σ^* under a partial commutativity relation, called Cartier-Foata languages [1]. In the description of these languages with Kleene-closure type expansions, words appear with negative signs attached to them. However such words are cancelled by those with positive signs, leaving only the sum of the words of the language. An example of this is $(a+b-ba)^*$ which is more familiarly denoted by the regular expression a^*b^*. The interesting aspect of Cartier-Foata languages is that the words with negative signs cancel out automatically, leaving only the representative words, each appearing exactly once.

Motivated by these languages, we consider grammars which are obtained from context-free grammars by allowing signed productions, i.e., normal productions (in the role of positive productions) and productions of the form $A \to -\alpha$ (negative productions). In this way, a derivation results in a signed word where the sign depends on the parity of the number of negative rules applied in the derivation. We consider those derivations equivalent that belong to the same derivation tree, and actually, the derivation tree itself defines the sign of the derived word. The language generated by such a grammar is obtained by taking all possible derivation trees for a given word (both its positive and negative derivations) and

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on
Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 28–37, doi:10.4204/EPTCS.388.5

requiring that the sum of the yields of all derivation trees over the grammar simply is a list of the words in a language $\mathscr{L}$. This means that the simplified formal sum is of the form $\sum_{w\in\mathscr{L}} w$, each word of the language appearing with multiplicity one. (Without loss of generality, in this study, we restrict ourselves to grammars having finitely many parse trees for each of the derived words.)

On one hand, the requirements in the specification of a language generated by a signed grammar may seem too restrictive. But at the same time this class of languages includes all unambiguous context-free languages and it is closed under complementation, and consequently can generate languages that are not even context-free. Therefore it is of interest to consider the interplay between the restrictions and various properties of languages generated by signed grammars.

2 Preliminaries

Given a language $\mathscr{L}$ over an alphabet Σ, we identify $\mathscr{L}$ with the formal sum of its words denoted by $f(\mathscr{L})$:

$$f(\mathscr{L}) = \sum_{w\in\mathscr{L}} w\,. \tag{1}$$

The sum in (1) is also referred to as the *listing series* of $\mathscr{L}$. A *weighted series of* $\mathscr{L}$ is a formal series of the form $\sum_{w\in\mathscr{L}} n_w w$ where n_w are integers. Thus a weighted series of Σ^*

$$\sum_{w\in\Sigma^*} n_w w$$

is the listing series of some language $\mathscr{L}$ over Σ iff

$$n_w = \begin{cases} 1 & \text{if } w\in\mathscr{L} \\ 0 & \text{if } w\notin\mathscr{L}\,. \end{cases} \tag{2}$$

We are allowed ordinary arithmetic operations on weighted series in a natural way. The important thing is that a weighted series is the listing series of a language $\mathscr{L}$ iff the coefficients of the words in $\mathscr{L}$ in the weighted series are 1, and all the others are 0. So for example over $\Sigma=\{a,b,c\}$, the weighted series $a+b+c+ba$ is the listing series of the finite language $\mathscr{L}=\{a,b,c,ba\}$, whereas the weighted series $a+b+c-ba$ does not correspond to a language over Σ. This is because in the latter example n_w does not satisfy (2) for $w=ba$. As another example, the difference of the weighted series $2a+3b-c+ba$ and $a+2b-2c+ba$ corresponds to the language $\mathscr{L}=\{a,b,c\}$.

2.1 CFGs and degree of ambiguity

Next we look at the usual CFGs $G=(V,\Sigma,P,S)$. Here the start symbol is $S\in V$. Let T be a parse (derivation) tree over G with root label S and terminal letters as labels of the leaves of T. Let $Y(T)\in\Sigma^*$ be the *yield* of T. Then the language generated by G is

$$\mathscr{L}(G) = \{Y(T) \mid T \text{ is a parse tree over } G\}\,.$$

This is equivalent to $\mathscr{L}(G)=\{w\in\Sigma^* \mid S \overset{*}{\underset{G}{\Longrightarrow}} w\}$. For a CFG G, we can define the formal weighted sum

$$f(G) = \sum_{T\in\mathscr{T}_G} Y(T) = \sum_{w\in\Sigma^*} n_w w \tag{3}$$

where $\mathscr{T}_G$ denotes all parse trees over G. Various notions of ambiguity for CFLs can be interpreted as the nature of the coefficients n_w that appear in (3). Rewriting some of the definitions in Harrison [8, pp. 240-242] in terms of these coefficients, we have

1. Given $k \geq 1$, G is *ambiguous of degree k* if $n_w \leq k$ for all $w \in \mathscr{L}(G)$.
2. $\mathscr{L}$ is *inherently ambiguous of degree* $k \geq 2$ if $\mathscr{L}$ cannot be generated by any grammar that is ambiguous of degree less than k but can be generated by by a grammar that is ambiguous of degree k. In other words the degree of ambiguity of a CFL is the least upper bound for the number of derivation trees which a word in the language can have.
3. $\mathscr{L}$ is *finitely inherently ambiguous* if there is some k and some G for $\mathscr{L}$ so that G is inherently ambiguous of degree k.
4. A CFG G is *infinitely ambiguous* if for each $i \geq 1$, there exists a word in $\mathscr{L}(G)$ which has at least i parse trees. A language L is *infinitely inherently ambiguous* if every grammar generating L is infinitely ambiguous.

The CFL $\mathscr{A} = \{a^i b^j c^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous of degree 2 [8, p. 240], $\mathscr{A}^m$ is inherently ambiguous of degree 2^m [8, Theorem 7.3.1], and $\mathscr{A}^*$ is infinitely inherently ambiguous [8, Theorem 7.3.3]. Another interesting CFL which is infinitely inherently ambiguous is Crestin's language [3] of double palindromes over a binary alphabet $\{w_1 w_2 \mid w_1, w_2 \in \{a,b\}^*, w_1 = w_1^R, w_2 = w_2^R\}$. Furthermore, for every $k \geq 1$, there exist inherently ambiguous CFLs of degree k. The behavior of the sequence n_w over all CFGs for a language was studied by Wich [25, 26].

3 Signed grammars

We consider *signed grammars* G which are like CFGs but with a sign associated with each production, that is, apart from the usual (say positive) productions, we allow productions of the form $A \to -\alpha$. In the derivation relation we use the signs as usual in a multiplicative manner: We start the derivation from the sentence symbol (with $+$ sign, but as usual we may not need to put it, as it is the default sign). The derivation steps, as rewriting steps, occur as they are expected in a CFG, the only extension is that we need to deal with also the sign. When a positive production is applied in a sentential form, its sign does not change, while whenever a negative production is applied, this derivation step switches the sign of the sentential form. Thus, in this case the yield of a parse tree of G is a word over Σ with a $\pm$ sign attached to it. Furthermore, the sign of a derived word depends only on the parity of the number of negative productions used during its derivation. Therefore, different derivation trees for the same word may lead to the word with different signs attached to it. We note that, in fact, any CFG is a signed grammar. For a signed grammar G, let $f(G)$ be defined as in (3), where again $\mathscr{T}_G$ denotes all parse trees over G. Without loss of generality, we may assume that in the grammar G there are only finitely many parse trees for any of the words generated by the grammar.

Definition 1 *We say that a signed grammar G* generates *a language $\mathscr{L}$ iff the weighted series $f(G)$ in (3) is the listing series of $\mathscr{L}$, i.e. $f(G) = f(\mathscr{L})$.*

3.1 Examples of languages generated by signed grammars

Example 1 For the signed grammar G_1 with start symbol A and productions $A \to -aA \,|\, \lambda$, we have

$$f(G_1) = \sum_{i \geq 0} a^{2i} - \sum_{i \geq 0} a^{2i+1}. \tag{4}$$

Therefore the signed grammar G with productions $S \to A \,|\, B$, $A \to -aA \,|\, \lambda$, $B \to aaB \,|\, a$ generates the regular language $(aa)^*$. As this is our first example, we provide details of the derivations in G:

- The empty word λ can be derived only in one way, by applying a positive production, thus it is in the language.
- By applying a negative and a positive production, $S \Rightarrow A \Rightarrow -aA \Rightarrow -a$ yields $-a$, and $S \Rightarrow B \Rightarrow a$ yields $+a$. These two are the only derivations over G for $\pm a$. This means that the word a is not in the language.
- For the word aa, the only derivation is $S \Rightarrow A \Rightarrow -aA \Rightarrow aaA \Rightarrow aa$. Consequently aa is in the generated language.
- Finally, by induction, one can see that an even number of a-s can only be produced by starting the derivation by $S \Rightarrow A$. Following this positive production, each usage of $A \to -aA$ introduces a negative sign. Therefore each word of the form a^{2i} is generated once this way with a $+$ sign. On the other hand there are two possible ways to produce a string a^{2i+1} of an odd number of a-s. One of these starts with $A \Rightarrow -aA$ as before and produces $-a^{2i+1}$ after an odd number of usages of $A \to -aA$; the other one starts with $S \Rightarrow B$ and produces a^{2i+1} after an even number of applications of $B \to aaB$, followed by $B \to a$. Therefore odd length words cancel each other out and are not in the language generated.

Another way to look at this is to note that for the (signed) grammar G_2 with the start symbol B and productions $B \to aaB \,|\, a$, we have

$$f(G_2) = \sum_{i \geq 0} a^{2i+1}\,, \tag{5}$$

and the words generated by G are given by the formal sum of (4) and (5).

Example 2 The signed grammar with productions $S \to aS \,|\, bS \,|\, -baS \,|\, \lambda$ generates the regular language denoted by the regular expression a^*b^*. First few applications of the productions give

$$\begin{aligned}
&\lambda;\\
&a+b-ba;\\
&a^2+ab-aba+ba+b^2-b^2a-ba^2-bab+baba;
\end{aligned}$$

in which the only immediate cancellation is of $-ba$, though all words carrying negative signs will eventually cancel out. This is a special case of the Cartier-Foata result [1], [5, Section 8.4].

Example 3 Over the decimal (or the binary) alphabet we can construct an unambiguous regular grammar G that generates all nonnegative even numbers, e.g., $S \to 9S \,|\, 8A \,|\, 7S \,|\, 6A \,|\, 5S \,|\, 4A \,|\, 3S \,|\, 2A \,|\, 1S \,|\, 0A$ and $A \to 9S \,|\, 8A \,|\, 7S \,|\, 6A \,|\, 5S \,|\, 4A \,|\, 3S \,|\, 2A \,|\, 1S \,|\, 0A \,|\, \lambda$. Let, further, a regular grammar G' be generating the numbers which are divisible by 6 (e.g., based on the deterministic finite automaton checking the sum of the digits to be divisible by 3 and the last digit must be even, we need states/nonterminals to count the sum of already read digits by mod 3 and take care to the last digit as we did for G).

Then $\mathscr{L}(G)$ consists of all even numbers and $\mathscr{L}(G')$ consists of all numbers divisible by 6. Now, from G', we may make a signed grammar G'' which allows us to derive every multiple of 6 with the sign $-$. Then by combining the two grammars G and G'', we can easily give a signed grammar that generates all even numbers that are not divisible by 3 (i.e., even numbers not divisible by 6).

Example 4 Over the alphabet $\{a,b\}$ consider the signed grammar with productions $S \to aSa \,|\, bSb \,|\, a \,|\, b$. This so far generates odd length palindromes. Let us add the productions $S \to -A,\ A \to -abAba \,|\, a$.

Then each odd length palindrome with the letter b in the middle has exactly one derivation tree with a $+$ sign. There are no cancellations for these and therefore all odd length palindromes with b in the middle are in the language. If the middle of an odd length palindrome w is a but not $ababa$, then w is not in $\mathscr{L}$ as it has also derivation tree with $-$ sign. Similarly, if the middle of w is $ababa$ but not $ababababa$, w is in $\mathscr{L}$. In general, if an odd length palindrome w has $(ab)^{2k-1}a(ba)^{2k-1}$ in the middle, but it does not have $(ab)^{2k}a(ba)^{2k}$ in its middle, then it is in $\mathscr{L}$. Here the number of derivation trees for a word with a $+$ sign is either equal to the number of derivation trees with a $-$ sign for the word, or it is exactly one more.

Example 5 For the following signed grammar

$$\begin{aligned} S_1 &\to -aA \,|\, Ba \,|\, a \\ A &\to -aA \,|\, Ba \,|\, a \\ B &\to -aB \,|\, Ba \,|\, -a \,|\, aa \end{aligned}$$

for n odd, there are 2^{n-1} parse trees for a^n and $2^{n-1}-1$ parse trees for $-a^n$. For n even, there are $2^{n-1}-1$ parse trees for a^n and 2^{n-1} parse trees for $-a^n$. In other words for the above grammar

$$\begin{aligned} f(G) &= \sum_{i\geq 0} 2^{2i}a^{2i+1} + \sum_{i\geq 0}(2^{2i}-1)a^{2i} - \sum_{i\geq 0}(2^{2i}-1)a^{2i+1} - \sum_{i\geq 0} 2^{2i}a^{2i} \\ &= \sum_{i\geq 0}(-1)^i a^{i+1}. \end{aligned}$$

If we add the productions $S \to S_1 \,|\, S_2,\ S_2 \to aaS_2 \,|\, aa$ then the resulting signed grammar generates the regular language $a(aa)^*$. Even though the language generated is very simple we see that signed grammars possess some interesting behavior.

4 Properties of languages generated by signed grammars

In this section our aim is twofold. On the one hand we give some closure properties of the class of languages generated by our new approach and, on the other hand, we give hierarchy like results by establishing where this family of languages is compared to various other classes.

We immediately observe that in the weighted sum (3) for a CFG G (i.e. a signed grammar G with no signed productions), the coefficient n_w is the number of parse trees for w over G, in other words the degree of ambiguity of w.

Proposition 1 *Any unambiguous CFL is generated by a signed grammar.*

Proof An unambiguous CFL $\mathscr{L}$ is generated by the signed grammar G where G is any unambiguous CFG for $\mathscr{L}$. •

As the class of unambiguous CFLs contains all deterministic CFLs, $LR(0)$ languages, regular languages, subsets of $w_1^*w_2^*$ [7, Theorem 7.1], all of these languages are generated by signed grammars. Further, all these classes are proper subsets of the class of languages generated by signed grammars.

Now we present a closure property.

Proposition 2 *Languages generated by signed grammars are closed under complementation.*

Proof Take an unambiguous CFG for Σ^* with start symbol S_1. If $\mathscr{L}$ is generated by a signed grammar with start symbol S_2 (and no common nonterminal in the two grammars), then the productions of the two grammars together with $S \to S_1 \mid -S_2$ with a new start symbol S generates $\overline{\mathscr{L}}$. •

We continue the section comparing our new class of languages with other well-known language class, the class of CFLs.

In 1966 Hibbard and Ullian constructed an unambiguous CFL whose complement is not a CFL [9, Theorem 2]. Recently Martynova and Okhotin constructed an unambiguous linear language whose complement is not context-free [14]. This shows that unambiguous linear CFLs are not closed under complementation while providing another proof of Hibbard and Ullian's result.

We know that languages generated by signed grammars are closed under complementation, and also every unambiguous CFL is generated by a signed grammar. A consequence of this is that signed grammars can generate languages that are not context-free.

Proposition 3 *There is a language generated by a signed grammar that is not context-free.*

Proof If $\mathscr{L}$ is the unambiguous CFL constructed by Hibbard and Ullian, then $\mathscr{L}$ and therefore $\overline{\mathscr{L}}$ are generated by signed grammars. But we know that $\overline{\mathscr{L}}$ is not context-free. •

Actually, our last proposition shows that the generative power of signed grammars is surprisingly large, it contains, e.g., all deterministic and unambiguous CFLs and their complements. Thus, one can easily generate some languages that are not in the class of CFLs.

Continuing with closure properties, recall that disjoint union is an operation that is defined only on disjoint sets which produces their union.

Proposition 4 *Languages generated by signed grammars are closed under disjoint union* $\uplus$.

Proof Let $\mathscr{L}_1$ and $\mathscr{L}_2$ be two languages over an alphabet Σ such that $\mathscr{L}_1 \cap \mathscr{L}_2 = \emptyset$. Let $\mathscr{L}_1$ be generated by a signed grammar with start symbol S_1 and $\mathscr{L}_2$ be generated by a signed grammar with start symbol S_2, such that the sets of nonterminals of these two grammars are disjoint. Then the productions of the two grammars together with $S \to S_1 \mid S_2$ with a new start symbol S generates the disjoint union $\mathscr{L}_1 \uplus \mathscr{L}_2$. •

Now, let us define the set theoretical operation "subset minus" ($\ominus$), as follows: let $A \subseteq B$, then $B \ominus A = B \setminus A$. This type of setminus operation is defined only for sets where the subset condition holds.

Proposition 5 *Languages generated by signed grammars are closed under subset minus* $\ominus$.

Proof Let $\mathscr{L}_1 \subseteq \mathscr{L}_2$ be two languages over a given alphabet Σ. Take the signed grammar for $\mathscr{L}_1$ with start symbol S_1. If $\mathscr{L}_2$ is generated by a signed grammar with start symbol S_2 (with no common nonterminals of the two grammars), then the productions of the two grammars together with $S \to S_1 \mid -S_2$ with a new start symbol S generates the language of $\mathscr{L}_2 \ominus \mathscr{L}_1$. •

Let $\mathscr{L}_1, \mathscr{L}_2 \subseteq \Sigma^*$ be two languages and $\$ \notin \Sigma$. The $-concatenation of $\mathscr{L}_1$ and $\mathscr{L}_2$ is the language $\mathscr{L}_1\$\mathscr{L}_2$ over the alphabet $\Sigma \cup \{\$\}$.

Proposition 6 *Languages generated by signed grammars are closed under* $-*concatenation.*

Proof The language $\mathscr{L}_1\$$ has the prefix property (i.e. it is prefix-free) due to the special role of the marker $. Let G_1 and G_3 be signed grammars with disjoint variables and start symbols S_1 and S_3 that generate $\mathscr{L}_1$ and $\mathscr{L}_2$, respectively. Consider also the signed grammar G_2 with the single production

$S_2 \to \$$. Then the signed grammar which have all the productions of G_1, G_2, G_3 together with the production $S \to S_1 S_2 S_3$ where S is a new start symbol generates the language $\mathscr{L}_1 \$ \mathscr{L}_2$. The proof follows by observing that for $u, u' \in \mathscr{L}_1$ and $v, v' \in \mathscr{L}_2$, $u\$v = u'\$v'$ iff $u = u'$ and $v = v'$, so that each word that appears in the expansion of

$$\left(\sum_{w \in \mathscr{L}_1} w \right) \$ \left(\sum_{w \in \mathscr{L}_2} w \right)$$

has coefficient 1. •

In a similar manner, it can also be seen that we have a similar statement for languages over disjoint alphabet, i.e., the class of languages generated by signed grammars is closed under "disjoint concatenation" $\boxdot$.

Proposition 7 *Let $\mathscr{L}_1 \subseteq \Sigma_1^*$ and $\mathscr{L}_2 \subseteq \Sigma_2^*$ be two languages that are generated by signed grammars, where $\Sigma_1 \cap \Sigma_2 = \emptyset$. Then, the language $\mathscr{L}_1 \boxdot \mathscr{L}_2 = \mathscr{L}_1 \mathscr{L}_2$ can be generated by a signed grammar.*

In the following proposition, $f(\mathscr{L})$ and $f(G)$ are as defined in (1) and (3).

Proposition 8 *Suppose $\mathscr{L}$ generated by a signed grammar. Then there are CFGs G_1 and G_2 such that $f(\mathscr{L}) = f(G_1) - f(G_2)$.*

Proof Given a signed grammar over Σ, add an extra letter t to Σ and replace all productions of the form $A \to -\alpha$ by $A \to t\alpha$. The words generated by this CFG over $\Sigma \cup \{t\}$ with an even number of occurrences of t is a CFL since it is the intersection of CFL and the regular language, i.e. all words over $\Sigma \cup \{t\}$ with an even number of occurrences of t. Similarly, the words generated with an odd number of occurrences of t is a CFL. We can then take homomorphic images of these two languages generated by replacing t by λ and obtain two CFLs generated by CFGs G_1 and G_2. The weighted series $f(G)$ is then the difference of two weighted series

$$f(G) = f(G_1) - f(G_2) = \sum_{w \in \Sigma^*} n_w w \; - \; \sum_{w \in \Sigma^*} n'_w w \; . \tag{6}$$

In (6), the coefficients n_w and n'_w are nonnegative integers for all $w \in \Sigma^*$ as they count the number of derivation trees for w over G_1 and G_2, respectively. •

Remark 1 *In Proposition 8, $f(G_1) - f(G_2)$ is the listing series of $\mathscr{L}$, and therefore $n_w - n'_w = 1$ or $n_w - n'_w = 0$ for all $w \in \Sigma^*$. In the first case $w \in \mathscr{L}$, and in the second $w \notin \mathscr{L}$. Note that these conditions do not imply that $\mathscr{L} = \mathscr{L}(G_1) \setminus \mathscr{L}(G_2)$.*

5 Partial commutativity

Addition of commutativity relations to CFGs was considered in [19]. Here we consider partial commutativity defined on Σ^* where $\Sigma = \{x_1, x_2, \ldots, x_m\}$. Given an $m \times m$ symmetric $\{0,1\}$-matrix $A = [a_{i,j}]$ with 1s down the diagonal, a pair of letters x_i, x_j is a commuting pair iff $a_{i,j} = 1$. This defines an equivalence relation and partitions Σ^* into equivalence classes, also known as traces. Thinking about the element of the alphabet as processes and traces as their scheduling, commuting processes are considered as independent from each other. In this way the theory of traces has been intensively studied in connection to parallel processes [11, 12]. A (linearization of a) trace language is a union of some of these equivalence classes. Trace languages based on regular, linear and context-free languages (adding a partial

commutativity relation to the language) were studied and accepted by various types of automata with translucent letters in [21, 23, 22], respectively. Traces and trajectories are also analyzed in various grids [15, 16, 20]. On the other hand, the *Cartier–Foata language* $\mathscr{L}_A$ corresponding to the matrix A of a partial commutativity relation is constructed by picking a representative word from each equivalence class.

Let us define a set $F \subseteq \Sigma$ to be *commuting* if any pair of letters in F commute. Let $\mathscr{C}(A)$ denote the collection of all nonempty commuting sets. Denote by $w(F)$ the word obtained by juxtaposing the letters of F. The order in which these letters are juxtaposed is immaterial since all arrangements are equivalent.

The central result is that the listing series $f(\mathscr{L}_A)$ can be constructed directly from the matrix A:

$$f(\mathscr{L}_A) = \left(\sum_{F \in \mathscr{C}(A)} (-1)^{\#F} w(F) \right)^* = \sum_{n \geq 0} \left(\sum_{F \in \mathscr{C}(A)} (-1)^{\#F} w(F) \right)^n , \tag{7}$$

where $\#F$ denotes the number of elements of F.

Over $\Sigma = \{a,b\}$ where a and b commute, the Cartier-Foata theorem gives $\mathscr{L}_A$ as $(a+b-ba)^*$, which is to be interpreted as the weighted series $\lambda + (a+b-ba) + (a+b-ba)^2 + \cdots$ In this case the representatives of the equivalence classes are seen to be the words in a^*b^*. The essence of the theorem is that this is a listing series, so there is exactly one representative word from each equivalence class that remains after algebraic cancellations are carried out.

Similarly over $\Sigma = \{a,b,c\}$ with a,b and a,c commuting pairs, the listing series is $\lambda + (a+b+c-ba-ca) + (a+b+c-ba-ca)^2 + \cdots$

The words in this second language are generated by the signed grammar

$$S \to \lambda \,|\, aS \,|\, bS \,|\, cS \,|\, -baS \,|\, -caS \,.$$

6 Conclusions and a conjecture

Proposition 8 provides an expression for the listing series of a language generated by a signed grammar in terms of weighted listed series of two CFLs. However this result is short of a characterization in terms of CFLs. It is also possible to change the way signed grammars generate languages by requiring $n_w \geq 1$ in (2) instead of equality. In this way, every signed grammar would generate a language, and obviously, the class of generated languages would also change. However, our consideration in this paper to allow only 0 and 1 to be the signed sum, gives a nice and immediate connection to Cartier-Foata languages in the regular case by special regular like expressions.

Since by signed grammars, we generate languages based on counting the number of (signed) derivation trees, it is straightforward to see the connection between our grammars and unambiguous CFLs. On the other hand, there may be more than one derivation tree for a given word w, with the proviso that the algebraic sum of the yields of derivation trees for it has multiplicity $n_w \in \{0,1\}$. Therefore signed grammars may also generate ambiguous CFLs. In this sense, the bottom of the hierarchy, the unambiguous CFLs are included in the class we have investigated. On the other hand, if there are multiple derivation trees for a word generated by a grammar, by playing with their signs, we have a chance to somehow have their signed sum to be in $\{0,1\}$. Thus, it may be possible to generate languages that are higher in the hierarchy based on ambiguity. However, this is still an open problem.

We have shown that signed grammars can generate languages that are not context-free. It would be of interest to use the fact that the languages generated by signed grammars are closed under complementation to show that signed grammars can generate inherently ambiguous CFLs. One way to do this

would be to start with an unambiguous CFL whose complement is an inherently ambiguous CFL. The standard examples of inherently ambiguous CFLs do not seem to have this property. By the Chomsky-Schützenberger theorem [2] the generating function of an unambiguous CFL is algebraic. Using the contrapositive and analytical methods, Flajolet [6] and later Koechlin [13] devised ingenious methods to show the transcendence of the generating function of a given language to prove its inherent ambiguity. However if the generating function of $\mathscr{L}$ is transcendental so is the generating function of its complement $\overline{\mathscr{L}}$. This means that one needs to look among inherently ambiguous languages with algebraic generating functions (e.g. $\{a^i b^j c^k \mid i = j \text{ or } j = k\}$, see [13, Proposition 14]) if the complement has any chance of being unambiguous.

So it would be nice to have an answer to the following question: *Is there an unambiguous CFL whose complement is an inherently ambiguous CFL?*

A related problem of showing the existence of an inherently ambiguous CFL whose complement is also an inherently ambiguous CFL was settled by Maurer [17].

References

[1] P. Cartier & D. Foata (1969): *Problèmes combinatoires de commutation et réarrangements*. Springer, doi:10.1007/BFb0079468.

[2] N. Chomsky & M. P. Schützenberger (1963): *The Algebraic Theory of Context-Free Languages*. In P. Braffort & D. Hirschberg, editors: *Computer Programming and Formal Systems*, *Studies in Logic and the Foundations of Mathematics* 35, Elsevier, pp. 118–161, doi:10.1016/S0049-237X(08)72023-8. Available at `https://www.sciencedirect.com/science/article/pii/S0049237X08720238`.

[3] J. P. Crestin (1972): *Un langage non ambigu dont le carré est d'ambiguité non bornée*. In Maurice Nivat, editor: *Automata, Languages and Programming, Colloquium, Paris, France, July 3-7, 1972*, North-Holland, Amsterdam, pp. 377–390. Available at `https://api.semanticscholar.org/CorpusID:44540005`.

[4] V. Diekert & G. Rozenberg, editors (1995): *The Book of Traces*. World Scientific, doi:10.1142/2563.

[5] Ö. Eğecioğlu & A. Garsia (2021): *Lessons in Enumerative Combinatorics*. Springer, Graduate Texts in Mathematics, doi:10.1007/978-3-030-71250-1.

[6] P. Flajolet (1987): *Analytic models and ambiguity of context-free languages*. *Theor. Comput. Sci.* 49(2), pp. 283–309, doi:10.1016/0304-3975(87)90011-9.

[7] S. Ginsburg & J. S. Ullian (1966): *Ambiguity in context free languages*. *J. ACM* 13, pp. 62–89, doi:10.1145/321341.321345.

[8] M. A. Harrison (1978): *Introduction to Formal Language Theory*. Addison-Wesley.

[9] T. N. Hibbard & J. S. Ullian (1966): *The independence of inherent ambiguity from complementedness among context-free languages*. *Journal of the ACM* 13(4), pp. 588–593, doi:10.1145/321356.321366.

[10] J. Hopcroft & J. Ullman (1979): *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

[11] Ryszard Janicki, Jetty Kleijn, Maciej Koutny & Lukasz Mikulski (2017): *Invariant Structures and Dependence Relations*. *Fundam. Informaticae* 155(1-2), pp. 1–29, doi:10.3233/FI-2017-1574.

[12] Ryszard Janicki, Jetty Kleijn, Maciej Koutny & Lukasz Mikulski (2019): *Classifying invariant structures of step traces*. *J. Comput. Syst. Sci.* 104, pp. 297–322, doi:10.1016/j.jcss.2017.05.002.

[13] F. Koechlin (2022): *New Analytic Techniques for Proving the Inherent Ambiguity of Context-Free Languages*. In Anuj Dawar & Venkatesan Guruswami, editors: *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2022)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 250, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany,

pp. 41:1–41:22, doi:10.4230/LIPIcs.FSTTCS.2022.41. Available at https://drops.dagstuhl.de/opus/volltexte/2022/17433.

[14] O. Martynova & A. Okhotin (2023): *Non-Closure under Complementation for Unambiguous Linear Grammars*. *Inf. Comput.* 292(C), doi:10.1016/j.ic.2023.105031.

[15] Alexandru Mateescu, Grzegorz Rozenberg & Arto Salomaa (1998): *Shuffle on Trajectories: Syntactic Constraints*. *Theor. Comput. Sci.* 197(1-2), pp. 1–56, doi:10.1016/S0304-3975(97)00163-1.

[16] Alexandru Mateescu, Kai Salomaa & Sheng Yu (2000): *On Fairness of Many-Dimensional Trajectories*. *J. Autom. Lang. Comb.* 5(2), pp. 145–157, doi:10.25596/jalc-2000-145.

[17] H. A. Maurer (1970): *A note on the complement of inherently ambiguous context-free languages*. *Commun. ACM* 13, p. 194, doi:10.1145/362052.362065.

[18] A. Mazurkiewicz (1977): *Concurrent Program Schemes and their Interpretations*. *DAIMI Report Series* 6(78), doi:10.7146/dpb.v6i78.7691. Available at https://tidsskrift.dk/daimipb/article/view/7691.

[19] Benedek Nagy (2009): *Languages generated by context-free grammars extended by type $AB \to BA$ rules*. *Journal of Automata, Languages and Combinatorics* 14, pp. 175–186, doi:10.25596/jalc-2009-175.

[20] Benedek Nagy & Arif A. Akkeles (2017): *Trajectories and Traces on Non-traditional Regular Tessellations of the Plane*. In Valentin E. Brimkov & Reneta P. Barneva, editors: *Combinatorial Image Analysis - 18th International Workshop, IWCIA 2017, Plovdiv, Bulgaria, June 19-21, 2017, Proceedings*, *Lecture Notes in Computer Science* 10256, Springer, pp. 16–29, doi:10.1007/978-3-319-59108-7_2.

[21] Benedek Nagy & Friedrich Otto (2010): *CD-Systems of Stateless Deterministic R(1)-Automata Accept All Rational Trace Languages*. In Adrian-Horia Dediu, Henning Fernau & Carlos Martín-Vide, editors: *Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010. Proceedings*, *Lecture Notes in Computer Science* 6031, Springer, pp. 463–474, doi:10.1007/978-3-642-13089-2_39.

[22] Benedek Nagy & Friedrich Otto (2011): *An Automata-Theoretical Characterization of Context-Free Trace Languages*. In Ivana Cerná, Tibor Gyimóthy, Juraj Hromkovic, Keith G. Jeffery, Rastislav Královic, Marko Vukolic & Stefan Wolf, editors: *SOFSEM 2011: Theory and Practice of Computer Science - 37th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 22-28, 2011. Proceedings*, *Lecture Notes in Computer Science* 6543, Springer, pp. 406–417, doi:10.1007/978-3-642-18381-2_34.

[23] Benedek Nagy & Friedrich Otto (2020): *Linear automata with translucent letters and linear context-free trace languages*. *RAIRO Theor. Informatics Appl.* 54, p. 3, doi:10.1051/ita/2020002.

[24] R. J. Parikh (1961): *Language generating devices*. *MIT Res. Lab., Quarterly Progress Report* 60, pp. 199–212.

[25] K. Wich (2000): *Sublinear Ambiguity*. In Mogens Nielsen & Branislav Rovan, editors: *Mathematical Foundations of Computer Science 2000*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 690–698, doi:10.1007/3-540-44612-5_64.

[26] K. Wich (2005): *Sublogarithmic ambiguity*. *Theoretical Computer Science* 345(2), pp. 473–504, doi:10.1016/j.tcs.2005.07.024.

Final Sentential Forms

Tomáš Kožár
Faculty of Information Technology
Brno University of Technology
Czech Republic
ikozar@fit.vut.cz

Zbyněk Křivka
Faculty of Information Technology
Brno University of Technology
Czech Republic
krivka@fit.vut.cz

Alexander Meduna
Faculty of Information Technology
Brno University of Technology
Czech Republic
meduna@fit.vut.cz

Let G be a context-free grammar with a total alphabet V, and let F be a *final language* over an alphabet $W \subseteq V$. A *final sentential form* is any sentential form of G that, after omitting symbols from $V - W$, it belongs to F. The string resulting from the elimination of all nonterminals from W in a final sentential form is in the *language of G finalized by F* if and only if it contains only terminals.

The language of any context-free grammar finalized by a regular language is context-free. On the other hand, it is demonstrated that L is a recursively enumerable language if and only if there exists a propagating context-free grammar G such that L equals the language of G finalized by $\{w\#w^R \mid w \in \{0,1\}^*\}$, where w^R is the reversal of w.

1 Introduction

The present paper introduces and studies *final sentential forms* of context-free grammars. These forms represent the sentential forms in which the sequences of prescribed symbols, possibly including nonterminals, belong to given *final languages*. If all the other symbols are terminals, these final forms are changed to the sentences of the generated languages by simply eliminating all nonterminals in them. Next, we sketch both a practical inspiration and a theoretical reason for introducing this new way of context-free language generation.

I. Indisputably, parsing represents a crucially important application area of ordinary context-free grammars (see Chapters 3 through 5 in [4]) as well as their modified versions, such as regulated grammars (see Section 20.3 in [6]). During the parsing process, the correctness of the source program syntax is often verified before all nonterminals are eliminated; nevertheless, most classically constructed parsers go on eliminating these nonterminals by using erasing rules until only terminals are derived. As a result, the entire parsing process is slowed down uselessly during this closing phase (for a simple, but straightforward illustration of this computational situation, see, for instance, Case Study 14/35 in [4] or Example 4.35 in [1]). Clearly, as the newly introduced way of language generation frees us from a necessity of this closing elimination of all nonterminals, the parsers that make use of it work faster.

II. From a theoretical viewpoint, in the present paper, we achieve a new representation for recursively enumerable languages based upon context-free languages. Admittedly, the theory of formal languages is overflown with many representations for recursively enumerable languages based upon operations over some context-free languages or their special cases (see Section 4.1.3 in [7]). Nonetheless, we believe this new representation is of some interest when compared with the previously demonstrated representations. Indeed, each of the already existing representations is demonstrated, in essence, by a proof that has the following general format. (i) First, given any recursively enumerable language L, it represents L by a suitable language model G, such as a phrase structure grammar in a normal form. (ii) Then, from G, it derives both operations and context-free

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 38–47, doi:10.4204/EPTCS.388.6

languages involved in the representation in question. (iii) Finally, it shows that the representation made in this way from G holds true. What is important from our standpoint is that in a proof like this, the specific form of all the operations as well as the languages involved in the representation always depend on G, which generates L. As opposed to this, the new representation achieved in the present paper is much less dependent on L or any of its language models . More precisely, we demonstrate the existence of a unique constant language C defined as $C = \{w\#w^R \mid w \in \{0,1\}^*\}$ and express any recursively enumerable language L by using C and a minimal linear language without any operation. Consequently, C always remains unchanged and, therefore, independent of L or its models. Considering this independency as well as the absence of any operations in the new representation, we believe this representation might be of some interest to formal language theory.

To give a more detailed insight into this study, we first informally recall the notion of an ordinary context-free grammar and its language (this paper assumes a familiarity with formal language theory). A context-free grammar G is based upon a grammatical alphabet V of symbols and a finite set of rules. The alphabet V is divided into two disjoint subalphabets—the alphabet of terminals T and the alphabet of nonterminals N. Each rule has the form $A \to x$, where A is a nonterminal and x is a string over V. Starting from a special start nonterminal, G repeatedly rewrites strings according to its rules, and in this way, it generates its sentential forms. Sentential forms that consist only of terminal symbols are called sentences, and the set of all sentences represents the language generated by G.

In this paper, we shortened the generating process sketched above by introducing a final language F over a subalphabet $W \subseteq V$. A *final sentential form* of G is any of the sentential forms in which the sequence of symbols from W belong to F. If in this form, all the symbols from $V - W$ are terminals, the string obtained by eliminating all nonterminals from $N \cap W$ results into a sentence of the generated language $L(G,F)$ finalized by F.

Next, we illustrate the newly introduced concept of final sentential forms by a simple example in linguistic morphology, which studies word formation, such as inflection and compounding, in natural languages.

Example 1. Consider an alphabet Σ of consonants and vowels. Suppose that a morphological study deals with a language L consisting of all possible words over Σ together with their consonant-vowel binary schemes in which every consonant and every vowel are represented by 1 and 0, respectively. Mathematically, $L = \{w\#\sigma(w) \mid w \in \Sigma^+\}$, where σ is the homomorphism from Σ^* to $\{0,1\}^*$ defined as $\sigma(x) = 1$ and $\sigma(y) = 0$ for every consonant x in Σ and every vowel y in Σ, respectively. For instance, considering Σ as the English alphabet, $the\#110 \in L$ while $the\#100 \notin L$. Define the context-free grammar G with the following rules.

- $S \to A\#B$, $B \to 0YB$, $B \to 0Y$, $B \to 1XB$, $B \to 1X$,
- $A \to aAY$, $A \to aY$ for all vowels a in Σ,
- $A \to bAX$, $A \to bX$ for all consonants b in Σ,

where the uppercase symbols are nonterminals with S being the start nonterminal, and the other symbols are terminals. Set $W = \{X,Y,\#\}$ and $F = \{w\#w^R \mid w \in \{X,Y\}^*\}$. For instance, take this step-by-step derivation

$$\begin{aligned} S &\Rightarrow A\#B \Rightarrow tAX\#B \Rightarrow thAXX\#B \Rightarrow theYXX\#B \\ &\Rightarrow theYXX\#1XB \Rightarrow theYXX\#1X1XB \Rightarrow theYXX\#1X1X0Y \end{aligned}$$

In $theYXX\#1X1X0Y$, $YXX\#XXY \in F$, and apart from $X,Y,\# \in W$, $theYXX\#1X1X0Y$ contains only terminals. The removal of all Xs and Ys in $theYXX\#1X1X0Y$ results into $the\#110$, which thus belongs to $L(G,F)$. On contrary,

$$S \Rightarrow^* theYXX\#1X1XB \Rightarrow theYXX\#1X1X0YB = \gamma \Rightarrow theYXX\#1X1X0Y0Y = \delta$$

Let $T = \Sigma \cup \{0,1\}$. Consider γ. Although $YXX\#XXY \in F$, $the\#110B \notin L(G,F)$ since $B \notin W \cup T$. On the other hand, considering δ, after omitting symbols from $W - T$, we have $the\#1100 \in T^*$, but since $YXX\#XXYY \notin F$, $the\#1100 \notin L(G,F)$.

Clearly, $L(G,F) = L$.

As its main result, the present paper demonstrates that L is a recursively enumerable language if and only if $L = L(G, \{w\#w^R \mid w \in \{0,1\}^*\})$, where G is a context-free grammar; observe that in this equivalence, the final language $\{w\#w^R \mid w \in \{0,1\}^*\}$ remains constant independently of L. On the other hand, the paper also proves that any $L(G,F)$ is context-free if G is a context-free grammar and F is regular.

The rest of the paper is organized as follows. First, Section 2 gives all the necessary terminology and defines the new notions, informally sketched in this introduction. Then, Section 3 establishes the above-mentioned results and points out an open problem related to the present study.

2 Preliminaries and Definitions

This paper assumes that the reader is familiar with the language theory (see [5]).

For a set Q, $card(Q)$ denotes the cardinality of Q. For an alphabet V, V^* represents the free monoid generated by V under the operation of concatenation. The unit of V^* is denoted by ε. Set $V^+ = V^* - \{\varepsilon\}$; algebraically, V^+ is thus the free semigroup generated by V under the operation of concatenation. For $w \in V^*$, $|w|$ and w^R denotes the length of w and the reversal of w, respectively. Let W be an alphabet and ω be a homomorphism from V^* to W^* (see [5] for the definition of homomorphism); ω is a *weak identity* if $\omega(a) \in \{a, \varepsilon\}$ for all $a \in V$.

A *context-free grammar* (CFG for short) is a quadruple $G = (V,T,P,S)$, where V is an alphabet, $T \subseteq V$, $P \subseteq (V - T) \times V^*$ is finite, and $S \in V - T$. Set $N = V - T$. The components V,T,N,P, and S are referred to as the total alphabet, the terminal alphabet, the nonterminal alphabet, the set of rules, and the start symbol of G, respectively. Instead of $(A,x) \in P$, we write $A \to x \in P$ throughout. For brevity, we often denote $A \to x$ by a unique label p as $p : A \to x$, and we briefly use p instead of $A \to x$ under this denotation. For every $p : A \to x \in P$, the *left-hand side of* p is defined as $lhs(p) = A$. The grammar G is *propagating* if $A \to x \in P$ implies $x \in V^+$. The grammar G is *linear* if no more than one nonterminal appears on the right-hand side of any rule in P. Furthermore, a linear grammar G is *minimal* (see page 76 in [8]) if $N = \{S\}$ and $S \to \# \in P$, $\# \in T$, is the only rule with no nonterminal on the right-hand side, whereas it is assumed that # does not occur in any other rule. In this paper, a minimal linear grammar G is called a *palindromial grammar* if $card(P) \geq 2$, and every rule of the form $S \to xSy$, where $x,y \in T^*$, satisfies $x = y$ and $x,y \in T$. For instance, $H = (\{S,0,1,\#\}, \{0,1,\#\}, \{S \to 0S0, S \to 1S1, S \to \#\}, S)$ is a palindromial grammar.

For every $u,v \in V^*$ and $p : A \to x \in P$, write $uAv \Rightarrow uxv[p]$ or, simply, $uAv \Rightarrow uxv$; $\Rightarrow$ is called the *direct derivation* relation over V^*. For $n \geq 0$, $\Rightarrow^n$ denotes the n-th power of $\Rightarrow$. Furthermore, $\Rightarrow^+$ and $\Rightarrow^*$ denote the transitive closure and the transitive-reflexive closure of $\Rightarrow$, respectively. Let $\phi(G) = \{w \in V^* \mid S \Rightarrow^* w\}$ denotes the set of all *sentential forms* of G. The language of G is denoted by $L(G)$ and defined as $L(G) = T^* \cap \phi(G)$. For example, $L(H) = \{w\#w^R \mid w \in \{0,1\}^*\}$, where H is defined as above.

Let $G = (V,T,P,S)$ be a CFG and $W \subseteq V$. Define the weak identity ${}_W\omega$ from V^* to W^* as ${}_W\omega(X) = X$ for all $X \in W$, and ${}_W\omega(X) = \varepsilon$ for all $X \in V - W$. Let $F \subseteq W^*$. Set

$$\phi(G,F) = \{x \mid x \in \phi(G), {}_W\omega(x) \in F\}$$
$$L(G,F) = \{{}_T\omega(y) \mid y \in \phi(G,F), {}_{(N-W)}\omega(y) = \varepsilon\}.$$

$\phi(G,F)$ and $L(G,F)$ are referred to as the set of *sentential forms of G finalized by F* and the *language of G finalized by F*, respectively. Members of $\phi(G,F)$ are called *final sentential forms*. **REG**, **PAL**, **LIN**, **CF**, and **RE** denote the families of regular, palindromial, linear, context-free, and recursively enumerable languages, respectively. Observe that

$$\mathbf{REG} \cap \mathbf{PAL} = \emptyset \text{ and } \mathbf{REG} \cup \mathbf{PAL} \subset \mathbf{LIN}.$$

Set

$$\mathbf{CF_{PAL}} = \{L(G,F) \mid G \text{ is a CFG}, F \in \mathbf{PAL}\}$$
$$\mathbf{CF_{REG}} = \{L(G,F) \mid G \text{ is a CFG}, F \in \mathbf{REG}\}$$

Example 2. Set $I = \{i(x) \mid x \in \{0,1\}^+\}$, where $i(x)$ denotes the integer represented by x in the standard way; for instance, $i(011) = 3$. Consider

$$L = \{u\#v \mid u,v \in \{0,1\}^+, i(u) > i(v) \text{ and } |u| = |v|\}.$$

Next, we define a CFG G and $F \in \mathbf{PAL}$ such that $L = L(G,F)$. Let $G = (V,T,P,S)$ be a context-free grammar. Set $V = \{S,X,\overline{X},Y,\overline{Y},A,B,C,D,0,1,\#\}$, $T = \{0,1,\#\}$, and set P as the following rules

- $S \to X\#\overline{X}$,
- $X \to 1AX, X \to 0BX, X \to 1CY, X \to 1C$,
- $\overline{X} \to 1\overline{X}A, \overline{X} \to 0\overline{X}B, \overline{X} \to 0\overline{Y}C, \overline{X} \to 0C$,
- $Y \to \alpha DY, Y \to \alpha D, \overline{Y} \to \alpha\overline{Y}D, \overline{Y} \to \alpha D$ for all $\alpha \in \{0,1\}$.

Set $W = \{A,B,C,D,\#\}$ and $F = \{w\#w^R \mid w \in \{A,B,C,D\}^+ \text{ and } n \geq 1\}$. Observe that $F = L(H)$, where $H = (\{S,A,B,C,D,\#\},\{A,B,C,D,\#\},\{S \to ASA, S \to BSB, S \to CSC, S \to DSD, S \to \#\},S)$ is a palindromial grammar. Therefore, $F \in \mathbf{PAL}$. For instance, take this step-by-step derivation

$$\begin{aligned} S &\Rightarrow X\#\overline{X} \Rightarrow 1AX\#\overline{X} \Rightarrow 1A0BX\#\overline{X} \Rightarrow 1A0B1CY\#\overline{X} \Rightarrow 1A0B1C0D\#\overline{X} \\ &\Rightarrow 1A0B1C0D\#1\overline{X}A \Rightarrow 1A0B1C0D\#10\overline{X}BA \Rightarrow 1A0B1C0D\#100\overline{Y}CBA \\ &\Rightarrow 1A0B1C0D\#1001\overline{Y}DCBA \Rightarrow 1A0B1C0D\#1001DCBA \end{aligned}$$

in G. Notice that ${}_W\omega(1A0B1C0D\#1001DCBA) \in F$, and ${}_T\omega(1A0B1C0D\#1001DCBA) \in L(G,F)$. The reader is encouraged to verify that $L = L(G,F)$.

A *queue grammar* (see [2]) is a sextuple, $Q = (V,T,U,D,s,P)$, where V and U are alphabets satisfying $V \cap U = \emptyset$, $T \subseteq V$, $D \subseteq U$, $s \in (V - T)(U - D)$, and $P \subseteq (V \times (U - D)) \times (V^* \times U)$ is a finite relation such that for for every $a \in V$, there exists an element $(a,b,z,c) \in P$. If $u,v \in V^*U$ such that $u = arb; v = rzc; a \in V; r,z \in V^*; b,c \in U$; and $(a,b,z,c) \in P$, then $u \Rightarrow v$ $[(a,b,z,c)]$ in G or, simply, $u \Rightarrow v$. In the standard manner, extend $\Rightarrow$ to $\Rightarrow^n$, where $n \geq 0$; then, based on $\Rightarrow^n$, define $\Rightarrow^+$ and $\Rightarrow^*$.

The language of Q, $L(Q)$, is defined as $L(Q) = \{w \in T^* \mid s \Rightarrow^* wf \text{ where } f \in D\}$. A *left-extended queue grammar* is a sextuple, $Q = (V,T,U,D,s,P)$, where V,T,U,D, and s have the same meaning as in a queue grammar, and $P \subseteq (V \times (U-D)) \times (V^* \times U)$ is a finite relation (as opposed to an ordinary queue grammar, this definition does not require that for every $a \in V$, there exists an element $(a,b,z,c) \in P$). Furthermore, assume that $\# \notin V \cup U$. If $u,v \in V^*\{\#\}V^*U$ so that $u = w\#arb$; $v = wa\#rzc$; $a \in V$; $r,z,w \in V^*$; $b,c \in U$; and $(a,b,z,c) \in P$, then $u \Rightarrow v[(a,b,z,c)]$ in G or, simply $u \Rightarrow v$. In the standard manner, extend $\Rightarrow$ to $\Rightarrow^n$, where $n \geq 0$; then, based on $\Rightarrow^n$, define $\Rightarrow^+$ and $\Rightarrow^*$. The language of Q, $L(Q)$, is defined as $L(Q) = \{v \in T^* \mid \#s \Rightarrow^* w\#vf \text{ for some } w \in V^* \text{ and } f \in D\}$. Less formally, during every step of a derivation, a left-extended queue grammar shifts the rewritten symbol over #; in this way, it records the derivation history, which plays a crucial role in the proof of Lemma 5 in the next section.

A *deterministic finite automaton* (DFA for short) is a quintuple $M = (Q,\Sigma,R,s,F)$, where Q is a finite *set of states*, Σ is an *alphabet of input symbols*, $Q \cap \Sigma = \emptyset$, $s \in Q$ is a special state called the *start state*, $F \subseteq Q$ is a *set of final states* in M, and R is a total function from $Q \times \Sigma$ to Q. Instead of $R(q,a) = p$, we write $qa \to p$, where $q,p \in Q$ and $a \in \Sigma \cup \{\varepsilon\}$; R is referred to as the *set of rules* in M. For any $x \in \Sigma^*$ and $qa \to p \in R$, we write $qax \Rightarrow px$. The *language of* M, $L(M)$, is defined as $L(M) = \{w \mid w \in \Sigma^*, sw \Rightarrow^* f, f \in F\}$, where $\Rightarrow^*$ denotes the reflexive-transitive closure of $\Rightarrow$. Recall that DFAs characterize **REG** (see page 29 in [5]).

3 Results

In this section, we show that every language generated by a context-free grammar finalized by a regular language is context-free (see Theorem 2). On the other hand, we prove that every recursively enumerable language can be generated by a propagating context-free grammar finalized by $\{w\#w^R \mid w \in \{0,1\}^*\}$ (see Theorem 9).

Lemma 1. Let $G = (V,T,P,S)$ be any CFG and $F \in$ **REG**. Then, $L(G,F) \in$ **CF**.

Proof. Let $G = (V,T,P,S)$ be any CFG and $F \in$ **REG**. Let $F = L(M)$, where $M = (Q,W,R,q_s,Q_F)$ is a DFA.

Construction. Introduce $U = \{\langle paq\rangle \mid p,q \in Q,\ a \in V\} \cup \{\langle q_sSQ_F\rangle\}$. From G and M, construct a new CFG H such that $L(H) = L(G,F)$ in the following way. Set

$$H = (\overline{V},T,\overline{P},\langle q_sSQ_F\rangle)$$

The components of H are constructed as follows. Set $\overline{V} = V \cup U$ and initialize $\overline{P}$ to $\emptyset$. Construct $\overline{P}$ as follows:

(0) Add $\langle q_sSQ_F\rangle \to \langle q_sSq_f\rangle$ for all $q_f \in Q_F$.

(1) Let $A \to y_0X_1y_1X_2 \ldots X_ny_n \in P$, where $A \in V-T, y_i \in (V-W)^*$ and $X_j \in V$, $0 \leq i \leq n$, $1 \leq j \leq n$, for some $n \geq 1$;
then, add $\langle q_1Aq_{n+1}\rangle \to y_0\langle q_1X_1q_2\rangle y_1\langle q_2X_2q_3\rangle \ldots \langle q_nX_nq_{n+1}\rangle y_n$ to $\overline{P}$, for all $q_1,q_2,\ldots,q_{n+1} \in Q$.

(2) Let $A \to \alpha \in P$, where $A \in V-(T \cup W), \alpha \in (V-W)^*$;
then, add $A \to \alpha$ to $\overline{P}$.

(3) Let $\langle paq\rangle \in U$, where $a \in W \cap T, pa \to q \in R$;
then, add $\langle paq\rangle \to a$ to $\overline{P}$.

(4) Let $\langle pBq\rangle \in U$, where $pB \to q \in R, B \in W \cap (V-T)$;
then, add $\langle pBq\rangle \to \varepsilon$ to $\overline{P}$.

To prove $L(G,F) = L(H)$, we first prove $L(H) \subseteq L(G,F)$; then, we establish $L(G,F) \subseteq L(H)$. To demonstrate $L(H) \subseteq L(G,F)$, we first make three observations—(i) through (iii)—concerning every derivation of the form $\langle q_s S q_f\rangle \Rightarrow^* y$ with $y \in T^*$.

(i) By using rules constructed in (1) and (2), H makes a derivation of the form

$$\langle q_s S q_f\rangle \Rightarrow^* x_0\langle q_1 Z_1 q_2\rangle x_1 \ldots \langle q_n Z_n q_{n+1}\rangle x_n$$

where $x_i \in (T-W)^*$, $0 \leq i \leq n$, $\langle q_j Z_j q_{j+1}\rangle \in U$, $Z_j \in W$, $1 \leq j \leq n$, $q_1 = q_s$, $q_{n+1} = q_f$, $q_1, \ldots, q_{n+1} \in Q$, $q_f \in Q_F$.

(ii) If

$$\langle q_s S q_f\rangle \Rightarrow^* x_0\langle q_1 Z_1 q_2\rangle x_1 \ldots \langle q_n Z_n q_{n+1}\rangle x_n$$

in H, then

$$S \Rightarrow^* x_0 Z_1 x_1 \ldots Z_n x_n$$

in G, where all the symbols have the same meaning as in (i).

(iii) Let H make

$$x_0\langle q_1 Z_1 q_2\rangle x_1 \ldots \langle q_n Z_n q_{n+1}\rangle x_n \Rightarrow^* y$$

by using rules constructed in (3) and (4), where $y \in T^*$, and all the other symbols have the same meaning as in (i). Then, for all $1 \leq j \leq n, q_j Z_j \to q_{j+1} \in R, y = x_0 U_1 x_1 \ldots U_n x_n$, where $U_j = {}_T\omega(Z_j)$. As $q_j Z_j \to q_{j+1} \in R$, $1 \leq j \leq n$, $q_1 = q_s$ and $q_{n+1} = q_f$, $q_f \in Q_F$, we have $Z_1 \ldots Z_n \in L(M)$.

Based on (i) through (iii), we are now ready to prove $L(H) \subseteq L(G,F)$. Let $y \in L(H)$. Thus, $\langle q_s S Q_F\rangle \Rightarrow^* y$, $y \in T^*$ in H. As H is an ordinary CFG, we can always rearrange the applications of rules during $\langle q_s S Q_F\rangle \Rightarrow^* y$ in such a way that

$$\begin{array}{lll} \langle q_s S Q_F\rangle & \Rightarrow & \langle q_s S q_f\rangle \qquad (\alpha) \\ & \Rightarrow^* & x_0\langle q_1 Z_1 q_2\rangle x_1 \ldots \langle q_m Z_m q_{m+1}\rangle x_m \qquad (\beta) \\ & \Rightarrow^* & y \qquad (\gamma) \end{array}$$

so that during (α), only a rule from (0) is used, during β only rules from (1) and (2) are used, and during (γ) only rules from (3) and (4) are used. Recall that $Z_1 Z_2 \ldots Z_n \in F$ (see (iii)). Consequently, ${}_W\omega(x_0 Z_1 x_1 \ldots Z_n x_n) \in F$. From (3), (4), (ii), and (iii), it follows that

$$S \Rightarrow^* x_0 Z_1 x_1 \ldots x_{n-1} Z_n x_n \text{ in } G.$$

Thus, as $L(M) = F$, we have $y \in L(G,F)$, so $L(H) \subseteq L(G,F)$.

To prove $L(G,F) \subseteq L(H)$, take any $y \in L(G,F)$. Thus,

$$S \Rightarrow^* x_0Z_1x_1 \dots x_{n-1}Z_nx_n \text{ in } G, \text{ and}$$
$$y = {}_T\omega(x_0Z_1x_1 \dots x_{n-1}Z_nx_n) \text{ with } Z_1 \dots Z_n \in F$$

where $x_i \in (T-W)^*, 0 \leq i \leq n, Z_j \in W, 1 \leq j \leq n$. As $Z_1 \dots Z_n \in F$, we have $q_1Z_1 \to q_2$, ..., $q_nZ_n \to q_{n+1} \in R$, $q_1, \dots, q_{n+1} \in Q$, $q_1 = q_s$, $q_{n+1} = q_f$, $q_f \in Q_F$. Consequently, from (0) through (4) of the Construction, we see that

$$\begin{aligned} \langle q_sSQ_f \rangle &\Rightarrow \langle q_sSq_f \rangle \\ &\Rightarrow^* x_0Z_1x_1 \dots Z_nx_n \\ &\Rightarrow^* x_0U_1x_1 \dots U_nx_n \end{aligned}$$

where $U_j = {}_T\omega(Z_j)$, $1 \leq j \leq n$. Hence, $y \in L(H)$, so $L(G,F) \subseteq L(H)$.

Thus, $L(G,F) = L(H)$. □

Theorem 2. $\mathbf{CF_{REG}} = \mathbf{CF}$.

Proof. Clearly, $\mathbf{CF} \subseteq \mathbf{CF_{REG}}$. From Lemma 1, $\mathbf{CF_{REG}} \subseteq \mathbf{CF}$. Thus, Theorem 2 holds true. □

Now, we prove that by using the constant palindromial language $\{w\#w^R \mid w \in \{0,1\}^*\}$ to finalize a propagating context-free grammar, we can represent any recursively enumerable language.

Lemma 3. Let $L \in \mathbf{RE}$. Then, there exists a left-extended queue grammar Q satisfying $L(Q) = L$.

Proof. See *Lemma* 1 in [3]. □

Lemma 4. Let H be a left-extended queue grammar. Then, there exists a left-extended queue grammar, $Q = (V,T,U,D,s,R)$, such that $L(H) = L(Q)$ and every $(a,b,x,c) \in R$ satisfies $a \in V-T$, $b \in U-D$, $x \in (V-T)^* \cup T^*$, and $c \in U$.

Proof. See *Lemma* 2 in [3]. □

Lemma 5. Let $Q = (V,T,U,D,s,R)$ be a left-extended queue grammar. Then, $L(Q) = L(G, \{w\#w^R \mid w \in \{0,1\}^*\})$, where G is a CFG.

Proof. Without any loss of generality, assume that Q satisfies the properties described in Lemma 4 and that $\{0,1\} \cap (V \cup U) = \emptyset$. For some positive integer, n, define an injection, ι, from Ψ^* to $(\{0,1\}^n - 1^n)$, where $\Psi = \{ab \mid (a,b,x,c) \in R,\ a \in V-T,\ b \in U-D,\ x \in (V-T)^* \cup T^*,\ c \in U\}$ so that ι is an injective homomorphism when its domain is extended to Ψ^*; after this extension, ι thus represents an injective homomorphism from Ψ^* to $(\{0,1\}^n - 1^n)^*$ (a proof that such an injection necessarily exists is simple and left to the reader). Based on ι, define the substitution, ν from V to $(\{0,1\}^n - 1^n)$ as $\nu(a) = \{\iota(aq) \mid q \in U\}$ for every $a \in V$. Extend domain of ν to V^*. Furthermore, define the substitution, μ, from U to $(\{0,1\}^n - 1^n)$ as $\mu(q) = \{\iota(aq)^R \mid a \in V\}$ for every $q \in U$. Extend the domain of μ to U^*. Set $J = \{\langle p,i \rangle \mid p \in U-D \text{ and } i \in \{1,2\}\}$.

Construction. Next, we introduce a CFG G so that $L(Q) = L(G, \{w\#w^R \mid w \in \{0,1\}^*\})$. Let $G = (\overline{V},T,P,S)$, where $\overline{V} = J \cup \{0,1,\#\} \cup T$. Construct P in the following way. Initially, set $P = \emptyset$; then, perform the following steps 1 through 5.

1. if $(a,q,y,p) \in R$, where $a \in V-T$, $p,q \in U-D$, $y \in (V-T)^*$ and $aq = s$,
   then add $S \to u\langle p,1 \rangle v$ to P, for all $u \in \nu(y)$ and $v \in \mu(p)$;

2. if $(a,q,y,p) \in R$, where $a \in V-T$, $p,q \in U-D$ and $y \in (V-T)^*$, then add $\langle q,1\rangle \to u\langle p,1\rangle v$ to P, for all $u \in \nu(y)$ and $v \in \mu(p)$;
3. for every $q \in U-D$, add $\langle q,1\rangle \to \langle q,2\rangle$ to P;
4. if $(a,q,y,p) \in R$, where $a \in V-T$, $p,q \in U-D$, $y \in T^*$, then add $\langle q,2\rangle \to y\langle p,2\rangle v$ to P, for all $v \in \mu(p)$;
5. if $(a,q,y,p) \in R$, where $a \in V-T$, $q \in U-D$, $y \in T^*$, and $p \in D$, then add $\langle q,2\rangle \to y\#$ to P.

Set $W = \{0,1,\#\}$ and $\Omega = \{xy\#z \in \phi(G) \,|\, x \in \{0,1\}^+, y \in T^*, z = x^R\}$.

Claim 6. Every $h \in \Omega$ is generated by G in this way

$$\begin{aligned}
& S \\
\Rightarrow\ & g_1\langle q_1,1\rangle t_1 \Rightarrow g_2\langle q_2,1\rangle t_2 \Rightarrow \cdots \Rightarrow g_k\langle q_k,1\rangle t_k \Rightarrow g_k\langle q_k,2\rangle t_k \\
\Rightarrow\ & g_k y_1\langle q_{k+1},2\rangle t_{k+1} \Rightarrow g_k y_1 y_2\langle q_{k+2},2\rangle t_{k+2} \Rightarrow \cdots \Rightarrow g_k y_1 y_2 \ldots y_{m-1}\langle q_{k+m-1},2\rangle t_{k+m-1} \\
\Rightarrow\ & g_k y_1 y_2 \ldots y_{m-1} y_m \# t_{k+m}
\end{aligned}$$

in G, where $k,m \geq 1$; $q_1,\ldots,q_{k+m-1} \in U-D$; $y_1,\ldots,y_m \in T^*$; $t_i \in \mu(q_i \ldots q_1)$ for $i = 1,\ldots,k+m$; $g_j \in \nu(d_1 \ldots d_j)$ with $d_1,\ldots,d_j \in (V-T)^*$ for $j = 1,\ldots,k$; $d_1 \ldots d_k = a_1 \ldots a_{k+m}$ with $a_1, \ldots, a_{k+m} \in V-T$ (that is, $g_k \in \nu(a_1 \ldots a_{k+m})$ with $g_k = (t_{k+m})^R$); $h = y_1 y_2 \ldots y_{m-1} y_m$.

Proof. Examine the construction of P. Observe that every derivation begins with an application of a rule having S on its left-hand side. Set 1-$J = \{\langle p,1\rangle \,|\, p \in U\}$, 2-$J = \{\langle p,2\rangle \,|\, p \in U\}$, 1-$P = \{p \,|\, p \in P$ and $lhs(p) \in$ 1-$J\}$, 2-$P = \{p \,|\, p \in P$ and $lhs(p) \in$ 2-$J\}$. Observe that in every successful derivation of h, all applications of rules from 1-P precede the applications of rules from 2-P. Thus, the generation of h can be expressed as

$$\begin{aligned}
& S \\
\Rightarrow\ & g_1\langle q_1,1\rangle t_1 \Rightarrow g_2\langle q_2,1\rangle t_2 \Rightarrow \cdots \Rightarrow g_k\langle q_k,1\rangle t_k \Rightarrow g_k\langle q_k,2\rangle t_k \\
\Rightarrow\ & g_k y_1\langle q_{k+1},2\rangle t_{k+1} \Rightarrow g_k y_1 y_2\langle q_{k+2},2\rangle t_{k+2} \Rightarrow \cdots \Rightarrow g_k y_1 y_2 \ldots y_{m-1}\langle q_{k+m-1},2\rangle t_{k+m-1} \\
\Rightarrow\ & g_k y_1 y_2 \ldots y_{m-1} y_m \# t_{k+m}
\end{aligned}$$

where all the involved symbols have the meaning stated in Claim 6. □

Claim 7. Every $h \in L(Q)$ is generated by Q in this way

$$\begin{aligned}
& \#a_0 q_0 && \\
\Rightarrow\ & a_0 \# x_0 q_1 && [(a_0,q_0,z_0,q_1)] \\
\Rightarrow\ & a_0 a_1 \# x_1 q_2 && [(a_1,q_1,z_1,q_2)] \\
& \vdots && \\
\Rightarrow\ & a_0 a_1 \ldots a_k \# x_k q_{k+1} && [(a_k,q_k,z_k,q_{k+1})] \\
\Rightarrow\ & a_0 a_1 \ldots a_k a_{k+1} \# x_{k+1} q_{k+2} && [(a_{k+1},q_{k+1},y_1,q_{k+2})] \\
& \vdots && \\
\Rightarrow\ & a_0 a_1 \ldots a_k a_{k+1} \ldots a_{k+m-1} \# x_{k+m-1} y_1 \ldots y_{m-1} q_{k+m} && [(a_{k+m-1},q_{k+m-1},y_{m-1},q_{k+m})] \\
\Rightarrow\ & a_0 a_1 \ldots a_k a_{k+1} \ldots a_{k+m} \# y_1 \ldots y_m q_{k+m+1} && [(a_{k+m},q_{k+m},y_m,q_{k+m+1})]
\end{aligned}$$

where $k,m \geq 1$, $a_i \in V-T$ for $i = 0,\ldots,k+m$, $x_j \in (V-T)^*$ for $j = 1,\ldots,k+m$, $s = a_0 q_0$, $a_j x_j = x_{j-1} z_j$ for $j = 1,\ldots,k$, $a_1 \ldots a_k x_{k+1} = z_0 \ldots z_k$, $a_{k+1} \ldots a_{k+m} = x_k$, $q_0, q_1, \ldots, q_{k+m} \in U-D$ and $q_{k+m+1} \in D$, $z_1,\ldots,z_k \in (V-T)^*$, $y_1,\ldots,y_m \in T^*$, $h = y_1 y_2 \ldots y_{m-1} y_m$.

Proof. Recall that Q satisfies the properties given in Lemma 4. These properties imply that Claim 7 holds true. □

Claim 8. $L(G,\{w\#w^R \mid w \in \{0,1\}^*\}) = L(Q)$.

Proof. To prove that $L(G,\{w\#w^R \mid w \in \{0,1\}^*\}) \subseteq L(Q)$, take any $h \in \Omega$ generated in the way described in Claim 6. From ${}_W\omega(h) \in \{w\#w^R \mid w \in \{0,1\}^*\}$ with $W = \{0,1,\#\}$, it follows that $xy\#z$ with $z = x^R$ where $x = g_k$, $y = y_1 \ldots y_m$, $z = t_{k+m}$. At this point, R contains (a_0, q_0, z_0, q_1), …, (a_k, q_k, z_k, q_{k+1}), $(a_{k+1}, q_{k+1}, y_1, q_{k+2})$, …, $(a_{k+m-1}, q_{k+m-1}, y_{m-1}, q_{k+m})$, $(a_{k+m}, q_{k+m}, y_m, q_{k+m+1})$, where $z_1, \ldots, z_k \in (V-T)^*$, and $y_1, \ldots, y_m \in T^*$. Then, Q makes the generation of ${}_T\omega(h)$ in the way described in Claim 7. Thus, ${}_T\omega(h) \in L(Q)$.

To prove $L(Q) \subseteq L(G,\{w\#w^R \mid w \in \{0,1\}^*\})$, take any $h \in L(Q)$. Recall that h is generated in the way described in Claim 7. Consider the rules used in this generation. Furthermore, consider the definition of ν and μ. Based on this consideration, observe that from the construction of P, it follows that $S \Rightarrow^* oh\#\overline{o}$ in G for some $o, \overline{o} \in \{0,1\}^+$ with $\overline{o} = o^R$. Thus, ${}_W\omega(oh\#\overline{o}) \in \{w\#w^R \mid w \in \{0,1\}^*\}$, so consequently, $h \in L(G,\{w\#w^R \mid w \in \{0,1\}^*\})$. □

Claims 6 through 8 imply that Lemma 5 holds true.

Theorem 9. A language $L \in$ **RE** if and only if $L = L(G,\{w\#w^R \mid w \in \{0,1\}^*\})$, where G is a propagating CFG.

Proof. This theorem follows from Lemmas 3 through 5.

Corollary 10. $\mathbf{RE} = \mathbf{CF_{PAL}}$.

Consider $\{w\#w^R \mid w \in \{0,1\}^*\}$ without #—that is $\{ww^R \mid w \in \{0,1\}^*\}$. On the one hand, this language is out of $\mathbf{CF_{PAL}}$ because the central symbol # does not occur in it. On the other hand, it is worth pointing out that Theorem 9 can be based upon this purely binary language as well.

Corollary 11. A language $L \in$ **RE** if and only if $L = L(G,\{ww^R \mid w \in \{0,1\}^*\})$, where G is propagating CFG.

Proof. Prove this corollary by analogy with the way Theorem 9 is demonstrated.

Before closing this paper, we point out an open problem. As its main results, the paper has demonstrated that every recursively enumerable language can be generated by a propagating context-free grammar G finalized by $\{w\#w^R \mid w \in \{0,1\}^*\}$ (see Theorem 9). Can this results be established with G having a limited number of nonterminals and/or rules?

Acknowledgement

This work was supported by Brno University of Technology grant FIT-S-23-8209.

References

[1] A. V. Aho, M. S. Lam, R. Sethi & J. D. Ullman (2006): *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2nd edition. Addison-Wesley.

[2] H. C. M. Kleijn & G. Rozenberg (1983): *On the Generative Power of Regular Pattern Grammars*. *Acta Informatica* 20(4), pp. 391–411, doi:10.1007/BF00264281.

[3] A. Meduna (2000): *Generative Power of Three-Nonterminal Scattered Context Grammars*. *Theoretical Computer Science* 246(1–2), pp. 279–284, doi:10.1016/S0304-3975(00)00153-5.

[4] A. Meduna (2008): *Elements of Compiler Design*. Taylor & Francis, doi:10.1201/9781420063257.

[5] A. Meduna (2014): *Formal Languages and Computation*. Taylor & Francis, doi:10.1201/b16376.

[6] A. Meduna & P. Zemek (2014): *Regulated Grammars and Automata*. Springer, doi:10.1007/978-1-4939-0369-6.

[7] G. Rozenberg & A. Salomaa, editors (1997): *Handbook of Formal Languages, Vol. 1: Word, Language, Grammar*. Springer.

[8] A. Salomaa (1973): *Formal Languages*. ACM monograph series, Academic Press.

Deterministic Real-Time Tree-Walking-Storage Automata

Martin Kutrib
Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
kutrib@informatik.uni-giessen.de

Uwe Meyer
Technische Hochschule Mittelhessen
Wiesenstr. 14, 35390 Giessen, Germany
uwe.meyer@mni.thm.de

We study deterministic tree-walking-storage automata, which are finite-state devices equipped with a tree-like storage. These automata are generalized stack automata, where the linear stack storage is replaced by a non-linear tree-like stack. Therefore, tree-walking-storage automata have the ability to explore the interior of the tree storage without altering the contents, with the possible moves of the tree pointer corresponding to those of tree-walking automata. In addition, a tree-walking-storage automaton can append (push) non-existent descendants to a tree node and remove (pop) leaves from the tree. Here we are particularly considering the capacities of deterministic tree-walking-storage automata working in real time. It is shown that even the non-erasing variant can accept rather complicated unary languages as, for example, the language of words whose lengths are powers of two, or the language of words whose lengths are Fibonacci numbers. Comparing the computational capacities with automata from the classical automata hierarchy, we derive that the families of languages accepted by real-time deterministic (non-erasing) tree-walking-storage automata is located between the regular and the deterministic context-sensitive languages. There is a context-free language that is not accepted by any real-time deterministic tree-walking-storage automaton. On the other hand, these devices accept a unary language in non-erasing mode that cannot be accepted by any classical stack automaton, even in erasing mode and arbitrary time. Basic closure properties of the induced families of languages are shown. In particular, we consider Boolean operations (complementation, union, intersection) and AFL operations (union, intersection with regular languages, homomorphism, inverse homomorphism, concatenation, iteration). It turns out that the two families in question have the same properties and, in particular, share all but one of these closure properties with the important family of deterministic context-free languages.

1 Introduction

Stack automata were introduced in [6] as a theoretical model motivated by compiler theory, and the implementation of recursive procedures with parameters. Their computational power lies between that of pushdown automata and Turing machines. Basically, a stack automaton is a finite-state device equipped with a generalization of a pushdown store. In addition to be able to push or pop at the top of the pushdown store, a stack automaton can move its storage head (stack pointer) *inside* the stack to read stack symbols, but without altering the contents. In this way, it is possible to read but not to change the stored information. Over the years, stack automata have aroused great interest and have been studied in different variants. Apart from distinguishing deterministic and nondeterministic computations, the original two-way input reading variant has been restricted to one-way [7]. Further investigated restrictions concern the usage of the stack storage. A stack automaton is said to be *non-erasing* if no symbol may be popped from the stack [13], and it is *checking* if it cannot push any symbols once the stack pointer has moved into the stack [9]. While the early studies of stack automata have extensively been done in relation with AFL theory as well as time and space complexity [11, 14, 15, 22, 25], more recent papers consider the computational power gained in generalizations by allowing the input head to jump [19], allowing multiple input heads, multiple stacks [18], and multiple reversal-bounded counters [17]. The stack size

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on
Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 48–62, doi:10.4204/EPTCS.388.7

required to accept a language by stack automata has been considered as well [16]. In [20] the property of working input-driven has been imposed to stack automata, and their capacities as transducer are studied in [2].

All these models have in common that their storage structures are linear. Therefore, it is a natural idea to generalize stack automata by replacing the stack storage by some non-linear data structure. In [21] tree-walking-storage automata have been introduced, which are essentially stack automata with a tree-like stack. As for classical stack automata, tree-walking-storage automata have the additional ability to move the storage head (here tree pointer) inside the tree without altering the contents. The possible moves of the tree pointer correspond to those of tree walking automata. In this way, it is possible to read but not to change the stored information. In addition, a tree-walking-storage automaton can append (push) a non-existent descendant to a tree node and remove (pop) a leaf from the tree. A main focus in [21] is on the comparisons of the different variants of tree-walking-storage automata as well as on the comparisons with classical stack automata. It turned out that the checking variant is no more powerful than classical checking stack automata. In particular it is shown that in the case of unlimited time deterministic tree-walking-storage automata are as powerful as Turing machines. This result suggested to consider time constraints for deterministic tree-walking-storage automata. The computational capacities of polynomial-time non-erasing tree-walking-storage automata and non-erasing stack automata are separated. Moreover, it is shown that non-erasing tree-walking-storage and tree-walking-storage automata are equally powerful.

Here we continue the study of tree-walking-storage automata by imposing a very strict time limit. We consider the minimal time to solve non-trivial problems, that is, we consider real-time computations. This natural limitation has been investigated from the early beginnings of complexity theory. Already before the seminal paper [12], Rabin considered computations such that if the problem (the input data) consists of n symbols then the computation must be performed in n basic steps, one step per input symbol [24].

Before we turn to our main results and the organization of the paper, we briefly mention different approaches to introduce tree-like stacks. So-called pushdown tree automata [10] extend the usual string pushdown automata by allowing trees instead of strings in both the input and the stack. So, these machines accept trees and may not explore the interior of the stack. Essentially, this model has been adapted to string inputs and tree-stacks where the so-called *tree-stack automaton* can explore the interior of the tree-stack in read-only mode [8]. However, in the writing-mode a new tree can be pushed on the stack employing the subtrees of the old tree-stack, that is, subtrees can be permuted, deleted, or copied. If the root of the tree-stack is popped, exactly one subtree is left in the store. Another model also introduced under the name *tree-stack automaton* gave up the bulky way of pushing and popping at the root of the tree-stack [5]. However, this model may alter the interior nodes of the tree-stack. Therefore, the tree-stack is actually a non-linear Turing tape. Therefore, we have chosen the name *tree-walking-storage automaton*, so as not to have one more model under the name of tree-stack automaton.

The idea of a tree-walking process originates from [1]. A tree-walking automaton is a sequential model that processes input trees. For example, it is known that deterministic tree-walking automata are strictly weaker than nondeterministic ones [3] and that even nondeterministic tree-walking automata cannot accept all regular tree languages [4].

The paper is organized as follows. The definition of the models and an illustrating example are given in Section 2. Section 3 is devoted to compare the computational capacity of real-time deterministic tree-walking-storage automata with some classical types of acceptors. It is shown that the possibility to create tree-storages of certain types in real time can be utilized to accept further, even unary, languages by real-time deterministic, even non-erasing, tree-walking-storage automata. To this end, the non-semilinear

unary language of the words whose lengths are double Fibonacci numbers is used as a witness.

Then, a technique for disproving that languages are accepted is established for real-time tree-walking-storage automata. The technique is based on equivalence classes which are induced by formal languages. If some language induces a number of equivalence classes which exceeds the number of classes distinguishable by a certain device, then the language is not accepted by that device. Applying these results, we show that there is a context-free language which is not accepted by any tree-walking-storage automaton in real time. For the comparison with classical deterministic one-way stack automata we show that the unary language $\{a^{n^3} \mid n \geq 0\}$ is a real-time tree-walking-storage automaton language. It is known from [23] that this language is not accepted by any classical deterministic one-way stack automaton. Finally, in Section 4 some basic closure properties of the language families in question are derived. It turns out that the two families in question have the same properties and, in particular, share all but one of these closure properties with the important family of deterministic context-free languages. In particular, we consider Boolean operations (complementation, union, intersection) and AFL operations (union, intersection with regular languages, homomorphism, inverse homomorphism, concatenation, iteration). The results are summarized in Table 1 at the end of the section.

2 Definitions and Preliminaries

Let Σ^* denote the *set of all words* over the finite alphabet Σ. The *empty word* is denoted by λ, and $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. The set of words of length $n \geq 0$ is denoted by Σ^n. The *reversal* of a word w is denoted by w^R. For the *length* of w we write $|w|$. We use $\subseteq$ for *inclusions* and $\subset$ for *strict inclusions*. We write $|S|$ for the cardinality of a set S. We say that two language families $\mathscr{L}_1$ and $\mathscr{L}_2$ are *incomparable* if $\mathscr{L}_1$ is not a subset of $\mathscr{L}_2$ and vice versa.

A tree-walking-storage automaton is an extension of a classical stack automaton to a tree storage. As for classical stack automata, tree-walking-storage automata have the additional ability to move the storage head (here tree pointer) inside the tree without altering the contents. The possible moves of the tree pointer correspond to those of tree walking automata. In this way, it is possible to read but not to change the stored information. However, a classical stack automaton can push and pop at the top of the stack. Accordingly, a tree-walking-storage automaton can append (push) a non-existent descendant to a tree node and remove (pop) a leaf from the tree.

Here we consider mainly deterministic one-way devices. The trees in this paper are finite, binary trees whose nodes are labeled by a finite alphabet Γ. A Γ-tree T is represented by a mapping from a finite, non-empty, prefix-closed subset of $\{l,r\}^*$ to $\Gamma \cup \{\bot\}$, such that $T(w) = \bot$ if and only if $w = \lambda$. The elements of the domain of T are called *nodes of the tree*. Each node of the tree has a *type* from $\mathsf{TYPE} = \{-,l,r\} \times \{-,+\}^2$, where the first component expresses whether the node *is* the root ($-$), a left descendant (l), or a right descendant (r), and the second and third components tell whether the node *has* a left and right descendant ($+$), or not ($-$). A *direction* is an element from $\mathsf{DIRECT} = \{u,s,d_l,d_r\}$, where u stands for 'up', s stands for 'stay', d_l stands for 'left descendant' and d_r for 'right descendant'.

A *deterministic tree-walking-storage automaton (twsDA)* is a system $M = \langle Q,\Sigma,\Gamma,\delta,q_0,\triangleleft,\bot,F\rangle$, where Q is the finite set of *internal states*, Σ is the finite set of *input symbols* not containing the *end-marker* $\triangleleft$, Γ is the finite set of *tree symbols*, $q_0 \in Q$ is the *initial state*, $\bot \notin \Gamma$ is the *root symbol*, $F \subseteq Q$ is the set of *accepting states*, and

$$\delta\colon Q \times (\Sigma \cup \{\lambda,\triangleleft\}) \times \mathsf{TYPE} \times (\Gamma \cup \{\bot\}) \to Q \times (\mathsf{DIRECT} \cup \{\mathtt{pop}\} \cup \{\mathtt{push}(x,d) \mid x \in \Gamma, d \in \{l,r\}\})$$

is the *transition function*. There must never be a choice of using an input symbol or of using λ input. So, it is required that for all q in Q, $(t_1,t_2,t_3) \in \mathsf{TYPE}$, and x in $\Gamma \cup \{\bot\}$: if $\delta(q,\lambda,(t_1,t_2,t_3),x)$ is defined, then $\delta(q,a,(t_1,t_2,t_3),x)$ is undefined for all a in $\Sigma \cup \{\lhd\}$.

A *configuration* of a twsDA is a quadruple (q,v,T,P), where $q \in Q$ is the current state, $v \in \Sigma^*\{\lhd,\lambda\}$ is the unread part of the input, T is the current Γ-tree, and P is an element of the domain of T, called the *tree pointer*, that is the current node of T. The *initial configuration* for input w is set to $(q_0,w\lhd,T_0,\lambda)$, where $T_0(\lambda) = \bot$ and T_0 is undefined otherwise.

During the course of its computation, M runs through a sequence of configurations. In a given configuration (q,v,T,P), M is in state q, reads the first symbol of v or λ, knows the type of the current node P, and sees the label $T(P)$ of the current node. Then it applies δ and, thus, enters a new state and either moves the tree pointer along a direction, removes the current node (if it is a leaf) by `pop`, or appends a new descendant to the current node (if this descendant does not exist) by `push`. Here and in the sequel it is understood that δ is well defined in the sense that it will never move the tree pointer to a non-existing node, will never pop a non-leaf node, and will never push an existing descendant. This normal form is always available through effective constructions.

One step from a configuration to its successor configuration is denoted by $\vdash$, and the reflexive and transitive (resp., transitive) closure of $\vdash$ is denoted by $\vdash^*$ (respectively $\vdash^+$). Let $q \in Q$, $av \in \Sigma^*\lhd$ with $a \in \Sigma \cup \{\lambda,\lhd\}$, T be a Γ-tree, P be a tree pointer of T, and $(t_1,t_2,t_3) \in \mathsf{TYPE}$ be the type of the current node P. We set

1. $(q,av,T,P) \vdash (q',v,T,P')$ with $P = P'l$ or $P = P'r$,
   if $t_1 \neq -$ and $\delta(q,a,(t_1,t_2,t_3),T(P)) = (q',u)$, (move the tree pointer up),

2. $(q,av,T,P) \vdash (q',v,T,P)$,
   if $\delta(q,a,(t_1,t_2,t_3),T(P)) = (q',s)$, (do not move the tree pointer),

3. $(q,av,T,P) \vdash (q',v,T,P')$ with $P' = Pl$,
   if $t_2 = +$ and $\delta(q,a,(t_1,t_2,t_3),T(P)) = (q',d_l)$, (move the tree pointer to the left descendant),

4. $(q,av,T,P) \vdash (q',v,T,P')$ with $P' = Pr$,
   if $t_3 = +$ and $\delta(q,a,(t_1,t_2,t_3),T(P)) = (q',d_r)$, (move the tree pointer to the right descendant),

5. $(q,av,T,P) \vdash (q',v,T',P')$ with $P = P'l$ or $P = P'r$, $T'(P)$ is undefined and $T'(w) = T(w)$ for $w \neq P$,
   if $t_2 = t_3 = -$ and $\delta(q,a,(t_1,t_2,t_3),T(P)) = (q',\mathtt{pop})$, (remove the current leaf node, whereby the tree pointer is moved up),

6. $(q,av,T,P) \vdash (q',v,T',P')$ with $P' = Pl$, $T'(Pl) = x$ and $T'(w) = T(w)$ for $w \neq Pl$,
   if $t_2 = -$ and $\delta(q,a,(t_1,t_2,t_3),T(P)) = (q',\mathtt{push}(x,l))$, (append a left descendant to the current node, whereby the tree pointer is moved to the descendant),

7. $(q,av,T,P) \vdash (q',v,T',P')$ with $P' = Pr$, $T'(Pr) = x$ and $T'(w) = T(w)$ for $w \neq Pr$,
   if $t_3 = -$ and $\delta(q,a,(t_1,t_2,t_3),T(P)) = (q',\mathtt{push}(x,r))$, (append a right descendant to the current node, whereby the tree pointer is moved to the descendant).

Figure 1 illustrates the transitions that move the tree pointer up, respectively to the left descendant.

Figure 2 illustrates the push, respectively the pop transitions. All remaining transitions are analogous.

So, a classical stack automaton can be seen as a tree-walking-storage automaton all of whose right descendants of the tree-storage are not present. In accordance with stack automata, a twsDA is said to be *non-erasing* (twsDNEA) if it is not allowed to pop from the tree.

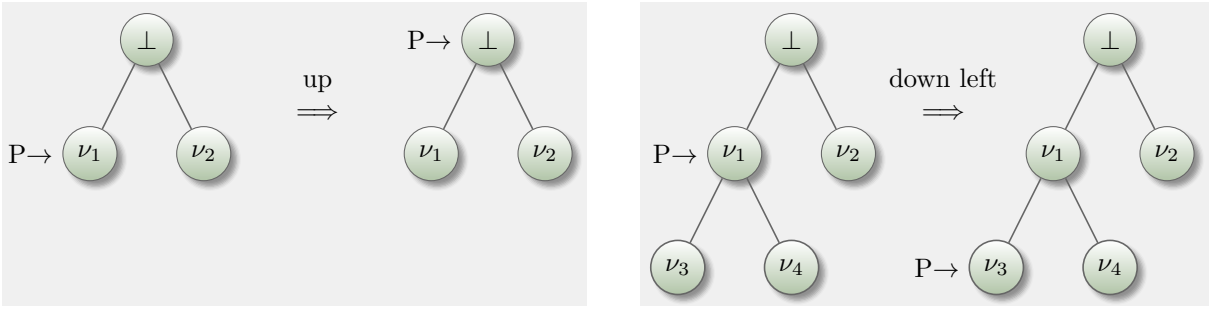


Figure 1: Up and left transitions

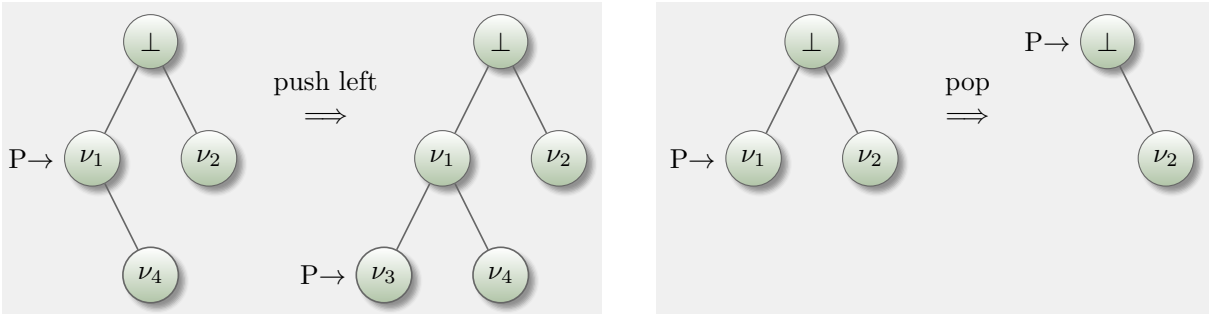


Figure 2: Push left and pop operations

A twsDCA M *halts* if the transition function is not defined for the current configuration. A word w is *accepted* if the machine halts in an accepting state after having read the input $w\triangleleft$ entirely, otherwise it is *rejected*. The *language accepted* by M is $L(M) = \{ w \in \Sigma^* \mid w \text{ is accepted by } M \}$.

A twsDA works in *real time* if its transition function is undefined for λ input. That is, it reads one symbol from the input at every time step, thus, halts on input w after at most $|w|+1$ steps.

We write DSA for deterministic one-way stack automata, DNESA for the non-erasing, and DCSA for the checking variant. The family of languages accepted by a device of type X is denoted by $\mathscr{L}(\mathsf{X})$. We write in particular $\mathscr{L}_{\mathrm{rt}}(\mathsf{X})$ if acceptance has to be in real time.

In order to clarify our notion, we continue with an example.

Example 1. The language $L_{\mathrm{expo}} = \{ a^{2^n} \mid n \geq 0 \}$ is accepted by some twsDNEA in real time.

The basic idea of the construction is to let a twsDNEA successively create tree-storages which are complete binary trees. To this end, we construct a twsDNEA $M = \langle Q, \{a\}, \{\bullet\}, \delta, q_l, \triangleleft, \bot, F \rangle$ with state set $Q = \{q_l, q_p, q_d, q_r\}$ that runs in phases. In each phase a complete level is added to the complete binary tree. So, at the outset of the computation the tree-storage of M forms a complete binary tree of level 1, that is a single node. After the $(\ell-1)$th phase, the tree-storage of M forms a complete binary tree of level ℓ, that is, the tree has $2^\ell - 1$ nodes. At the beginning and at the end of each phase the tree pointer is at the root of the tree-storage. For simplicity, we construct M such that it works on empty input only. Later, it will be extended.

Next, we explain how a level is added when the tree-storage of M forms a complete binary tree of level $\ell \geq 1$ and the tree pointer is at the root.

Let a star $*$ as component of the type of the current node in the tree-walking-storage of M denote an arbitrary entry and $\gamma \in \Gamma \cup \{\bot\}$. We set:

1. $\delta(q_l, \lambda, (*,+,*), \gamma) = (q_l, d_l)$
2. $\delta(q_l, \lambda, (*,-,-), \gamma) = (q_p, \mathtt{push}(\bullet, l))$
3. $\delta(q_p, \lambda, (l,-,-), \gamma) = (q_r, u)$

First, state q_l is used to move the tree pointer as far as possible to the left (Transition 1). The leaf reached is the first node that gets descendants. After pushing a left descendant (Transition 2), M enters state q_p to indicate that the last tree operation was a push. If the new leaf was pushed as left descendant, the tree pointer is moved up while state q_r is entered (Transition 3). State q_r indicates that the right subtree of the current node has still to be processed.

4. $\delta(q_r, \lambda, (*,+,-), \gamma) = (q_p, \mathtt{push}(\bullet, r))$
5. $\delta(q_p, \lambda, (r,-,-), \gamma) = (q_d, u)$

If the current leaf has no right descendant and M is in state q_r, a right descendant is pushed (Transition 4). Again, state q_p is entered. If the new leaf was pushed as right descendant, the tree pointer is moved up while state q_d is entered (Transition 5). State q_d indicates that the current node has been processed entirely.

6. $\delta(q_d, \lambda, (l,*,*), \gamma) = (q_r, u)$
7. $\delta(q_d, \lambda, (r,*,*), \gamma) = (q_d, u)$

In state q_d, the tree pointer is moved to the ancestor. However, if it comes to the ancestor from the left subtree, the right subtree is still to be processed. In this case, Transition 6 sends the tree pointer to the ancestor in state q_r. If the tree pointer comes to the ancestor from the right subtree, the ancestor has entirely be processed and the tree pointer is moved up in the appropriate state q_d (Transition 7).

8. $\delta(q_r, \lambda, (*,+,+), \gamma) = (q_l, d_r)$

If there is a right descendant of the node visited in state q_r then the process is recursively applied to the right subtree by moving the tree pointer to the right descendant in state q_l (Transition 8).

The end of the phase that can uniquely be detected by M when its tree pointer comes back to the root in state q_d from the right.

Before we next turn to the extension of M, we consider the number of steps taken to generate the complete binary trees. The total number of nodes in such a tree of level $\ell \geq 1$ is $2^\ell - 1$. Since all nodes except the root are connected by exactly one edge, the number of edges is $2^\ell - 2$. In order to increase the level of the tree-storage from ℓ to $\ell + 1$, the tree pointer takes a tour through the tree as for a depth-first traversal. So, every edge is moved along twice. In addition, each of the 2^ℓ new nodes is connected whereby for each new node the connecting (new) edge is also moved along twice. In total, we obtain $2(2^\ell - 2 + 2^\ell) = 2^{\ell+2} - 4$ moves to increase the level. Summing up the moves yields the number of moves taken by M to increase the level of the tree-storage from initially 1 to ℓ as

$$\sum_{i=1}^{\ell-1} 2^{i+2} - 4 = -4(\ell - 1) + 2^{\ell+2} - 8 = 2^{\ell+2} - 4\ell - 4.$$

Now, the construction of M is completed as follows. Initially, M performs 8 moves without any operation on the tree-storage. That is, the tree pointer stays at the root. This can be realized by additional

states. Next, M starts to run through the phases described above, where at the end of phase $\ell-1$ the tree-storage forms a complete binary tree of level ℓ. Before each phase, M performs additionally 4 moves without any operation on the tree-storage, respectively.

Finally, it remains to be described how the input is read and possibly accepted. We let M read an input symbol at every move. An input word is accepted if and only if its length is 2^0, 2^1, 2^2, 2^3, or if M reads the last input symbol exactly at the end of some phase.

In order to give evidence that M works correctly, assume that the input length is 2^x, for some $x \geq 4$. Then M starts to generate a tree-storage that forms a complete binary tree of level $x-2$. The generation takes $2^x - 4(x-2) - 4$ moves plus the initial delay of 8 moves plus the delay of totally $4(x-3)$ moves before each phase. Altogether, this makes 2^x moves. Since M reads one input symbol at every move, it reads exactly 2^x symbols and works in real time. ■

3 Computational Capacity

This section is devoted to compare the computational capacity of real-time deterministic tree-walking-storage automata with some classical types of acceptors. On the bottom of the automata hierarchy there are finite state automata characterizing the family of regular languages REG. Trivially, we have the inclusion $\mathrm{REG} \subset \mathscr{L}_{\mathrm{rt}}(\mathsf{twsDNEA})$ whose properness follows from Example 1.

On the other end, we consider the deterministic linear bounded automata that are characterizing the family of deterministic context-sensitive languages DCSL, that is, the complexity class $\mathsf{DSPACE}(n)$. In a real-time computation of some twsDA, the tree-storage can grow not beyond $n+1$ nodes, where n is the length of the input. Since a binary tree with n nodes can be encoded with $O(n)$ bits, the tree-storage can be simulated in deterministic space n. Therefore, a real-time twsDA can be simulated by a deterministic linear bounded automaton and we obtain the inclusion $\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDA}) \subseteq \mathrm{DCSL}$.

We continue the investigation by showing that the possibility to create tree-storages of certain types in real time can be utilized to accept further, even unary, languages by real-time deterministic, even non-erasing, tree-walking-storage automata. To this end, we make the construction of Example 1 more involved and consider the non-semilinear unary language of the words whose lengths are double Fibonacci numbers.

The Fibonacci numbers form a sequence in which each number is the sum of the two preceding ones. The sequence starts from 1 and 1 (sometimes in the literature it starts from 0 and 1). A prefix of the sequence is $1,1,2,3,5,8,13,21,34,55,89$. Correspondingly, we are speaking of the ith Fibonacci number f_i, where i is the position in the sequence starting from 1. So, for example, f_6 is the number 8. We are going to prove that the language $L_{\mathrm{fib}} = \{\, a^{2n} \mid n \text{ is a Fibonacci number} \,\}$ is accepted by some twsDNEA in real time by showing that a twsDNEA can successively create tree-storages that are Fibonacci trees. *Fibonacci trees* are recursively defined as follows. The Fibonacci tree F_0 of level 0 is the empty tree. The Fibonacci tree F_1 of level 1 is the tree that consists of one node only. The Fibonacci tree F_ℓ of level $\ell \geq 2$ consists of the root whose left subtree is a Fibonacci tree of level $\ell-1$ and whose right subtree is a Fibonacci tree of level $\ell-2$ (see Figure 3). For our purposes, the number of nodes of a Fibonacci tree is important. It is well known that the number of nodes of Fibonacci tree F_ℓ, for $\ell \geq 2$, is $\nu_\ell = \nu_{\ell-1} + \nu_{\ell-2} + 1$. In other words, we obtain $\nu_\ell = f_{\ell+2} - 1$.

Theorem 2. *The language L_{fib} is accepted by some twsDNEA in real time.*

Proof. We proceed as in Example 1 and construct a twsDNEA $M = \langle Q, \{a\}, \{\bullet\}, \delta, q_l, \lhd, \bot, F \rangle$ with state set $Q = \{q_l, q_p, q_d, q_r\}$ that runs in phases. Again, at the outset of the computation the tree-storage of M

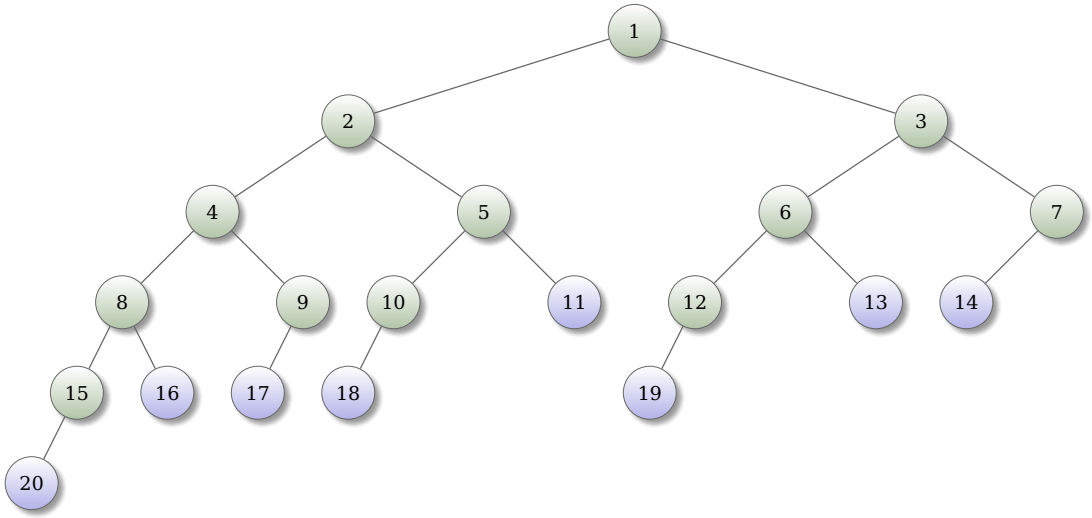


Figure 3: A Fibonacci tree of level 6. Removing the blue nodes (the leaves) yields a Fibonacci tree of level 5.

forms a Fibonacci tree of level 1. After the $(\ell-1)$th phase, the tree-storage of M forms a Fibonacci tree of level ℓ. At the beginning and at the end of each phase the tree pointer is at the root of the tree-storage. Again, we first construct M such that it works on empty input and extend it later.

So, assume that the tree-storage of M forms a Fibonacci tree of level $\ell \geq 1$ and that its tree pointer is at the root. According to the recursive definition of Fibonacci trees, M will increase the levels of the subtrees of every node by one in a bottom-up fashion. To this end, first state q_l is used to move the tree pointer as far as possible to the left. The leaf reached is the first node to be dealt with. In particular, this leaf gets a left descendant. See Figure 3 for an example, where the Fibonacci tree of level 5 depicted by the green nodes is extended to the entire Fibonacci tree of level 6 by adding the blue nodes.

Let a star $*$ as component of the type of the current node in the tree-walking-storage of M denote an arbitrary entry and $\gamma \in \Gamma \cup \{\bot\}$. We set:

1. $\delta(q_l, \lambda, (*,+,*), \gamma) = (q_l, d_l)$
2. $\delta(q_l, \lambda, (*,-,-), \gamma) = (q_p, \texttt{push}(\bullet, l))$
3. $\delta(q_p, \lambda, (*,*,*), \gamma) = (q_d, u)$

Essentially, the meaning of the states are as in Example 1. State q_p indicates that the last tree operation was a push, and the meaning of state q_d is to indicate that the current node has entirely be processed and that its ancestor is the next node to consider. So far, in Figure 3 node 20 has been pushed and the tree pointer is back at node 15 in state q_d.

4. $\delta(q_d, \lambda, (l,*,*), \gamma) = (q_r, u)$
5. $\delta(q_d, \lambda, (r,*,*), \gamma) = (q_d, u)$

Node 15 has entirely be processed, since it got a new left subtree of level 1 and, thus, stick with a right subtree of level 0 (the empty tree). By Transitions 4 and 5 the tree pointer is moved to the ancestor. However, if it comes to the ancestor from the left subtree, the right subtree is still to be processed. In this case, Transition 4 sends the tree pointer to the ancestor in state q_r. If the tree pointer comes to the

ancestor from the right subtree, the ancestor has entirely be processed and the tree pointer is moved up in the appropriate state q_d (Transition 5).

6. $\delta(q_r, \lambda, (*,+,-), \gamma) = (q_p, \mathtt{push}(\bullet, r))$
7. $\delta(q_r, \lambda, (*,+,+), \gamma) = (q_l, d_r)$

If there is a right descendant of the node visited in state q_r then the process is recursively applied to the right subtree by moving the tree pointer to the right descendant in state q_l (Transition 7). Otherwise, if there is no right descendant of the node visited in state q_r then this empty right subtree has to be replaced by a subtree of level 1. This is simply done by pushing a single node (Transition 6). In Figure 3, node 16 has been pushed as right descendant of node 8. Then, after the next few steps, node 8 has entirely processed and node 4 is reached in state q_r. Continuing, this process will end when node 3 has entirely been processed and the root is reached from the right subtree in state q_d. This is the end of the phase that can uniquely be detected by M when its tree pointer comes back to the root from the right.

Before we next turn to the extension of M, we consider the number of steps taken to generate the Fibonacci tree.

To this end, let $\ell \geq 1$ and recall that the number of nodes of Fibonacci tree F_ℓ is $\nu_\ell = f_{\ell+2} - 1$. Since all nodes except the root are connected by exactly one edge, the number of edges of Fibonacci tree F_ℓ is $\kappa_\ell = f_{\ell+2} - 2$. We derive that the number of nodes of $F_{\ell+1}$ is $\nu_{\ell+1} = f_{\ell+3} - 1 = f_{\ell+2} - 1 + f_{\ell+1}$ and the number of its edges is $\kappa_{\ell+1} = f_{\ell+2} - 2 + f_{\ell+1}$. In order to increase the level of the tree-storage from ℓ to $\ell+1$, the tree pointer takes a tour through the tree as for a depth-first traversal. So, every edge of F_ℓ is moved along twice. In addition, each new node is connected whereby for each new node the connecting (new) edge is also moved along twice. In total, we obtain $2(f_{\ell+2} - 2)$ plus $2f_{\ell+1}$ moves, that is, $2f_{\ell+3} - 4$ moves. Summing up the moves yields the number of moves taken by M to increase the level of the tree-storage from initially 1 to ℓ as

$$\sum_{i=1}^{\ell-1} 2f_{i+3} - 4 = -4(\ell-1) + 2\sum_{i=1}^{\ell-1} f_{i+3} = -4(\ell-1) - 8 + 2\sum_{i=1}^{\ell+2} f_i$$
$$= 2(f_{\ell+4} - 1) - 4\ell - 4 = 2f_{\ell+4} - 4\ell - 6,$$

since, in general, $\sum_{i=1}^{\ell} f_i = f_{\ell+2} - 1$.

Now, the construction of M is completed as follows. Initially, M performs 6 moves without any operation on the tree-storage. That is, the tree pointer stays at the root. This can be realized by additional states. Next, M starts to run through the phases described above, where at the end of phase ℓ the tree-storage forms a Fibonacci tree of level $\ell+1$. Before the first and after each phase, M performs additionally 4 moves without any operation on the tree-storage, respectively.

Finally, it remains to be described how the input is read and possibly accepted. We let M read an input symbol at every move. An input word is accepted if and only if its length is $2f_1$, $2f_2$, $2f_3$, $2f_4$, or if M reads the last input symbol exactly at the end of some phase. In order to give evidence that M works correctly, assume that the input length is $2f_x$, for some $x \geq 5$. Then M starts to generate a tree-storage that forms a Fibonacci tree of level $x-4$. The generation takes $2f_x - 4(x-4) - 6$ moves plus the initial delay of 6 moves plus the delay of totally $4(x-4)$ moves before the first and after each phase. Altogether, this makes $2f_x - 4(x-4) - 6 + 6 + 4(x-4) = 2f_x$ moves. Since M reads one input symbol at every move, it reads exactly $2f_x$ symbols. Clearly, M works in real time. □

Now we turn to a technique for disproving that languages are accepted. In general, the method is based on equivalence classes which are induced by formal languages. If some language induces a number

of equivalence classes which exceeds the number of classes distinguishable by a certain device, then the language is not accepted by that device. First we give the definition of an equivalence relation which applies to real-time twsDAs.

Let $L \subseteq \Sigma^*$ be a language and $\ell \geq 1$ be an integer constant. Two words $w \in \Sigma^*$ and $w' \in \Sigma^*$ are *ℓ-equivalent with respect to L* if and only if $wu \in L \iff w'u \in L$ for all $u \in \Sigma^*$, $|u| \leq \ell$. The number of ℓ-equivalence classes with respect to L is denoted by $E(L,\ell)$.

Lemma 3. *Let $L \subseteq \Sigma^*$ be a language accepted by some twsDA in real time. Then there exists a constant $p \geq 1$ such that $E(L,\ell) \leq 2^{p \cdot 2^\ell}$.*

Proof. The number of different binary trees with n nodes is known to be the nth Catalan number C_n. We have $C_0 = 1$ and $C_{n+1} = \frac{4n+2}{n+2} C_n$ (see, for example, [26]). So, we obtain $C_n \leq 4^n$, which is a rough but for our purposes good enough estimation.

Now, let M be a real-time twsDA with state set Q and tree symbols Γ. In order to determine an upper bound for the number of ℓ-equivalence classes with respect to $L(M)$, we consider the possible configurations of M after reading all but ℓ input symbols. The remaining computation depends on the last ℓ input symbols, the current state of M, the current Γ-tree as well as the current tree pointer P. Since M works in real time, in its last at most $\ell+1$ steps it can only access at most $\ell+1$ tree nodes, starting with the current node. These may be located in the upper ℓ levels of the tree rooted in the current node, or at the upper $\ell-1$ levels of the tree rooted in the ancestor of the current node, etc. So, there are no more than $2^{\ell+1}-1+2^{\ell-1}+2^{\ell-2}+\cdots+2^0 \leq 2^{\ell+2}$ nodes that can be accessed. Though the corresponding part of the tree can have certain structures only, we consider all non-isomorphic binary trees with $2^{\ell+2}$ nodes. Each node may be labeled by a symbol of Γ or by $\perp$. Together, there are at most

$$|Q| \cdot C_{2^{\ell+2}} \cdot (|\Gamma|+1)^{2^{\ell+2}} \leq 2^{\log(|Q|)+2\cdot 2^{\ell+2}+\log(|\Gamma|+1)\cdot 2^{\ell+2}} = 2^{\log(|Q|)+4\cdot(2+\log(|\Gamma|+1))\cdot 2^\ell}$$

different possibilities. Setting $p = \log(|Q|)+4(2+\log(|\Gamma|+1))$, we derive

$$2^{\log(|Q|)+4\cdot(2+\log(|\Gamma|+1))\cdot 2^\ell} \leq 2^{p\cdot 2^\ell}.$$

Since the number of equivalence classes is not affected by the last ℓ input symbols, there are at most $2^{p\cdot 2^\ell}$ equivalence classes. □

Next, we turn to apply Lemma 3 to show that there is a context-free language which is not accepted by any twsDA in real time. To this end, we consider the homomorphism $h\colon \{\alpha_0,\alpha_1,\alpha_2,\alpha_3\}^* \to \{a,b\}^*$ defined as $h(\alpha_0)=aa$, $h(\alpha_1)=ab$, $h(\alpha_2)=ba$, $h(\alpha_3)=bb$, and the witness language

$$L_h = \{\, x_1\$x_2\$\cdots\$x_k\#y \mid k \geq 0, x_i \in \{a,b\}^*, 1 \leq i \leq k, \text{ and there exists } j \text{ such that } x_j^R = h(y) \,\}.$$

Theorem 4. *The language L_h is not accepted by any twsDA in real time.*

Proof. We consider some integer constant $\ell \geq 1$ and show that $E(L_h,\ell)$ exceeds the number of equivalence classes distinguishable by any real-time twsDA. To this end, let $L_h^{(\ell)} \subset L_h$ be the language of words from L_h whose factors x_i, $1 \leq i \leq k$, all have length 2ℓ.

There are $2^{2^{2\ell}}$ different subsets of $\{a,b\}^{2\ell}$. For every subset $P = \{v_1, v_2, \ldots, v_k\} \subseteq \{a,b\}^{2\ell}$, we define a word $w_P = \$v_1\$v_2\$\cdots\$v_k\#$. Now, let P and S be two different subsets. Then there is some word $u \in \{a,b\}^{2\ell}$ such that u belongs to the symmetric difference of P and S. Say, u belongs to $P \setminus S$. Setting $\hat{u} = h^{-1}(u)$ We have $w_P\hat{u}^R \in L_h$ and $w_S\hat{u}^R \notin L_h$. Therefore, language L_h induces at least $2^{2^{2\ell}}$ equivalence classes in $E(L_h,\ell)$.

On the other hand, if L would be accepted by some real-time twsDA, then, by Lemma 3, there is a constant $p \geq 1$ such that $E(L_h, \ell) \leq 2^{p \cdot 2^\ell}$. Since L_h is infinite, we may choose ℓ large enough such that $2^{2\ell} > p \cdot 2^\ell$. □

Since the language L_h is context free and, on the other hand, the non-semilinear unary language of Proposition 2 belongs to $\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDNEA})$, we have the following incomparabilities.

Theorem 5. *The families $\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDA})$ and $\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDNEA})$ are both incomparable with the family of context-free languages.*

Next, we consider classical deterministic one-way stack automata. It has been shown that the unary language $L_{\mathrm{cub}} = \{ a^{n^3} \mid n \geq 0 \}$ is not accepted by any DSA [23].

Proposition 6. *The language L_{cub} is accepted by some twsDA in real time.*

Proposition 6 and the result in [23] yield the following corollary.

Corollary 7. *There is a language belonging to $\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDA})$ that does not belong to $\mathscr{L}(\mathsf{DSA})$.*

4 Basic Closure Properties

The goal of this section is to collect some basic closure properties of the families $\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDA})$ and $\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDNEA})$. In particular, we consider Boolean operations (complementation, union, intersection) and AFL operations (union, intersection with regular languages, homomorphism, inverse homomorphism, concatenation, iteration). The results are summarized in Table 1 at the end of the section.

It turns out that the two families in question have the same properties and, in particular, share all but one of these closure properties with the important family of deterministic context-free languages.

We start by mentioning the only two positive closure properties which more or less follow trivially from the definitions.

Proposition 8. *The families $\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDA})$ and $\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDNEA})$ are closed under complementation and intersection with regular languages.*

Proof. For acceptance it is required that the tree-walking-storage automata halt accepting after having read the input entirely. Due to the real-time requirement the machines halt in any case. Should this happen somewhere in the input, the remaining input can be read in an extra state. So, interchanging accepting and non-accepting states is sufficient to accept the complement of a language.

For the intersection with regular languages, it is enough to simulate a deterministic finite automaton in the states which is a standard construction for automata. □

In order to prepare for further (non-)closure properties, we now tweak the language L_h of Section 3 and define

$$L_p = \{ x_1 \$^{|x_1|} x_2 \$^{|x_2|} \cdots x_k \$^{|x_k|} \# y \mid k \geq 0, x_i \in \{a,b\}^*, 1 \leq i \leq k, \\ \text{no } x_i \text{ is proper prefix of } x_j, \text{ for } 1 \leq j < i, \text{ and there exists } m \text{ such that } x_m = y \}.$$

These little changes have a big impact. The language becomes now real-time acceptable by some twsDNEA M. The basic idea of the construction of M is that it can accept L_p by building a trie from $x_1, x_2, \ldots, x_k$, observing that the \$ padding allows it to return to the root between each part, and then on encountering # it matches y to the trie.

Theorem 9. *The language L_p is accepted by some twsDNEA in real time.*

The construction in the proof of Theorem 9 can straightforwardly be extended to show that the following language $\hat{L}_p$ is also accepted by some twsDNEA in real time.

$$\hat{L}_p = \{ x_1\$^{|x_1|}x_2\$^{|x_2|}\cdots x_k\$^{|x_k|}¢z\#_1 y \mid k \geq 0, x_i \in \{a,b\}^*, 1 \leq i \leq k, z \in \{a,b,\$\}^* \\ \text{no } x_i \text{ is proper prefix of } x_j, \text{ for } 1 \leq j < i, \text{ and there exists } m \text{ such that } x_m = y \}.$$

The language

$$\hat{L}_{\mathrm{mi}} = \{ x¢v\$v^R\#_2 \mid x \in \{a,b,\$\}^*, v \in \{a,b\}^* \}$$

is accepted by some deterministic pushdown automaton in real time. Therefore, it is accepted by some real-time twsDNEA as well.

The proof of the next Proposition first shows the non-closure under union. Then the non-closure under intersection follows from the closure under complementation by De Morgan's law. A witness for the non-closure under union is $L = \hat{L}_p \cup \hat{L}_{\mathrm{mi}}$. No real-time twsDA can accept L as any deterministic automaton would have to represent a tree with arbitrary height representing a potential v from $\hat{L}_{\mathrm{mi}}$, which makes it impossible for it to reach whatever representation it has built of $x_1, x_2, \ldots, x_k$ if it turns out to be trying to accept $\hat{L}_p$.

Proposition 10. *The families $\mathscr{L}_{\mathrm{rt}}$(twsDA) and $\mathscr{L}_{\mathrm{rt}}$(twsDNEA) are neither closed under union nor under intersection.*

We turn to the catenation operations.

Proposition 11. *The families $\mathscr{L}_{\mathrm{rt}}$(twsDA) and $\mathscr{L}_{\mathrm{rt}}$(twsDNEA) are neither closed under concatenation nor under iteration.*

Proof. To make the language $\hat{L}_p \cup \hat{L}_{\mathrm{mi}}$ more manageable we add a hint to the left of the words. So, let $\bullet$ be a new symbol and set $L_1 = \bullet\hat{L}_p \cup \hat{L}_{\mathrm{mi}}$. Since $\hat{L}_p$ and $\hat{L}_{\mathrm{mi}}$ do belong to $\mathscr{L}$(twsDNEA), L_1 is accepted by some real-time twsDNEA as well. The second language used here is the finite language $L_2 = \{\bullet, \bullet\bullet\}$ that certainly also belongs to $\mathscr{L}$(twsDNEA).

We consider the concatenation $L_2 \cdot L_1$ and assume that it belongs to $\mathscr{L}$(twsDA). Since $\mathscr{L}$(twsDA) is closed under intersection with regular languages, $(L_2 \cdot L_1) \cap \bullet\bullet\{a,b,\$,¢,\#_1,\#_2\}^* = \bullet\bullet(\hat{L}_p \cup \hat{L}_{\mathrm{mi}})$ belongs to $\mathscr{L}$(twsDA). Since $\mathscr{L}$(twsDA) is straightforwardly closed under left quotient by a singleton, we obtain $\hat{L}_p \cup \hat{L}_{\mathrm{mi}} \in \mathscr{L}$(twsDA), a contradiction.

The non-closure under iteration follows similarly. Since L_2 is regular, we derive that $L_1 \cup L_2 \in \mathscr{L}$(twsDA). However, $(L_1 \cup L_2)^* \cap \bullet\bullet\{a,b,\$,¢,\#_1,\#_2\}^+$ equals again $\bullet\bullet(\hat{L}_p \cup \hat{L}_{\mathrm{mi}})$. So, as for the concatenation we obtain a contradiction to the assumption that $\mathscr{L}$(twsDA) is closed under iteration. □

Proposition 12. *The families $\mathscr{L}_{\mathrm{rt}}$(twsDA) and $\mathscr{L}_{\mathrm{rt}}$(twsDNEA) are not closed under length-preserving homomorphisms.*

Proof. The idea to show the non-closure is first to provide some hint that allows a language to be accepted, and then to make the hint worthless by applying a homomorphism.

So, let us provide a hint that makes the language $\hat{L}_p \cup \hat{L}_{\mathrm{mi}}$ acceptable by some real-time twsDNEA. We use two new symbols $\bullet_1$ and $\bullet_2$ and set $L = \bullet_1\hat{L}_p \cup \bullet_2\hat{L}_{\mathrm{mi}}$. In this way, L belongs to $\mathscr{L}_{\mathrm{rt}}$(twsDNEA). However applying the homomorphism $h\colon \{a,b,\$,¢,\#_1,\#_2,\bullet_1,\bullet_2\}^* \to \{a,b,\$,¢,\#_1,\#_2,\bullet\}^*$, that maps $\bullet_1$ and $\bullet_2$ to $\bullet$ and all other symbols to itself, to language L yields $h(L) = \bullet(\hat{L}_p \cup \hat{L}_{\mathrm{mi}})$ which does not belong to $\mathscr{L}_{\mathrm{rt}}$(twsDA). □

Proposition 13. *The families $\mathscr{L}_{\mathrm{rt}}(\mathit{twsDA})$ and $\mathscr{L}_{\mathrm{rt}}(\mathit{twsDNEA})$ are not closed under inverse homomorphisms.*

Proof. Previously, we have taken the language $L_h \notin \mathscr{L}_{\mathrm{rt}}(\mathsf{twsDA})$ and tweaked it to $L_p \in \mathscr{L}_{\mathrm{rt}}(\mathsf{twsDNEA})$. Now we merge both languages to

$$\tilde{L}_h = \{ x_1\$^{|x_1|}x_2\$^{|x_2|}\cdots x_k\$^{|x_k|}\#y \mid k \geq 0, x_i \in \{a,b\}^*, 1 \leq i \leq k, \\ \text{no } x_i \text{ is proper prefix of } x_j, \text{ for } 1 \leq j < i, \text{ and there exists } m \text{ such that } x_m = h(y) \},$$

where $h\colon \{\alpha_0,\alpha_1,\alpha_2,\alpha_3\}^* \to \{a,b\}^*$ is defined as $h(\alpha_0) = aa$, $h(\alpha_1) = ab$, $h(\alpha_2) = ba$, and $h(\alpha_3) = bb$. The main ingredients to show that $L_h \notin \mathscr{L}_{\mathrm{rt}}(\mathsf{twsDA})$ (Theorem 4) are kept such that $\tilde{L}_h \notin \mathscr{L}_{\mathrm{rt}}(\mathsf{twsDA})$ immediately follows.

Similarly, if we require that $y \in \{a',b'\}$ has to match a factor x_i after being unprimed then the corresponding language

$$\tilde{L}_p = \{ x_1\$^{|x_1|}x_2\$^{|x_2|}\cdots x_k\$^{|x_k|}\#y \mid k \geq 0, x_i \in \{a,b\}^*, 1 \leq i \leq k, \\ \text{no } x_i \text{ is proper prefix of } x_j, \text{ for } 1 \leq j < i, \text{ and there exists } m \text{ such that } x_m = h_1(y) \}$$

where $h_1\colon \{a',b'\}^* \to \{a,b\}^*$ is defined as $h_1(a') = a$, and $h_1(b') = b$, still belongs to $\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDNEA})$.

We define the homomorphism $h_2\colon \{\alpha_0,\alpha_1,\alpha_2,\alpha_3,a,b,\$,¢,\#_1,\#_2\}^* \to \{a,b,\$,¢,\#_1,\#_2,a',b'\}^*$ as $h_2(\alpha_0) = a'a'$, $h_2(\alpha_1) = a'b'$, $h_2(\alpha_2) = b'a'$, $h_2(\alpha_3) = b'b'$, and $h_2(x) = x$, for $x \in \{a,b,\$,¢,\#_1,\#_2\}$.

So, we have $h_2^{-1}(\tilde{L}_p) = \tilde{L}_h$ which implies the non-closure under inverse homomorphisms. □

Finally, we consider the reversal.

Proposition 14. *The families $\mathscr{L}_{\mathrm{rt}}(\mathit{twsDA})$ and $\mathscr{L}_{\mathrm{rt}}(\mathit{twsDNEA})$ are not closed under reversal.*

Proof. A witness for the non-closure under reversal is the language $L = \hat{L}_p \cup \hat{L}_{\mathrm{mi}}$. By Proposition 10, it is not accepted by any real-time twsDA.

Concerning L^R, the first symbol of an input decides to which language it still may belong. If the symbol is $\#_2$ the input may only belong to $\hat{L}_{\mathrm{mi}}^R$. If it is from $\{a,b,\#_2\}$ then the input may only belong to $\hat{L}_p^R$.

The language $\hat{L}_{\mathrm{mi}}^R$ is accepted by some real-time deterministic pushdown automaton and, thus, by some real-time twsDNEA. Furthermore, it is not hard to see that $\hat{L}_p^R$ belongs to $\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDNEA})$ as well. We conclude the non-closures under reversal. □

Family	—	$\cup$	$\cap$	$\cap_{\mathrm{reg}}$	$\cdot$	$*$	$h_{\mathrm{len.pres.}}$	h^{-1}	R
$\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDA})$	✓	✗	✗	✓	✗	✗	✗	✗	✗
$\mathscr{L}_{\mathrm{rt}}(\mathsf{twsDNEA})$	✓	✗	✗	✓	✗	✗	✗	✗	✗
DCFL	✓	✗	✗	✓	✗	✗	✗	✓	✗

Table 1: Closure properties of the language families discussed. DCFL denotes the family of deterministic context-free languages.

5 Future Work

We made some first steps to investigate deterministic real-time tree-walking-storage automata. Several possible lines of future research may be tackled. First of all, it would be natural to consider the *nondeterministic* variants of the model. Decision problems and their computational complexities are an untouched area. Another question is how and to which extent the capacities and complexities are changing in case of a unary input alphabet and/or a unary set of tree symbols (which lead to the notion of counters in the classical models).

References

[1] Alfred V. Aho & Jeffrey D. Ullman (1971): *Translations on a Context-Free Grammar*. Inform. Control 19(5), pp. 439–475, doi:10.1016/S0019-9958(71)90706-6.

[2] Suna Bensch, Johanna Björklund & Martin Kutrib (2017): *Deterministic Stack Transducers*. Int. J. Found. Comput. Sci. 28, pp. 583–601, doi:10.1142/S0129054117400081.

[3] Mikołaj Bojańczyk & Thomas Colcombet (2006): *Tree-walking automata cannot be determinized*. Theor. Comput. Sci. 350, pp. 164–173, doi:10.1016/j.tcs.2005.10.031.

[4] Mikołaj Bojańczyk & Thomas Colcombet (2008): *Tree-Walking Automata Do Not Recognize All Regular Languages*. SIAM J. Comput. 38, pp. 658–701, doi:10.1137/050645427.

[5] Tobias Denkinger (2016): *An Automata Characterisation for Multiple Context-Free Languages*. In Srecko Brlek & Christophe Reutenauer, editors: *Developments in Language Theory (DLT 2016)*, *LNCS* 9840, Springer, pp. 138–150, doi:10.1007/978-3-662-53132-7_12.

[6] Seymour Ginsburg, Sheila A. Greibach & M. A. Harrison (1967): *Stack automata and compiling*. J. ACM 14, pp. 172–201, doi:10.1145/321371.321385.

[7] Seymour Ginsburg, Sheila A. Greibach & Michael A. Harrison (1967): *One-Way Stack Automata*. J. ACM 14, pp. 389–418, doi:10.1145/321386.321403.

[8] Wolfgang Golubski & Wolfram-Manfred Lippe (1996): *Tree-Stack Automata*. Math. Systems Theory 29, pp. 227–244, doi:10.1007/BF01201277.

[9] Sheila A. Greibach (1969): *Checking Automata and One-Way Stack Languages*. J. Comput. Syst. Sci. 3, pp. 196–217, doi:10.1016/S0022-0000(69)80012-7.

[10] Irène Guessarian (1983): *Pushdown Tree Automata*. Math. Systems Theory 16, pp. 237–263, doi:10.1007/BF01744582.

[11] Eitan M. Gurari & Oscar H. Ibarra (1982): *(Semi)Alternating Stack Automata*. Math. Systems Theory 15, pp. 211–224, doi:10.1007/BF01786980.

[12] J. Hartmanis & R. E. Stearns (1965): *On the Computational Complexity of Algorithms*. Trans. Amer. Math. Soc. 117, pp. 285–306, doi:10.1090/S0002-9947-1965-0170805-7.

[13] John E. Hopcroft & Jeffrey D. Ullman (1967): *Nonerasing Stack Automata*. J. Comput. Syst. Sci. 1, pp. 166–186, doi:10.1016/S0022-0000(67)80013-8.

[14] John E. Hopcroft & Jeffrey D. Ullman (1968): *Deterministic Stack Automata and the Quotient Operator*. J. Comput. Syst. Sci. 2, pp. 1–12, doi:10.1016/S0022-0000(68)80003-0.

[15] Oscar H. Ibarra (1971): *Characterizations of Some Tape and Time Complexity Classes of Turing Machines in Terms of Multihead and Auxiliary Stack Automata*. J. Comput. Syst. Sci. 5(2), pp. 88–117, doi:10.1016/S0022-0000(71)80029-6.

[16] Oscar H. Ibarra, Jozef Jirásek, Ian McQuillan & Luca Prigioniero (2021): *Space Complexity of Stack Automata Models*. Int. J. Found. Comput. Sci. 32, pp. 801–823, doi:10.1142/S0129054121420090.

[17] Oscar H. Ibarra & Ian McQuillan (2018): *Variations of checking stack automata: Obtaining unexpected decidability properties*. *Theor. Comput. Sci.* 738, pp. 1–12, doi:10.1016/j.tcs.2018.04.024.

[18] Oscar H. Ibarra & Ian McQuillan (2021): *Generalizations of Checking Stack Automata: Characterizations and Hierarchies*. *Int. J. Found. Comput. Sci.* 32, pp. 481–508, doi:10.1142/S0129054121410045.

[19] S. Rao Kosaraju (1974): *1-Way Stack Automaton with Jumps*. *J. Comput. Syst. Sci.* 9, pp. 164–176, doi:10.1016/S0022-0000(74)80005-X.

[20] Martin Kutrib, Andreas Malcher & Matthias Wendlandt (2017): *Tinput-Driven Pushdown, Counter, and Stack Automata*. *Fund. Inform.* 155, pp. 59–88, doi:10.3233/FI-2017-1576.

[21] Martin Kutrib & Uwe Meyer (2023): *Tree-Walking-Storage Automata*. In Frank Drewes & Mikhail Volkov, editors: *Developments in Language Theory (DLT 2023)*, *LNCS* 13911, Springer, pp. 182–194, doi:10.1007/978-3-031-33264-7_15.

[22] Klaus-Jörn Lange (2010): *A Note on the P-completeness of Deterministic One-way Stack Language*. *J. UCS* 16, pp. 795–799, doi:10.3217/jucs-016-05-0795.

[23] William F. Ogden (1969): *Intercalation Theorems for Stack Languages*. In: *Proceedings of the First Annual ACM Symposium on Theory of Computing (STOC 1969)*, ACM Press, New York, pp. 31–42, doi:10.1145/800169.805419.

[24] Michael Oser Rabin (1963): *Real time computation*. *Israel J. Math.* 1, pp. 203–211, doi:10.1007/BF02759719.

[25] Eli Shamir & Catriel Beeri (1974): *Checking Stacks and Context-Free Programmed Grammars Accept P-complete Languages*. In Jacques Loeckx, editor: *International Colloquium on Automata, Languages and Programming (ICALP 1974)*, *LNCS* 14, Springer, pp. 27–33, doi:10.1007/3-540-06841-4_50.

[26] Richard P. Stanley (2015): *Catalan Numbers*. Cambridge University Press, doi:10.1017/CBO9781139871495.

Latvian Quantum Finite State Automata for Unary Languages

Carlo Mereghetti Beatrice Palano Priscilla Raucci
Dipartimento di Informatica "Giovanni Degli Antoni"
Università degli Studi di Milano, via Celoria 18, 20135 Milano – Italy
{carlo.mereghetti, beatrice.palano, priscilla.raucci}@unimi.it

We design *Latvian quantum finite state automata* (LQFAs for short) recognizing unary regular languages with isolated cut point $\frac{1}{2}$. From an architectural point of view, we combine two LQFAs recognizing with isolated cut point, respectively, the finite part and the ultimately periodic part of any given unary regular language L. In both modules, we use a component addressed in the literature and here suitably adapted to the unary case, to discriminate strings on the basis of their length. The number of basis states and the isolation around the cut point of the resulting LQFA for L exponentially depends on the size of the minimal deterministic finite state automaton for L.

1 Introduction

Quantum finite automata (QFAs for short) represent a theoretical model for a quantum computer with finite memory [3, 4]. While we can hardly expect to see a full-featured quantum computer in the near future, small quantum components, modeled by QFAs, seem to be promising from a physical implementation viewpoint (see, e.g., [7, 15]).

Very roughly speaking, a QFA is obtained by imposing the quantum paradigm — superposition, unitary evolution, observation — to a classical finite state automaton. The state of the QFA can be seen as a linear combination of classical states, called superposition. The QFA steps from a superposition to the next one by a unitary (reversible) evolution. Superpositions can transfer the complexity of a computation from a large number of sequential steps to a large number of coherently superposed classical states (this phenomenon is sometimes referred as quantum parallelism). Along its computation, the QFA can be "observed", i.e., some features, called observables, can be measured. From measuring an observable, an outcome is obtained with a certain probability and the current superposition irreversibly "collapses", with the same probability, to a particular superposition (coherent with the observed outcome).

QFAs exhibit both advantages and disadvantages with respect to their classical (deterministic or probabilistic) counterpart. Basically, quantum superposition offers some computational advantages on probabilistic superposition. On the other hand, quantum dynamics are reversible: because of limitation of memory, it is sometimes impossible to simulate deterministic finite state automata (DFAs for short) by quantum automata. Limitations due to reversibility can be partially attenuated by systematically introducing measurements of suitable observables as computational steps.

In the literature, several models of QFAs are proposed, which mainly differ in their measurement policy. The first and most simple model is the *measure-once* QFA (MO-QFA for short) [6, 16], where the probability of accepting strings is evaluated by "observing" just once, at the end of input processing. In *measure-many* QFAs (MM-QFAs for short) [11], instead, the acceptance probability is evaluated by observing after each move, thus allowing the possibility of halting the computation in the middle of input processing. An additional model is the *Latvian* QFA (LQFA for short) [1], which can be regarded as

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 63–78, doi:10.4204/EPTCS.388.8

"intermediate" between MO-QFAs and MM-QFAs. In fact, as in the MM-QFA model, LQFAs are observed after each move; on the other hand, as in the MO-QFA model, acceptance probability is evaluated at the end of the computation only. From a language recognition point of view, it is well known that MO-QFAs are strictly less powerful than LQFAs, which are strictly less powerful than MM-QFAs, which are strictly less powerful than DFAs. This hierarchy is established, e.g., in [1, 6, 11].

In this paper, we investigate the architecture and size of LQFAs processing *unary languages*, i.e. languages built over a single-letter alphabet. A similar investigation is presented in [5], where MM-QFAs recognizing unary regular languages with isolated cut point are exhibited, whose size (number of basis states) is linear in the size of equivalent minimal DFAs. Here, we show that unary regular languages can be recognized with isolated cut point by the less powerful model of LQFAs as well, paying by an exponential size increase. A relevant module in our construction is a LQFA recognizing with isolated cut point the strings of length exceeding a fixed threshold. For its design, we adapt a construction in [1, 14] to the unary case. Such a module is then suitably combined with two LQFAs taking care, respectively, of the finite part and the ultimately periodic part any unary regular language consists of. The architecture of the resulting LQFA turns out to be significantly different from the equivalent MM-QFAs in [5]. Moreover, while in the MM-QFA case the isolation around the cut point is constant, for LQFAs it exponentially decreases with respect to the size of the DFA for the finite part of the target unary regular language. However, it should be stressed that the less powerful model of MO-QFAs cannot recognize with isolated cut point all unary regular languages. Our results constructively prove that LQFAs and MM-QFAs have the same recognition power, whenever restricted to recognize unary languages with isolated cut point.

The paper is organized as follows. In Section 2, we overview basics on formal language theory, linear algebra, and quantum finite state automaton models. In Section 3, we design isolated cut point LQFAs recognizing the strings whose length is greater than or equal to a fixed value. Then, in Section 4, we provide the full architecture of isolated cut point LQFAs for unary regular languages, analyzing their size, cut point, and isolation. Finally, in Section 5, we draw some concluding remarks and offer possible research hints.

2 Preliminaries

2.1 Formal Languages

We assume familiarity with basic notions of formal language theory (see, e.g., [9]). Given a set S, we let $|S|$ denote its cardinality. The set of all words or strings (including the empty string ε) over a finite alphabet Σ is denoted by Σ^*, and we let $\Sigma^+ = \Sigma^* \setminus \varepsilon$. For a string $w \in \Sigma^*$, we let $|w|$ denote its length and w_i its ith symbol. For any given $i \geq 0$, we let Σ^i be the set of strings over Σ of length i, with $\Sigma^0 = \{\varepsilon\}$. We let $\Sigma^{\leq i} = \bigcup_{j=0}^{i} \Sigma^j$; sets $\Sigma^{>i}$ and $\Sigma^{\geq i}$ are defined accordingly. A language over Σ is any subset $L \subseteq \Sigma^*$; its complement is the language $L^c = \Sigma^* \setminus L$. A *deterministic finite state automaton* (DFA) is formally defined as a 5-tuple $D = (Q, \Sigma, q_0, \delta, F)$, where Q is the finite set of states, Σ the finite input alphabet, $q_0 \in Q$ the initial state, $F \subseteq Q$ the set of accepting states, and $\delta : Q \times \Sigma \to Q$ the transition function. Denoting by δ^* the canonical extension of δ to Σ^*, the language recognized by D is the set $L_D = \{w \in \Sigma^* | \delta^*(q_0, w) \in F\}$. It is well known that DFAs characterize the class of regular languages.

A *unary language* is any language built over a single letter alphabet, e.g., $\Sigma = \{\sigma\}$, and thus has the general form $L \subseteq \sigma^*$. Unary *regular* languages form *ultimately periodic sets*, as stated by the following

Theorem 1. ([9, 17]) *Let $L \subseteq \sigma^*$ be a unary regular language. Then, there exist two integers $T \geq 0$ and $P > 0$ such that, for any $k \geq T$, we have $\sigma^k \in L$ if and only if $\sigma^{k+P} \in L$.*

By Theorem 1, it is easy to see that any unary regular language L can be recognized by a (minimal) DFA consisting of an initial path of T states joined to a cycle of P states; accepting states are suitably settled on both the path and the cycle. Unary regular languages satisfying Theorem 1 with $T = 0$ are called *periodic languages* of period P.

2.2 Linear Algebra

We quickly recall some notions of linear algebra, useful to describe quantum computational devices. For more details, we refer the reader to, e.g., [19]. The fields of real and complex numbers are denoted by $\mathbb{R}$ and $\mathbb{C}$, respectively. Given a complex number $z = a + ib$, with $a, b \in \mathbb{R}$, its *conjugate* is denoted by $z^* = a - ib$, and its *modulus* by $|z| = \sqrt{z \cdot z^*}$. We let $\mathbb{C}^{n\times m}$ denote the set of $n \times m$ matrices with entries in $\mathbb{C}$. Given a matrix $M \in \mathbb{C}^{n\times m}$, for $1 \leq i \leq n$ and $1 \leq j \leq m$, we denote by M_{ij} its (i,j)th entry. The *transpose* of M is the matrix $M^T \in \mathbb{C}^{m\times n}$ satisfying ${M^T}_{ij} = M_{ji}$, while we let M^* be the matrix satisfying ${M^*}_{ij} = (M_{ij})^*$. The *adjoint* of M is the matrix $M^\dagger = (M^T)^*$. For matrices $A, B \in \mathbb{C}^{n\times m}$, their *sum* is the $n \times m$ matrix $(A+B)_{ij} = A_{ij} + B_{ij}$. For matrices $C \in \mathbb{C}^{n\times m}$ and $D \in \mathbb{C}^{m\times r}$, their *product* is the $n \times r$ matrix $(C \cdot D)_{ij} = \sum_{k=1}^{m} C_{ik} \cdot D_{kj}$. For matrices $A \in \mathbb{C}^{n\times m}$ and $B \in \mathbb{C}^{p\times q}$, their *direct (or tensor or Kronecker) product* is the $n{\cdot}p \times m{\cdot}q$ matrix defined as

$$A \otimes B = \begin{pmatrix} A_{11}B & \cdots & A_{1m}B \\ \vdots & \ddots & \vdots \\ A_{n1}B & \cdots & A_{nm}B \end{pmatrix}.$$

When operations are allowed by matrix dimensions, we have that $(A \otimes B) \cdot (C \otimes D) = A \cdot C \otimes B \cdot D$.

A *Hilbert space* of dimension n is the linear space $\mathbb{C}^n$ of n-dimensional complex row vectors equipped with sum and product by elements in $\mathbb{C}$, where the *inner product* $\langle \varphi, \psi \rangle = \varphi \cdot \psi^\dagger$ is defined, for vectors $\varphi, \psi \in \mathbb{C}^n$. The ith component of φ is denoted by φ_i, its *norm* is given by $\|\varphi\| = \sqrt{\langle \varphi, \varphi \rangle} = \sqrt{\sum_{i=1}^{n} |\varphi_i|^2}$. If $\langle \varphi, \psi \rangle = 0$ (and $\|\varphi\| = 1 = \|\psi\|$), then φ and ψ are orthogonal (orthonormal). An orthonormal basis of $\mathbb{C}^n$ is any set of n orthonormal vectors in $\mathbb{C}^n$. In particular, the *canonical basis* of $\mathbb{C}^n$ is the set $\{ \boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n \}$, where $\boldsymbol{e}_i \in \mathbb{C}^n$ is the vector having 1 at the ith component and 0 elsewhere. Clearly, any vector $\varphi \in \mathbb{C}^n$ can be univocally expressed as a linear combination of the vectors in the canonical basis as $\varphi = \sum_{i=1}^{n} \varphi_i \cdot \boldsymbol{e}_i$. This latter fact is usually addressed by saying that $\mathbb{C}^n$ is *spanned* by $\{ \boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n \}$. Two subspaces $X, Y \subseteq \mathbb{C}^n$ are orthogonal if any vector in X is orthogonal to any vector in Y. In this case, we denote by $X \dot{+} Y$ the linear space generated by $X \cup Y$. For vectors $\varphi \in \mathbb{C}^n$ and $\psi \in \mathbb{C}^m$, their direct (or tensor or Kronecker) product is the vector $\varphi \otimes \psi = (\varphi_1 \cdot \psi, \ldots, \varphi_n \cdot \psi)$; we have $\|\varphi \otimes \psi\| = \|\varphi\| \cdot \|\psi\|$.

A matrix $M \in \mathbb{C}^{n\times n}$ is said to be *unitary* if $M \cdot M^\dagger = I^{(n)} = M^\dagger \cdot M$, where $I^{(n)}$ is the $n \times n$ identity matrix. Equivalently, M is unitary if it preserves the norm, i.e., $\|\varphi \cdot M\| = \|\varphi\|$ for any vector $\varphi \in \mathbb{C}^n$. Direct products of unitary matrices are unitary as well. The matrix M is said to be *Hermitian (or self-adjoint)* if $M = M^\dagger$. Let $\mathscr{O} \in \mathbb{C}^{n\times n}$ be an Hermitian matrix, $\nu_1, \nu_2, \ldots, \nu_s$ its eigenvalues, and $E_1, E_2, \ldots, E_s$ the corresponding eigenspaces. It is well known that each eigenvalue ν_k is real, that E_i is orthogonal to E_j for every $1 \leq i \neq j \leq s$, and that $E_1 \dot{+} E_2 \dot{+} \cdots \dot{+} E_s = \mathbb{C}^n$. Thus, every vector $\varphi \in \mathbb{C}^n$ can be uniquely decomposed as $\varphi = \varphi_{(1)} + \varphi_{(2)} + \cdots + \varphi_{(s)}$, for unique $\varphi_{(j)} \in E_j$. The linear transformation $\varphi \mapsto \varphi_{(j)}$ is the *projector* P_j onto the subspace E_j. Actually, the Hermitian matrix $\mathscr{O}$ is biunivocally determined by its eigenvalues and projectors as $\mathscr{O} = \nu_1 \cdot P_1 + \nu_2 \cdot P_2 + \cdots + \nu_s \cdot P_s$. We recall that a matrix $P \in \mathbb{C}^{n\times n}$ is a projector if and only if P is Hermitian and idempotent, i.e. $P^2 = P$.

Let $\omega = \mathrm{e}^{i\frac{2\pi}{n}}$ be the nth root of the unity ($\omega^n = 1$) and define the Vandermonde matrix $W \in \mathbb{C}^{n\times n}$ whose (r,c)th component is ω^{rc}, for $0 \leq r, c < n$. Let the $n \times n$ complex matrix $F_n = \frac{1}{\sqrt{n}} \cdot W$. It is easy

to see that F_n is the unitary matrix implementing the *quantum Fourier transform*. Throughout the paper, it will be useful to recall that operating F_n on the jth vector of the canonical basis yields the vector $\boldsymbol{e}_j \cdot F_n = \frac{1}{\sqrt{n}} \cdot (\omega^0, \omega^{(j-1)\cdot 1}, \ldots, \omega^{(j-1)\cdot(n-1)})$. We remark that $|(\boldsymbol{e}_j \cdot F_n)_k|^2 = \frac{1}{n}$, for every $1 \leq k \leq n$.

As we will see in the next section, in accordance with quantum mechanics principles (see, e.g., [10]), the state of a quantum finite state automaton at any given time during its computation is represented by a norm 1 vector from a suitable Hilbert space, the state evolution of the automaton is modeled by unitary matrices, and information on certain characteristics of the automaton are probabilistically extracted by measuring some "observables" represented by Hermitian matrices.

2.3 Quantum Finite State Automata

Here, we recall the model of a *Latvian quantum finite state automaton* [1] we are mostly interested in. We then quickly introduce *measure-once* quantum finite state automata [6, 16] as a particular case of Latvian automata. Finally, we overview *measure-many* quantum finite state automata [11].

Definition 1. *Let Σ be an input alphabet, $\sharp \notin \Sigma$ an endmarker symbol, and set $\Gamma = \Sigma \cup \{\sharp\}$. A* Latvian quantum finite automaton (LQFA for short) *is a system $\mathscr{A} = (Q, \Sigma, \pi_0, \{U(\sigma)\}_{\sigma\in\Gamma}, \{\mathscr{O}_\sigma\}_{\sigma\in\Gamma}, Q_{acc})$, where*

- *$Q = \{q_1, q_2, \ldots, q_n\}$ is the finite set of basis states; the elements of Q span*[1] *the Hilbert space $\mathbb{C}^n$,*
- *$Q_{acc} \subseteq Q$ is the set of accepting basis states,*
- *$\pi_0 \in \mathbb{C}^n$ is the initial amplitude vector (superposition) satisfying $\|\pi_0\| = 1$,*
- *$U(\sigma) \in \mathbb{C}^{n\times n}$ is the unitary evolution matrix, for any $\sigma \in \Gamma$,*
- *for any $\sigma \in \Sigma$, we let $\mathscr{O}_\sigma = \sum_{i=0}^{k_\sigma - 1} c_i(\sigma) \cdot P_i(\sigma)$ be an observable (Hermitian matrix) on $\mathbb{C}^n$, where $\{c_0(\sigma), \ldots, c_{k_\sigma-1}(\sigma)\}$ is the set of all possible outcomes (eigenvalues) of measuring $\mathscr{O}_\sigma$, and $\{P_0(\sigma), \ldots, P_{k_\sigma-1}(\sigma)\}$ are the projectors onto the corresponding eigenspaces,*
- *we let $\mathscr{O}_\sharp = a \cdot P_{acc}(\sharp) + r \cdot (I^{(n)} - P_{acc}(\sharp))$ be the final observable, where $P_{acc}(\sharp)$ is the projector onto the subspace of $\mathbb{C}^n$ spanned by the states in Q_{acc}.*

Let us briefly describe the behavior of $\mathscr{A}$ on an input word $w\sharp \in \Sigma^*\sharp$. At any given time, the *state* of $\mathscr{A}$ is a *superposition of basis states* in Q which is represented by a norm 1 vector $\xi \in \mathbb{C}^n$. We have that $\xi_i \in \mathbb{C}$ is the *amplitude* of the basis state q_i, while $|\xi_i|^2 \in [0,1]$ is the *probability* of observing $\mathscr{A}$ in the basis state q_i. The computation of $\mathscr{A}$ on $w\sharp$ starts in the initial superposition π_0 by reading the first input symbol. Then, the transformations associated with each input symbol are applied in succession. The transformation corresponding to a symbol $\sigma \in \Gamma$ consists of two steps:

1. *Evolution:* the matrix $U(\sigma)$ acts on the current state ξ of $\mathscr{A}$, yielding the next state $\xi' = \xi \cdot U(\sigma)$.
2. *Observation:* the observable $\mathscr{O}_\sigma$ is measured and the outcome $c_i(\sigma)$ is seen with probability $\|\xi' \cdot P_i(\sigma)\|^2$; upon seeing $c_i(\sigma)$, according to Copenhagen interpretation of quantum mechanics [10], the state of $\mathscr{A}$ "collapses" to (norm 1) state $\xi' \cdot P_i(\sigma) / \|\xi' \cdot P_i(\sigma)\|$ and the computation continues, unless we are processing the endmarker $\sharp$.

Upon processing the endmarker $\sharp$, the final observable $\mathscr{O}_\sharp$ is measured yielding the probability of seeing $\mathscr{A}$ in an accepting basis state. Therefore, the probability of accepting $w \in \Sigma^*$ is given by

$$p_{\mathscr{A}}(w) = \sum_{i_1=0}^{k_{w_1}-1} \cdots \sum_{i_{|w|}=0}^{k_{w_{|w|}}-1} \left\| \pi_0 \cdot U(w_1) \cdot P_{i_1}(w_1) \cdot \cdots \cdot U(w_{|w|}) \cdot P_{i_{|w|}}(w_{|w|}) \cdot U(\sharp) \cdot P_{acc}(\sharp) \right\|^2.$$

[1] We can associate with the set $Q = \{q_1, q_2, \ldots, q_n\}$ of basis states the canonical basis $\{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n\}$ of the Hilbert space $\mathbb{C}^n$ (see Section 2.2) where, for each $1 \leq i \leq n$, we let $\boldsymbol{e}_i$ represent the basis state q_i. As the canonical basis spans $\mathbb{C}^n$, with a slight abuse of notation, we say that the elements of Q spans $\mathbb{C}^n$.

The function $p_{\mathscr{A}} : \Sigma^* \to [0,1]$ is the *stochastic event induced by* $\mathscr{A}$. The *language recognized by* $\mathscr{A}$ *with cut point* $\lambda \in [0,1]$ is the set of words $L_{\mathscr{A},\lambda} = \{w \in \Sigma^* \mid p_{\mathscr{A}}(w) > \lambda\}$. The cut point λ is said to be *isolated* whenever there exists $\rho \in \left(0, \frac{1}{2}\right]$ such that $|p_{\mathscr{A}}(w) - \lambda| \geq \rho$, for any $w \in \Sigma^*$. The parameter ρ is usually referred to as *radius of isolation*.

In general, a language $L \subseteq \Sigma^*$ is recognized with isolated cut point by a LQFA whenever there exists a LQFA $\mathscr{A}$ such that $(\inf\{p_{\mathscr{A}}(w) \mid w \in L\} - \sup\{p_{\mathscr{A}}(w) \mid w \notin L\}) > 0$. In this case, we can compute the cut point as being $\lambda = \frac{1}{2} \cdot (\inf\{p_{\mathscr{A}}(w) \mid w \in L\} + \sup\{p_{\mathscr{A}}(w) \mid w \notin L\})$, with radius of isolation $\rho = \frac{1}{2} \cdot (\inf\{p_{\mathscr{A}}(w) \mid w \in L\} - \sup\{p_{\mathscr{A}}(w) \mid w \notin L\})$. Throughout the rest of the paper, for the sake of conciseness, we will sometimes be writing "isolated cut point quantum finite automaton for a language" instead of "quantum finite automaton recognizing a language with isolated cut point". Isolated cut point turns out to be one of the main language recognition policies within the literature of probabilistic devices. Its relevance in the realm of finite state automata is due to the fact that we can arbitrarily reduce the classification error probability of an input word w by repeating a constant number of times (not depending on the length of w) its parsing and taking the majority of the answers. We refer the reader to ,e.g., [18, Sec. 5], where the notion of isolated cut point recognition is introduced and carefully analyzed.

One of the two original and most studied models of a quantum finite state automaton is the *measure-once* model (MO-QFA for short). An MO-QFA can be seen as a particular LQFA where, for any $\sigma \in \Sigma$, we have that $\mathscr{O}_\sigma = I^{(n)}$. Basically, this amounts to leave the computation of $\mathscr{A}$ undisturbed up to the final observation for acceptance. Thus, an MO-QFA can be formally and more succinctly written as $\mathscr{A} = (Q, \Sigma, \pi_0, \{U(\sigma)\}_{\sigma \in \Gamma}, Q_{acc})$. The probability of $\mathscr{A}$ accepting the word $w \in \Sigma^*$ now simplifies as

$$p_{\mathscr{A}}(w) = \left\| \pi_0 \cdot U(w_1) \cdot \cdots \cdot U(w_{|w|}) \cdot U(\sharp) \cdot P_{acc}(\sharp) \right\|^2.$$

Let us now switch to the other original model, namely a *measure-many quantum finite state automaton* (MM-QFA for short). Roughly speaking, an MM-QFA $\mathscr{A}$ is defined as LQFA but with the possibility of accepting/rejecting the input string *before* reaching the endmarker. More precisely, the set Q of the basis states of $\mathscr{A}$ can be partitioned into *halting states*, which can be either *accepting* or *rejecting*, and *non halting states*, also called *go* states, i.e., $Q = Q_{acc} \cup Q_{rej} \cup Q_{go}$. Following such a state partition, the sole observable $\mathscr{O} = a \cdot P_{acc} + r \cdot P_{rej} + g \cdot P_{go}$, whose projectors map onto the subspaces spanned by the corresponding basis states, is associated with *each* symbol in Γ. At each step, the observable $\mathscr{O}$ is measured and the computation of $\mathscr{A}$ continues (unless we are processing $\sharp$) only if the outcome g is seen. Instead, if the outcome a (r) is seen, then $\mathscr{A}$ halts and accepts (rejects). Formally, the MM-QFA $\mathscr{A}$ can be written as $\mathscr{A} = (Q, \Sigma, \pi_0, \{U(\sigma)\}_{\sigma \in \Gamma}, \mathscr{O}, Q_{acc})$, and the probability of accepting the word $w\sharp = w_1 \cdots w_n w_{n+1}$ is

$$p_{\mathscr{A}}(w) = \textstyle\sum_{k=1}^{n+1} \| \pi_0 \cdot (\prod_{i=1}^{k-1} U(w_i) \cdot P_{go}) \cdot U(w_k) \cdot P_{acc} \|^2.$$

It is well known (see, e.g., [6, 16]) that the class of languages recognized by isolated cut point MO-QFAs coincides with the class of group languages. Notice that finite languages are not group languages, and hence they cannot be accepted by isolated cut point MO-QFAs. Isolated cut point LQFAs are proved in [1, 14] to be strictly more powerful than isolated cut point MO-QFAs, since their recognition power coincides with the class of block group languages. An equivalent characterization states that a language is recognized by an isolated cut point LQFA if and only if it belongs to the boolean closure of languages of the form $L_1 a_1 L_2 a_2 \cdots a_k L_{k+1}$, for $a_i \in \Sigma$, group language $L_i \subseteq \Sigma^*$, and $|\Sigma| > 1$. Finally, the recognition power of isolated cut point MM-QFAs still remains an open question. However, it is know that MM-QFAs are strictly more powerful than LQFAs but strictly less powerful than DFAs. In fact, isolated cut point MM-QFAs can recognize the language $a\Sigma^*$ which cannot be accepted by isolated cut point LQFAs [1]. On the other hand, isolated cut point MM-QFAs cannot recognize the language $\Sigma^* a$, for $|\Sigma| > 1$ and $a \in \Sigma$ [11].

3 Isolated Cut Point LQFAs for Words Longer than T

Here, we design an isolated cut point LQFA recognizing the unary language $\sigma^{\geq T}$, for any given $T > 0$ (i.e., the set of unary strings whose length is greater than or equal to T). As it will be clear in the next section, this LQFA will be a relevant component in the modular construction of isolated cut point LQFAs for unary regular languages.

Our design pattern is inspired by [1, 14], where the authors provide an isolated cut point LQFA for the language $\Sigma^* a_1 \Sigma^* a_2 \cdots a_k \Sigma^*$, with $a_i \in \Sigma$ and $|\Sigma| > 1$. So, we focus on recognizing the unary version of $\Sigma^* a_1 \Sigma^* a_2 \cdots a_T \Sigma^*$ yielded by fixing $a_1 = \cdots = a_T = \sigma$ and $\Sigma = \{\sigma\}$, namely, the desired language $a_1 \cdots a_T \Sigma^* = \sigma^{\geq T}$. We adapt the construction in [1, 14], and inductively exhibit a family $\{M^{(\ell)}\}_{\ell \geq 1}$ of LQFAs such that: *(i)* $M^{(\ell)}$ recognizes the language $\sigma^{\geq \ell}$ with isolated cut point, and *(ii)* $M^{(\ell)}$ is constructed by "expanding" $M^{(\ell-1)}$. So, the desired isolated cut point LQFA for $\sigma^{\geq T}$ will result after T "expansions", starting from the LQFA $M^{(1)}$. We provide a detailed analysis of the stochastic behavior of $M^{(\ell)}$ machines, emphasizing cut points, isolations and their size (i.e., number of their basis states). In this section, to have a convenient notation, we will be using A_σ for the evolution operator of our LQFAs.

Base of the construction: For the induction base, we define the LQFA $M^{(1)}$ for the language $\sigma^{\geq 1}$ as

$$M^{(1)} = (Q^{(1)}, \{\sigma\}, \pi_0, \{A_\sigma^{(1)}, A_\sharp^{(1)}\}, \{\mathscr{O}_\sigma^{(1)}, \mathscr{O}_\sharp^{(1)}\}, Q_{acc}^{(1)}),$$

where $Q^{(1)} = \{q_0, \ldots, q_{n-1}\}$ is the set of n basis states, $\pi_0 = \boldsymbol{e}_1$ meaning that $M^{(1)}$ starts in the state q_0, $Q_{acc}^{(1)} = Q^{(1)} \setminus \{q_0\}$ is the set of $n-1$ accepting states. For the evolution matrices, we let $A_\sigma^{(1)} = F_n$ (the quantum Fourier transform) and $A_\sharp^{(1)} = I$ (the identity matrix). The observable $\mathscr{O}_\sigma^{(1)}$ is the *canonical observable* defined by the projectors $\{\boldsymbol{e}_1{}^\dagger \cdot \boldsymbol{e}_1, \boldsymbol{e}_2{}^\dagger \cdot \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n{}^\dagger \cdot \boldsymbol{e}_n\}$. By measuring $\mathscr{O}_\sigma^{(1)}$ on $M^{(1)}$ being in the superposition $\xi \in \mathbb{C}^n$, we will see $M^{(1)}$ in the basis state q_{i-1} with probability $\|\xi \cdot (\boldsymbol{e}_i{}^\dagger \cdot \boldsymbol{e}_i)\|^2 = |\xi_i|^2$. Upon such an outcome, the state of $M^{(1)}$ clearly collapses to $\boldsymbol{e}_i$. The final observation $\mathscr{O}_\sharp^{(1)}$ projects onto the subspace spanned by the accepting basis states $\{q_1, \ldots, q_{n-1}\}$.

The automaton $M^{(1)}$ behaves as follows: when the first input symbol is read, the state of $M^{(1)}$ becomes $\pi_0 \cdot A_\sigma^{(1)} = \boldsymbol{e}_1 \cdot F_n$, upon which the canonical observation is measured. As noticed at the end of Section 2.2, such a measurement will cause $M^{(1)}$ to move from q_0 to some basis state q_i, with $0 \leq i \leq n-1$, uniformly at random (i.e., with probability $|(\boldsymbol{e}_1 \cdot F_n)_{i+1}|^2 = \frac{1}{n}$). After processing (again, by quantum Fourier transform followed by measuring the canonical observable) the next input symbol from being in the state $\boldsymbol{e}_i$, we again find $M^{(1)}$ in a basis state uniformly at random. Such a dynamics continues unaltered, until the endmarker is reached and processed by the identity matrix. At this point, the final observation $\mathscr{O}_\sharp$ is measured, and an accepting state is easily seen to be reached with probability $|Q_{acc}^{(1)}| \cdot \frac{1}{n} = (\frac{n-1}{n})$. Clearly, processing the empty string leaves $M^{(1)}$ in the non accepting state q_0 with certainty. Therefore, $p_{M^{(1)}}(\varepsilon) = 0$, while for $k > 0$ we have $p_{M^{(1)}}(\sigma^k) = (\frac{n-1}{n})$. So, $M^{(1)}$ recognizes the language $\sigma^{\geq 1}$ with isolated cut point.

Inductive step of the construction: For the inductive step, we show how to build the isolated cut point LQFA $M^{(\ell)}$ for the language $\sigma^{\geq \ell}$ from the LQFA $M^{(\ell-1)}$ for the language $\sigma^{\geq \ell-1}$, this latter LQFA being given by inductive hypothesis. We define

$$M^{(\ell)} = (Q^{(\ell)}, \{\sigma\}, \pi_0, \{A_\sigma^{(\ell)}, A_\sharp^{(\ell)}\}, \{\mathscr{O}_\sigma^{(\ell)}, \mathscr{O}_\sharp^{(\ell)}\}, Q_{acc}^{(\ell)}),$$

where the set $Q^{(\ell)}$ of basis states consists of the previous set $Q^{(\ell-1)}$ of basis states, plus $(n-1)$ new basis states per each state in $Q_{acc}^{(\ell-1)}$. We let $Q_{acc}^{(\ell)}$ be the set containing these $(n-1) \cdot |Q_{acc}^{(\ell-1)}|$ new states, with $|Q_{acc}^{(\ell)}| = (n-1)^\ell$. Therefore, $Q^{(\ell)} = Q^{(\ell-1)} \cup Q_{acc}^{(\ell)} = \{q_0\} \cup Q_{acc}^{(1)} \cup Q_{acc}^{(2)} \cup \cdots \cup Q_{acc}^{(\ell)}$ with $|Q_{acc}^{(i)}| = (n-1)^i$,

so that $|Q^{(\ell)}| = \sum_{i=0}^{\ell}(n-1)^i = \frac{(n-1)^{(\ell+1)}-1}{n-2}$. The initial superposition is $\pi_0 = \boldsymbol{e}_1$. We let $A_\sharp^{(\ell)} = I$ and $A_\sigma^{(\ell)} = B^{(\ell)} \cdot \tilde{A}^{(\ell-1)}$, where $\tilde{A}^{(\ell-1)}$ is the transformation acting as $A_\sigma^{(\ell-1)}$ on $Q^{(\ell-1)} \subset Q^{(\ell)}$, and as the identity elsewhere. Instead, $B^{(\ell)}$ is an additional operator working as follows. For any $\tilde{q} \in Q_{acc}^{(\ell-1)}$, let $Q_{\tilde{q}} = \{\tilde{q}_1, \ldots, \tilde{q}_{n-1}\} \subset Q_{acc}^{(\ell)}$ be the set of the $n-1$ new added accepting states associated with $\tilde{q}$. Thus, the operator $B^{(\ell)}$ first acts as F_n on $\{\tilde{q}\} \cup Q_{\tilde{q}}$ for every $\tilde{q} \in Q_{acc}^{(\ell-1)}$, then it measures $\mathscr{O}_\sigma^{(\ell)}$ being the canonical observable on $Q_{acc}^{(\ell-1)} \cup Q_{acc}^{(\ell)}$ plus the identity projector on the remaining basis states. The final observable $\mathscr{O}_\sharp^{(\ell)}$ as usual projects onto the subspace spanned by $Q_{acc}^{(\ell)}$. Actually, the automaton so far constructed does not perfectly comply with the definition of a LQFA given in Section 2.3 since $A_\sigma^{(\ell)}$ is not a unitary matrix. However, [1, Claim 1] ensures that the action of the operator $B^{(\ell)} \cdot \tilde{A}^{(\ell-1)}$ followed by measuring $\tilde{\mathscr{O}}_\sigma^{(\ell-1)}$ (the observable of $M^{(\ell-1)}$ extended to $Q^{(\ell)}$ by the identity projector onto $Q_{acc}^{(\ell)}$) can be expressed as a unitary matrix followed by measuring a suitable observable. This last detail possibly enlarges the dimension of the Hilbert space for $M^{(\ell)}$ by a factor bounded by n^ℓ. The stochastic event induced by $M^{(\ell)}$ will be discussed later.

To clarify the architecture and behavior of this family of automata, we now describe the LQFA $M^{(3)}$ recognizing the language $\sigma^{\geq 3}$ with isolated cut point. We have

$$M^{(3)} = (Q^{(3)}, \{\sigma\}, \pi_0, \{A_\sigma^{(3)}, A_\sharp^{(3)}\}, \{\mathscr{O}_\sigma^{(3)}, \mathscr{O}_\sharp^{(3)}\}, Q_{acc}^{(3)}),$$

where we let the set of basis states be $Q^{(3)} = \{q_0\} \cup Q_{acc}^{(1)} \cup Q_{acc}^{(2)} \cup Q_{acc}^{(3)}$ with $Q_{acc}^{(1)} = \{q_i \mid 1 \leq i \leq n-1\}$, $Q_{acc}^{(2)} = \{q_{i,j} \mid 1 \leq i,j \leq n-1\}$, and $Q_{acc}^{(3)} = \{q_{i,j,k} \mid 1 \leq i,j,k \leq n-1\}$. We remark that $Q_{acc}^{(3)}$ is the set of $(n-1)^3$ accepting basis states of $M^{(3)}$. We can regard basis states as partitioned into three groups reflected by the number of subscripts attributed to each basis state; each group of states is added in a subsequent step of the inductive construction. The general structure of the state (superposition) of $M^{(3)}$ is a norm 1 vector in $\mathbb{C}^{|Q^{(3)}|}$ of the following form, with $\alpha(q)$ denoting the amplitude of the basis state q:

$$\begin{aligned}
&[\alpha(q_0), \alpha(q_1), \alpha(q_{1,1}), [\ldots \alpha(q_{1,1,k}) \ldots], \alpha(q_{1,2}), [\ldots \alpha(q_{1,2,k}) \ldots], \ldots, \alpha(q_{1,n-1}), [\ldots \alpha(q_{1,n-1,k}) \ldots],\\
&\quad \alpha(q_2), \alpha(q_{2,1}), [\ldots \alpha(q_{2,1,k}) \ldots], \alpha(q_{2,2}), [\ldots \alpha(q_{2,2,k}) \ldots], \ldots, \alpha(q_{2,n-1}), [\ldots \alpha(q_{2,n-1,k}) \ldots],\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&\alpha(q_{n-1}), \alpha(q_{n-1,1}), [\ldots \alpha(q_{n-1,1,k}) \ldots], \alpha(q_{n-1,2}), [\ldots \alpha(q_{n-1,2,k}) \ldots], \ldots, \alpha(q_{n-1,n-1}), [\ldots \alpha(q_{n-1,n-1,k}) \ldots]].
\end{aligned}$$

(*) Form of states (superpositions) of $M^{(3)}$.

As usual, we let $\pi_0 = \boldsymbol{e}_1$. The evolution matrices of $M^{(3)}$ are $A_\sharp^{(3)} = I$, while we have $A_\sigma^{(3)} = B^{(3)} \cdot \tilde{B}^{(2)} \cdot \tilde{A}^{(1)}$, where each matrix in the product acts on levels of the basis states as follows: $\tilde{A}^{(1)}$ affects the states in $\{q_0\} \cup Q_{acc}^{(1)}$, $\tilde{B}^{(2)}$ the states in $Q_{acc}^{(1)} \cup Q_{acc}^{(2)}$, and $B^{(3)}$ the states in $Q_{acc}^{(2)} \cup Q_{acc}^{(3)}$. From now on, it will be useful to describe the dynamic of $M^{(3)}$ by displaying the sequence of the stochastic vectors obtained by squaring the amplitudes in the superpositions of the form in (*). In such vectors, the value $|\alpha(q)|^2$ of the component associated with q represents the probability for $M^{(3)}$ of being in the basis state q. This stochastic dynamic description turns out to be appropriate as $M^{(3)}$ uses the canonical observable after each quantum Fourier transform operation. Upon reading a symbol σ, the LQFA $M^{(3)}$ executes $A_\sigma^{(3)}$ followed by measuring $\tilde{\mathscr{O}}_\sigma^{(1)}$: formally, we write $A_\sigma^{(3)} \downarrow \tilde{\mathscr{O}}_\sigma^{(1)}$. This operation distributes the probability differently in the three group of basis states $Q^{(1)}$, $Q_{acc}^{(2)}$ and $Q_{acc}^{(3)}$. In particular, the probability values turn out to be identical within each group of basis states, for each step of computation (except for the initial

superposition π_0). Therefore, the form of the stochastic vector at each step of computation is

$$\begin{aligned}
&[x, x, y, [\cdots z \cdots], y, [\cdots z \cdots], \ \ldots\ , y, [\cdots z \cdots]],\\
&\quad x, y, [\cdots z \cdots], y, [\cdots z \cdots], \ \ldots\ , y, [\cdots z \cdots]],\\
&\qquad\qquad \vdots\\
&x, y, [\cdots z \cdots], y, [\cdots z \cdots], \ \ldots\ , y, [\cdots z \cdots]]],
\end{aligned}$$

where x is the probability value for the states in $Q^{(1)}$, y for the states in $Q^{(2)}_{acc}$, and z for the (accepting) states in $Q^{(3)}_{acc}$. Thus, the current accepting probability is $(n-1)^3 \cdot z$.

Now, let $x(k)$, $y(k)$, and $z(k)$ be the above basis states probabilities after processing the kth input symbol. We are going to establish the dependence of such values from $x(k-1)$, $y(k-1)$, and $z(k-1)$ in order to single out a closed formula for the stochastic event $p_{M^{(3)}}$. To this aim, for reader's ease of mind, a graphical representation is given in Figure 1, of how one step of the evolution-plus-observation $A^{(3)}_\sigma \downarrow \tilde{\mathcal{O}}^{(1)}_\sigma$ affects the probability values in each different group of basis states.

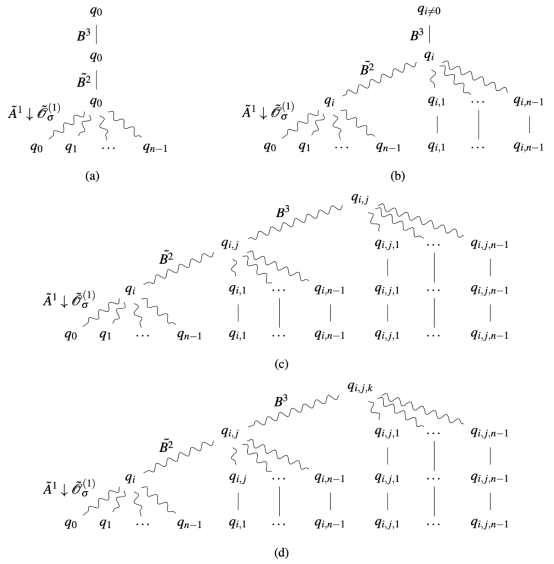


Figure 1: Stochastic representation of a computation step of $M^{(3)}$ on the symbol σ for basis states of different groups. The notation $\tilde{A}^1 \downarrow \tilde{\mathcal{O}}^{(1)}_\sigma$ means that $\tilde{A}^1$ is applied and then the observable $\tilde{\mathcal{O}}^{(1)}_\sigma$ is measured. Wave (straight) edges indicate basis state transitions occurring with probability $\frac{1}{n}$ (with certainty). For instance, the tree in (b) says that, starting from $q_{i\neq 0}$ for a fixed i and after one step of computation, we will observe $M^{(3)}$ in q_0 with probability $\frac{1}{n^2}$. Note that there exist $n-1$ trees of the form (b) leading to q_0.

Let us focus, e.g., on $x(k)$. The probability $x(k)$ depends on $x(k-1)$, $y(k-1)$, and $z(k-1)$ as follows:

- Figure 1(a) shows that the basis state q_0 contributes with $\frac{1}{n} \cdot x(k-1)$.
- Figure 1(b) shows the contribution of each basis states in $Q^{(1)}_{acc}$, which is $\frac{1}{n^2} \cdot x(k-1)$; given that $|Q^{(1)}_{acc}| = (n-1)$, the total contribution is $\frac{(n-1)}{n^2} \cdot x(k-1)$.
- Figure 1(c) shows that the total contribution given by $y(k-1)$ elements (i.e., by the $(n-1)^2$ basis states in $Q^{(2)}_{acc}$) is $\frac{(n-1)^2}{n^3} \cdot y(k-1)$.
- Figure 1(d) shows that the total contribution given by $z(k-1)$ elements (i.e., by the $(n-1)^3$ basis states in $Q^{(3)}_{acc}$) is $\frac{(n-1)^3}{n^3} \cdot z(k-1)$.

By analogous reasonings, we can obtain recurrences for $y(k)$ and $z(k)$, globally yielding the system

$$\begin{cases} x(k) = \frac{1}{n} \cdot x(k-1) + \frac{(n-1)}{n^2} \cdot x(k-1) + \frac{(n-1)^2}{n^3} \cdot y(k-1) + \frac{(n-1)^3}{n^3} \cdot z(k-1) \\ y(k) = \frac{1}{n} \cdot x(k-1) + \frac{(n-1)}{n^2} \cdot y(k-1) + \frac{(n-1)^2}{n^2} \cdot z(k-1) \\ z(k) = \frac{1}{n} \cdot y(k-1) + \frac{(n-1)}{n} \cdot z(k-1). \end{cases} \tag{1}$$

The base for this system of recurrences is the probability distribution after reading the first symbol σ, i.e.:

$$\begin{cases} x(1) = \frac{1}{n} \\ y(1) = z(1) = 0. \end{cases} \tag{2}$$

From the system (1), the reader may verify that at each computation step the probability "shifts" towards the next deeper level of the basis states until reaching the basis states in $Q^{(3)}_{acc}$. In fact, after the first step (yielding probabilities in (2)), only the x-components have non null values. After the second step, only the x- and y-components have values different from 0, while the value of the z-components is still 0. This shows that $M^{(3)}$ rejects with certainty the strings in $\sigma^{\leq 2}$. After the third step, all the components have non null values; in particular, $z(3) = \frac{1}{n^3}$, so that the accepting probability of the string σ^3 attains $|Q^{(3)}_{acc}| \cdot z(3) = (\frac{n-1}{n})^3$. By solving the system (1), we get a closed formula for $z(k)$, with $k \geq 2$, as

$$z(k) = \frac{1}{n(n-1)^2} \cdot \left(1 - \frac{(\frac{2n-2}{n^2})^{k-2} \cdot (n-1)^2 + 1}{(n-1)^2 + 1} \right).$$

This allows us to evaluate the accepting probability of $M^{(3)}$ for any string in σ^* as

$$p_{M^{(3)}}(\sigma^k) = |Q^{(3)}_{acc}| \cdot z(k) = \begin{cases} 0 & \text{if } k \leq 2 \\ \frac{n-1}{n} \cdot \left(1 - \frac{(\frac{2n-2}{n^2})^{k-2} \cdot (n-1)^2 + 1}{(n-1)^2 + 1} \right) & \text{if } k \geq 3. \end{cases} \tag{3}$$

Equation (3) shows that $M^{(3)}$ recognizes $\sigma^{\geq 3}$ with isolated cut point. Clearly, the stochastic event induced by $M^{(3)}$ depends on the number n of the basis states of $M^{(1)}$, the initial automaton of the inductive construction. Figure 2 displays $p_{M^{(3)}}$ for some values of n. As expected, the higher n grows, the better the isolation around the cut point becomes.

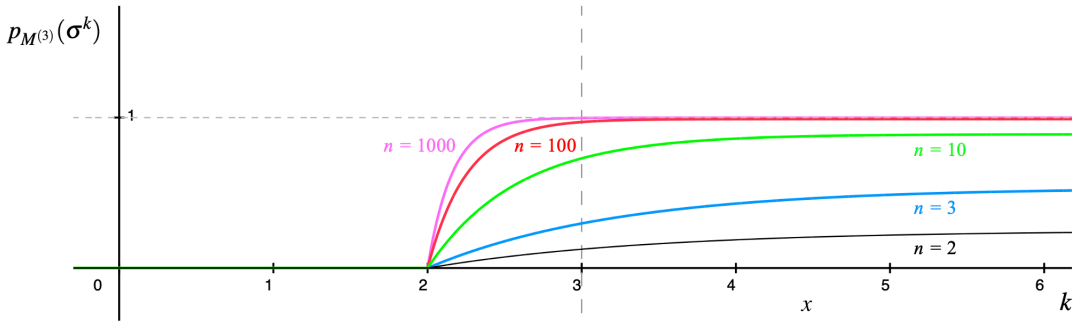


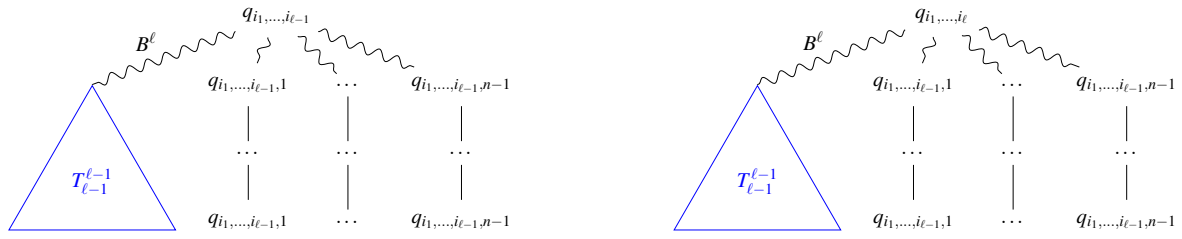
Figure 2: The ("continuous version" of the) stochastic events induced by $M^{(3)}$ according to Equation 3, for different values of the number n of basis states of $M^{(1)}$, the base module inductively leading to $M^{(3)}$.

Now, we consider the general LQFA $M^{(\ell)}$, and derive the system of recurrences for its stochastic dynamic. The set of basis states of $M^{(\ell)}$ is now partitioned into ℓ groups. For $1 \leq h \leq \ell$, we denote by $x_h(k)$ the probability for $M^{(\ell)}$ of being in a basis state of the hth group, after processing k input symbols. The system of recurrences for $M^{(\ell)}$ generalizes the system (1) as follows:

$$\begin{cases} x_1(k) = \frac{1}{n} \cdot x_1(k-1) + \sum_{j=1}^{\ell-1} \frac{(n-1)^j}{n^{j+1}} \cdot x_j(k-1) + \frac{(n-1)^\ell}{n^\ell} \cdot x_\ell(k-1) \\ x_2(k) = \left(\sum_{j=0}^{\ell-2} \frac{(n-1)^j}{n^{j+1}} \cdot x_{j+1}(k-1) \right) + \frac{(n-1)^{\ell-1}}{n^{\ell-1}} \cdot x_\ell(k-1) \\ \quad \vdots \\ x_h(k) = \left(\sum_{j=0}^{\ell-h} \frac{(n-1)^j}{n^{j+1}} \cdot x_{j+h-1}(k-1) \right) + \frac{(n-1)^{\ell-h+1}}{n^{\ell-h+1}} \cdot x_\ell(k-1) \\ \quad \vdots \\ x_\ell(k) = \frac{1}{n} \cdot x_{\ell-1}(k-1) + \frac{(n-1)}{n} \cdot x_\ell(k-1), \end{cases} \tag{4}$$

with initial values $x_1(1) = \frac{1}{n}$, and $x_h(1) = 0$ for every $2 \leq h \leq \ell$. We show the validity of this system of recurrences by induction, having, e.g., the system (1) for the automaton $M^{(3)}$ as base case. By inductive hypothesis, we assume the system of recurrences for $M^{(\ell-1)}$, and we build the system (4) for $M^{(\ell)}$. We consider the set of trees representing one step of the computation of our automata, starting from basis states of different groups. E.g., Figure 1 displays the four different types of trees for $M^{(3)}$, one per each group of basis states, plus one for the evolution from the state q_0. So, for $M^{(\ell)}$ we are going to provide ℓ of such trees, plus the one for q_0. Let us explain how to obtain them from the trees of $M^{(\ell-1)}$. Let $T_j^{(\ell-1)}$ be a tree representing one step of the evolution of $M^{(\ell-1)}$ on a basis state of group $1 \leq j \leq \ell-1$, namely, a basis state from $Q_{acc}^{(j)}$. Moreover, let $T_0^{(\ell-1)}$ be the tree for q_0. The evolution for $M^{(\ell)}$ is $A_\sigma^{(\ell)} = B^{(\ell)} \cdot \tilde{A}^{(\ell-1)}$. Thus, the behavior of $M^{(\ell)}$ is described by $\ell+1$ trees with the following structure:

- The trees $T_j^{(\ell)}$ for $0 \leq j < \ell-1$ are basically the trees $T_j^{(\ell-1)}$ with a preliminary step due to the action of $B^{(\ell)}$. Since in these trees the root is labeled by a basis state of level $j < \ell-1$, such a preliminary step coincides with the identity evolution.
- Even the trees $T_{\ell-1}^{(\ell)}$ and $T_\ell^{(\ell)}$ have the action of $B^{(\ell)}$ as a preliminary step. However, in these cases, $B^{(\ell)}$ acts as F_n on the basis states of groups $\ell-1$ in the tree $T_{\ell-1}^{(\ell)}$, and ℓ in the tree $T_\ell^{(\ell)}$. The structure of these two trees, both containing the tree $T_{\ell-1}^{\ell-1}$ as a sub-tree, is presented in Figure 3.



(a) (b)

Figure 3: The form of the tree $T_{\ell-1}^{\ell}$ in (a), and of the tree T_{ℓ}^{ℓ} in (b) for the automaton $M^{(\ell)}$. Within both these two trees, the tree $T_{\ell-1}^{\ell-1}$ turns out to be a sub-tree.

It is now possible to properly justify the system (4) by using the induction step. Starting from the system of recurrences for $M^{(\ell-1)}$, we show how it modifies towards the system for $M^{(\ell)}$. Clearly, a new recurrence for $x_\ell(k)$ (i.e., the probabilities for basis states of group ℓ, the accepting states for $M^{(\ell)}$) is added at the end of the system. This component receives contributions only from the trees $T_{\ell-1}^{(\ell)}$ and $T_{\ell}^{(\ell)}$ weighted, respectively, by $x_{\ell-1}(k-1)$ and $x_\ell(k-1)$. Precisely, from the former tree we get the contribution $\frac{1}{n}\cdot x_{\ell-1}(k-1)$, from the latter ($n-1$ different trees) the contribution is $\frac{n-1}{n}\cdot x_\ell(k-1)$. For $x_h(k)$, with $1\leq h\leq \ell-1$, we note that the only modified contribution is the one carried by $x_{\ell-1}(k-1)$; moreover a new contribution from $x_\ell(k-1)$ is added. Even in this case, the trees $T_{\ell-1}^{(\ell)}$ and $T_{\ell}^{(\ell)}$ account for these modifications: the new coefficient of $x_{\ell-1}(k-1)$ is the old one for $x_{\ell-1}(k-1)$ (i.e., the one associated with $x_{\ell-1}(k-1)$ in the system for $M^{(\ell-1)}$) multiplied by $\frac{1}{n}$, while the coefficient of the new contribution $x_\ell(k-1)$ is the old one for $x_{\ell-1}(k-1)$ multiplied by $\frac{(n-1)}{n}$.

By simply applying repeated substitutions in the system (4), one may verify that, for $1\leq k\leq \ell$, the value $x_k(k)$ always equals $\frac{1}{n^k}$, while we have $x_{k+1}(k)=\cdots=x_\ell(k)=0$. Nevertheless, this implies that the acceptance probability of $M^{(\ell)}$ for the string σ^k is zero for $k<\ell$, while is $|Q_{acc}^{(\ell)}|\cdot x_\ell(\ell)=(\frac{n-1}{n})^\ell$ for $k=\ell$. We are now going to prove that for the strings in the language $\sigma^{\geq\ell}$ the acceptance probability never goes below $(\frac{n-1}{n})^\ell$. To this aim, it suffices to show

Theorem 2. *On the input string $\sigma^{\ell+s}$, with $s\geq 0$, the probability for $M^{(\ell)}$ of being in one of the accepting basis states in $Q_{acc}^{(\ell)}$ while processing the suffix σ^s is greater than or equal to $\frac{1}{n^\ell}$.*

Proof. We split the proof into two parts, both proved by induction. In the first part, we focus on the input prefix σ^ℓ. We show by induction on $1\leq k\leq \ell$ that $x_h(k)\geq\frac{1}{n^k}$ in the system (4) holds true for every $1\leq h\leq k$. This will enables us to obtain that $x_1(\ell),\ldots,x_\ell(\ell)\geq\frac{1}{n^\ell}$. For the base case $k=1$, we recall that $x_1(1)=\frac{1}{n}$. So, let us assume by inductive hypothesis that $x_h(k)\geq\frac{1}{n^k}$ for a given $k<\ell$ and every $1\leq h\leq k$, and prove the property for $k+1$. From the system (4), we have

$$x_h(k+1)=\frac{1}{n}\cdot x_{h-1}(k)+\frac{n-1}{n^2}\cdot x_h(k)+\frac{(n-1)^2}{n^3}\cdot x_{h+1}(k)+\cdots+\frac{(n-1)^{k-h+1}}{n^{k-h+2}}\cdot x_k(k)+\cdots+\frac{(n-1)^{\ell-h+1}}{n^{\ell-h+1}}\cdot x_\ell(k).$$

Since $x_h(k)\geq\frac{1}{n^k}$ for $1\leq h\leq k$, and 0 otherwise, we can bound $x_h(k+1)$ from below as

$$x_h(k+1)\geq\frac{1}{n^k}\cdot\frac{1}{n}\cdot\left(1+\frac{n-1}{n}+\frac{(n-1)^2}{n^2}+\cdots+\frac{(n-1)^{k-h+1}}{n^{k-h+1}}\right)\geq\frac{1}{n^{k+1}}.$$

Now, the second part of the proof comes, where we show, again by induction on k, that $x_h(k)\geq\frac{1}{n^\ell}$ for $k\geq\ell$ and $1\leq h\leq\ell$. By the first part of the proof, we have $x_1(\ell),x_2(\ell),\ldots x_\ell(\ell)\geq\frac{1}{n^\ell}$, and so the base

case holds true. We prove $x_h(k+1) \geq \frac{1}{n^\ell}$ assuming such a property for k by inductive hypothesis. From the system (4), we get

$$x_h(k+1) = \frac{1}{n} \cdot x_{h-1}(k) + \frac{n-1}{n^2} \cdot x_h(k) + \frac{(n-1)^2}{n^3} \cdot x_{h+1}(k) + \ldots + \frac{(n-1)^{\ell-h}}{n^{\ell-h+1}} \cdot x_{\ell-1}(k) + \frac{(n-1)^{\ell-h+1}}{n^{\ell-h+1}} \cdot x_\ell(k).$$

Since we are assuming all $x_j(k)$'s to be greater than or equal to $\frac{1}{n^\ell}$, we can bound $x_h(k+1)$ from below as

$$x_h(k+1) \geq \frac{1}{n^\ell} \cdot \left(\frac{1}{n} + \frac{n-1}{n^2} + \frac{(n-1)^2}{n^3} + \cdots + \frac{(n-1)^{\ell-h}}{n^{\ell-h+1}} + \left(\frac{n-1}{n} \right)^{\ell-h+1} \right) = \frac{1}{n^\ell},$$

whence, the claimed result follows. □

We can conclude that $M^{(\ell)}$ induces the following stochastic event:

$$p_{M^{(\ell)}}(\sigma^k) = |Q^{(\ell)}_{acc}| \cdot x_\ell(k) \begin{cases} = 0 & \text{if } k < \ell \\ \geq \left(\frac{n-1}{n}\right)^\ell & \text{if } k \geq \ell. \end{cases} \tag{5}$$

This shows that the automaton $M^{(\ell)}$ recognizes $\sigma^{\geq \ell}$ with isolated cut point and $n^{O(\ell)}$ basis states. As expected, for $n \to \infty$, the event in (5) approximates a deterministic behavior. In fact, for growing values of n, we have $p_{M^{(\ell)}}(\sigma^k) \to 1$ for $k \geq \ell$, and $p_{M^{(\ell)}}(\sigma^k) = 0$ for $k < \ell$.

To sum up, let us get back to our initial purpose, i.e., building an isolated cut point LQFA for the language $\sigma^{\geq T}$. Such a LQFA is obtained by pushing T steps ahead from $M^{(1)}$ the inductive construction to finally get the LQFA $M^{(T)}$. As noted, $M^{(T)}$ features $n^{O(T)}$ basis states, n being the number of basis states of $M^{(1)}$. From Equation (5), we can fix a cut point $\frac{1}{2} \cdot \left(\frac{n-1}{n}\right)^T$ isolated by $\frac{1}{2} \cdot \left(\frac{n-1}{n}\right)^T$. By increasing n, we widen such an isolation, tending to a deterministic recognition of the language $\sigma^{\geq T}$.

Focusing on the size of $M^{(T)}$, we observe that its number of basis states exponentially depends on T. As a matter of fact, we can avoid such an exponential blow up by noticing that even the LQFA $M^{(3)}$ can actually accept with isolated cut point the language $\sigma^{\geq T}$, for $T \geq 4$. This is due to the fact that the stochastic event induced by $M^{(3)}$ is an increasing function, as one may readily infer from Equation (3) and Figure 2. By this property, we can fix the isolated cut point between $p_{M^{(3)}}(\sigma^{T-1})$ and $p_{M^{(3)}}(\sigma^T)$, thus recognizing $\sigma^{\geq T}$ with $n^{O(1)}$ basis states, not depending on T any more. Nevertheless, such a dramatic size reduction comes at a price. In fact, the isolation around the cut point shrinks from $\frac{1}{2} \cdot \left(\frac{n-1}{n}\right)^T$ to $\frac{p_{M^{(3)}}(\sigma^T) - p_{M^{(3)}}(\sigma^{T-1})}{2} = \frac{1}{2} \cdot \left(\frac{2}{n}\right)^{T-3} \cdot \left(\frac{n-1}{n}\right)^{T-1} \cdot \left(\frac{n+1}{n}\right)$. This isolation vanishes as n grows, thus suggesting to consider small values of n. E.g., for $n = 2$ we obtain an isolation of $\frac{3}{2} \cdot \left(\frac{1}{2}\right)^T$; for $n = 3$ we get $\frac{27}{8} \cdot \left(\frac{4}{9}\right)^T$.

4 Isolated Cut Point LQFAs for Unary Regular Languages

Here, we are going to use the LQFAs designed in the previous section as modules in a more general construction yielding isolated cut point LQFAs for unary regular languages. This investigation is inspired by [5] where the same problem is tackled for MM-QFAs. Our result constructively shows that isolated cut point MM-QFAs and LQFAs are equivalent on unary inputs, in sharp contrast to the case for general alphabets where MM-QFAs outperform LQFAs (see Section 2.3).

We start by observing that, according to Theorem 1, any unary regular language $L \subseteq \sigma^*$ can viewed as the disjoint union of two unary languages, namely, the finite language $L_T = L \cap \sigma^{\leq T}$ plus the ultimately periodic language $L_P = L \cap \sigma^{\geq T+1}$. So, we are going to design two LQFA modules recognizing these two languages with isolated cut point, and then suitably assemble such modules into a final isolated cut point LQFA A_L for the unary regular language L.

The finite language L_T: We define the "$(T+1)$-periodic continuation" $L_{T^\circlearrowleft}$ of L_T, namely, the language obtained from L_T by adding all the strings of the form $\sigma^{i+h\cdot(T+1)}$, with $h \geq 0$, for $\sigma^i \in L_T$. Formally, $L_{T^\circlearrowleft} = \{\sigma^{i+h\cdot(T+1)} \mid h \geq 0 \text{ and } \sigma^i \in L_T\}$. Clearly, $L_{T^\circlearrowleft}$ is a periodic language of period $(T+1)$, and we have that $L_T = L_{T^\circlearrowleft} \cap \sigma^{\leq T}$. Therefore, in order to recognize L_T, we start by defining the isolated cut point LQFA $A_{T^\circlearrowleft}$ for $L_{T^\circlearrowleft}$. We let $A_{T^\circlearrowleft} = (Q, \{\sigma\}, \pi_0, \{U(\sigma), U(\sharp)\}, \{\mathscr{O}_\sigma, \mathscr{O}_\sharp\}, Q_{acc})$, where: $Q = \{q_0, \ldots, q_T\}$ is the set of basis states, $Q_{acc} = \{q_i \mid 0 \leq i \leq T \text{ and } \sigma^i \in L_T\}$ is the set of accepting basis states, $\pi_0 = \boldsymbol{e}_1$ is the initial superposition, $U(\sigma) = S$, where $S \in \{0,1\}^{(T+1)\times(T+1)}$ is the matrix representing the cyclic permutation: S has 1 at the $(i,i+1)$th entries for $1 \leq i \leq T$ and at the $(T+1,1)$th entry, all the other entries are 0, $U(\sharp) = I^{(T+1)}$, $\mathscr{O}_\sigma$ is the observable having the identity as sole projector, $\mathscr{O}_\sharp$ is the usual final observable projecting onto the subspace spanned by Q_{acc}. Given the observable $\mathscr{O}_\sigma$, we have that $A_{T^\circlearrowleft}$ is basically a MO-QFA whose induced event writes as $p_{A_{T^\circlearrowleft}}(\sigma^k) = \|\pi_0 \cdot U(\sigma)^k \cdot U(\sharp) \cdot P_{acc}(\sharp)\|^2$. After processing the input $\sigma^k\sharp$, the state $\xi(k)$ of $A_{T^\circlearrowleft}$ is

$$\xi(k) = \pi_0 \cdot U(\sigma)^k \cdot U(\sharp) = \boldsymbol{e}_1 \cdot U(\sigma)^k \cdot U(\sharp) = \boldsymbol{e}_{(k \bmod (T+1))+1}. \tag{6}$$

Let us now discuss measuring by the final observable, i.e., the action of the projector $P_{acc}(\sharp)$ on the final superposition $\xi(k)$. By (6), $\xi(k)$ is $\boldsymbol{e}_{(k \bmod (T+1))+1}$, representing the basis state $q_{k \bmod (T+1)}$. By definition of Q_{acc} we have that $q_{k \bmod (T+1)}$ is an accepting state if and only if $\sigma^{k \bmod (T+1)} \in L_T$ if and only if $\sigma^k \in L_{T^\circlearrowleft}$. Therefore, we can rewrite the stochastic event induced by $A^{T^\circlearrowleft}$ as $p_{A_{T^\circlearrowleft}}(\sigma^k) = \|\xi(k) \cdot P_{acc}(\sharp)\|^2 = 1$ if $\sigma^k \in L_{T^\circlearrowleft}$, and 0 otherwise. Whence, the LQFA $A_{T^\circlearrowleft}$ recognizes $L_{T^\circlearrowleft}$ by a deterministic event. Now, we need $A_{T^\circlearrowleft}$ to work simultaneously with a module which checks whether or not the input string has length not exceeding T, so that the resulting accepted language is $L_{T^\circlearrowleft} \cap \sigma^{\leq T} = L_T$. Such a module can be obtained by complementing the LQFA $M^{(T+1)}$ for $\sigma^{\geq T+1}$ presented in Section 3 (basically, by taking $Q \setminus Q_{acc}$ as the set of accepting basis states). The resulting LQFA $\overline{M}^{(T+1)}$ induces the complement of the event in Equation (5) with $\ell = T+1$:

$$p_{\overline{M}^{(T+1)}}(\sigma^k) = 1 - p_{M^{(T+1)}}(\sigma^k) \begin{cases} = 1 & \text{if } k \leq T \\ \leq 1 - \left(\frac{n-1}{n}\right)^{(T+1)} & \text{if } k \geq T+1, \end{cases}$$

thus recognizing the language $\sigma^{\leq T}$ with isolated cut point and $n^{O(T)}$ basis states. Finally, we build the LQFA $A_{T^\circlearrowleft} \otimes \overline{M}^{(T+1)}$ (basically by taking the direct product component wise of the two LQFAs $A_{T^\circlearrowleft}$ and $\overline{M}^{(T+1)}$) inducing the product event

$$p_{A_{T^\circlearrowleft} \otimes \overline{M}^{(T+1)}}(\sigma^k) = p_{A_{T^\circlearrowleft}} \cdot p_{\overline{M}^{(T+1)}}(\sigma^k) \begin{cases} = 1 & \text{if } \sigma^k \in L_T \\ \leq 1 - \left(\frac{n-1}{n}\right)^{(T+1)} & \text{otherwise,} \end{cases}$$

defining L_T with $(T+1)\cdot n^{O(T)}$ basis states, and cut point $1 - \frac{1}{2}\cdot\left(\frac{n-1}{n}\right)^{(T+1)}$ isolated by $\frac{1}{2}\cdot\left(\frac{n-1}{n}\right)^{(T+1)}$. Notice that, for large values of n, the LQFA $A_{T^\circlearrowleft} \otimes \overline{M}^{(T+1)}$ approximates a deterministic recognition of L_T.

The ultimately periodic language L_P: It suites our goal to rewrite L_P as $L_P = L_{P^\circlearrowleft} \cap \sigma^{\geq T+1}$, where we let $L_{P^\circlearrowleft} = \{\sigma^{(T+1+i) \bmod P + h\cdot P} \mid 0 \leq i < P,\ h \geq 0, \text{ and } \sigma^{T+1+i} \in L_P\}$. Clearly, $L_{P^\circlearrowleft}$ is a periodic language of period P. So, for recognizing L_P, we first focus on building the isolated cut point LQFA $A_{P^\circlearrowleft}$ for $L_{P^\circlearrowleft}$. We let $A_{P^\circlearrowleft} = (Q, \{\sigma\}, \pi_0, \{U(\sigma), U(\sharp)\}, \{\mathscr{O}_\sigma, \mathscr{O}_\sharp\}, Q_{acc})$, where: $Q = \{q_0, \ldots, q_{P-1}\}$ is the set of basis states, $Q_{acc} = \{q_i \mid 0 \leq i < P \text{ and } \sigma^i \in L_{P^\circlearrowleft}\}$ is the set of accepting basis states, $\pi_0 = \boldsymbol{e}_1$ is the initial superposition, $U(\sigma) = S$, where $S \in \{0,1\}^{P\times P}$ is the cyclic permutation matrix, $U(\sharp) = I^{(P)}$, $\mathscr{O}_\sigma$

is the observable having the identity as sole projector, $\mathscr{O}_\sharp$ is the usual final observable projecting onto the subspace spanned by Q_{acc}. Given the observable $\mathscr{O}_\sigma$, we have that $A_{P^\circlearrowleft}$ is basically a MO-QFA whose induced event writes as $p_{A^{P\circlearrowleft}}(\sigma^k) = \|\pi_0 \cdot U(\sigma)^k \cdot U(\sharp) \cdot P_{acc}(\sharp)\|^2$. After processing the input $\sigma^k\sharp$, the state $\xi(k)$ of $A_{P^\circlearrowleft}$ is

$$\xi(k) = \pi_0 \cdot U(\sigma)^k \cdot U(\sharp) = \boldsymbol{e}_1 \cdot U(\sigma)^k \cdot U(\sharp) = \boldsymbol{e}_{(k \bmod P)+1}. \tag{7}$$

Let us now measure the final observable on the final superposition $\xi(k)$. By (7), $\xi(k)$ is $\boldsymbol{e}_{(k \bmod P)+1}$, representing the basis state $q_{k \bmod P}$. By definition of Q_{acc}, we have that $q_{k \bmod P}$ is an accepting state if and only if $\sigma^k \in L_{P^\circlearrowleft}$. Therefore, the stochastic event induced by $A_{P^\circlearrowleft}$ is $p_{P^\circlearrowleft}(\sigma^k) = \|\xi(k) \cdot P_{acc}(\sharp)\|^2 = 1$, if $\sigma^k \in L_{P^\circlearrowleft}$, and 0 otherwise. whence, the LQFA $A_{P^\circlearrowleft}$ recognizes $L_{P^\circlearrowleft}$ by a deterministic event. Now, we need $A_{P^\circlearrowleft}$ to work simultaneously with a module which checks whether or not the input string has length exceeding T, so that the resulting accepted language is $L_{P^\circlearrowleft} \cap \sigma^{\geq T+1} = L_P$. Such a module is the LQFA $M^{(T+1)}$ for $\sigma^{\geq T+1}$ presented in Section 3, and inducing the event

$$p_{M^{(T+1)}}(\sigma^k) \begin{cases} \geq (\frac{n-1}{n})^{T+1} & \text{if } k \geq T+1 \\ = 0 & k \leq T, \end{cases}$$

thus recognizing the language $\sigma^{\geq T+1}$ with isolated cut point and $n^{O(T)}$ basis states. Finally, we build the LQFA $A_{P^\circlearrowleft} \otimes M^{(T+1)}$, inducing the product event

$$p_{A_{P^\circlearrowleft} \otimes M^{(T+1)}}(\sigma^k) = p_{A_{P^\circlearrowleft}} \cdot p_{M^{(T+1)}}(\sigma^k) \begin{cases} \geq (\frac{n-1}{n})^{T+1} & \text{if } \sigma^k \in L_P \\ = 0 & \text{otherwise,} \end{cases}$$

defining L_P with $P \cdot n^{O(T)}$ basis states, and cut point $\frac{1}{2} \cdot \left(\frac{n-1}{n}\right)^{(T+1)}$ isolated by $\frac{1}{2} \cdot \left(\frac{n-1}{n}\right)^{(T+1)}$. Notice that, for large values of n, the LQFA $A_{P^\circlearrowleft} \otimes M^{(T+1)}$ approximates a deterministic recognition of L_P.

Putting things together: We are now ready to suitably assemble the two LQFAs $A_T = A_{T^\circlearrowleft} \otimes \overline{M}^{(T+1)}$ and $A_P = A_{P^\circlearrowleft} \otimes M^{(T+1)}$ so far described to obtain an isolated cut point LQFA A_L for the unary regular language L. We notice that $L = L_T \cup L_P = (L_T^c \cap L_P^c)^c$. This suggests first to construct LQFAs for L_T^c and L_P^c by building $\overline{A}_T$ and $\overline{A}_P$ inducing the complement events $p_{\overline{A}_T} = 1 - p_{A_T}$ and $p_{\overline{A}_P} = 1 - p_{A_P}$, respectively. Next, to account for the intersection, we construct the LQFA $\overline{A}_L = \overline{A}_T \otimes \overline{A}_P$ inducing the product event $p_{\overline{A}_L} = (1 - p_{A_T}) \cdot (1 - p_{A_P})$. Finally, the desired LQFA A_L will be obtained by complementing $\overline{A}_L$, so that $p_{A_L} = (1 - p_{\overline{A}_L}) = 1 - (1 - p_{A_T}) \cdot (1 - p_{A_P}) = p_{A_T} + p_{A_P} - p_{A_T} \cdot p_{A_P}$.

Let us now explain how p_{A_L} behaves on input string σ^k:

- $\sigma^k \in L = L_T \cup L_P$: Clearly, we have either $\sigma^k \in L_T$ or $\sigma^k \in L_P$. Suppose $\sigma^k \in L_T$. Then, we have that $p_{A_T}(\sigma^k) = 1$ since $A_T = A_{T^\circlearrowleft} \otimes \overline{M}^{(T+1)}$ and both its sub-modules will accept with certainty; correspondingly, $p_{A_P}(\sigma^k) = 0$ since $A_P = A_{P^\circlearrowleft} \otimes M^{(T+1)}$ and the sub-module $M^{(T+1)}$ accepts with 0 probability the input strings of length less than or equal to T. Globally, we have $p_{A_L}(\sigma^k) = 1$. Suppose $\sigma^k \in L_P$. Then, we have that $p_{A_P}(\sigma^k) \geq \left(\frac{n-1}{n}\right)^{T+1}$ since $A_{P^\circlearrowleft}$ accepts with certainty, while the sub-module $M^{(T+1)}$ accepts with probability not less than $\left(\frac{n-1}{n}\right)^{T+1}$. Let us now focus on A_T. The sub-module $A_{T^\circlearrowleft}$ could accept with probability either 0 or 1. In the former case, globally we have $p_{A_L}(\sigma^k) \geq \left(\frac{n-1}{n}\right)^{T+1}$, in the latter, the sub-module $\overline{M}^{(T+1)}$ accepts with a probability bounded above by $1 - \left(\frac{n-1}{n}\right)^{T+1}$. By letting $(1-y)$ the acceptance probability of $\overline{M}^{(T+1)}$, with $0 \leq y \leq \left(\frac{n-1}{n}\right)^{T+1}$, we get $p_{A_L}(\sigma^k) \geq \left(\frac{n-1}{n}\right)^{T+1} + (1-y) - \left(\frac{n-1}{n}\right)^{T+1} \cdot (1-y) \geq \left(\frac{n-1}{n}\right)^{T+1}$.
  In conclusion, for any $\sigma^k \in L$, we have $p_{A_L}(\sigma^k) \geq \left(\frac{n-1}{n}\right)^{T+1}$.

- $\sigma^k \notin L = L_T \cup L_P$: Clearly, both $\sigma^k \notin L_T$ and $\sigma^k \notin L_P$. By assuming $k \leq T$, we must have $\sigma^k \notin L_{T^\circlearrowleft}$. Therfore, the sole acceptance probability contribution could come from the module $A_P = A_{P^\circlearrowleft} \otimes M^{(T+1)}$. However, since $k \leq T$, the sub-module $M^{(T+1)}$ accepts with 0 probability. So, $p_{A_L}(\sigma^k) = 0$. Instead, by assuming $k \geq T+1$, we must have that $\sigma^k \notin L_{P^\circlearrowleft}$. Thus, the sole acceptance probability could come from the module A_T. However, the acceptance probability yielded by the sub-module $\overline{M}^{(T+1)}$ turns out to be at most $1 - \left(\frac{n-1}{n}\right)^{T+1}$.

  In conclusion, for any $\sigma^k \notin L$, we have $p_{A_L}(\sigma^k) \leq 1 - \left(\frac{n-1}{n}\right)^{T+1}$.

Summing up, the stochastic event induced by the LQFA A_L is

$$p_{A_L}(\sigma^k) \begin{cases} \geq \left(\frac{n-1}{n}\right)^{T+1} & \text{if } \sigma^k \in L \\ \leq 1 - \left(\frac{n-1}{n}\right)^{T+1} & \text{otherwise.} \end{cases} \tag{8}$$

By the event in Equation (8), we get that A_L recognizes L with the following cut point and isolation radius:

$$\lambda = \frac{1}{2} \cdot \left(\left(\frac{n-1}{n}\right)^{T+1} + 1 - \left(\frac{n-1}{n}\right)^{T+1} \right) = \frac{1}{2}, \quad \rho = \frac{1}{2} \cdot \left(\left(\frac{n-1}{n}\right)^{T+1} - 1 + \left(\frac{n-1}{n}\right)^{T+1} \right) = \left(\frac{n-1}{n}\right)^{T+1} - \frac{1}{2}.$$

Clearly, to have an isolation around λ, we must require that $\rho > 0$. This can always be achieved on any $T > 0$ by imposing $\left(\frac{n-1}{n}\right)^{T+1} > \frac{1}{2}$, which is attained whenever $n > \frac{1}{1 - \sqrt[T+1]{\frac{1}{2}}}$. This latter condition is satisfied, e.g., by letting $n = 4T$ for any $T > 0$. Nevertheless, the isolation radius ρ tends to $\frac{1}{2}$ as n grows.

Let us inspect the size of the LQFA $A_L = \overline{\overline{A}_T \otimes \overline{A}_P}$. As above pointed out, A_T and A_P have, respectively, $(T+1) \cdot n^{O(T)}$ and $P \cdot n^{O(T)}$ basis states. The complements $\overline{A}_T$ and $\overline{A}_P$ maintain the same number of basis states, while the product $\overline{A}_T \otimes \overline{A}_P$ requires $((T+1) \cdot n^{O(T)}) \cdot (P \cdot n^{O(T)}) \leq T \cdot P \cdot n^{O(T)}$ basis states. The final complement $\overline{\overline{A}_T \otimes \overline{A}_P}$ maintains the same number of basis states. By replacing n with $4T$, as above suggested, the number of basis states of the isolated cut point LQFA A_L for L becomes $P \cdot T^{O(T)}$.

5 Conclusions

In this work, we have exhibited a modular framework for building isolated cut point LQFAs for unary regular languages. By suitably adapting to the unary case an inductive construction in [1, 14], we have first designed LQFAs discriminating unary inputs on the basis of their length. These devices have then been plugged into two sub-modules recognizing the finite part and the ultimately periodic part any unary regular language consists of. The resulting LQFA recognizes a unary regular language L with isolated cut point $\frac{1}{2}$, and a number of basis states which is exponential in the number of states of the minimal DFA for L. In spite of this exponential size blow up, it should be stressed that more restricted models of quantum finite automata in the literature, such as MO-QFAs, cannot recognize all unary regular languages. On the other hand, a linear amount of basis states is sufficient for the more powerful model of isolated cut point MM-QFAs [5]. Thus, it would be worth investigating whether a more size efficient construction for unary LQFAs could be provided. Another interesting line of research might explore the descriptional power (see, e.g., [2, 8, 12, 13] for topics in descriptional complexity) of isolated cut point LQFAs with respect to other relevant classes of subregular languages such as, e.g., commutative regular languages [20].

Acknowledgements. The authors wish to thank the anonymous referees for their valuable comments.

References

[1] Andris Ambainis, Martin Beaudry, Marats Golovkins, Arnolds Kikusts, Mark Mercer & Denis Thérien (2006): *Algebraic results on quantum automata*. *Theory of Computing Systems* 39, pp. 165–188, doi:10.1007/s00224-005-1263-x.

[2] Zuzana Bednárová, Viliam Geffert, Carlo Mereghetti & Beatrice Palano (2017): *Boolean language operations on nondeterministic automata with a pushdown of constant height*. *Journal of Computer and System Sciences* 90, pp. 99 – 114, doi:10.1016/j.jcss.2017.06.007.

[3] Maria Paola Bianchi, Carlo Mereghetti & Beatrice Palano (2014): *Size lower bounds for quantum automata*. *Theoretical Computer Science* 551, p. 102 – 115, doi:10.1016/j.tcs.2014.07.004.

[4] Maria Paola Bianchi, Carlo Mereghetti & Beatrice Palano (2017): *Quantum finite automata: Advances on Bertoni's ideas*. *Theoretical Computer Science* 664, pp. 39–53, doi:10.1016/j.tcs.2016.01.045.

[5] Maria Paola Bianchi & Beatrice Palano (2010): *Behaviours of unary quantum automata*. *Fundamenta Informaticae* 104, pp. 1–15, doi:10.3233/FI-2010-333.

[6] Alex Brodsky & Nicholas Pippenger (2002): *Characterizations of 1-way quantum finite automata*. *SIAM Journal on Computing* 31, pp. 1456–1478, doi:10.1137/S0097539799353443.

[7] Alessandro Candeloro, Carlo Mereghetti, Beatrice Palano, Simone Cialdi, Matteo G. A. Paris & Stefano Olivares (2021): *An enhanced photonic quantum finite automaton*. *Applied Sciences* 11, p. 8768, doi:10.3390/app11188768.

[8] Markus Holzer & Martin Kutrib (2011): *Descriptional and computational complexity of finite automata - A survey*. *Information and Computation* 209, pp. 456–470, doi:10.1016/j.ic.2010.11.013.

[9] John E. Hopcroft, Rajeev Motwani & Jeffrey D. Ullman (2006): *Introduction to Automata Theory, Languages, and Computation*, 3rd edition. Addison-Wesley.

[10] R.I.G. Hughes (1992): *The Structure and Interpretation of Quantum Mechanics*. Harvard University Press.

[11] Attila Kondacs & John Watrous (1997): *On the power of quantum finite state automata*. In: *Proc. 38th Symp. on Found. Comp. Sci. (FOCS)*, IEEE Computer Society, pp. 66–75, doi:10.1109/SFCS.1997.646094.

[12] Martin Kutrib, Andreas Malcher, Carlo Mereghetti & Beatrice Palano (2020): *Deterministic and nondeterministic iterated uniform finite-state transducers: computational and descriptional power*. In: *Proc. 16th Int. Conf. Comp. Europe (CiE 2020)*, *LNCS* 12098, Springer, pp. 87–99, doi:10.1007/978-3-030-51466-2_8.

[13] Martin Kutrib, Andreas Malcher, Carlo Mereghetti & Beatrice Palano (2020): *Iterated uniform finite-state transducers: descriptional complexity of nondeterminism and two-way motion*. In: *Proc. 22nd Int. Conf. Des. Comp. Form. Sys. (DCFS 2020)*, *LNCS* 12442, Springer, pp. 117–129, doi:10.1007/978-3-030-62536-8_10.

[14] Mark Mercer (2007): *Applications of Algebraic Automata Theory to Quantum Finite Automata*. Ph.D. thesis, McGill University, Montreal, Quebec, Canada.

[15] Carlo Mereghetti, Beatrice Palano, Simone Cialdi, Valeria Vento, Matteo G. A. Paris & Stefano Olivares (2020): *Photonic realization of a quantum finite automaton*. *Physical Review Research* 2, p. 013089, doi:10.1103/PhysRevResearch.2.013089.

[16] Cristopher Moore & James P. Crutchfield (2000): *Quantum automata and quantum grammars*. *Theoretical Computer Science* 237, pp. 275–306, doi:10.1016/S0304-3975(98)00191-1.

[17] Rohit Parikh (1966): *On context-free languages*. *J. ACM* 13, pp. 570–581, doi:10.1145/321356.321364.

[18] Michael O. Rabin (1963): *Probabilistic automata*. *Information and Control* 6, pp. 230–245, doi:10.1016/S0019-9958(63)90290-0.

[19] Georgy E. Shilov (1971): *Linear Algebra*. Prentice-Hall. Reprinted by Dover, 1977.

[20] Bianca Truthe (2018): *Hierarchy of Subregular Language Families*. Technical Report, Universitätsbibliothek Gießen, Institut für Informatik, doi:10.22029/JLUPUB-6984.

Constituency Parsing as an Instance of the M-monoid Parsing Problem

Richard Mörbitz
Faculty of Computer Science
Technische Universität Dresden
01062 Dresden
richard.moerbitz@tu-dresden.de

We consider the constituent parsing problem which states: given a final state normalized constituent tree automaton (CTA) and a string, compute the set of all constituent trees that are inductively recognized by the CTA and yield the string. We show that this problem is an instance of the M-monoid parsing problem. Moreover, we show that we can employ the generic M-monoid parsing algorithm to solve the constituency parsing problem for a meaningful class of CTA.

1 Introduction

Constituency, sometimes also referred to as phrase structure, is an important aspect of natural language processing (NLP). Given a phrase of natural language, the task of constituency parsing consists in computing a tree-like structure which describes the syntactic composition of the phrase. These structures are usually visualized as trees where the words of the phrase occur as leaves. Figure 1 (left) shows such a constituent tree for the German phrase "hat schnell gearbeitet". The ordering of the phrase is indicated below the tree where dashed lines link the leaves to their corresponding positions in the phrase. A special phenomenon that may occur in the scope of constituency parsing are discontinuous constituents. These span non-contiguous parts of a phrase; for instance, cf. the constituent labeled V which spans the subphrases "hat" and "gearbeitet" in our example. In the usual illustration, discontinuity manifests itself by crossing lines between the leaves of the tree and the ordering of the phrase.

Usual formal models employed in NLP, such as context-free grammars (CFG) and finite-state tree automata (FTA), are not adequate for modeling discontinuous constituents. This problem has been solved on the grammar side by exploring more powerful grammar formalisms such as tree adjoining grammars (TAG; [7]) and linear context-free rewriting systems (LCFRS; [15, 8]). On the automaton side, hybrid tree automata [2] have recently been introduced. In this context, hybrid trees are usual trees where labels can be extended by a positive number, called *index*, which indicates their position in the phrase. (Each index may only occur once per hybrid tree.) Thus, constituent trees are a particular type of hybrid trees where a label has an index if and only if it occurs at a leaf position. Cf. the tree ξ in Fig. 1 (center) which corresponds to the constituent tree from above. The previously mentioned discontinuity in its first subtree is resembled by the fact that the set of indices occurring in this subtree is not contiguous. Given a constituent tree, we can obtain its phrase by reading off the labels at the leaves in the order of their indices; we call this operation *yield*. Non-contiguous indices lead to phrases with gaps that are formalized using a comma. For instance, the first subtree of ξ (whose root is labeled by V) yields the string tuple $(\mathrm{hat}, \mathrm{gearbeitet})$. In contrast to this formalization of constituent trees, the usual representation of constituent trees in NLP does not feature indices and is thus more abstract.

We briefly recall the automaton model of [2]. In essence, a hybrid tree automaton (HTA) is an FTA where each transition additionally has an *index constraint* which describes the acceptable combinations

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 79–94, doi:10.4204/EPTCS.388.9

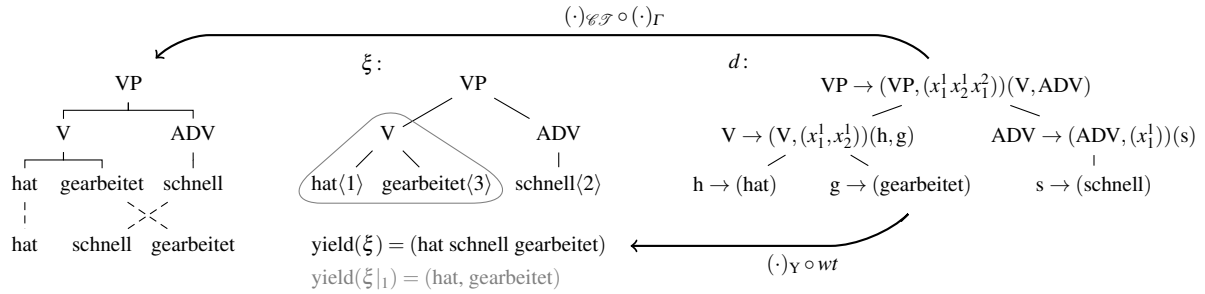


Figure 1: Left: constituent tree for the German phrase "hat schnell gearbeitet". It is discontinuous as the phrase of the left subtree is interleaved with a word from the right subtree. Center: formalization of this constituent tree in the framework of [2]. Right: AST of an RTG and its evaluation in the algebras of our M-monoid parsing problem.

of indices. Such a constraint may refer to both the indices occurring in the subtrees of the position where the transition is applied and the index occurring at that position itself. If unrestricted, these general constraints lead to an overly expressive automaton model. This is why [2] also introduced constituent tree automata (CTA) as a restricted form of HTA to recognize languages of constituent trees. Here, the index constraints are given by *word tuples* as they occur in LCFRS. For instance, the word tuple $(x_1^1 x_2^1 x_1^2)$ states that the indices of the first subtree form two separate intervals, i.e., sets of contiguous numbers, referred to by x_1^1 and x_1^2, and the indices of the second subtree (x_2^1) lie in between. Thus, discontinuous constituents can also be modeled. In essence, a CTA is final state normalized if the constituent trees it recognizes may only yield contiguous phrases (of course, there may be discontinuity in the subtrees). Drewes et al. (2022) [2] showed that the yields of the languages inductively recognized by final state normalized CTA are equal to the languages generated by LCFRS. Thus, CTA provide a meaningful framework for specifying constituency analyses. They did, however, not tackle the following problem which we call the *constituency parsing problem*: given a final state normalized CTA $\mathscr{A}$ and a string u, compute the set of all constituent trees that are inductively recognized by $\mathscr{A}$ and yield u. In this paper, we will solve this problem by showing that it is an instance of the M-monoid parsing problem to which the generic M-monoid parsing algorithm can be applied, provided that $\mathscr{A}$ fulfils a certain condition.

M-monoid parsing [12, 13] is an algebraic framework for weighted parsing. Its kernel is a *weighted RTG-based language model* (wRTG-LM) $\bar{G}$; each wRTG-LM consists of a regular tree grammar (RTG) $\mathscr{G}$, a Γ-algebra $(\mathscr{L}, \phi)$ called *language algebra*, a complete M-monoid $\mathbb{K}$ called *weight algebra*, and a weight mapping *wt* from the set of rules of $\mathscr{G}$ to the signature of $\mathbb{K}$. Moreover, the terminal alphabet of $\mathscr{G}$ is required to be a subset of Γ. The algebraic computations are based on the abstract syntax trees (ASTs) of $\mathscr{G}$; these are trees over rules which represent valid derivations. In the language algebra, each AST can be evaluated to an element of $\mathscr{L}$ by first projecting it to a tree over Γ and then applying the unique homomorphism from the Γ-term algebra to $\mathscr{L}$. In the weight algebra, each AST can be evaluated to an element of $\mathbb{K}$ by first applying *wt* to every rule and then applying the unique homomorphism from the Ω-term algebra to $\mathbb{K}$.

The *M-monoid parsing problem* states the following: given a wRTG-LM $\bar{G}$ and an element $u \in \mathscr{L}$ of the language algebra, compute the sum of the weights (in $\mathbb{K}$) of all ASTs of $\mathscr{G}$ which have the initial nonterminal as the left-hand side of the rule in their root and evaluate to u in the language algebra.

Our first contribution is the instantiation of the M-monoid parsing problem to constituency parsing. In attempting this instantiation, the constituent trees by [2] turn out to be not suitable for such an algebraic

framework. Instead, we introduce *partitioned constituent trees* which are inspired by Nederhof and Vogler (2014) [14] (also cf. [5]). They are tuples consisting of a (usual) tree, a strict total order on its leaves, and a partitioning of its leaves. Compared to constituent trees, partitioned constituent trees abstract from particular indices and only preserve information about the order of the leaves and their groupings (where leaves with consecutive indices fall into the same component of the partitioning). This is also closer to the usual notion of constituent trees in NLP. The yield of partitioned constituent trees is defined analogously to constituent trees: now, the order of the labels is determined by the total order on the leaves and commas are placed between labels whose positions belong to different subsets of the partitioning.

The main part of the instantiation is the following construction. Given a final state normalized CTA $\mathscr{A}$, we construct a wRTG-LM $\bar{G}$ such that the M-monoid parsing problem for $\bar{G}$ is equal to the constituency parsing problem for $\mathscr{A}$. For the definition of $\bar{G}$, we introduce two algebras: one for computing partitioned constituent trees and one for computing their yield. Both use the same signature Γ where each operator consists of a symbol from some ranked alphabet Σ and a word tuple. The first algebra, called *constituent tree algebra*, operates on partitioned constituent trees over Σ by performing top concatenation on their tree components (using the operator's symbol from Σ) and merging their total orders and partitionings using the operator's word tuple. The second one, called *constituent tree yield algebra*, operates on Σ-string tuples by combining them using the operator's word tuple in the same way as the language generated by an LCFRS is computed. Both algebras are many-sorted to ensure that the operators and the arguments fit.

Now, given a CTA $\mathscr{A}$, we define the $\mathscr{A}$-wRTG-LM $\bar{G}$ as follows. Its RTG $\mathscr{G}$ is a syntactical variant of $\mathscr{A}$, where the nonterminals, terminals, and the initial nonterminal of $\mathscr{G}$ are the states of $\mathscr{A}$, a particular subset of Γ, and the final state of $\mathscr{A}$, respectively. Each transition $(q_1 \cdots q_k, a, e, q)$ of $\mathscr{A}$ becomes a rule $q \to (a,e)(q_1, \ldots, q_k)$ in G. Moreover, the language algebra of $\bar{G}$ is the constituent tree yield algebra, the weight algebra is the constituent tree algebra lifted to sets, and the weight mapping maps each rule to its Γ-symbol. This leads to the following M-monoid parsing problem. Given $\bar{G}$ and $u \in \Sigma^*$, compute the set of all partitioned constituent trees that are results of evaluating an AST d of $\mathscr{G}$ in the constituent tree algebra, provided that d evaluates to u in the constituent tree yield algebra. We can prove that this set equals, modulo particular indices, the set of all constituent trees that are inductively recognized by $\mathscr{A}$ and yield u. Since adding (resp. removing) these indices is trivial, the M-monoid parsing problem for $\bar{G}$ and u is equivalent to the constituency parsing problem for $\mathscr{A}$ and u. In Figure 1 we indicate the result of this construction by showing an AST d of some RTG $\mathscr{G}$ which could be obtained as the result of the above construction for a given CTA $\mathscr{A}$ that inductively recognizes ξ. Moreover, we show the evaluation of d in the constituent tree yield algebra as well as in the constituent tree algebra (where the partitioned constituent tree is shown via the typical illustration of constituent trees in NLP) via the homomorphisms $(\cdot)_{\mathrm{Y}}$ and $(\cdot)_{\mathscr{CT}}$, resp. (Here, $(\cdot)_\Gamma$ projects rules to their Γ-symbols.)

Our second contribution concerns the applicability of the generic M-monoid parsing algorithm [13] to the M-monoid parsing problem defined above. We find that the algorithm is in general not applicable, where the problem lies in monadic cycles: if the CTA $\mathscr{A}$ contains transitions of the form (q_1, a_1, e_1, q_2), (q_2, a_2, e_2, q_3), $\ldots$, (q_n, a_n, e_n, q_1), then termination of the algorithm is not guaranteed. Otherwise, if $\mathscr{A}$ is free of such cycles, the M-monoid parsing algorithm is applicable to the M-monoid parsing problem constructed from $\mathscr{A}$ and thus solves the constituency parsing problem for $\mathscr{A}$.

This paper is structured as follows. In Section 2 we fix the basic notions and repeat some mathematic foundations, especially from the area of algebra. In Sections 3 and 4, we recall the central ideas of CTA and the M-monoid parsing problem, respectively. In Section 5, we detail the definition of the wRTG-LM $\bar{G}$ we use to model constituency parsing and we show that the corresponding M-monoid parsing problem

is equivalent to the constituency parsing problem. Finally, in Section 6, we discuss the applicability of the M-monoid parsing algorithm.

2 Preliminaries

Mathematical notions. The set of natural numbers (including 0) is denoted by $\mathbb{N}$ and we let $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$. For every $k, \ell \in \mathbb{N}$, we let $[k, \ell]$ denote the interval $\{i \in \mathbb{N} \mid k \leq i \leq \ell\}$ and we abbreviate $[1, \ell]$ by $[\ell]$. The set of all nonempty intervals of $\mathbb{N}_+$ is denoted by $\mathbb{I}$. For every $I, I' \in \mathbb{I}$, the expression $I < I'$ holds if $\max I < \min I'$ and $I \frown I'$ holds if $\max I + 1 = \min I'$. Thus $I \frown I'$ implies $I < I'$. For each set A, we let $\mathscr{P}(A)$ denote the power set of A. We extend a mapping $f\colon A \to B$ in the canonical way to a mapping $f\colon \mathscr{P}(A) \to \mathscr{P}(B)$. A *family* $(a_i \mid i \in I)$ is a mapping $f\colon I \to A$ with $f(i) = a_i$ for each $i \in I$. Let A, B, and C be sets. The composition of two mappings $f\colon A \to B$ and $g\colon B \to C$ is denoted by $g \circ f$. Whenever we deal with a partitioning $(A_1, \ldots, A_n)$ of a set A, we require A_i to be non-empty (for each $i \in [n]$). An *alphabet* is a finite and non-empty set.

Strings and tuples. Let A be a set and $k \in \mathbb{N}$. We let A^k denote the set of all strings $w = a_1 \cdots a_k$ of length k, where $a_1, \ldots, a_k \in A$, and we let $A^* = \bigcup_{k \in \mathbb{N}} A^k$. The empty string ($k = 0$) is denoted by ε. We denote substrings of a string $w = a_1 \cdots a_k$ in A^* as follows: for every $i \in [k]$ and $j \in [k - i + 1]$, we let $w[i; j] = a_i \cdots a_{i+j-1}$, and $w[i]$ abbreviates $w[i; 1]$. Let $\ell \in \mathbb{N}$. The *concatenation* of two strings $v = a_1 \cdots a_k$ in A^k and $w = b_1 \cdots b_\ell$ in A^ℓ, denoted by $v \cdot w$, is the string $a_1 \cdots a_k b_1 \cdots b_\ell$ in $A^{k+\ell}$; we drop $\cdot$ if it is clear from the context. Moreover, we lift concatenation to sets of strings in the obvious way.

We let $\mathrm{Tup}_k(A)$ denote the k-fold Cartesian product of A; its elements are called *k-tuples over A*. Moreover, we let $\mathrm{Tup}(A) = \bigcup_{k \in \mathbb{N}} \mathrm{Tup}_k(A)$. In the obvious way, we transfer the notion of substrings from strings to tuples.

Sorted sets, trees, and regular tree grammars. Let S be a set; its elements are usually called *sorts*. An *S-sorted set* is a pair (A, sort) where A is a set and $\mathrm{sort}\colon A \to S$ is a mapping. For each $s \in S$, we let $A^{(s)} = \{a \in A \mid \mathrm{sort}(a) = s\}$. We call an S-sorted set *single-sorted* if $|S| = 1$; thus, each (usual) set can be viewed as a single-sorted set. A *ranked set* is an $\mathbb{N}$-sorted set; its sort mapping is usually denoted by rk. In examples, we will show the rank of a symbol as a superscript in parentheses, e.g., $a^{(k)}$ if $\mathrm{rk}(a) = k$. An S-sorted (resp. ranked) alphabet is an S-sorted (resp. ranked) set which is an alphabet.

An $(S^* \times S)$-sorted set Γ is called *S-signature*. Whenever we write $\gamma \in \Gamma^{(s_1 \cdots s_k, s)}$ we assume that $k \in \mathbb{N}$ and $s, s_1, \ldots, s_k \in S$ are universally quantified if not specified otherwise. Now let H be an S-sorted set. The set of *S-sorted trees over Γ and H*, denoted by $\mathrm{T}_\Gamma(H)$, is the smallest S-sorted set T such that, for each $s \in S$, we have $H^{(s)} \subseteq T^{(s)}$ and, for every $\gamma \in \Gamma^{(s_1 \cdots s_k, s)}$ and $t_1 \in T^{(s_1)}, \ldots, t_k \in T^{(s_k)}$, we have $\gamma(t_1, \ldots, t_k) \in T^{(s)}$. We abbreviate $\mathrm{T}_\Gamma(\emptyset)$ by T_Γ. Since we can view each $(S^* \times S)$-sorted set as a ranked set by, for every $\gamma \in \Gamma^{(s_1 \cdots s_k, s)}$, letting $\mathrm{rk}(\gamma) = k$, the above definition also covers the usual trees over ranked alphabets.

The *set of positions* of a tree is defined by the mapping $\mathrm{pos}\colon \mathrm{T}_\Gamma(H) \to \mathscr{P}((\mathbb{N}_+)^*)$ as usual. Let $t \in \mathrm{T}_\Gamma(H)$ and $w \in \mathrm{pos}(t)$. The set of *leaves* of t, the *label of t at w*, and the *subtree of t at w* are also defined as usual, and are denoted by $\mathrm{leaves}(t)$, $t(w)$, and $t|_w$, respectively.

An *S-sorted regular tree grammar* (RTG; [1]) is a tuple $\mathscr{G} = (N, \Gamma, A_0, R)$ where N is an S-sorted alphabet (*nonterminals*), Γ is an $(S^* \times S)$-sorted alphabet (*terminals*) with $N \cap \Gamma = \emptyset$, $A_0 \in N$ (*initial nonterminal*), and R is a finite set of *rules* where each rule r has the form $A \to \gamma(A_1, \ldots, A_k)$ with $k \in \mathbb{N}$,

$\gamma \in \Gamma^{(s_1 \cdots s_k, s)}$, and $A \in N^{(s)}, A_1 \in N^{(s_1)}, \ldots, A_k \in N^{(s_k)}$. (Thus, we only consider RTGs in normal form.) We call A the *left-hand side* of r; it is denoted by $\mathrm{lhs}(r)$.

We view R as an $(N^* \times N)$-sorted set where each rule $A \to \gamma(A_1, \ldots, A_k)$ has sort $(A_1 \cdots A_k, A)$. Thus, for every $d \in \mathrm{T}_R$ and $w \in \mathrm{pos}(d)$, the following holds: if $d(w)$ is $A \to \gamma(A_1, \ldots, A_k)$, then, for each $i \in [k]$, we have $\mathrm{lhs}(d(w \cdot i)) = A_i$. We call T_R the set of *abstract syntax trees* (short: ASTs) of $\mathscr{G}$. We define the mapping $(\cdot)_\Gamma \colon \mathrm{T}_R \to \mathrm{T}_\Gamma$ such that $(d)_\Gamma$ is obtained from d by replacing each $A \to \gamma(A_1, \ldots, A_k)$ by γ. The *tree language generated by* $\mathscr{G}$ is the set $\mathrm{L}(\mathscr{G}) = (\mathrm{T}_R)_\Gamma$.

S-sorted Γ-algebras. Let S be a set and Γ be an S-signature. An *S-sorted Γ-algebra* (short: algebra) is a pair $(\mathscr{A}, \phi)$ where $\mathscr{A}$ is an S-sorted set (*carrier set*) and ϕ is a mapping which maps each $\gamma \in \Gamma^{(s_1 \cdots s_k, s)}$ to a mapping $\phi(\gamma) \colon \mathscr{A}^{(s_1)} \times \cdots \times \mathscr{A}^{(s_k)} \to \mathscr{A}^{(s)}$. We will sometimes identify $\phi(\gamma)$ and γ (as it is usual).

The *S-sorted Γ-term algebra* is the S-sorted Γ-algebra $(\mathrm{T}_\Gamma, \phi_\Gamma)$ where, for every $\gamma \in \Gamma^{(s_1 \cdots s_k, s)}$ and $t_1 \in \mathrm{T}_\Gamma^{(s_1)}, \ldots, t_k \in \mathrm{T}_\Gamma^{(s_k)}$, we let $\phi_\Gamma(\gamma)(t_1, \ldots, t_k) = \gamma(t_1, \ldots, t_k)$. For each Γ-algebra $(\mathscr{A}, \phi)$ there is a unique homomorphism, denoted by $(\cdot)_\mathscr{A}$, from the Γ-term algebra to $(\mathscr{A}, \phi)$ [16]. We write its application to an argument $t \in \mathrm{T}_\Gamma$ as $(t)_\mathscr{A}$. Intuitively, $(\cdot)_\mathscr{A}$ evaluates a tree t in $(\mathscr{A}, \phi)$, in the same way as arithmetic expressions are evaluated to numbers. For instance, the expression $3 + 2 \cdot (4 + 5)$ is evaluated to 21 in the $\{+, \cdot\}$-algebra $(\mathbb{N}, +, \cdot)$. Often we abbreviate an algebra $(\mathscr{A}, \phi)$ by $\mathscr{A}$. For every $a \in \mathscr{A}$ we let $\mathrm{factors}(a) = \{b \in \mathscr{A} \mid b <_{\mathrm{factor}}{}^* a\}$ where, for every $a, b \in \mathscr{A}$, $b <_{\mathrm{factor}} a$ if there is a $\gamma \in \Gamma$ such that b occurs in some tuple $(b_1, \ldots, b_k)$ with $\phi(\gamma)(b_1, \ldots, b_k) = a$. That is, $\mathrm{factors}(a)$ is the set of all values that occur in a term which evaluates to a. We call $(\mathscr{A}, \phi)$ *finitely decomposable* if $\mathrm{factors}(a)$ is finite for every $a \in \mathscr{A}$.

Word tuples. Let $k \in \mathbb{N}$ and $\kappa = (\ell_1, \ldots, \ell_k)$ in $\mathrm{Tup}_k(\mathbb{N}_+)$. We let $\mathbb{X}_\kappa = \{x_i^j \mid i \in [k], j \in [\ell_i]\}$ and call each element x_i^j of $\mathbb{X}_\kappa$ a *variable*. Moreover, let $n \in \mathbb{N}_+$ and Δ be an alphabet. Then we denote by $\mathbb{W}_\kappa^n(\Delta)$ the set of all tuples $e = (s_1, \ldots, s_n)$ such that (1) for each $i \in [n]$, the component s_i is a string over Δ and $\mathbb{X}_\kappa$, (2) each variable in $\mathbb{X}_\kappa$ occurs exactly once in e, and (3) for all $x_i^{j_1}, x_i^{j_2} \in \mathbb{X}_\kappa$ with $j_1 < j_2$, the variable $x_i^{j_1}$ occurs left of $x_i^{j_2}$ in e. Each element of $\mathbb{W}_\kappa^n(\Delta)$ is a *monotone* (n, κ)*-word tuple*.[1] We let $\mathbb{W}(\Delta) = \bigcup_{n \in \mathbb{N}_+, k \in \mathbb{N}, \kappa \in \mathrm{Tup}_k(\mathbb{N}_+)} \mathbb{W}_\kappa^n(\Delta)$ and we drop '$(\emptyset)$' for empty Δ.

Let $e = (s_1, \ldots, s_n)$ be in $\mathbb{W}_\kappa^n(\Delta)$. The *word function induced by* e is the mapping

$$[\![e]\!] \colon \mathrm{Tup}_{\ell_1}(\Delta^*) \times \cdots \times \mathrm{Tup}_{\ell_k}(\Delta^*) \to \mathrm{Tup}_n(\Delta^*)$$

which is defined, for every $(w_1^1, \ldots, w_1^{\ell_1}) \in \mathrm{Tup}_{\ell_1}(\Delta^*)$, …, $(w_k^1, \ldots, w_k^{\ell_k}) \in \mathrm{Tup}_{\ell_k}(\Delta^*)$, by

$$[\![e]\!]((w_1^1, \ldots, w_1^{\ell_1}), \ldots, (w_k^1, \ldots, w_k^{\ell_k})) = (v_1, \ldots, v_n)$$

where each v_m ($m \in [n]$) is obtained from s_m by replacing every occurrence of a variable x_i^j by w_i^j. For instance, let $\Delta = \{a, b, c, d\}$. The word tuple $e = (bx_2^1x_1^1ax_1^2, acx_2^2x_1^3a)$ in $\mathbb{W}_{(3,2)}^2(\Delta)$ induces the word function $[\![e]\!] \colon \mathrm{Tup}_3(\Delta^*) \times \mathrm{Tup}_2(\Delta^*) \to \mathrm{Tup}_2(\Delta^*)$ with

$$[\![e]\!]((w_1^1, w_1^2, w_1^3), (w_2^1, w_2^2)) = (bw_2^1w_1^1aw_1^2, acw_2^2w_1^3a).$$

We view $\mathbb{W}(\Delta)$ as a $(\mathbb{N}_+^* \times \mathbb{N}_+)$-sorted set in the obvious way (i.e., $e \in \mathbb{W}_\kappa^n(\Delta)$ has sort (κ, n)) and we denote the unique homomorphism from the $\mathbb{N}_+$-sorted $\mathbb{W}(\Delta)$-term algebra to the $\mathbb{N}_+$-sorted algebra $(\mathrm{Tup}(\Delta^*), [\![\cdot]\!])$ also by $[\![\cdot]\!]$. Intuitively, it evaluates trees over word tuples to elements of $\mathrm{Tup}(\Delta^*)$ by applying in a bottom-up way the word functions induced by their word tuples.

[1] Monotonicity is expressed by condition (3); in this paper, we do not deal with non-monotone word tuples.

Monoids. A *monoid* is an algebra $(\mathbb{K}, \oplus, \mathbb{0})$ such that $\oplus$ is a binary, associative operation on $\mathbb{K}$ and $\mathbb{0} \oplus \Bbbk = \Bbbk = \Bbbk \oplus \mathbb{0}$ for each $\Bbbk \in \mathbb{K}$. The monoid is *commutative* if $\oplus$ is commutative and it is *idempotent* if $\Bbbk \oplus \Bbbk = \Bbbk$. It is *complete* if, for each countable set I, there is an operation $\sum_I^{\oplus}$ which maps each family $(\Bbbk_i \mid i \in I)$ to an element of $\mathbb{K}$, coincides with $\oplus$ when I is finite, and otherwise satisfies axioms which guarantee commutativity and associativity [4, p. 124]. We abbreviate $\sum_I^{\oplus}(\Bbbk_i \mid i \in I)$ by $\sum_{i \in I}^{\oplus} \Bbbk_i$. A complete monoid is *d-complete* [9] if, for every $\Bbbk \in \mathbb{K}$ and family $(\Bbbk_i \mid i \in \mathbb{N})$ of elements of $\mathbb{K}$, the following holds: if there is an $n_0 \in \mathbb{N}$ such that for every $n \in \mathbb{N}$ with $n \geq n_0$, $\sum^{\oplus}_{i \in \mathbb{N}:\, i \leq n} \Bbbk_i = \Bbbk$, then $\sum^{\oplus}_{i \in \mathbb{N}} \Bbbk_i = \Bbbk$. A complete monoid is *completely idempotent* if for every $\Bbbk \in \mathbb{K}$ and countable set I it holds that $\sum^{\oplus}_{i \in I} \Bbbk = \Bbbk$. An easy proof shows that if $\mathbb{K}$ is completely idempotent, it is also d-complete.

M-monoids. A *multioperator monoid* (M-monoid; [11]) is an algebra $(\mathbb{K}, \oplus, \mathbb{0}, \Omega, \phi)$ where $(\mathbb{K}, \oplus, \mathbb{0})$ is a commutative monoid (*additive monoid*), Ω is a ranked set, and $(\mathbb{K}, \phi)$ is an Ω-algebra. An M-monoid inherits the properties of its monoid (e.g., being complete). We denote a complete M-monoid by $(\mathbb{K}, \oplus, \mathbb{0}, \Omega, \phi, \sum^{\oplus})$. An M-monoid is *distributive* if, for every $\omega \in \Omega^{(m)}$, $i \in [m]$, and $\Bbbk, \Bbbk_1, \ldots, \Bbbk_m \in \mathbb{K}$,

$$\omega(\Bbbk_1, \ldots, \Bbbk_{i-1}, \Bbbk_i \oplus \Bbbk, \Bbbk_{i+1}, \ldots, \Bbbk_m) = \omega(\Bbbk_1, \ldots, \Bbbk_{i-1}, \Bbbk_i, \Bbbk_{i+1}, \ldots, \Bbbk_m) \oplus \omega(\Bbbk_1, \ldots, \Bbbk_{i-1}, \Bbbk, \Bbbk_{i+1}, \ldots, \Bbbk_m).$$

If $\mathbb{K}$ is complete, then we only call it distributive if the above equation also holds for each countable set of summands. We sometimes refer to an M-monoid only by its carrier set.

Example 1. Let S be a set, Ω be an S-signature, and $(\mathscr{A}, \phi)$ be an S-sorted set. We will now define an M-monoid which lifts the computations from $(\mathscr{A}, \phi)$ to sets of elements of $\mathscr{A}$. Its carrier set will be $B = \bigcup_{s \in S} \mathscr{P}(\mathscr{A}^{(s)}) \cup \{\bot\}$ where $\bot$ is a new element. Thus, B contains all single-sorted subsets of $\mathscr{A}$ and an element $\bot$ which will be used whenever an operation is applied to arguments which do not match its sort. Formally, we define the M-monoid $(B, \mathbin{\mathfrak{S}}, \emptyset, \Omega, \psi)$ where, for every $B_1, B_2 \in B$,

$$B_1 \mathbin{\mathfrak{S}} B_2 = \begin{cases} B_1 \cup B_2 & \text{if there exists } s \in S \text{ such that } B_1, B_2 \subseteq \mathscr{A}^{(s)} \\ \bot & \text{otherwise} \end{cases}$$

and, for every $\gamma \in \Gamma^{(s_1 \cdots s_k, s)}$ and $B_1, \ldots, B_k \in B$,

$$\psi(\gamma)(B_1, \ldots, B_k) = \begin{cases} \phi(\gamma)(B_1, \ldots, B_k) & \text{if } B_1 \subseteq \mathscr{A}^{(s_1)}, \ldots, B_k \subseteq \mathscr{A}^{(s_k)} \\ \bot & \text{otherwise.} \end{cases}$$

We consider $\sum^{\mathfrak{S}}$ which is defined for every index set I as $\bigcup_I$. It is easy to see that B together with $\sum^{\mathfrak{S}}$ is complete and distributive. Moreover, since the monoid $(B, \mathfrak{S}, \emptyset, \sum^{\mathfrak{S}})$ is completely idempotent, we obtain that $(B, \mathfrak{S}, \emptyset, \Omega, \psi, \sum^{\mathfrak{S}})$ is d-complete. ◁

3 Constituent tree automata

Hybrid trees and, as a special case thereof, constituent trees are certain trees over potentially indexed symbols where, intuitively, an indexed symbol is a symbol equipped with a positive number. Formally, let Σ be an alphabet. The *set of indexed Σ-symbols*, denoted by $\Sigma\langle\mathbb{N}_+\rangle$, is the ranked set defined by $\Sigma\langle\mathbb{N}_+\rangle^{(k)} = \{a\langle n\rangle \mid a \in \Sigma^{(k)}, n \in \mathbb{N}_+\}$ for each $k \in \mathbb{N}$. An element $a\langle n\rangle$ is called *indexed symbol* and n is the *index of* $a\langle n\rangle$. We write $(a\langle n\rangle)_\Sigma$ for a and $(a\langle n\rangle)_\mathbb{N}$ for n.

Here we only define constituent trees; for a general definition of hybrid trees, cf. [2]. A *constituent tree* is a tree $\xi \in \mathrm{T}_\Sigma(\Sigma\langle\mathbb{N}_+\rangle)$ such that, for every $w, w' \in \mathrm{leaves}(\xi)$, we have that $(\xi(w))_\mathbb{N} = (\xi(w'))_\mathbb{N}$

implies $w = w'$. In words, a symbol is indexed if and only if it occurs at a leaf and no index occurs twice. We let $(\xi)_\Sigma$ denote the tree in T_Σ obtained from ξ by removing all indices. The set of all constituent trees over Σ is denoted by C_Σ.

We extract the linear phrase from a constituent tree ξ using the mapping yield: $\mathrm{C}_\Sigma \to \mathrm{Tup}(\Sigma^*)$ which we define as follows. We order the set of indexed symbols occurring in ξ into a sequence according to their indices, then we drop each comma between neighbored symbols with consecutive indices, and finally we drop the indices. Thus, the tuple yield(ξ) has one more component than the number of gaps in the set of indices occurring in ξ. For instance, consider the constituent tree ξ in Figure 1. The ordering of its set of indexed symbols is $(\mathrm{hat}\langle 1\rangle, \mathrm{schnell}\langle 2\rangle, \mathrm{gearbeitet}\langle 3\rangle)$ and all commas are dropped as there are no gaps between indices.

A *constituent tree automaton* (short: CTA) is a tuple $\mathscr{A} = (Q, \Sigma, \delta, q_f)$ where

- Q is a ranked alphabet with $Q^{(0)} = \emptyset$ (*states*),
- Σ is a ranked alphabet,
- δ is a finite set of *transitions*, each of which having either form (ε, a, q) where $a \in \Sigma^{(0)}$ and $q \in Q^{(1)}$ or form $(q_1 \cdots q_k, a, e, q)$ where $k \in \mathbb{N}_+$, $e \in \mathbb{W}^n_{(\ell_1,\ldots,\ell_k)}$, $q_1 \in Q^{(\ell_1)}, \ldots, q_k \in Q^{(\ell_k)}, q \in Q^{(n)}$, and $a \in \Sigma^{(k)}$; and
- $q_f \in Q$ (*final state*).

We call $\mathscr{A}$ *final state normalized* if $q_f \in Q^{(1)}$.

We note that this definition of CTA simplifies the definition by [2] in three regards. First, we opted to define CTA directly and not as a special case of HTA. Second, their nullary transitions contain an additional object, the universal index constraint $\mathrm{UIC}_{0,1}$, which we have dropped for the sake of clarity. Third, to achieve coherence with RTGs, our CTA has only a single final state q_f. This is not a restriction, since each CTA of [2] with a set of final states can be transformed into an equivalent CTA with a single final state using a standard construction from automata theory (cf., e.g., [3, L. 4.8]).

Example 2. Let $\mathscr{A} = (Q, \Sigma, \delta, q_f)$ be a CTA where the states are $Q = \{q^{(3)}, q_l^{(2)}, q_r^{(2)}, q_a^{(1)}, q_b^{(1)}, q_c^{(1)}, q_f^{(1)}\}$, the terminal alphabet is $\Sigma = \{a^{(0)}, b^{(0)}, c^{(0)}, d^{(3)}, e^{(2)}\}$, and δ consists of the following transitions:

$$(q_l q q_c, d, (x_1^1 x_2^1 x_1^2 x_2^2 x_3^1 x_2^3), q_f) \qquad (q_a q q_r, d, (x_1^1 x_2^1 x_3^1 x_2^2 x_3^2 x_2^3), q_f)$$
$$(q_l q q_c, d, (x_1^1 x_2^1, x_1^2 x_2^2, x_3^1 x_2^3), q) \qquad (q_a q q_r, d, (x_1^1 x_2^1, x_3^1 x_2^2, x_3^2 x_2^3), q)$$
$$(q_a q_b q_c, d, (x_1^1, x_2^1, x_3^1), q) \qquad (q_a q_b, e, (x_1^1, x_2^1), q_l) \qquad (q_b q_c, e, (x_1^1, x_2^1), q_r)$$
$$(\varepsilon, a, q_a) \qquad (\varepsilon, b, q_b) \qquad (\varepsilon, c, q_c).$$

We note that $\mathscr{A}$ is final state normalized. We will use $\mathscr{A}$ to illustrate the semantics of CTA which we define next. ◁

While there are two semantics of CTA in [2], we are only interested in one of them, called the *hybrid tree language inductively recognized* by CTA. In this paper, we refer to it simply as *language inductively recognized by* $\mathscr{A}$ and define it in the following.

Let $k, \ell_1, \ldots, \ell_k \in \mathbb{N}_+$. We let $\kappa = (\ell_1, \ldots, \ell_k)$. A κ-*assignment* is a mapping $\varphi\colon \mathbb{X}_\kappa \to \mathbb{I}$ such that, for every $x, x' \in \mathbb{X}_\kappa$ with $x \neq x'$, it holds that $\varphi(x) \cap \varphi(x') = \emptyset$. Now let $n \in \mathbb{N}_+$ and $e \in \mathbb{W}^n_\kappa$. We say that φ *models* e, denoted by $\varphi \models e$, if the expression e' holds where e' is obtained from e by (1) writing $\frown$ between each occurrence of two consecutive variables, (2) replacing each comma by $<$, and (3) replacing each variable x by $\varphi(x)$. As an example, consider the word tuple $e = (x_1^1 x_2^1, x_3^1 x_2^2, x_3^2 x_2^3)$ which occurs at position 2 of ρ in Figure 2. We define the $(1,3,2)$-assignment φ with $\varphi(x_1^1) = \{2\}$, $\varphi(x_2^1) = \{3\}$,

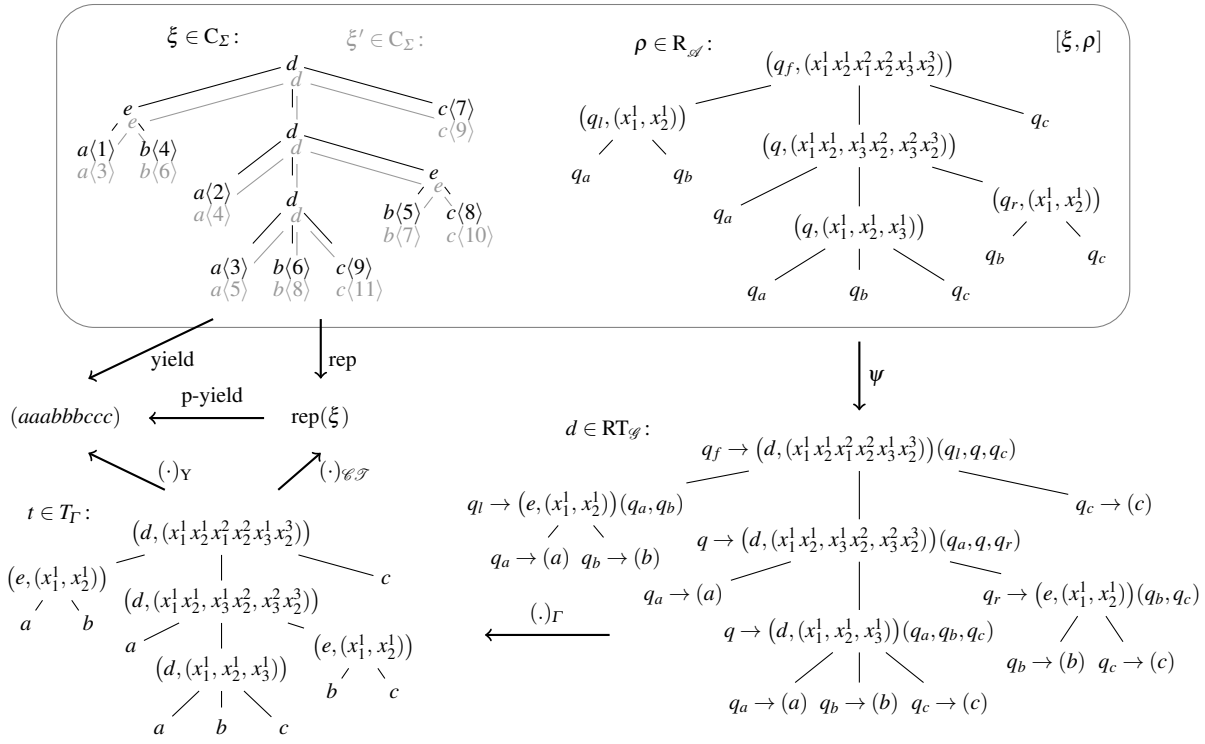


Figure 2: Top: constituent tree ξ and run ρ of the CTA $\mathscr{A}$ from Example 2 such that $(\xi,\rho) \in \mathrm{CR}_{\mathscr{A}}$. We also show a constituent tree ξ' with $(\xi',\rho) \sim (\xi,\rho)$ in gray. Bottom: AST d of the $\mathscr{A}$-RTG $\mathscr{G}$ which is the image of $[\xi,\rho]$ under the bijection ψ. The mappings introduced in this paper commute; in particular, $\mathrm{yield}(\xi) = ((\psi([\xi,\rho]))_\Gamma)_\mathrm{Y}$.

$\varphi(x_3^1) = \{5\}$, $\varphi(x_2^2) = \{6\}$, $\varphi(x_3^2) = \{8\}$, $\varphi(x_2^3) = \{9\}$, where the indices are taken from the constituent tree ξ in Figure 2. We obtain the expression $e' = (\{2\} \frown \{3\} < \{5\} \frown \{6\} < \{8\} \frown \{9\})$ which is valid and hence $\varphi \models e$.

Let $\mathscr{A} = (Q, \Sigma, \delta, q_f)$ be a CTA. A *run of* $\mathscr{A}$ is a tree $\rho \in \mathrm{T}_{Q\times\mathbb{W}}(Q)$ where, for $(q,e) \in Q \times \mathbb{W}$, we let $\mathrm{rk}(q,e) = k$ if $e \in \mathbb{W}^n_{(\ell_1,\ldots,\ell_k)}$. We let $\mathrm{R}_{\mathscr{A}}$ denote the set of runs of $\mathscr{A}$. We define $\Theta_{\mathscr{A}} \subseteq \mathrm{C}_\Sigma \times \mathrm{R}_{\mathscr{A}} \times \mathrm{Tup}(\mathbb{I})$ to be the smallest set T that satisfies the following:

- For every $(\varepsilon, a, q) \in \delta$ and $i \in \mathbb{N}_+$ it holds that $\big(a\langle i\rangle, q, \{i\}\big) \in T$.
- For every $(q_1 \cdots q_k, a, e, q) \in \delta$ and $(\xi_1,\rho_1,J_1),\ldots,(\xi_k,\rho_k,J_k) \in T$ where q_i is the state at $\rho_i(\varepsilon)$ (for $i \in [k]$), we let κ denote $(\mathrm{rk}(q_1),\ldots,\mathrm{rk}(q_k))$ and consider the mapping $\varphi\colon \mathbb{X}_\kappa \to \mathbb{I}$ defined, for every $i \in [k]$ and $j \in [\mathrm{rk}(q_i)]$, by $\varphi(x_i^j) = J_i[j]$. (The fact that $J_i[j]$ is indeed an interval can easily be verified by induction.) Now, if φ is a κ-assignment (i.e., its image consists of pairwise disjoint sets) and $\varphi \models e$, then $\big(a(\xi_1,\ldots,\xi_k), \rho, (U_1,\ldots,U_{\mathrm{rk}(q)})\big) \in T$ where we let $\rho = (q,e)(\rho_1,\ldots,\rho_k)$ and, for each $m \in [\mathrm{rk}(q)]$, $U_m = \bigcup_{i,j\colon x_i^j \text{ occurs in the } m\text{-th component of } e} \varphi(x_i^j)$.

We define the following projection of $\Theta_{\mathscr{A}}$ (where CR stands for "constituent (trees and) runs"):

$$\mathrm{CR}_{\mathscr{A}} = \{(\xi,\rho) \mid (\exists J \in \mathrm{Tup}(\mathbb{I}))\,.\,(\xi,\rho,J) \in \Theta_{\mathscr{A}}\}.$$

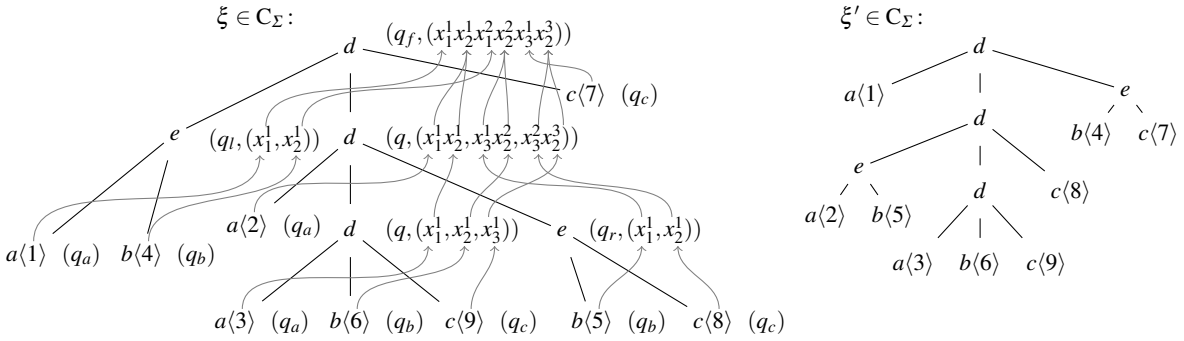


Figure 3: Left: constituent tree ξ and run ρ of the CTA $\mathscr{A}$ from Example 2 such that $(\xi,\rho) \in \mathrm{CR}_{\mathscr{A}}$ where the states and word tuples of ρ have been written next to the positions of ξ. Arrows indicate the family of assignments which witnesses $(\xi,\rho) \in \mathrm{CR}_{\mathscr{A}}$ where, at each non-leaf position of ρ, a variable is assigned the set of all indices whose arrows reach it. Right: another constituent tree $\xi' \in \mathrm{L}_{\mathrm{ind}}(\mathscr{A})$ such that $\mathrm{yield}(\xi) = \mathrm{yield}(\xi')$.

The language *inductively recognized by* $\mathscr{A}$, denoted by $\mathrm{L}_{\mathrm{ind}}(\mathscr{A})$, is the set

$$\mathrm{L}_{\mathrm{ind}}(\mathscr{A}) = \{\xi \mid (\xi,\rho) \in \mathrm{CR}_{\mathscr{A}}, \rho(\varepsilon) \text{ has state } q_f\}.$$

Example 3. Recall the CTA $\mathscr{A}$ of Example 2. The top left of Figure 2 shows a constituent tree ξ and a run ρ of $\mathscr{A}$ such that $(\xi,\rho) \in \mathrm{CR}_{\mathscr{A}}$. In order to show that $(\xi,\rho) \in \mathrm{CR}_{\mathscr{A}}$ indeed holds, in Figure 3 (left), we illustrate the assignments used at each position of ξ in the inductive definition of $\Theta_{\mathscr{A}}$. For this, we use arrows starting at the indices in the leaves of ξ. At every non-leaf position w of ρ, we show the κ-assignment φ which witnesses the existence of $J \in \mathrm{Tup}(\mathbb{I})$ such that $(\xi|_w, \rho|_w, J) \in \Theta_{\mathscr{A}}$ as follows: for each variable x_i^j in the word tuple at $\rho(w)$, it holds that $\varphi(x_i^j)$ consists of all indices whose arrows reach x_i^j. In the way these arrows pass through the word tuples at subtrees of $\rho|_w$, it is shown that φ is consistent with the assignments in the subtrees. This stresses the inductive nature of $\mathrm{CR}_{\mathscr{A}}$.

The constituent tree ξ exemplifies the form of each constituent tree inductively recognized by $\mathscr{A}$. The backbone is a monadic chain where each position is labeled with d. The bottom of the chain has three leaf children, labeled by a, b, and c. Each inner position of the chain has three children as well, the second of which continues the chain. Moreover, the symbols a, b, and c are distributed as leaves among the first and third child, where e serves as an intermediate node under the child receiving two symbols (cf. positions ε and 2 of ξ). The indices are placed such that, for each of a, b, and c, the indices occurring with this symbol form an interval where a has the lowest and c has the highest interval. Thus, $\mathrm{yield}(\mathrm{L}_{\mathrm{ind}}(\mathscr{A})) = \{a^n b^n c^n \mid n \in \mathbb{N}_+\}$ which is not context-free. As there are two patterns for inner positions of the backbone, $\mathscr{A}$ may recognize several constituent trees with the same yield (an example is given in the right of Figure 3). ◁

The *constituency parsing problem* states:

Given: a final state normalized CTA $\mathscr{A} = (Q,\Sigma,\delta,q_f)$ and $u \in (\Sigma^{(0)})^*$

Compute: $\{\xi \in \mathrm{L}_{\mathrm{ind}}(\mathscr{A}) \mid \mathrm{yield}(\xi) = (u)\}$.

We note that, since $\mathscr{A}$ is final state normalized, every $\xi \in \mathrm{L}_{\mathrm{ind}}(\mathscr{A})$ has $\mathrm{yield}(\xi) \in \Sigma^*$. Hence we did not allow string tuples consisting of more than one component in the specification of the constituency parsing problem.

4 Weighted RTG-based language models and the M-monoid parsing problem

The M-monoid parsing problem [12, 13] builds on RTG-based language models which are inspired by the initial algebra approach [6].

An *RTG-based language model* (RTG-LM) is a tuple $(\mathscr{G},(\mathscr{L},\phi))$ where, for some S-signature Γ,

- $(\mathscr{L},\phi)$ is a Γ-algebra (*language algebra*), we call the elements of $\mathscr{L}$ *syntactic objects*, and
- $\mathscr{G}=(N,\Lambda,A_0,R)$ is an S-sorted RTG with $\Lambda\subseteq\Gamma$.

The *language generated by* $(\mathscr{G},(\mathscr{L},\phi))$ is the set

$$(\mathrm{L}(\mathscr{G}))_{\mathscr{L}}=\{(t)_{\mathscr{L}}\mid t\in\mathrm{L}(\mathscr{G})\}\subseteq\mathscr{L},$$

i.e., the set of all syntactic objects obtained by evaluating trees of $\mathrm{L}(\mathscr{G})$ in the language algebra $\mathscr{L}$. We note that $(\mathrm{L}(\mathscr{G}))_{\mathscr{L}}\subseteq\mathscr{L}^{\mathrm{sort}(A_0)}$, i.e., each syntactic object in the language generated by $(\mathscr{G},(\mathscr{L},\phi))$ has the sort of A_0.

A *weighted RTG-based language model* (wRTG-LM) is a tuple

$$\big((\mathscr{G},(\mathscr{L},\phi)),\ \ (\mathbb{K},\oplus,\mathbb{0},\Omega,\psi,\textstyle\sum^{\oplus}),\ \ wt\big)$$

where

- $(\mathscr{G},(\mathscr{L},\phi))$ is an RTG-LM,
- $(\mathbb{K},\oplus,\mathbb{0},\Omega,\psi,\sum^{\oplus})$ is a complete M-monoid (*weight algebra*), and
- wt maps each rule of $\mathscr{G}$ with rank k to a k-ary operation in Ω. In the obvious way, we lift wt to the mapping $wt'\colon \mathrm{T}_R\to\mathrm{T}_\Omega$ and let wt also denote wt'.

The *M-monoid parsing problem* states:

Given: a wRTG-LM $((\mathscr{G},(\mathscr{L},\phi)),(\mathbb{K},\oplus,\mathbb{0},\Omega,\psi,\sum^{\oplus}),wt)$ with $G=(N,\Lambda,A_0,R)$ and $a\in\mathscr{L}$

Compute: the value $\mathrm{parse}_{(\mathscr{G},\mathscr{L})}(a)\in\mathbb{K}$ where

$$\mathrm{parse}_{(\mathscr{G},\mathscr{L})}(a)=\sum_{\substack{d\in\mathrm{T}_R:\\((d)_\Gamma)_{\mathscr{L}}=a,\mathrm{lhs}(d(\varepsilon))=A_0}}^{\oplus}(wt(d))_{\mathbb{K}}.$$

The computation of $\mathrm{parse}_{(\mathscr{G},\mathscr{L})}(a)$ employs the homomorphisms of both algebras. Each AST of $\mathscr{G}$ is mapped to an element of $\mathscr{L}$ via the homomorphisms $(\cdot)_\Gamma$ and $(\cdot)_{\mathscr{L}}$ and it is mapped to an element of $\mathbb{K}$ via the homomorphisms wt and $(\cdot)_{\mathbb{K}}$. Given a syntactic object a, the M-monoid parsing problem states to first compute a collection of ASTs[2] via the inverse of the homomorphisms $(\cdot)_\Gamma$ and $(\cdot)_{\mathscr{L}}$. These ASTs are filtered for those where the left-hand side of the rule at the root is the initial nonterminal. Then, values in $\mathbb{K}$ are computed from the remaining ASTs via the homomorphisms wt and $(\cdot)_{\mathbb{K}}$. Finally, these values are accumulated to a single value using $\sum^{\oplus}$.

[2]Due to ambiguity, an AST may occur several times in the computation of $\mathrm{parse}_{(\mathscr{G},\mathscr{L})}(a)$ [13].

5 Constituency parsing as an M-monoid parsing problem

In this section, we give the formal details of the definition of the constituent tree algebra, the constituent tree yield algebra, and the wRTG-LM we construct for a given CTA to model its constituency parsing problem. Moreover, we sketch the proof of the statement that the corresponding M-monoid parsing problem is equal to that constituency parsing problem. We start by defining partitioned constituent trees which are inspired by the hybrid trees of Nederhof and Vogler (2014) [14] (also cf. [5, 10]).

Let Σ be a ranked alphabet. A *partitioned constituent tree (over Σ)* is a tuple $\xi = (t, <, (U_1, \ldots, U_n))$ where $t \in \mathrm{T}_\Sigma$, $<$ is a strict total order on leaves(t), $n \in \mathbb{N}_+$, and $(U_1, \ldots, U_n)$ is a partitioning of leaves(ξ) such that, for every $i \in [n-1]$, $w_1 \in U_i$, and $w_2 \in U_{i+1}$, we have that $w_1 < w_2$. Intuitively, this condition on the partitioning enforces consistency with $<$, i.e., positions further left in $(U_1, \ldots, U_n)$ are smaller. We say that ξ *has n segments*. The set of all partitioned constituent trees over Σ is denoted by pC_Σ.

Compared to the constituent trees of [2], partitioned constituent trees abstract from particular indices. Thus, each partitioned constituent tree represents infinitely many constituent trees. To formalize this, we define the mapping $\mathrm{rep}\colon \mathrm{C}_\Sigma \to \mathrm{pC}_\Sigma$ as follows. Let $\xi \in \mathrm{C}_\Sigma$. If ξ is of the form $a\langle n \rangle$, we let $\mathrm{rep}(a\langle n \rangle) = (a, \emptyset, (\{\varepsilon\}))$. Otherwise, ξ is of the form $a(\xi_1, \ldots, \xi_k)$ and we let $\mathrm{rep}(\xi) = ((\xi)_\Sigma, <, (U_1, \ldots, U_n))$ where, for every $w_1, w_2 \in \mathrm{leaves}(\xi)$, we let $w_1 < w_2$ if and only if $(\xi(w_1))_\mathbb{N} < (\xi(w_2))_\mathbb{N}$ and $(U_1, \ldots, U_n)$ is the unique partitioning of leaves(ξ) such that, for each $m \in [n]$, the set $\{(\xi(w))_\mathbb{N} \mid w \in U_m\}$ is an interval and, for each $m \in [n-1]$, $\max_{w \in U_m}(\xi(w))_\mathbb{N} + 1 < \min_{w \in U_{m+1}}(\xi(w))_\mathbb{N}$. Intuitively, $<$ orders the leaves of ξ by their indices and $(U_1, \ldots, U_n)$ groups the leaves such that, for each subset of the partitioning, the indices of the leaves in that subset form an interval and this interval is as large as possible.

We remark that our partitioned constituent trees differ from the hybrid trees by [14] in three regards. (1) In the first component, we only allow a tree ξ rather than a sequence of trees. (2) The total order $<$ is defined on the set of leaves of ξ rather than the set of all positions of ξ whose labels are from a particular subset Γ of Σ. We note that [5] defined constituent trees[3] as a special case of hybrid trees where Γ makes up the leaf labels, hence that difference is only syntactical (also, this was already indicated by [14]). (3) Their hybrid trees did not feature a partitioning, so phrases with gaps cannot be modeled. Compared to the segmented totally ordered terms (tots) of [10], the total order of our partitioned constituent trees only regards the leaves rather than the entire set of positions.

Intuitively, the linear phrase represented by a partitioned constituent tree $(t, <, (U_1, \ldots, U_n))$ can be obtained analogously to the yield of constituent trees in C_Σ; we merely order the symbols at the leaves according to $<$ rather than by their index and we place commas according to $(U_1, \ldots, U_n)$ rather than gaps in the indices. We formalize this by defining the mapping $\text{p-yield}\colon \mathrm{pC}_\Sigma \to \mathrm{Tup}(\Sigma^*)$ as follows. Let $\xi = (t, <, (U_1, \ldots, U_n))$ be in pC_Σ. Then

$$\text{p-yield}(\xi) = (f_{t,<}(U_1), \ldots, f_{t,<}(U_n))$$

where the auxiliary mapping $f_{t,<}$ is inductively defined by $f_{t,<}(\emptyset) = \varepsilon$ and, for nonempty $U \subseteq \mathrm{leaves}(t)$, $f_{t,<}(U) = t(\min_< U) \cdot f_{t,<}(U \setminus \{\min_< U\})$.

It is easy prove that, for every $\xi \in \mathrm{C}_\Sigma$, we have

$$\mathrm{yield}(\xi) = \text{p-yield}(\mathrm{rep}(\xi)), \tag{1}$$

i.e., intuitively, the mapping rep preserves yield.

[3] They refer to constituent trees as phrase structure trees.

5.1 The constituent tree algebra and the constituent tree yield algebra

Prior to the definition of the algebras we give the formal definition of their signature Γ. The intuition behind our choice of sorts is the observation that the elements of both algebras, partitioned constituent trees and string tuples, have a certain "arity": each partitioned constituent tree has n segments, i.e., groups of leaves, and each string tuple consists of n strings where, in both cases, $n \in \mathbb{N}_+$.

We define the $((\mathbb{N}_+)^* \times \mathbb{N}_+)$-sorted set $\Gamma = \Gamma^{(\varepsilon,1)} \cup \bigcup_{n,k,\ell_1,\ldots,\ell_k \in \mathbb{N}_+} \Gamma^{(\ell_1 \cdots \ell_k, n)}$ where

- $\Gamma^{(\varepsilon,1)} = \Sigma^{(0)}$ and
- for every $n, k, \ell_1, \ldots, \ell_k \in \mathbb{N}_+$, we let

$$\Gamma^{(\ell_1 \cdots \ell_k, n)} = \{(a, e) \mid a \in \Sigma^{(k)}, e \in \mathbb{W}^n_{(\ell_1,\ldots,\ell_k)}\}.$$

Now we can approach the definition of the constituent tree algebra as a Γ-algebra whose carrier set is pC_Σ. For this, we consider pC_Σ as an $\mathbb{N}_+$-sorted set by letting, for every $n \in \mathbb{N}_+$,

$$(\mathrm{pC}_\Sigma)^{(n)} = \{\xi \in \mathrm{pC}_\Sigma \mid \xi \text{ has } n \text{ segments}\}.$$

The *constituent tree algebra* is the $\mathbb{N}_+$-sorted Γ-algebra $\mathscr{CT} = (\mathrm{pC}_\Sigma, \theta_\Sigma)$ where

- for each $a \in \Sigma^{(0)}$, we let $\theta_\Sigma(a) = (a, \emptyset, (\{\varepsilon\}))$ and
- for every $(a, e) \in \Gamma^{(\ell_1 \cdots \ell_k, n)}$ and $\xi_1 \in (\mathrm{pC}_\Sigma)^{(\ell_1)}, \ldots, \xi_k \in (\mathrm{pC}_\Sigma)^{(\ell_k)}$ with $\xi_i = (t_i, <_i, (U_1^{(i)}, \ldots, U_i^{(\ell_i)}))$ (for $i \in [k]$), we let

$$\theta_\Sigma(a, e)(\xi_1, \ldots, \xi_k) = (t, <, (U_1, \ldots, U_n))$$

where $t = a(t_1, \ldots, t_k)$ and, for each $m \in [n]$, we let U_m be the union of all sets $\{i\} \cdot U_i^{(j)}$ such that x_i^j occurs in the m-th component of e. Thus, clearly, $(U_1, \ldots, U_n)$ is a partitioning of $\mathrm{leaves}(t)$. Hence, for each $w \in \mathrm{leaves}(t)$, there exist exactly one $i \in [k]$ and $j \in [\ell_i]$ such that $w \in \{i\} \cdot U_i^{(j)}$; we let $\mathrm{var}(w)$ denote x_i^j. For the definition of $<$, let $w_1, w_2 \in \mathrm{leaves}(t)$. If $\mathrm{var}(w_1) \neq \mathrm{var}(w_2)$, then we let $w_1 < w_2$ if and only if $\mathrm{var}(w_1)$ occurs left of $\mathrm{var}(w_2)$ in e. Otherwise, we let $i \in [k]$ and $j \in [\ell_i]$ such that $\mathrm{var}(w_1) = x_i^j$. Then $w_1 < w_2$ if and only if $w_1' <_i w_2'$ where $w_1', w_2' \in \mathrm{pos}(t_i)$ such that $w_1 = i \cdot w_1'$ and $w_2 = i \cdot w_2'$.

We let $(\cdot)_{\mathscr{CT}}$ denote the unique Γ-homomorphism from T_Γ to pC_Σ.

We note that the definition of $\mathscr{CT}$ is semantically close to the algebra of segmented tots by [10], but the operations of $\mathscr{CT}$ are defined using word tuples and the non-nullary symbols of Γ do not add tree positions to the total order or the partitioning since, in our case, these components only refer to the leaves. Moreover, one cannot define a Γ-algebra similar to $\mathscr{CT}$ but with C_Σ as its carrier set. For this, one would need to fix a mapping $\mathrm{sort}\colon \mathrm{C}_\Sigma \to \mathbb{N}_+$. An appropriate choice could be assigning to each $\xi \in \mathrm{C}_\Sigma$ the smallest number n such that the indices of ξ form n intervals. For instance, let $\xi_1 = a\langle 2 \rangle$ and $\xi_2 = b(a\langle 1 \rangle, a\langle 4 \rangle)$ be constituent trees over Σ. Then we have $\mathrm{sort}(\xi_1) = 1$ and $\mathrm{sort}(\xi_2) = 2$. In essence, this mimics the sort mapping of pC_Σ but considers intervals of indices rather than the partitioning of the set of leaves. However, this approach bears the following problem. Let $c \in \Sigma^{(2)}$ and $e = (x_2^1 x_1^1, x_2^2)$. We compute $\theta_\Sigma(c, e)(\xi_1, \xi_2) = a(\xi_1, \xi_2)$ and have $\mathrm{sort}(a(\xi_1, \xi_2)) = 2$. On the other hand, if we also consider $\xi_3 = b(a\langle 1 \rangle, a\langle 3 \rangle)$, then $\theta_\Sigma(c, e)(\xi_1, \xi_3)$ has sort 1 which contradicts the sort of (c, e). Moreover, this sort mapping falls short of inhibiting that constituent trees with overlapping indices are passed as arguments to $\theta_\Sigma(e)$. The rich field of many-sorted algebra surely provides means to remedy these problems by choosing a more complex signature rather than Γ. However, we believe that circumventing these problems by dealing with pC_Σ is a cleaner solution.

We define the *constituent tree yield algebra* to be the $\mathbb{N}_+$-sorted Γ-algebra $(\mathrm{Tup}(\Sigma^*), \theta_Y)$ where

- for each $n \in \mathbb{N}_+$, we let $\mathrm{sort}(\mathrm{Tup}_n(\Sigma^*)) = n$,
- for each $a \in \Sigma^{(0)}$, we let $\theta_{\mathrm{Y}}(a) = (a)$, and
- for each $(a,e) \in \Gamma^{(\ell_1 \cdots \ell_k, n)}$, we let $\theta_{\mathrm{Y}}(a,e) = [\![e]\!]$.

Let $(\cdot)_{\mathrm{Y}}$ denote the unique homomorphism from T_Γ to $\mathrm{Tup}(\Sigma^*)$. We can also show that the mapping p-yield is a Γ-homomorphism from pC_Σ to $\mathrm{Tup}(\Sigma^*)$. Thus, by the laws of universal algebra (cf., e.g., [16]), we obtain that, for every $t \in \mathrm{T}_\Gamma$,

$$\text{p-yield}((t)_{\mathscr{C}\mathscr{T}}) = (t)_{\mathrm{Y}}. \tag{2}$$

5.2 The wRTG-LM for constituency parsing

Here we show, given a CTA $\mathscr{A}$ and a string u, how to construct a wRTG-LM such that the corresponding M-monoid parsing problem is equivalent to the constituency parsing problem for $\mathscr{A}$ and u. We start with the RTG and afterwards add the algebras from the previous subsection.

Let $\mathscr{A} = (Q, \Sigma, \delta, q_f)$ be a CTA. We define the *$\mathscr{A}$-RTG* to be the RTG $\mathscr{G} = (Q, \Lambda, R, q_f)$ where

(1) $\Lambda = \Lambda^{(\varepsilon,1)} \cup \bigcup_{k \in \mathbb{N}_+, n, \ell_1, \ldots, \ell_k \in \mathrm{rk}(Q)} \Lambda^{(\ell_1 \cdots \ell_k, n)}$ where we let $\Lambda^{(\varepsilon,1)} = \Sigma^{(0)}$ and, for each $k \in \mathbb{N}_+$ and every $n, \ell_1, \ldots, \ell_k \in \mathrm{rk}(Q)$, we let $\Lambda^{(\ell_1 \cdots \ell_k, n)} = \Sigma^{(k)} \times \mathbb{W}^n_{(\ell_1,\ldots,\ell_k)}$,[4]

(2) for every $a \in \Sigma^{(0)}$ and $q \in Q$ it holds that $(\varepsilon, a, q) \in \delta$ if and only if $(q \to (a)) \in R$, and

(3) for every $k \in \mathbb{N}_+$, $a \in \Sigma^{(k)}$, $e \in \mathbb{W}$, and $q_1, \ldots, q_k, q \in Q$ it holds that $(q_1 \ldots q_k, a, e, q) \in \delta$ if and only if $(q \to (a,e)(q_1, \ldots, q_k)) \in R$.

Example 4. Recall the CTA $\mathscr{A} = (Q, \Sigma, \delta, q_f)$ from Example 2. The $\mathscr{A}$-RTG is $\mathscr{G} = (Q, \Lambda, R, q_f)$ where $\Lambda^{(\varepsilon,1)} = \{a, b, c\}$, for every $n, \ell_1, \ell_2, \ell_3 \in [3]$, $\Lambda^{(\ell_1\ell_2, n)} = \{e\} \times \mathbb{W}^n_{(\ell_1,\ell_2)}$ and $\Lambda^{(\ell_1\ell_2\ell_3, n)} = \{d\} \times \mathbb{W}^n_{(\ell_1,\ell_2,\ell_3)}$; and R consists of the following rules:

$$q_f \to (d, (x^1_1 x^1_2 x^2_1 x^2_2 x^1_3 x^3_2))(q_l, q, q_c) \qquad q_f \to (d, (x^1_1 x^1_2 x^1_3 x^2_2 x^2_3 x^3_2))(q_a, q, q_r)$$
$$q \to (d, (x^1_1 x^1_2, x^2_1 x^2_2, x^1_3 x^3_2))(q_l, q, q_c) \qquad q \to (d, (x^1_1 x^1_2, x^1_3 x^2_2, x^2_3 x^3_2))(q_a, q, q_r)$$
$$q \to (d, (x^1_1, x^1_2, x^1_3))(q_a, q_b, q_c) \qquad q_l \to (e, (x^1_1, x^1_2))(q_a, q_b) \qquad q_r \to (e, (x^1_1, x^1_2))(q_b, q_c)$$
$$q_a \to (a) \qquad q_b \to (b) \qquad q_c \to (c).$$

The bottom right of Figure 2 shows an AST of $\mathscr{G}$. ◁

As the language algebra of our wRTG-LM we use the constituent tree yield algebra $(\mathrm{Tup}(\Sigma^*), \theta_{\mathrm{Y}})$. For the weight algebra we point out that each of its operations computes a single partitioned constituent tree. However, our goal as determined by the constituency parsing problem is to compute a *set* of constituent trees. Hence, we lift the constituent tree algebra to sets. Formally, we define the M-monoid

$$\mathbb{C} = (\bigcup_{n \in \mathbb{N}_+} \mathscr{P}(\mathrm{pC}^{(n)}_\Sigma) \cup \{\bot\}, \mathbb{N}, \emptyset, \Gamma, \theta'_\Sigma, \Sigma^{\mathbb{N}})$$

where $\mathbb{N}$ and θ'_Σ are defined like their counterparts in Example 1. In the following, we will write θ_Σ rather than θ'_Σ and we let $(\cdot)_{\mathscr{C}\mathscr{T}}$ denote also the unique Γ-homomorphism from T_Γ to this algebra.

Combining these components, we define the *$\mathscr{A}$-wRTG-LM* to be the wRTG-LM

$$\bar{G} = ((\mathscr{G}, (\mathrm{Tup}(\Sigma^*), \theta_{\mathrm{Y}})), \mathbb{C}, wt)$$

where $\mathscr{G}$ is the $\mathscr{A}$-RTG and, for every $r = (A \to \gamma(A_1, \ldots, A_k))$ in R, we let $wt(r) = \gamma$.

[4] Thus Λ is a finite subset of Γ.

5.3 Constituency parsing is an instance of the M-monoid parsing problem

Let $\mathscr{A} = (Q, \Sigma, \delta, q_f)$ be a CTA and let $\bar{G} = ((\mathscr{G}, (\mathrm{Tup}(\Sigma^*), \theta_Y)), \mathbb{C}, wt)$ with $\mathscr{G} = (Q, \Lambda, R, q_f)$ be the $\mathscr{A}$-wRTG-LM. The M-monoid parsing problem for $\bar{G}$ is, given some $(u) \in \mathrm{Tup}(\Sigma^*)$, to compute

$$\mathrm{parse}_{(\mathscr{G},\mathrm{C}_\Sigma)}(u) = \bigcup_{\substack{d \in \mathrm{T}_R: \\ ((d)_\Gamma)_Y = (u), \mathrm{lhs}(d(\varepsilon)) = q_f}} (wt(d))_{\mathscr{C}\mathscr{T}}.$$

For a given phrase u, this instance of the M-monoid parsing problem enumerates the set of all ASTs of $\mathscr{G}$ that have the initial nonterminal at the root and evaluate to u in the constituent tree yield algebra. Each of these ASTs is evaluated in the constituent tree algebra.

In order to show that this M-monoid parsing problem is equal to the constituency parsing problem for $\mathscr{A}$ (and u), we seek a bijection ψ between the set $\mathrm{CR}_\mathscr{A}$ and the set of abstract syntax trees of $\mathscr{G}$. However, similar to [2], we only find such a bijection if we consider certain elements of $\mathrm{CR}_\mathscr{A}$ equivalent.

Formally, we define the equivalence relation $\sim$ as follows. For every $(\xi_1, \rho_1), (\xi_2, \rho_2) \in \mathrm{CR}_\mathscr{A}$, we let $(\xi_1, \rho_1) \sim (\xi_2, \rho_2)$ if and only if $(\xi_1)_\Sigma = (\xi_2)_\Sigma$ and $\rho_1 = \rho_2$. Clearly, $\sim$ is indeed an equivalence relation. Let $\mathrm{CR}_\mathscr{A}/_\sim$ denote the quotient set of $\mathrm{CR}_\mathscr{A}$ by $\sim$. For each $(\xi, \rho) \in \mathrm{CR}_\mathscr{A}$, we let $[\xi, \rho]$ denote the equivalence class (ξ, ρ) belongs to. An example for $\sim$ is given in the top of Figure 2.

We define the mapping $\psi\colon \mathrm{CR}_\mathscr{A}/_\sim \to \mathrm{T}_R$ inductively as follows. Let $(\xi, \rho) \in \mathrm{CR}_\mathscr{A}$. If (ξ, ρ) is of the form $(a\langle n\rangle, q)$, we let $\psi([a\langle n\rangle, q]) = q \to (a)$. Otherwise, ξ is of the form $a(\xi_1, \ldots, \xi_k)$ and ρ is of the form $(q, e)(\rho_1, \ldots, \rho_k)$, then we let

$$\psi([\xi, \rho]) = q \to (a, e)(\psi([\xi_1, \rho_1]), \ldots, \psi([\xi_k, \rho_k])).$$

We illustrate ψ for the CTA $\mathscr{A}$ from Example 2 and the $\mathscr{A}$-RTG $\mathscr{G}$ from Example 4 in Figure 2.

Using a method similar to [2] we can show that ψ is indeed a bijection. Moreover, we can prove the following auxiliary statement. Let $(\xi, \rho) \in \mathrm{CR}_\mathscr{A}$. If $((\psi([\xi, \rho]))_\Gamma)_{\mathscr{C}\mathscr{T}} = (t_1, <_1, (U_1^{(1)}, \ldots, U_1^{(\ell_1)}))$ and $\mathrm{rep}(\xi) = (t_2, <_2, (U_2^{(1)}, \ldots, U_2^{(\ell_2)}))$, then

$$t_1 = t_2 \quad \text{and} \quad <_1 = <_2. \tag{3}$$

Intuitively, $((\psi([\xi, \rho]))_\Gamma)_{\mathscr{C}\mathscr{T}}$ and $\mathrm{rep}(\xi)$ may only differ in the partitioning. For instance, consider the constituent tree ξ and the run ρ in Figure 2 where we even have $((\psi([\xi, \rho]))_\Gamma)_{\mathscr{C}\mathscr{T}} = \mathrm{rep}(\xi)$.

We note that since $\mathscr{A}$ is final state normalized, ψ implies that each AST d of $\mathscr{G}$ with $\mathrm{lhs}(d(\varepsilon)) = q_f$ has $((d)_\Gamma)_Y \in \Sigma^*$. Thus, $\mathrm{parse}_{(\mathscr{G},\mathrm{C}_\Sigma)}(u)$ is only non-empty if u is a string. This resembles the fact that the constituency parsing problem is only defined for strings. We will assume that $u \in \Sigma^*$ in the following.

After these preparations, we can show that the M-monoid parsing problem for $\bar{G}$ and u relates to the constituency parsing problem for $\mathscr{A}$ and u in the following way:

$$\begin{aligned}
\mathrm{parse}_{(\mathscr{G},\mathrm{C}_\Sigma)}(u) &= \{(wt(d))_{\mathscr{C}\mathscr{T}} \mid d \in \mathrm{T}_R, ((d)_\Gamma)_Y = u, \mathrm{lhs}(d(\varepsilon)) = q_f\} \\
&\overset{(2)}{=} \{(wt(d))_{\mathscr{C}\mathscr{T}} \mid d \in \mathrm{T}_R, \text{p-yield}(((d)_\Gamma)_{\mathscr{C}\mathscr{T}}) = u, \mathrm{lhs}(d(\varepsilon)) = q_f\} \\
&\overset{\text{bij.}}{=} \{(wt(\psi([\xi, \rho])))_{\mathscr{C}\mathscr{T}} \mid (\xi, \rho) \in \mathrm{CR}_\mathscr{A}, \text{p-yield}(((\psi([\xi, \rho]))_\Gamma)_{\mathscr{C}\mathscr{T}}) = u, \\
&\qquad\qquad \mathrm{lhs}(\psi([\xi, \rho])(\varepsilon)) = q_f\} \\
&\overset{\star_1}{=} \{(wt(\psi([\xi, \rho])))_{\mathscr{C}\mathscr{T}} \mid (\xi, \rho) \in \mathrm{CR}_\mathscr{A}, \text{p-yield}(((\psi([\xi, \rho]))_\Gamma)_{\mathscr{C}\mathscr{T}}) = u, \\
&\qquad\qquad q_f \text{ is the state at } \rho(\varepsilon)\}
\end{aligned}$$

$$\overset{\star_2}{=} \{\mathrm{rep}(\xi) \mid (\xi,\rho) \in \mathrm{CR}_{\mathscr{A}}, \text{p-yield}(\mathrm{rep}(\xi)) = u, q_f \text{ is the state at } \rho(\varepsilon)\}$$
$$\overset{(1)}{=} \{\mathrm{rep}(\xi) \mid (\xi,\rho) \in \mathrm{CR}_{\mathscr{A}}, \mathrm{yield}(\xi) = u, q_f \text{ is the state at } \rho(\varepsilon)\}$$
$$= \{\mathrm{rep}(\xi) \mid \xi \in \mathrm{L}_{\mathrm{ind}}(\mathscr{A}), \mathrm{yield}(\xi) = u\}$$

where $\star_1$ holds by definition of ψ and $\star_2$ follows from (3) (using $wt = (\cdot)_\Gamma$) and both $\mathrm{rep}(\xi)$ and $(wt(\psi([\xi,\rho])))_{\mathscr{CT}}$ being in $\mathrm{pC}_\Sigma^{(1)}$ (which is a consequence of $\mathscr{A}$ being final state normalized). We illustrate this equality by showing how the mapping rep commutes with $(\cdot)_{\mathscr{CT}} \circ (\cdot)_\Gamma \circ \psi$ in Figure 2.

We note that $\mathrm{parse}_{(\mathscr{G},\mathrm{C}_\Sigma)}(u)$ is a subset of pC_Σ (i.e., constituent trees without particular indices) whereas the constituency parsing problem computes a subset of C_Σ. To bridge this gap, we note that the set $T = \{\xi \in \mathrm{C}_\Sigma \mid \mathrm{rep}(\xi) \in \mathrm{parse}_{(\mathscr{G},\mathrm{C}_\Sigma)}(u)\}$ can be easily constructed. We sketch this construction by letting $\xi = (t, <, (U_1, \ldots, U_n)) \in \mathrm{parse}_{(\mathscr{G},\mathrm{C}_\Sigma)}(u)$. Since $\mathscr{A}$ is final state normalized, we have $n = 1$. Now we let $m \in \mathbb{N}_+$ and fix an interval $[m, m+|U_1|]$, then we obtain $\xi' \in \mathrm{C}_\Sigma$ from t by adding the indices $m, m+1, \ldots, m+|U_1|$ to the symbols at the leaves of t in the order determined by $<$. Clearly, $\mathrm{rep}(\xi') = \xi$. By letting m range over $\mathbb{N}_+$ we obtain the set $\{\xi' \in \mathrm{C}_\Sigma \mid \mathrm{rep}(\xi') = \xi\}$. Clearly, for every $\xi \in T$ we have $\mathrm{yield}(\xi) = u$ and $\xi \in \mathrm{L}_{\mathrm{ind}}(\mathscr{A})$. Thus T is the solution of the constituency parsing problem of $\mathscr{A}$ and u.

6 Applicability of the M-monoid parsing algorithm

The M-monoid parsing algorithm [13] is a two-phase pipeline which is applicable to a large class of M-monoid parsing problems, where applicability means that the algorithm is terminating and correct. Due to space restrictions, we cannot repeat the algorithm here and only investigate its applicability to our scenario. For this, we let $\mathscr{W}(\mathrm{CTA})$ be the set of all $\mathscr{A}$-wRTG-LMs for each final state normalized CTA $\mathscr{A}$. We let $\bar{G} \in \mathscr{W}(\mathrm{CTA})$ and $u \in \Sigma^*$.

The first phase of the M-monoid parsing algorithm applies a weighted deduction system to $\bar{G}$ and u, thus obtaining a new wRTG-LM $\bar{G}'$. Mörbitz and Vogler (2021) [13] provide the canonical weighted deduction system which is applicable in all situations where the language algebra of the input wRTG-LM is finitely decomposable. Since this is clearly the case for $(\mathrm{Tup}(\Sigma^*), \theta_{\mathrm{Y}})$, we obtain that the first phase of the M-monoid parsing algorithm is applicable to every $\bar{G} \in \mathscr{W}(\mathrm{CTA})$ and $u \in \Sigma^*$.

The second phase, called value computation algorithm, uses $\bar{G}'$ to compute an element in the weight algebra. There are two independent sufficient conditions for this value to be equal to $\mathrm{parse}_{(\mathscr{G},\mathrm{C}_\Sigma)}(u)$. The first condition requires $\bar{G}$ to fulfil a property called closed. Without giving details on this property, we state that not every wRTG-LM in $\mathscr{W}(\mathrm{CTA})$ is closed.[5] The second condition requires $\bar{G}$ to fulfil a property called nonlooping and the weight algebra to be distributive and d-complete. Now distributivity of $\mathbb{C}$ is easy to see and d-completeness of $\mathbb{C}$ follows from the fact that its additive monoid is completely idempotent. In essence, $\bar{G}$ is nonlooping if for each AST d of its RTG the following holds: if there is a proper subtree $d|_w$ of d which evaluates to the same syntactic object as d in the language algebra, then $d(w)$ must have a different label than $d(\varepsilon)$. As our language algebra is $(\mathrm{Tup}(\Sigma^*), \theta_{\mathrm{Y}})$, this property can only be violated if each node in d from ε to w is monadic. Then, by pumping the monadic chain from ε to w, we can construct infinitely many ASTs with the same yield, each of which is evaluated to a different constituent tree in the weight algebra. However, a terminating algorithm cannot compute an infinite set of constituent trees. By the construction of $\bar{G}$, we find that this situation is only possible if the CTA $\mathscr{A}$ contains transitions of the form (q_1, a_1, e_1, q_2), (q_2, a_2, e_2, q_3), …, (q_n, a_n, e_n, q_1). Thus, if $\mathscr{A}$ is free of such monadic cycles, $\bar{G}$ is nonlooping and the M-monoid parsing algorithm is correct for $\bar{G}$ and u.

[5] The interested reader may see that for themselves in a way similar to [13, Appendix A.7].

7 Future work

Dependency is another important syntactical analysis in NLP. Dependency trees are also introduced by [2] where dependency tree automata are mentioned as another possible special case of HTA, mirroring CTA. We believe that the corresponding dependency parsing problem can be shown to be an instance of M-monoid parsing in a way very similar to the present paper.

The constituency parsing problem considered here states to compute the set of all suitable constituent trees. However, parsing problems often occur in weighted settings where the weights are, e.g., probabilities, and compute only the best analysis. A constituency parsing problem with such additional weights also falls in the scope of the M-monoid parsing problem. Moreover, the underlying CTA could even have transitions that allow monadic cycles as long as they lead to a decrease in weight.

References

[1] W.S. Brainerd (1969): *Tree generating regular systems*. *Information and Control* 14(2), pp. 217–231, doi:10.1016/S0019-9958(69)90065-5.

[2] F. Drewes, R. Mörbitz & H. Vogler (2022): *Hybrid Tree Automata and the Yield Theorem for Constituent Tree Automata*. In: *26th International Conference on Implementation and Application of Automata*, *LNCS* 13266, Springer, pp. 93–105, doi:10.1007/978-3-031-07469-1_7.

[3] M. Droste, C. Pech & H. Vogler (2005): *A Kleene theorem for weighted tree automata*. *Theory of Computing Systems* 38(1), pp. 1–38, doi:10.1007/s00224-004-1096-z.

[4] S. Eilenberg (1974): *Automata, languages, and machines*. Academic press.

[5] K. Gebhardt, M.-J. Nederhof & H. Vogler (2017): *Hybrid Grammars for Parsing of Discontinuous Phrase Structures and Non-Projective Dependency Structures*. *Computational Linguistics* 43(3), pp. 465–520, doi:10.1162/COLI_a_00291.

[6] J.A. Goguen, J.W. Thatcher, E.G. Wagner & J.B. Wright (1977): *Initial algebra semantics and continuous algebras*. *Journal of the ACM (JACM)* 24(1), pp. 68–95, doi:10.1145/321992.321997.

[7] A.K. Joshi & Y. Schabes (1997): *Tree-Adjoining Grammars*. In: *Handbook of Formal Languages*, Springer, pp. 69–123, doi:10.1007/978-3-642-59126-6_2.

[8] L. Kallmeyer (2010): *Parsing beyond context-free grammars*. Springer, doi:10.1007/978-3-642-14846-0.

[9] G. Karner (1992): *On limits in complete semirings*. *Semigroup Forum* 45(1), doi:10.1007/BF03025757.

[10] Kuhlmann & Niehren (2008): *Logics and Automata for Totally Ordered Trees*. In: *Rewriting Techniques and Applications*, Springer Berlin Heidelberg, Berlin, pp. 217–231, doi:10.1007/978-3-540-70590-1_15.

[11] W. Kuich (1999): *Linear systems of equations and automata on distributive multioperator monoids*. *Contributions to general algebra* 12, pp. 247–256.

[12] R. Mörbitz & H. Vogler (2019): *Weighted parsing for grammar-based language models*. In: *Proceedings of the 14th International Conference on Finite-State Methods and Natural Language Processing*, Association for Computational Linguistics, Dresden, Germany, pp. 46–55, doi:10.18653/v1/W19-3108.

[13] R. Mörbitz & H. Vogler (2021): *Weighted parsing for grammar-based language models over multioperator monoids*. *Information and Computation* 281, doi:10.1016/j.ic.2021.104774.

[14] M.-J. Nederhof & H. Vogler (2014): *Hybrid grammars for discontinuous parsing*. In: *Proc. of 25th International Conference on Computational Linguistics*. Available at `https://aclanthology.org/C14-1130`.

[15] H. Seki, T. Matsumura, M. Takashi, M. Fujii & T. Kasami (1991): *On multiple context-free grammars* 88(2), pp. 191–229. doi:10.1016/0304-3975(91)90374-B.

[16] W. Wechler (1992): *Universal Algebra for Computer Scientists*, first edition. *Monogr. Theoret. Comput. Sci. EATCS Ser.* 25, Springer-Verlag, Heidelberg/Berlin, doi:10.1007/978-3-642-76771-5_3.

Forgetting 1-Limited Automata

Giovanni Pighizzini
Dipartimento di Informatica
Università degli Studi di Milano, Italy
pighizzini@di.unimi.it

Luca Prigioniero
Department of Computer Science
Loughborough University, UK
l.prigioniero@lboro.ac.uk

We introduce and investigate *forgetting* 1*-limited automata*, which are single-tape Turing machines that, when visiting a cell for the first time, replace the input symbol in it by a fixed symbol, so forgetting the original contents. These devices have the same computational power as finite automata, namely they characterize the class of regular languages. We study the cost in size of the conversions of forgetting 1-limited automata, in both nondeterministic and deterministic cases, into equivalent one-way nondeterministic and deterministic automata, providing optimal bounds in terms of exponential or superpolynomial functions. We also discuss the size relationships with two-way finite automata. In this respect, we prove the existence of a language for which forgetting 1-limited automata are exponentially larger than equivalent minimal deterministic two-way automata.

1 Introduction

Limited automata have been introduced in 1967 by Hibbard, with the aim of generalizing the notion of determinism for context-free languages [6]. These devices regained attention in the last decade, mainly from a descriptional complexity point of view, and they have been considered in several papers, starting with [14, 15]. (For a recent survey see [13].)

In particular, 1*-limited automata* are single-tape nondeterministic Turing machines that are allowed to rewrite the content of each tape cell only in the first visit. They have the same computational power as finite automata [24, Thm. 12.1], but they can be extremely more succinct. Indeed, in the worst case the size gap from the descriptions of 1-limited automata to those of equivalent one-way deterministic finite automata is double exponential [14].

In order to understand this phenomenon better, we recently studied two restrictions of 1-limited automata [17]. In the first restriction, called *once-marking* 1*-limited automata*, during each computation the machine can make only one change to the tape, just marking exactly one cell during the first visit to it. We proved that, under this restriction, a double exponential size gap to one-way deterministic finite automata remains possible.

In the second restriction, called *always-marking* 1*-limited automata*, each tape cell is marked during the first visit. In this way, at each step of the computation, the original content in the cell remains available, together with the information saying if it has been already visited at least one time. In this case, the size gap to one-way deterministic finite automata reduces to a single exponential. However, the information about which cells have been already visited still gives extra descriptional power. In fact, the conversion into equivalent two-way finite automata in the worst case costs exponential in size, even if the original machine is deterministic and the target machine is allowed to make nondeterministic choices.

A natural way to continue these investigations is to ask what happens if in each cell the information about the original input symbol is lost after the first visit. This leads us to introduce and study the subject of this paper, namely *forgetting* 1*-limited automata*. These devices are 1-limited automata in which, during the first visit to a cell, the input symbol in it is replaced with a unique fixed symbol. Forgetting

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 95–109, doi:10.4204/EPTCS.388.10

automata have been introduced in the literature longtime ago [8]. Similarly to the devices we consider here, they can use only one fixed symbol to replace symbols on the tape. However, the replacement is not required to happen in the first visit, so giving the possibility to recognize more than regular languages. In contrast, being a restriction of 1-limited automata, forgetting 1-limited automata recognize only regular languages.

In this paper, first we study the size costs of the simulations of forgetting 1-limited automata, in both nondeterministic and deterministic versions, by one-way finite automata. The upper bounds we prove are exponential, when the simulated and the target machines are nondeterministic and deterministic, respectively. In the other cases they are superpolynomial. These bounds are obtained starting from the conversions of always-marking 1-limited automata into one-way finite automata presented in [17], whose costs, in the case we are considering, can be reduced using techniques and results derived in the context of automata over a one-letter alphabet [2, 11]. We also provide witness languages showing that these upper bounds cannot be improved asymptotically.

In the last part of the paper we discuss the relationships with the size of two-way finite automata, which are not completely clear. We show that losing the information on the input content can reduce the descriptional power. In fact, we show languages for which forgetting 1-limited automata, even if nondeterministic, are exponentially larger than minimal two-way deterministic finite automata. We conjecture that also the converse can happen. In particular we show a family of languages for which we conjecture that two-way finite automata, even if nondeterministic, must be significantly larger than minimal deterministic forgetting 1-limited automata.

2 Preliminaries

In this section we recall some basic definitions useful in the paper. Given a set S, $\#S$ denotes its cardinality and 2^S the family of all its subsets. Given an alphabet Σ and a string $w \in \Sigma^*$, $|w|$ denotes the length of w, $|w|_a$ the number of occurrences of a in w, and Σ^k the set of all strings on Σ of length k.

We assume the reader to be familiar with notions from formal languages and automata theory, in particular with the fundamental variants of finite automata (1DFAs, 1NFAs, 2DFAs, 2NFAs, for short, where 1/2 mean *one-way/two-way* and D/N mean *deterministic/nondeterministic*, respectively). For any unfamiliar terminology see, e.g., [7].

A 1-*limited automaton* (1-LA, for short) is a tuple $A = (Q, \Sigma, \Gamma, \delta, q_I, F)$, where Q is a finite *set of states*, Σ is a finite *input alphabet*, Γ is a finite *work alphabet* such that $\Sigma \cup \{\rhd, \lhd\} \subseteq \Gamma$, $\rhd, \lhd \notin \Sigma$ are two special symbols, called the *left* and the *right end-markers*, $\delta : Q \times \Gamma \to 2^{Q \times (\Gamma \setminus \{\rhd, \lhd\}) \times \{-1, +1\}}$ is the *transition function*, and $F \subseteq Q$ is a set of final states. At the beginning of the computation, the input word $w \in \Sigma^*$ is stored onto the tape surrounded by the two end-markers, the left end-marker being in position zero and the right end-marker being in position $|w| + 1$. The head of the automaton is on cell 1 and the state of the finite control is the *initial state* q_I.

In one move, according to δ and the current state, A reads a symbol from the tape, changes its state, replaces the symbol just read from the tape with a new symbol, and moves its head to one position forward or backward. Furthermore, the head cannot pass the end-markers, except at the end of computation, to accept the input, as explained below. Replacing symbols is allowed to modify the content of each cell only during the first visit, with the exception of the cells containing the end-markers, which are never modified. Hence, after the first visit, a tape cell is "frozen". More technical details can be found in [14].

The automaton A accepts an input w if and only if there is a computation path that starts from the initial state q_I with the input tape containing w surrounded by the two end-markers and the head on the

first input cell, and which ends in a *final state* $q \in F$ after passing the right end-marker. The device A is said to be *deterministic* (D-1-LA, for short) whenever $\#\delta(q,\sigma) \leq 1$, for every $q \in Q$ and $\sigma \in \Gamma$.

We say that the 1-LA A is a *forgetting* 1-LA (for short F-1-LA or D-F-1-LA in the deterministic case), when there is only one symbol Z that is used to replace symbols in the first visit, i.e., the work alphabet is $\Gamma = \Sigma \cup \{Z\} \cup \{\rhd, \lhd\}$, with $Z \notin \Sigma$ and if $(q,A,d) \in \delta(p,a)$ and $a \in \Sigma$ then $A = Z$.

Two-way finite automata are limited automata in which no rewritings are possible; one-way finite automata can scan the input in a one-way fashion only. A finite automaton is, as usual, a tuple (Q,Σ,δ,q_I,F), where, analogously to 1-LAs, Q is the finite set of states, Σ is the finite input alphabet, δ is the transition function, q_I is the initial state, and F is the set of final states. We point out that for two-way finite automata we assume the same accepting conditions as for 1-LAs.

Two-way machines in which the direction of the head can change only at the end-markers are said to be *sweeping* [22].

In this paper we are interested in comparing the size of machines. The *size* of a model is given by the total number of symbols used to write down its description. Therefore, the size of 1-LAs is bounded by a polynomial in the number of states and of work symbols, while, in the case of finite automata, since no writings are allowed, the size is linear in the number of instructions and states, which is bounded by a polynomial in the number of states and in the number of input symbols. We point out that, since F-1-LAs use work alphabet $\Gamma = \Sigma \cup \{Z\} \cup \{\rhd, \lhd\}$, $Z \notin \Sigma$, the relevant parameter for evaluating the size of these devices is their number of states, differently than 1-LAs, in which the size of the work alphabet is not fixed, i.e., depends on the machine.

We now shortly recall some notions and results related to number theory that will be useful to obtain our cost estimations. First, given two integers m and n, let us denote by $\gcd(m,n)$ and by $\mathrm{lcm}(m,n)$ their *greatest common divisor* and *least common multiple*, respectively.

We remind the reader that each integer $\ell > 1$ can be factorized in a unique way as product of powers of primes, i.e., as $\ell = p_1^{k_1} \cdots p_r^{k_r}$, where $p_1 < \cdots < p_r$ are primes, and $k_1, \ldots, k_r > 0$.

In our estimations, we shall make use of the *Landau's function* $F(n)$ [9, 10], which plays an important role in the analysis of simulations among different types of unary automata (e.g. [2, 4, 11]). Given a positive integer n, let

$$F(n) = \max\{\mathrm{lcm}(\lambda_1, \ldots, \lambda_r) \mid \lambda_1 + \cdots + \lambda_r = n\},$$

where $\lambda_1, \ldots, \lambda_r$ denote, for the time being, arbitrary positive integers. Szalay [23] gave a sharp estimation of $F(n)$ that, after some simplifications, can be formulated as follows:

$$F(n) = e^{(1+o(1))\cdot\sqrt{n \cdot \ln n}}.$$

Note that the function $F(n)$ grows less than e^n, but more than each polynomial in n. In this sense we say that $F(n)$ is a *superpolynomial function*.

As observed in [5], for each integer $n > 1$ the value of $F(n)$ can also be expressed as the maximum product of powers of primes, whose sum is bounded by n, i.e.,

$$F(n) = \max\{p_1^{k_1} \cdots p_r^{k_r} \mid p_1^{k_1} + \cdots + p_r^{k_r} \leq n, \, p_1, \ldots, p_r \text{ are primes, and } k_1, \ldots, k_r > 0\}.$$

3 Forgetting 1-Limited Automata vs. One-Way Automata

When forgetting 1-limited automata visit a cell for the first time, they replace the symbol in it with a fixed symbol Z, namely they forget the original content. In this way, each input prefix can be rewritten in

a unique way. As already proved for *always-marking* 1-LAs, this prevents a double exponential size gap in the conversion to 1DFAs [17]. However, in this case the upper bounds obtained for always-marking 1-LAs, can be further reduced, using the fact that only one symbol is used to replace input symbols:

Theorem 1 *Let M be an n-state* F-1-LA. *Then M can be simulated by a* 1NFA *with at most $n \cdot (5n^2 + F(n)) + 1$ states and by a complete* 1DFA *with at most $(2^n - 1) \cdot (5n^2 + F(n)) + 2$ states.*

Proof. First of all, we recall the argument for the conversion of 1-LAs into 1NFAs and 1DFAs presented [14, Thm. 2] that, in turn, is derived from the technique to convert 2DFAs into equivalent 1DFAs, presented in [21], and based on *transitions tables*.

Let us start by supposing that $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ is an n-state 1-LA.

Roughly, transition tables represent the possible behaviors of M on "frozen" tape segments. More precisely, given $z \in \Gamma^*$, the *transition table* associated with z is the binary relation $\tau_z \subseteq Q \times Q$, consisting of all pairs (p,q) such that M has a computation path that starts in the state p on the rightmost symbol of a tape segment containing $\triangleright z$, ends reaching the state q by leaving the same tape segment to the right side, i.e., by moving from the rightmost cell of the segment to the right, and does not visit any cell outside the segment.

A 1NFA A can simulate M by keeping in the finite control two components:

- The transition table corresponding to the part of the tape at the left of the head. This part has been already visited and, hence, it is frozen.
- The state in which the simulated computation of M reaches the current tape position.

Since the number of transition tables is at most 2^{n^2}, the number of states in the resulting 1NFA A is bounded by $n \cdot 2^{n^2}$.

Applying the subset construction, this automaton can be converted into an equivalent deterministic one, with an exponential increasing in the number of states, so obtaining a double exponential number of states in n. In the general case, this number cannot be reduced due to the fact that different computations of A, after reading the same input, could keep in the control different transition tables, depending on the fact that M could replace the same input by different strings.

We now suppose that M is a F-1-LA. In this case each input string can be replaced by a unique string. This would reduce the cost of the conversion to 1DFAs to a single exponential. Indeed, it is possible to convert the 1NFA A obtained from M into an equivalent 1DFA that keeps in its finite control the *unique* transition table for the part of the tape scanned so far (namely, the same first component as in the state of A), and the set of states that are reachable by M when entering the current tape cell (namely, a set of states that can appear in the second component of A, while entering the current tape cell). This leads to an upper bound of $2^n \cdot 2^{n^2}$ states for the resulting 1DFA. We can make a further improvement, reducing the number of transition tables used during the simulation. Indeed we are going to prove that only a subset of all the possible 2^{n^2} transition tables can appear during the simulation.

Since only a fixed symbol Z is used to replace input symbols on the tape, the transition table when the head is in a cell depends only on the position of the cell and not on the initial tape content.

For each integer $m \geq 0$, let us call τ_m the transition table corresponding to a frozen tape segment of length m, namely the transition table when the head of the simulating one-way automaton is on the tape cell $m+1$. We are going to prove that the sequence $\tau_0, \tau_1, \ldots, \tau_m, \ldots$ is ultimately periodic, with period length bounded by $F(n)$ and, more precisely, $\tau_m = \tau_{m+F(n)}$ for each $m > 5n^2$.

The proof is based on the analysis of computation paths in unary 2NFAs carried on in [11, Section 3]. Indeed, we can see the parts of the computation on a frozen tape segment as computation paths of a unary 2NFA. More precisely, by definition, for $p, q \in Q$, $\tau_m(p,q) = 1$ if and only if there is a computation

path C that enters the frozen tape segment of length m from the right in the state p and, after some steps, exits the segment to the right in the state q. Hence, during the path C the head can visit only frozen cells (i.e., the cells in positions $1,\ldots,m$) of the tape, and the left end-marker. There are two possible cases:

- *In the computation path C the head never visits the left end-marker.*
  A path of this kind is also called *left U-turn*. Since it does not depend on the position of the left end-marker, this path will also be possible, suitably shifted to the right, on each frozen segment of length $m' > m$. Hence $\tau_{m'}(p,q) = 1$ for each $m' \geq m$. Furthermore, it has been proven that if there is a left U-turn which starts in the state p on cell m, and ends in state q, then there exists another left U-turn satisfying the same constraints, in which the head never moves farther than n^2 positions to the left of the position m [11, Lemma 3.1]. So, such a "short" U-turn can be shifted to the left, provided that the tape segment is longer than n^2.

  Hence, in this case $\tau_m(p,q) = 1$ implies $\tau_{m'}(p,q) = 1$ for each $m' > n^2$.
- *In the computation path C the head visits at least one time the left end-marker.*
  Let $s_0, s_1, \ldots, s_k$, $k \geq 0$, be the sequence of the states in which C visits the left end-marker. We can decompose C in a sequence of computation paths $C_0, C_1, \ldots, C_k, C_{k+1}$, where:
  - C_0 starts from the state p with the head on the cell m and ends in s_0 when the head reaches the left end-marker. C_0 is called *right-to-left traversal* of the frozen segment.
  - For $i = 1,\ldots,k$, C_i starts in state s_{i-1} with the head on the left end-marker and ends in s_i, when the head is back to the left end-marker. C_i is called *right U-turn*. Since, as seen before for left U-turns, each right U-turn can always be replaced by a "short" right U-turn, without loss of generality we suppose that C_i does not visit more than n^2 cells to the right of the left end-marker.
  - C_{k+1} starts from the state s_k with the head on the left end-marker and ends in q, when the head leaves the segment, moving to the right of the cell m. C_{k+1} is called *left-to-right traversal* of the frozen segment.

  From [11, Theorem 3.5], there exists a set of positive integers $\{\ell_1,\ldots,\ell_r\} \subseteq \{1,\ldots,n\}$ satisfying $\ell_1 + \cdots + \ell_r \leq n$ such that for $m \geq n$, if a frozen tape segment of length m can be (left-to-right or right-to-left) traversed from a state s to a state s' then there is an index $i \in \{1,\ldots,r\}$ such that, for each $\mu > \frac{5n^2-m}{\ell_i}$, a frozen tape segment of length $m + \mu\ell_i$ can be traversed (in the same direction) from state s to state s'. This was proved by showing that for $m > 5n^2$ a traversal from s to s' of a segment of length m can always be "pumped" to obtain a traversal of a segment of length $m' = m + \mu\ell_i$, for $\mu > 0$, and, furthermore, the segment can be "unpumped" by taking $\mu < 0$, provided that the resulting length m' is greater than $5n^2$.

  Let ℓ be the least common multiple of $\ell_1,\ldots,\ell_r$. If $m > 5n^2$, from the original computation path C, by suitably pumping or unpumping the parts C_0 and C_{k+1}, and without changing C_i, for $i = 1,\ldots,k$, for each $m' = m + \mu\ell > 5n^2$, with $\mu \in \mathbb{Z}$, we can obtain a computation path that enters a frozen segment of length m' from the right in the state p and exits the segment to the right in the state q.

By summarizing, from the previous analysis we conclude that for all $m, m' > 5n^2$, if $m \equiv m' \pmod{\ell}$ then $\tau_m = \tau_{m'}$. Hence, the transition tables used in the simulation are at most $5n^2 + \ell$. Since, by definition, ℓ cannot exceed $F(n)$, we obtain the number of different transitions tables that are used in the simulation is bounded by $5n^2 + 1 + F(n)$.

According with the construction outlined at the beginning of the proof, from the F-1-LA M we can obtain a 1NFA A that, when the head reaches the tape cell $m+1$, has in the first component of its finite

control the transition table τ_m, and in the second component the state in which the cell $m+1$ is entered for the first time during the simulated computation. Hence the total number of states of A is bounded by $n \cdot (5n^2 + 1 + F(n))$.

We observe that, at the beginning of the computation, the initial state is the pair containing the transition matrix τ_0 and the initial state of M. Hence, we do not need to consider other states with τ_0 as first component, unless τ_0 occurs in the sequence $\tau_1, \ldots, \tau_{5n^2+F(n)}$. This allows to reduce the upper bound to $n \cdot (5n^2 + F(n)) + 1$

If the simulating automaton A is a 1DFA, then first component does not change, while the second component contains the set of states in which the cell $m+1$ is entered for the first time during all possible computations of M. This would give a $2^n \cdot (5n^2 + F(n)) + 1$ state upper bound. However, if the set in the second component is empty then the computation of M is rejecting, regardless what is the remaining part of the input and what has been written on the tape. Hence, in this case, the simulating 1DFA can enter a sink state. This allows to reduce the upper bound to $(2^n - 1) \cdot (5n^2 + F(n)) + 2$. □

Optimality: The Language $\mathscr{L}_{n,\ell}$

We now study the optimality of the state upper bounds presented in Theorem 1. To this aim, we introduce a family of languages $\mathscr{L}_{n,\ell}$, that are defined with respect to integer parameters $n, \ell > 0$.

Each language in this family is composed by all strings of length multiple of ℓ belonging to the language L_{MF_n} which is accepted by the n-state 1NFA $A_{MF_n} = (Q_n, \{a,b\}, \delta_n, q_0, \{q_0\})$ depicted in Figure 1, i.e., $\mathscr{L}_{n,\ell} = L_{MF_n} \cap (\{a,b\}^\ell)^*$.

The automaton A_{MF_n} was proposed longtime ago by Meyer and Fischer as a witness of the exponential state gap from 1NFAs to 1DFAs [12]. Indeed, it can be proved that the smallest 1DFA accepting it has exactly 2^n states. In the following we shall refer to some arguments given in the proof of such result presented in [20, Thm. 3.9.6].

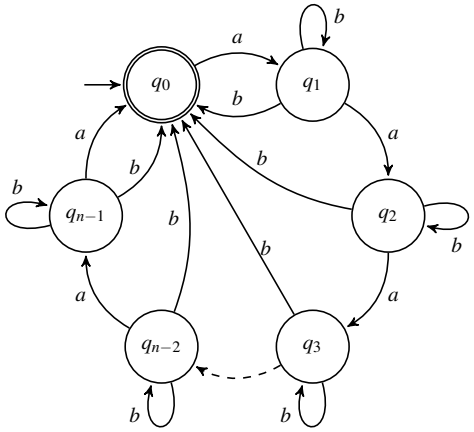


Figure 1: The 1NFA A_{MF_n} accepting the language of Meyer and Fischer.

Let us start by presenting some simple state upper bounds for the recognition of $\mathscr{L}_{n,\ell}$ by one-way finite automata.

Theorem 2 *For every two integers $n, \ell > 0$, there exists a complete* 1DFA *accepting $\mathscr{L}_{n,\ell}$ with $(2^n - 1) \cdot \ell + 1$ states and a* 1NFA *with $n \cdot \ell$ states.*

Proof. We apply the subset construction to convert the 1NFA A_{MF_n} into a 1DFA with 2^n states and then, with the standard product construction, we intersect the resulting automaton with the trivial ℓ-state automaton accepting $(\{a,b\}^\ell)^*$. In this way we obtain a 1DFA with $2^n \cdot \ell$ states for $\mathscr{L}_{n,\ell}$. However, all the states obtained from the sink state, corresponding to the empty set, are equivalent, so they can be replaced by a unique sink state. This allows to reduce the number of states to $(2^n-1)\cdot\ell+1$.

In the case of 1NFAs we apply the product construction to A_{MF_n} and the ℓ-state automaton accepting $(\{a,b\}^\ell)^*$, so obtaining a 1NFA with $n\cdot\ell$ states. □

We now study how to recognize $\mathscr{L}_{n,\ell}$ using two-way automata and F-1-LAs. In both cases we obtain sweeping machines.

Theorem 3 *Let $\ell > 0$ be an integer that factorizes $\ell = p_1^{k_1}\cdots p_r^{k_r}$ as a product of prime powers and $o = r \bmod 2$. Then:*

- *$\mathscr{L}_{n,\ell}$ is accepted by a sweeping 2NFA with $n + p_1^{k_1} + \cdots + p_r^{k_r} + o$ states, that uses nondeterministic transitions only in the first sweep.*
- *$\mathscr{L}_{n,\ell}$ is accepted by a sweeping F-1-LA with $\max(n, p_1^{k_1} + \cdots + p_r^{k_r} + o)$ states that uses nondeterministic transitions only in the first sweep.*
- *$\mathscr{L}_{n,\ell}$ is accepted by a sweeping 2DFA with $2n + p_1^{k_1} + \cdots + p_r^{k_r} + o$ states.*

Proof. In the first sweep, the 2NFA for $\mathscr{L}_{n,\ell}$, using n states, simulates the 1NFA A_{MF_n} to check if the input belongs to L_{MF_n}. Then, it makes one sweep for each $i = 1,\ldots,r$ (alternating a right-to-left sweep with a left-to-right sweep), using $p_i^{k_i}$ states in order to check whether $p_i^{k_i}$ divides the input length. If the outcomes of all these tests are positive, then the automaton accepts. When r is even, the last sweep ends with the head on the right end-marker. Then, moving the head one position to the right, the automaton can reach the accepting configuration. However, when r is odd, the last sweep ends on the left end-marker. Hence, using an extra state, the head can traverse the entire tape to finally reach the accepting configuration.

A F-1-LA can implement the same strategy. However, to check if the tape length is a multiple of ℓ, it can reuse the n states used in the first sweep, plus $p_1^{k_1} + \cdots + p_r^{k_r} + o - n$ extra states when $n < p_1^{k_1} + \cdots + p_r^{k_r} + o$. This is due to the fact that the value of the transition function depends on the state and on the symbol in the tape cell and that, in the first sweep, all the input symbols have been replaced by Z.

Finally, we can implement a 2DFA that recognizes $\mathscr{L}_{n,\ell}$ by firstly making r sweeps to check whether $p_i^{k_i}$ divides the input length, $i = 1,\ldots,r$. If so, then the automaton, after moving the head from the left to the right end-marker in case of r even, makes a further sweep from right to left, to simulate a 1DFA accepting the reversal of L_{MF_n}, which can be accepted using $2n$ states [19]. If the simulated automaton accepts, then the machine can make a further sweep, by using a unique state to move the head from the left end-marker to the right one, and then accept. The total number of states is $2n + p_1^{k_1} + \cdots + p_r^{k_r} + 2 - o$. This number can be slightly reduced as follows: in the first sweep (which is from left to right) the automaton checks the divisibility of the input length by $p_1^{k_1}$; in the second sweep (from right to left) the automaton checks the membership to L_{MF_n}; in the remaining $r-1$ sweeps (alternating left-to-right with right-to-left sweeps), it checks the divisibility for $p_i^{k_i}$, $i = 2,\ldots,r$. So, the total number of sweeps for these checks is $r+1$. This means that, when r is even, the last sweep ends on the right end-marker and the machine can immediately move to the accepting configuration. Otherwise the head needs to cross the input from left to right, using an extra state. □

As a consequence of Theorem 3, in the case of F-1-LAs we immediately obtain:

Corollary 1 *For each $n > 0$ the language $\mathscr{L}_{n,F(n)}$ is accepted by a* F-1-LA *with at most $n+1$ states.*

Proof. If $F(n) = p_1^{k_1} \cdots p_r^{k_r}$ then $p_1^{k_1} + \cdots + p_r^{k_r} \leq n \leq F(n)$. Hence, the statement follows from Theorem 3. □

We are now going to prove lower bounds for the recognition of $\mathscr{L}_{n,\ell}$, in the case n and ℓ are relatively primes.

Let us start by considering the recognition by 1DFAs.

Theorem 4 *Given two integers $n, \ell > 0$ with $\gcd(n,\ell) = 1$, each* 1DFA *accepting $\mathscr{L}_{n,\ell}$ must have at least $(2^n - 1) \cdot \ell + 1$ states.*

Proof. Let $Q_n = \{q_0, q_1, \ldots, q_{n-1}\}$ be the set of states of A_{MF_n} (see Figure 1). First, we briefly recall some arguments from the proof presented in [20, Thm. 3.9.6]. For each subset S of Q_n, we define a string w_S having the property that $\delta_n(q_0, w_S) = S$. Furthermore, it is proved that all the strings so defined are pairwise distinguishable, so obtaining the state lower bound 2^n for each 1DFA equivalent to A_{MF_n}. In particular, the string w_S is defined as follows:

$$w_S = \begin{cases} b & \text{if } S = \emptyset; \\ a^i & \text{if } S = \{q_i\}; \\ a^{e_k - e_{k-1}} b a^{e_{k-1} - e_{k-2}} b \cdots a^{e_2 - e_1} b a^{e_1}, & \text{otherwise;} \end{cases} \tag{1}$$

where in the second case $S = \{q_i\}$, $0 \leq i < n$, while in the third case $S = \{q_{e_1}, q_{e_2}, \ldots, q_{e_k}\}$, $1 < k \leq n$, and $0 \leq e_1 < e_2 < \cdots < e_k < n$.

To obtain the claimed state lower bound in the case of the language $\mathscr{L}_{n,\ell}$, for each nonempty subset S of Q_n and each integer j, with $0 \leq j < \ell$, we define a string $w_{S,j}$ which is obtained by suitably padding the string w_S in such a way that the set of states reachable from the initial state by reading $w_{S,j}$ remains S and the length of $w_{S,j}$, divided by ℓ, gives j as reminder. Then we shall prove that all the so obtained strings are pairwise distinguishable. Unlike (1), when defining $w_{S,j}$ we do not consider the case $S = \emptyset$.

In the following, let us denote by $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ a function satisfying $f(i,j) \bmod n = i$ and $f(i,j) \bmod \ell = j$, for $i, j \in \mathbb{N}$. Since $\gcd(n,\ell) = 1$, by the Chinese Reminder Theorem, such a function always exists.

For each non-empty subset S of Q_n and each integer j, with $0 \leq j < \ell$, the string $w_{S,j}$ is defined as:

$$w_{S,j} = \begin{cases} a^{f(i,j)} & \text{if } S = \{q_i\}; \\ a^{e_k - e_{k-1}} b a^{e_{k-1} - e_{k-2}} b \cdots a^{e_2 - e_1} b^{H\ell - k - e_k + 2 + j} a^{e_1}, & \text{otherwise;} \end{cases} \tag{2}$$

where in the first case $S = \{q_i\}$, $0 \leq i < n$, while in the second case $S = \{q_{e_1}, q_{e_2}, \ldots, q_{e_k}\}$, $1 < k \leq n$, $0 \leq e_1 < e_2 < \cdots < e_k < n$, and $H \geq 1$ is a fixed integer such that $H\ell > 2n$ (this last condition is useful to have $H\ell - k - e_k + 2 + j > 0$, in such a way that the last block of b's is always well defined and not empty).

We claim and prove the following facts:

1. $|w_{S,j}| \bmod \ell = j$.
   If $S = \{q_i\}$, then by definition $|w_{S,j}| \bmod \ell = f(i,j) \bmod \ell = j$. Otherwise, according to the second case in (2), $S = \{q_{e_1}, q_{e_2}, \ldots, q_{e_k}\}$ and $|w_{S,j}| = e_k - e_{k-1} + 1 + e_{k-1} - e_{k-2} + 1 + \cdots + e_2 - e_1 + H\ell - k - e_k + 2 + j + e_1$, which is equal to $H\ell + j$.
2. $\delta_n(q_0, w_{S,j}) = S$.
   In the automaton A_{MF_n}, all the transitions on the letter a are deterministic. Furthermore, by reading

the string a^x, $x > 0$, from the state q_0, the only reachable state is $q_{x \bmod n}$. Hence, for the first case $S = \{q_i\}$ in (2) we have $\delta_n(q_0, w_{S,j}) = \{q_{f(i,j) \bmod n}\} = \{q_i\}$.

For the second case, we already mentioned that $\delta_n(q_0, w_S) = S$. Furthermore $w_{S,j}$ is obtained from w_S by replacing the rightmost b by a block of more than one b. From the transition diagram of A_{MF_n} we observe that from each state q_i, with $i > 0$, reading a b the automaton can either remain in q_i or move to q_0. Furthermore, from q_0 there are no transitions on the letter b. This allows to conclude that the behavior does not change when one replaces an occurrence of b in a string with a sequence of more than one b. Hence, $\delta_n(q_0, w_{S,j}) = \delta_n(q_0, w_S) = S$.

3. *For $i = 0, \ldots, n-1$ and $x \geq 0$, $\delta_n(q_i, a^x) = \{q_{i'}\}$ where $i' = 0$ if and only if $x \bmod n = n - i$. Hence a^x is accepted by some computation path starting from q_i if and only if $x \bmod n = n - i$.*
It is enough to observe that all the transitions on the letter a are deterministic and form a loop visiting all the states. More precisely $i' = (i + x) \bmod n$. Hence, $i' = 0$ if and only if $x \bmod n = n - i$.

We now prove that all the strings $w_{S,j}$ are pairwise distinguishable. To this aim, let us consider two such strings $w_{S,j}$ and $w_{T,h}$, with $(S, j) \neq (T, h)$. We inspect the following two cases:

- $S \neq T$. Without loss of generality, let us consider a state $q_s \in S \setminus T$. We take $z = a^{f(n-s,\ell-j)}$. By the previous claims, we obtain that $w_{S,j} \cdot z \in L_{MF_n}$, while $w_{T,h} \cdot z \notin L_{MF_n}$. Furthermore, $|w_{S,j} \cdot z| \bmod \ell = (j + \ell - j) \bmod \ell = 0$. Hence $w_{S,j} \cdot z \in (\{a,b\}^\ell)^*$. This allows to conclude that $w_{S,j} \cdot z \in \mathscr{L}_{n,\ell}$, while $w_{T,h} \cdot z \notin \mathscr{L}_{n,\ell}$.
- $j \neq h$. We choose a state $q_s \in S$ and, again, the string $z = a^{f(n-s,\ell-j)}$. Exactly as in the previous case we obtain $w_{S,j} \cdot z \in \mathscr{L}_{n,\ell}$. Furthermore, being $j \neq h$ and $0 \leq j, h < \ell$, we get that $|w_{T,h} \cdot z| \bmod \ell = (h + \ell - j) \bmod \ell \neq 0$. Hence $w_{T,h} \cdot z \notin (\{a,b\}^\ell)^*$, thus implying $w_{T,h} \cdot z \notin \mathscr{L}_{n,\ell}$.

By summarizing, we have proved that all the above defined $(2^n - 1) \cdot \ell$ strings $w_{S,j}$ are pairwise distinguishable. We also observe that each string starting with the letter b is not accepted by the automaton A_{MF_n}.[1] This implies that the string b and each string $w_{S,j}$ are distinguishable. Hence, we are able to conclude that each 1DFA accepting $\mathscr{L}_{n,\ell}$ has at least $(2^n - 1) \cdot \ell + 1$ states. □

Concerning 1NFAs, we prove the following:

Theorem 5 *Given two integers $n, \ell > 0$ with $\gcd(n, \ell) = 1$, each 1NFA accepting $\mathscr{L}_{n,\ell}$ must have at least $n \cdot \ell$ states.*

Proof. The proof can be easily given by observing that $X = \{(a^i, a^{n \cdot \ell - i}) \mid i = 0, \ldots, n \cdot \ell - 1\}$ is a *fooling set* for $\mathscr{L}_{n,\ell}$ [1]. Hence, the number of states of each 1NFA for $\mathscr{L}_{n,\ell}$ cannot be lower than the cardinality of X. □

As a consequence of Theorems 4 and 5 we obtain:

Theorem 6 *For each prime $n > 4$, every 1DFA and every 1NFA accepting $\mathscr{L}_{n,F(n)}$ needs $(2^n - 1) \cdot F(n) + 1$ and $n \cdot F(n)$ states, respectively.*

Proof. First, we prove that $\gcd(n, F(n)) = 1$ for each prime $n > 4$. To this aim, we observe that by definition $F(n) \geq 2 \cdot (n-2)$ for each prime n. Furthermore, if $n > 4$ then $2 \cdot (n-2) > n$. Hence $F(n) > n$ for each prime $n > 4$. Suppose that $\gcd(n, F(n)) \neq 1$. Then n, being prime and less than $F(n)$, should

[1] We point out that two strings that in A_{MF_n} lead to the emptyset are not distinguishable. This is the reason why we did not considered strings of the form $w_{\emptyset,j}$ in (2).

divide $F(n)$. By definition of $F(n)$, this would imply $F(n) = n$; a contradiction. This allows us to conclude that $\gcd(n, F(n)) = 1$, for each prime $n > 4$.

Using Theorems 4 and 5, we get that, for all such n's, a 1DFA needs at least $(2^n - 1) \cdot F(n) + 1$ states to accept $\mathscr{L}_{n,F(n)}$, while an equivalent 1NFA needs at least $n \cdot \ell$ states. □

As a consequence of Theorem 6, for infinitely many n, the 1DFA and 1NFA for the language $\mathscr{L}_{n,F(n)}$ described in Theorem 2 are minimal.

By combining the results in Corollary 1 and Theorem 6, we obtain that the costs of the simulations of F-1-LAs by 1NFAs and 1DFAs presented in Theorem 1 are asymptotically optimal:

Corollary 2 *For infinitely many integers n there exists a language which is accepted by a* F-1-LA *with at most* $n+1$ *states and such that all equivalent* 1DFA*s and* 1NFA*s require at least* $(2^n - 1) \cdot F(n) + 1$ *and* $n \cdot F(n)$ *states, respectively.*

4 Deterministic Forgetting 1-Limited Automata vs. One-Way Automata

In Section 3 we studied the size costs of the conversions from F-1-LAs to one-way finite automata. We now restrict our attention to the simulation of deterministic machines. By adapting to this case the arguments used to prove Theorem 1, we obtain a superpolynomial state bound for the conversion into 1DFAs, which is not so far from the bound obtained starting from nondeterminstic machines:

Theorem 7 *Let M be an n-state* D-F-1-LA*. Then M can be simulated by a* 1DFA *with at most* $n \cdot (n + F(n)) + 2$ *states.*

Proof. We can apply the construction given in the proof of Theorem 1 to build, from the given D-F-1-LA M, a one-way finite automaton that, when the head reaches the tape cell $m+1$, has in its finite control the transition table τ_m associated with the tape segment of length m and the state in which the cell is reached for the first time. Since the transitions of M are deterministic, each tape cell is reached for the first time by at most one computation and the resulting automaton is a (possible partial) 1DFA, with no more than $n \cdot (5n^2 + F(n)) + 1$ states. However, in this case the number of transition tables can be reduced, so decreasing the upper bound. In particular, due to determinism and the unary content in the frozen part, we can observe that left and right U-turns cannot visit more than n tape cells. Furthermore, after visiting more than n tape cells, a traversal is repeating a loop. This allows to show that the sequence of transition matrices starts to be periodic after the matrix τ_n, i.e, for $m, m' > n$, if $m \equiv m' \pmod{F(n)}$ then $\tau_m = \tau_{m'}$. Hence, the number of different transition tables used during the simulation is at most $n + 1 + F(n)$, and the number of states of the simulating (possibly partial) 1DFA is bounded by $n \cdot (n + F(n)) + 1$. By adding one more state we finally obtain a complete 1DFA. □

Optimality: The Language $\mathscr{J}_{n,\ell}$

We now present a family of languages for which we prove a size gap very close to the upper bound in Theorem 7. Given two integers $n, \ell > 0$, let us consider:

$$\mathscr{J}_{n,\ell} = \{w \in \{a,b\}^* \mid |w|_a \bmod n = 0 \text{ and } |w| \bmod \ell = 0\}.$$

First of all, we observe that it is not difficult to recognize $\mathscr{J}_{n,\ell}$ using a 1DFA with $n \cdot \ell$ states that counts the number of a's using one counter modulo n, and the input length using one counter modulo ℓ. This number of states cannot be reduced, even allowing nondeterministic transitions:

Theorem 8 *Each* 1NFA *accepting* $\mathscr{J}_{n,\ell}$ *has at least* $n \cdot \ell$ *states.*

Proof. Let $H > \ell + n$ be a multiple of ℓ. For $i = 1, \ldots, \ell$, $j = 0, \ldots, n-1$, consider $x_{ij} = a^j b^{H+i-j}$ and $y_{ij} = b^{H-i-n+j} a^{n-j}$. We are going to prove that the set

$$X = \{(x_{ij}, y_{ij}) \mid 1 \leq i \leq \ell, 0 \leq j < n\}$$

is an *extended fooling set* for $\mathscr{J}_{n,\ell}$. To this aim, let us consider $i, i' = 1, \ldots, \ell$, $j, j' = 0, \ldots, n-1$. We observe that the string $x_{ij} y_{ij}$ contains n a's and has length $j + H + i - j + H - i - n + j + n - j = 2H$ and hence it belongs to $\mathscr{J}_{n,\ell}$. For $i, i' = 1, \ldots, \ell$, if $i \neq i'$ then the string $x_{ij} y_{i'j} \notin \mathscr{J}_{n,\ell}$ because it has length $2H + i - i'$ which cannot be a multiple of ℓ. On the other hand, if $j < j'$, the string $x_{ij} y_{i'j'}$ contains $j + n - j' < n$ many a's, so it cannot belong to $\mathscr{J}_{n,\ell}$, □

Concerning the recognition of $\mathscr{J}_{n,\ell}$ by F-1-LAs we prove the following:

Theorem 9 *Let* $\ell > 0$ *be an integer that factorizes* $\ell = p_1^{k_1} \cdots p_s^{k_r}$ *as product of prime powers,* $o = r \bmod 2$*, and* $n > 0$*. Then* $\mathscr{J}_{n,\ell}$ *is accepted by a sweeping* 2DFA *with* $n + p_1^{k_1} + \cdots + p_r^{k_r} + o$ *states and by a sweeping* D-F-1-LA *with* $\max(n, p_1^{k_1} + \cdots + p_r^{k_r} + o)$ *states.*

Proof. A 2DFA can make a first sweep of the input, using n states, to check if the number of a's in the input is a multiple of n. Then, in further r sweeps, alternating right-to-left with left-to-right sweeps, it can check the divisibility of the input length by $p_i^{k_i}$, $i = 1, \ldots, r$. If r is odd this process ends with the head on the left end-marker. Hence, in this case, when all tests are positive, a further sweep (made by using a unique state) is used to move the head from the left to the right end-marker and then reach the accepting configuration.

We can implement a D-F-1-LA that uses the same strategy. However, after the first sweep, all input symbols are replaced by Z. Hence, as in the proof of Theorem 3, the machine can reuse the n states of the first sweep. So, the total number of states reduces to $\max(n, p_1^{k_1} + \cdots + p_r^{k_r} + o)$. □

As a consequence of Theorem 9, we obtain:

Corollary 3 *For each integer* $n > 0$ *the language* $\mathscr{J}_{n,F(n)}$ *is accepted by a* D-F-1-LA *with at most* $n + 1$ *states.*

By combining the upper bound in Corollary 3 with the lower bound in Theorem 8, we obtain that the superpolynomial cost of the simulation of D-F-1-LAs by 1DFAs given in Theorem 7 is asymptotically optimal and it cannot be reduced even if the resulting automaton is nondeterministic:

Corollary 4 *For each integer* $n > 0$ *there exists a language accepted by a* D-F-1-LA *with at most* $n + 1$ *states and such that all equivalent* 1DFA*s and* 1NFA*s require at least* $n \cdot F(n)$ *states.*

5 Forgetting 1-Limited vs. Two-Way Automata

Up to now, we have studied the size costs of the transformations of F-1-LAs and D-F-1-LAs into one-way automata. We proved that they cannot be significantly reduced, by providing suitable witness languages. However, we can notice that such languages are accepted by two-way automata whose sizes are not so far from the sizes of F-1-LAs and D-F-1-LAs we gave. So we now analyze the size relationships between forgetting and two-way automata. On the one hand, we show that forgetting input symbols can dramatically reduce the descriptional power. Indeed, we provide a family of languages for which F-1-LAs are exponentially larger than 2DFAs. On the other hand, we guess that also in the opposite direction at least a superpolynomial gap can be possible. To this aim we present a language accepted by a D-F-1-LA of size $O(n)$ and we conjuncture that each 2NFA accepting it requires more than $F(n)$ states.

From Two-way to Forgetting 1-Limited Automata

For each integer $n > 0$, let us consider the following language

$$\mathscr{E}_n = \{w \in \{a,b\}^* \mid \exists x \in \{a,b\}^n, \exists y,z \in \{a,b\}^* : w = x \cdot y = z \cdot x^R\},$$

i.e., the set of strings in which the prefix of length n is equal to the reversal of the suffix. As we shall see, it is possible to obtain a 2DFA with $O(n)$ states accepting it. Furthermore, each equivalent F-1-LA requires 2^n states.

To achieve this result, first we give a lower bound technique for the number of states of F-1-LAs, which is inspired by the *fooling set technique* for 1NFAs [1].

Lemma 1 *Let $L \subseteq \Sigma^*$ be a language and $X = \{(x_i, y_i) \mid i = 1, \ldots, n\}$ be a set of words such that the following hold:*

- $|x_1| = |x_2| = \cdots = |x_n|$,
- $x_i y_i \in L$, *for* $i = 1, \ldots, n$,
- $x_i y_j \notin L$ *or* $x_j y_i \notin L$, *for* $i, j = 1, \ldots, n$ *with* $i \neq j$.

Then each F-1-LA*s accepting L has at least n states.*

Proof. Let M be a F-1-LAs accepting L. Let C_i be an accepting computation of M on input $x_i y_i$, $i = 1, \ldots, n$. We divide C_i into two parts C_i' and C_i'', where C_i' is the part of C_i that starts from the initial configuration and ends when the head reaches for the first time the first cell to the right of x_i, namely the cell containing the first symbol of y_i, while C_i'' is the remaining part of C_i. Let q_i be the state reached at the end of C_i', namely the state from which C_i'' starts.

If $q_i = q_j$, for some $1 \leq i, j \leq n$, then the computation obtained concatenating C_i' and C_j'' accepts the input $x_i y_j$. Indeed, at the end of C_i' and of C_j', the content of the tape to the left of the head is replaced by the same string $Z^{|x_i|} = Z^{|x_j|}$. So M, after inspecting x_i, can perform exactly the same moves as on input $x_j y_j$ after inspecting x_j and hence it can accept $x_i y_j$. In a similar way, concatenating C_j' and C_i'' we obtain an accepting computation on $x_j y_i$. If $i \neq j$, then this is a contradiction.

This allows to conclude that n different states are necessary for M. □

We are now able to prove the claimed separation.

Theorem 10 *The language $\mathscr{E}_n$ is accepted by a* 2DFA *with $O(n)$ states, while each* F-1-LA *accepting it has at least 2^n states.*

Proof. We can build a 2DFA that on input $w \in \Sigma^*$ tests the equality between the symbols in positions i and $|w| - i$ of w, for $i = 1, \ldots, n$. If one of the tests fails, then the automaton stops and rejects, otherwise it finally accepts. For each i, the test starts with the head on the left end-marker and the value of i in the finite control. Then, the head is moved to the right, while decrementing i, to locate the ith input cell and remember its content in the finite control. At this point, the head is moved back to the left end-marked, while counting input cells to restore the value of i. The input is completely crossed from left to right, by keeping this value in the control. When the right end-marker is reached, a similar procedure is applied to locate the symbol in position $|w| - i$, which is then compared with that in position i, previously stored in the control. If the two symbols are equal, then the head is moved again to the right end-marker, while restoring i. If $i = n$, then the machine moves in the accepting configuration, otherwise the value of i is incremented and the head is moved to the left end-marker to prepare the next test. From the above description we can conclude that $O(n)$ states are enough for a 2DFA to accept $\mathscr{E}_n$.

For the lower bound, we observe that the set $X = \{(x, x^R) \mid x \in \{a,b\}^n\}$, whose cardinality is 2^n, satisfies the requirements of Lemma 1. □

From Forgetting 1-limited to Two-way Automata

We wonder if there is some language showing an exponential, or at least superpolynomial, size gap from F-1-LAs to two-way automata. Here we propose, as a possible candidate, the following language, where $n, \ell > 0$ are integers:

$$\mathscr{H}_{n,\ell} = \{ ub^n v \mid u \in (a+b)^* a, v \in (a+b)^*, |u|_a \bmod n = 0, \text{ and } |u| \bmod \ell = 0 \}.$$

We prove that $\mathscr{H}_{n,F(n)}$ can be recognized by a D-F-1-LA with a number of states linear in n.

Theorem 11 *For each integer $n > 1$ the language $\mathscr{H}_{n,F(n)}$ is accepted by a* D-F-1-LA *with $O(n)$ states.*

Proof. A D-F-1-LA M can start to inspect the input from left to right, while counting modulo n the a's. In this way it can discover each prefix u that ends with an a and such that $|u|_a \bmod n = 0$. When such a prefix is located, M verifies whether $|u|$ is a multiple of $F(n)$ and it is followed by b^n. We will discuss how to do that below. If the result of the verification is positive, then M moves to the accepting configuration, otherwise it continues the same process.

Now we explain how the verification can be performed. Suppose $F(n) = p_1^{k_1} \cdots p_r^{k_r}$, where $p_1^{k_1}, \ldots, p_r^{k_r}$ are prime powers. First, we point out that when the verification starts, exactly the first $|u|$ tape cells have been rewritten. Hence, the rough idea is to alternate right-to-left and left-to-right sweeps on such a portion of the tape, to check the divisibility of $|u|$ by each $p_i^{k_i}$, $i = 1, \ldots, r$. A right-to-left sweep stops when the head reaches the left end-marker. On the other hand, a left-to-right sweep can end only when the head reaches the first cell to the right of the frozen segment. This forces the replacement of the symbol in it with the symbol Z, so increasing the length of the frozen segment by 1. In the next sweeps, the machine has to take into account how much the frozen segment increased. For instance, after checking divisibility by $p_1^{k_1}$ and by $p_2^{k_2}$, in the next sweep the machine should verify that the length of the frozen segment, modulo $p_3^{k_3}$, is 1. Because the machine has to check r divisors and right-to-left sweeps alternate with left-to-right sweeps, when all r sweeps are done, exactly $\lfloor r/2 \rfloor$ extra cells to the right of the original input prefix u are frozen. Since $n > r/2$, if the original symbol in all those cells was b, to complete the verification phase the machine has to check whether the next $n - \lfloor r/2 \rfloor$ not yet visited cells contain b. However, the verification fails if a cell containing an a or the right end-marker is reached during some point of the verification phase. This can happen either while checking the length of the frozen segment or while checking the last $n - \lfloor r/2 \rfloor$ cells. If the right end-marker is reached, then the machine rejects. Otherwise it returns to the main procedure, i.e., resumes the counting of the a's.

The machine uses a counter modulo n for the a's. In the verification phase this counter keeps the value 0. The device first has to count the length of the frozen part modulo $p_i^{k_i}$, iteratively for $i = 1, \ldots, r$, and to verify that the inspected prefix is followed by b^n, using again a counter. Since $p_1^{k_1} + \cdots + p_r^{k_r} \leq n$, by summing up we conclude that the total number of states is $O(n)$. □

By using a modification of the argument in the proof of Theorem 8, we can show that each 1NFA accepting $\mathscr{H}_{n,F(n)}$ cannot have less than $n \cdot F(n)$ states.[2] We guess that such a number cannot be substantially reduced even having the possibility of moving the head in both directions. In fact, a two-way automaton using $O(n)$ states can easily locate on the input tape a "candidate" prefix u. However, it cannot remember in which position of the tape u ends, in order to check $|u|$ in several sweeps of u. So we do not see how the machine could verify whether $|u|$ is a multiple of $F(n)$ using less than $F(n)$ states.

[2] It is enough to consider the set $X' = \{(x_{ij}, y_{ij} b^n) \mid 1 \leq i \leq \ell, 0 \leq j < n\}$, instead of X.

6 Conclusion

We compared the size of forgetting 1-limited automata with that of finite automata, proving exponential and superpolynomial gaps. We did not discuss the size relationships with 1-LAs. However, since 2DFAs are D-1-LAs that never write, as a corollary of Theorem 10 we get an exponential size gap from D-1-LAs to F-1-LAs. Indeed, the fact of having a unique symbol to rewrite the tape content dramatically reduces the descriptional power.

We point out that this reduction happens also in the case of F-1-LAs accepting languages defined over a one-letter alphabet, namely unary languages. To this aim, for each integer $n > 0$, let us consider the language $(a^{2^n})^*$. This language can be accepted with a D-1-LA having $O(n)$ states and a work alphabet of cardinality $O(n)$, and with a D-1-LA having $O(n^3)$ states and a work alphabet of size not dependent on n [16, 18]. However, each 2NFA accepting it requires at least 2^n states [16]. Considering the cost of the conversion of F-1-LAs into 1NFAs (Theorem 1), we can conclude that such a language cannot be accepted by any F-1-LA having a number of states polynomial in n.

References

[1] Jean-Camille Birget (1992): *Intersection and Union of Regular Languages and State Complexity*. *Inf. Process. Lett.* 43(4), pp. 185–190, doi:10.1016/0020-0190(92)90198-5.

[2] Marek Chrobak (1986): *Finite Automata and Unary Languages*. *Theor. Comput. Sci.* 47(3), pp. 149–158. Available at http://dx.doi.org/10.1016/0304-3975(86)90142-8. Errata: [3].

[3] Marek Chrobak (2003): *Errata to: Finite automata and unary languages: [Theoret. Comput. Sci. 47 (1986) 149-158]*. *Theor. Comput. Sci.* 302(1-3), pp. 497 – 498, doi:10.1016/S0304-3975(03)00136-1.

[4] Viliam Geffert (2007): *Magic numbers in the state hierarchy of finite automata*. *Inf. Comput.* 205(11), pp. 1652–1670. Available at http://dx.doi.org/10.1016/j.ic.2007.07.001.

[5] Viliam Geffert & Giovanni Pighizzini (2012): *Pairs of Complementary Unary Languages with "Balanced" Nondeterministic Automata*. *Algorithmica* 63(3), pp. 571–587, doi:10.1007/s00453-010-9479-9.

[6] Thomas N. Hibbard (1967): *A Generalization of Context-Free Determinism*. *Inf. Control.* 11(1/2), pp. 196–238, doi:10.1016/S0019-9958(67)90513-X.

[7] John E. Hopcroft & Jeffrey D. Ullman (1979): *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[8] Petr Jancar, Frantisek Mráz & Martin Plátek (1993): *A Taxonomy of Forgetting Automata*. In: *MFCS'93*, *Lecture Notes in Computer Science* 711, Springer, pp. 527–536, doi:10.1007/3-540-57182-5_44.

[9] Edmund Landau (1903): *Über die Maximalordnung der Permutation gegebenen Grades*. *Archiv der Mathematik und Physik* 3, pp. 92–103.

[10] Edmund Landau (1909): *Handbuch der Lehre von der Verteilung der Primzahlen I*. Teubner, Leipzig/Berlin.

[11] Carlo Mereghetti & Giovanni Pighizzini (2001): *Optimal Simulations between Unary Automata*. *SIAM J. Comput.* 30(6), pp. 1976–1992, doi:10.1137/S009753979935431X.

[12] Albert R. Meyer & Michael J. Fischer (1971): *SWAT 1971*. IEEE Computer Society, pp. 188–191, doi:10.1109/SWAT.1971.11.

[13] Giovanni Pighizzini (2019): *Limited Automata: Properties, Complexity and Variants*. In: *DCFS 2019*, *Lecture Notes in Computer Science* 11612, Springer, pp. 57–73, doi:10.1007/978-3-030-23247-4_4.

[14] Giovanni Pighizzini & Andrea Pisoni (2014): *Limited Automata and Regular Languages*. *Int. J. Found. Comput. Sci.* 25(7), pp. 897–916, doi:10.1142/S0129054114400140.

[15] Giovanni Pighizzini & Andrea Pisoni (2015): *Limited Automata and Context-Free Languages*. *Fundam. Inform.* 136(1-2), pp. 157–176, doi:10.3233/FI-2015-1148.

[16] Giovanni Pighizzini & Luca Prigioniero (2019): *Limited automata and unary languages*. Inf. Comput. 266, pp. 60–74, doi:10.1016/j.ic.2019.01.002.

[17] Giovanni Pighizzini & Luca Prigioniero (2023): *Once-Marking and Always-Marking* 1*-Limited Automata*. In: *AFL 2023*, *Electronic Proceedings in Theoretical Computer Science* 386, pp. 215–227, doi:10.4204/EPTCS.386.17.

[18] Giovanni Pighizzini & Luca Prigioniero (2023): *Two-way Machines and de Bruijn Words*. In: *CIAA 2023*, *Lecture Notes in Computer Science* 14151, pp. 254–65, doi:10.1007/978-3-031-40247-0_19.

[19] Giovanni Pighizzini, Luca Prigioniero & Simon Šádovský (2022): *1-Limited Automata: Witness Languages and Techniques*. J. Autom. Lang. Comb. 27(1-3), pp. 229–244, doi:10.25596/jalc-2022-229.

[20] Jeffrey O. Shallit (2008): *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, doi:10.1017/CBO9780511808876.

[21] John C. Shepherdson (1959): *The Reduction of Two-Way Automata to One-Way Automata*. IBM J. Res. Dev. 3(2), pp. 198–200, doi:10.1147/rd.32.0198.

[22] Michael Sipser (1980): *Lower Bounds on the Size of Sweeping Automata*. J. Comput. Syst. Sci. 21(2), pp. 195–202, doi:10.1016/0022-0000(80)90034-3.

[23] Mihály Szalay (1980): *On the maximal order in S_n and S_n^**. Acta Arithmetica 37, pp. 321–331, doi:10.4064/aa-37-1-321-331.

[24] Klaus W. Wagner & Gerd Wechsung (1986): *Computational complexity*. D. Reidel Publishing Company, Dordrecht.

Sweeping Permutation Automata

Maria Radionova
Department of Mathematics and Computer Science
St. Petersburg State University
Saint Petersburg, Russia
radmarale@gmail.com

Alexander Okhotin
Department of Mathematics and Computer Science
St. Petersburg State University
Saint Petersburg, Russia
alexander.okhotin@spbu.ru

This paper introduces sweeping permutation automata, which move over an input string in alternating left-to-right and right-to-left sweeps and have a bijective transition function. It is proved that these automata recognize the same family of languages as the classical one-way permutation automata (Thierrin, "Permutation automata", *Mathematical Systems Theory*, 1968). An n-state two-way permutation automaton is transformed to a one-way permutation automaton with $F(n) = \max_{k+\ell=n, m\leqslant\ell} k \cdot \binom{\ell}{m} \cdot \binom{k-1}{\ell-m} \cdot (\ell-m)!$ states. This number of states is proved to be necessary in the worst case, and its growth rate is estimated as $F(n) = n^{\frac{n}{2} - \frac{1+\ln 2}{2} \frac{n}{\ln n}(1+o(1))}$.

1 Introduction

Permutation automata, introduced by Thierrin [15], are one-way deterministic finite automata, in which the transition function by each symbol forms a permutation of the set of states. They recognize a proper subfamily of regular languages: for instance, no finite language is recognized by any permutation automaton. The language family recognized by permutation automata is known as the *group languages*, because their syntactic monoid is a group, and it has received some attention in the literature on algebraic automata theory [9]. Recently, Hospodár and Mlynárčik [3] determined the state complexity of operations on these automata, while Rauch and Holzer [12] investigated the effect of operations on permutation automata on the number of accepting states.

Permutation automata are reversible, in the sense that, indeed, knowing the current state and the last read symbol one can always reconstruct the state at the previous step. The more general *reversible automata*, studied by Angluin [1] and by Pin [10], additionally allow undefined transitions, so that the transition function by each symbol is injective. Reversible automata still cannot recognize all regular languages [11], but since they can recognize all finite languages, they are a more powerful model than permutation automata.

The notion of reversible computation has also been studied for two-way finite automata. In general, a two-way automaton (2DFA) operates on a string delimited by a left end-marker ($\vdash$) and a right end-marker ($\dashv$), and may move its head to the left or to the right in any transition. For the reversible subclass of two-way finite automata (2RFA), Kondacs and Watrous [7] proved that 2RFA can recognize every regular language. Later, Kunc and Okhotin [8, Sect. 8.1] showed that every regular language can still be recognized by 2RFA with no undefined transitions on symbols of the alphabet, and with injective functions on the end-markers. But since the latter automata, in spite of having some kind of bijections in their transition functions, recognize all regular languages, they are no longer a model for the group languages. And there seems to be no reasonable way to have 2RFA act bijectively on both end-markers, because in this case it would be impossible to define both an accepting and a rejecting state.

Can permutation automata have any two-way generalization at all? This paper gives a positive answer by investigating *sweeping permutation automata*. This new model is a subclass of sweeping automata,

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 110–124, doi:10.4204/EPTCS.388.11

that is, two-way automata that may turn only at the end-markers. In a sweeping automaton, there are left-moving and right-moving states, and any transition in a right-moving state by any symbol other than an end-marker must move the head to the right and lead to another right-moving state (same for left-moving states). The transitions by each symbol other than an end-marker form two functions, one acting on the right-moving states, and the other on the left-moving states. In the proposed sweeping permutation automata, both functions must be bijections, whereas the transition functions at the end-markers must be injective. A formal definition of the new model is given in Section 2.

The main motivation for the study of sweeping permutation automata (2PerFA) is that these automata recognize the same family of languages as the classical one-way permutation automata (1PerFA). This result is established in Section 3 by showing that if the optimal transformation of two-way automata to one-way, as defined by Kapoutsis [5, 6], is carefully applied to a 2PerFA, then it always produces a 1PerFA.

The next question studied in this paper is the number of states in a 1PerFA needed to simulate an n-state 2PerFA. The number of states used in the transformation in Section 3 depends on the partition of n states of the 2PerFA into k right-moving and ℓ left-moving states, and also on the number m of left-moving states in which there are no transitions by the left end-marker ($\vdash$). The resulting 1PerFA has $k \cdot \binom{\ell}{m} \cdot \binom{k-1}{\ell-m} \cdot (\ell-m)!$ states. A matching lower bound for each triple (k,ℓ,m), with $k > \ell$ and $m > 0$, is established in Section 4, where it is proved that there exists a 2PerFA with k right-moving states and ℓ left-moving states, and with the given value of m, such that every one-way deterministic automaton (1DFA) recognizing the same language must have at least $k \cdot \binom{\ell}{m} \cdot \binom{k-1}{\ell-m} \cdot (\ell-m)!$ states.

The desired state complexity of transforming two-way permutation automata to one-way should give the number of states in a 1PerFA that is sufficient and in the worst case necessary to simulate every 2PerFA with n states. Note that the minimal 1DFA for a group language is always a 1PerFA [3], and hence this is the same state complexity tradeoff as from 2PerFA to 1DFA. The following function gives an upper bound on this state complexity.

$$F(n) = \max_{\substack{k+\ell=n \\ m \leqslant \ell}} k \cdot \binom{\ell}{m} \cdot \binom{k-1}{\ell-m} \cdot (\ell-m)!$$

This bound is proved to be precise in Section 5, where it is shown that the maximum in the formula is reached for $k = \lfloor \frac{n+2}{2} \rfloor$ and $\ell = \lceil \frac{n-2}{2} \rceil$, that is, for $k > \ell$. Since Section 4 provides witness languages for these values of k and ℓ that require $k \cdot \binom{\ell}{m} \cdot \binom{k-1}{\ell-m} \cdot (\ell-m)!$ states in every 1PerFA, this gives a lower bound $F(n)$. Finally, the growth rate of the function $F(n)$ is estimated as $F(n) = n^{\frac{n}{2} - \frac{1+\ln 2}{2} \frac{n}{\ln n}(1+o(1))}$ using Stirling's approximation.

An alternative, more general definition of sweeping permutation automata, which allows acceptance both at the left end-marker and at the right end-marker, is presented in Section 6. A proof that they still can be transformed to 1PerFA is sketched, but the generalized transformation uses more states.

2 Definition

A one-way permutation automaton (1PerFA) is a one-way deterministic finite automaton (1DFA) in which the transition function by every symbol is a bijection [15].

This restriction is adapted to the more general *sweeping automata* [14], in which the set of states is divided into disjoint classes of right-moving (Q_+) and left-moving (Q_-) states, so that the automaton may turn only at the end-markers. In the proposed *sweeping permutation automata*, the transition function

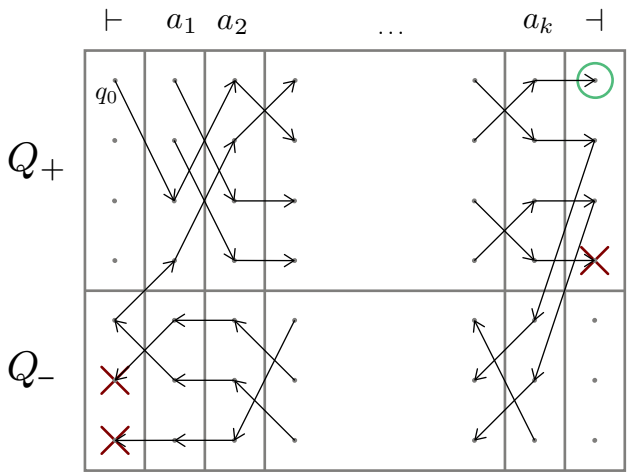


Figure 1: Transitions of a 2PerFA on an input string.

by each symbol forms one left-to-right bijection and another right-to-left bijection. Transitions at the end-markers are injective partial functions.

Definition 1. A sweeping permutation automaton (2PerFA) is a 9-tuple $\mathscr{A} = (\Sigma, Q_+, Q_-, q_0, \langle \delta_a^+ \rangle_{a\in\Sigma}, \langle \delta_a^- \rangle_{a\in\Sigma}, \delta_\vdash, \delta_\dashv, F)$, where

- Σ is the alphabet;
- $Q_+ \cup Q_-$ is the set of states, where $Q_+ \cap Q_- = \varnothing$;
- $q_0 \in Q_+$ is the initial state;
- for each symbol $a \in \Sigma$, $\delta_a^+ \colon Q_+ \to Q_+$ and $\delta_a^- \colon Q_- \to Q_-$ are bijective transition functions;
- the transition functions at the end-markers $\delta_\vdash \colon (Q_- \cup \{q_0\}) \to Q_+$, $\delta_\dashv \colon Q_+ \to Q_-$ are partially defined and injective on their respective domains;
- $F \subseteq Q_+$ is the set of accepting states, with $\delta_\dashv(q)$ undefined for all $q \in F$.

The computation of the automaton is defined in the same way as for sweeping automata of the general form. Given an input string $w = a_1 \dots a_m \in \Sigma^*$, the automaton operates on a tape $\vdash a_1 \dots a_m \dashv$. Its computation is a uniquely defined sequence of configurations, which are pairs (q,i) of a current state $q \in Q_+ \cup Q_-$ and a position $i \in \{0,1,\dots,m+1\}$ on the tape. It starts in the configuration $(q_0,0)$ and makes a transition to $(\delta_\vdash(q_0),1)$. If the automaton is in a configuration (q,i) with $q \in Q_+$ and $i \in \{1,\dots,m\}$, it moves to the next configuration $(\delta_{a_i}^+(q), i+1)$. Once the automaton is in a configuration $(q,m+1)$, it accepts if $q \in F$, or moves to $(\delta_\dashv(q),m)$ if $\delta_\dashv(q)$ is defined, and rejects otherwise. In a configuration (q,i) with $q \in Q_-$ and $i \in \{1,\dots,m\}$, the automaton moves to $(\delta_{a_i}^-(q), i-1)$. Finally, in a configuration $(q,0)$ with $q \in Q_-$, the automaton turns back to $(\delta_\vdash(q),1)$ or rejects if this transition is undefined.

The language recognized by an automaton $\mathscr{A}$, denoted by $L(\mathscr{A})$, is the set of all strings it accepts.

A one-way permutation automaton (1PerFA) is a 2PerFA in which $Q_- = \varnothing$ and $\delta_\dashv$ is undefined on every state. The left end-marker can be removed, making $\delta_\vdash(q_0)$ the new initial state.

Note that a 2PerFA never loops. If it did, then some configuration would appear twice in some computation. Consider the earliest such configuration. If it is not the initial configuration, then there exists only one possible previous configuration. It appears at least twice in the computation, and it

precedes the configuration considered before, a contradiction. The repeated configuration cannot be the initial configuration, in which the 2PerFA is at the left end-marker in the state $q_0 \in Q_+$, because the automaton may return to $\vdash$ only in the states from Q_-.

3 Transformation to one-way

Since a 2PerFA is a 2DFA, the well-known transformation to a one-way automaton can be applied to it [5, 13]: after reading a prefix of a string u, the 1DFA stores the first state in which the 2PerFA eventually goes right from the last symbol of the prefix, and the function which encodes the outcomes of all computations starting at the last symbol of the prefix and ending with the transition from that symbol to the right. For a sweeping automaton, all computations encoded by the functions start in Q_- and end in Q_+. Moreover, computations starting in different states should end in different states. Therefore, a one-way automaton has to remember fewer different functions of a simpler form, and eventually turns out to be a permutation automaton.

Lemma 1. *For every 2PerFA* $\mathscr{A} = (\Sigma, Q_+, Q_-, q_0, \langle \delta_a^+ \rangle_{a\in\Sigma}, \langle \delta_a^- \rangle_{a\in\Sigma}, \delta_\vdash, \delta_\dashv, F)$ *with*

$$|Q_+| = k, \quad |Q_-| = \ell, \quad |Q_-^\times| = m,$$

where $Q_-^\times \subseteq Q_-$ *is the set of states from which there is no transition by* $\vdash$*, there exists a 1PerFA recognizing the same language which uses states of the form* (q, f) *satisfying the following restrictions:*

- $q \in Q_+$,
- $f\colon Q_- \to Q_+$ *is a partially defined function,*
- $q \notin Im f$,
- f *is injective,*
- f *is undefined on exactly* m *states.*

Proof. We will construct a 1PerFA $\mathscr{B} = (\Sigma, Q, \widetilde{q}_0, \widetilde{\delta}, \widetilde{F})$; states of $\mathscr{B}$ shall be pairs (q, f), where $q \in Q_+$ and $f\colon Q_- \to Q_+$ is a partial function.

After reading a prefix $s \in \Sigma^*$, the automaton $\mathscr{B}$ should come to a state (q, f), where q and f describe the outcomes of the following computations of $\mathscr{A}$ on s:

- if the 2PerFA starts on $\vdash s$ in its initial configuration, then it eventually moves from the last symbol of $\vdash s$ to the right in the state q,
- if the 2PerFA starts at the last symbol of $\vdash s$ in a state p from Q_-, then it eventually leaves s in the state $f(p) \in Q_+$. If the computation reaches an undefined transition at $\vdash$, then the value $f(p)$ is undefined.

The initial state is defined as

$$\widetilde{q}_0 = (\delta_\vdash(q_0), \delta_\vdash\big|_{Q_-})$$

where $\delta_\vdash\big|_{Q_-}$ is $\delta_\vdash$ restricted to the domain Q_-.

The definition of the transition function is as follows:

$$\widetilde{\delta}_a((q, f)) = (\delta_a^+(q), \delta_a^+ \circ f \circ \delta_a^-), \qquad \text{for all } a \in \Sigma.$$

Claim 1. *Every pair* (q, f) *reachable from* $\widetilde{q}_0$ *by transitions in* $\widetilde{\delta}$ *satisfies the following conditions:*

- $q \in Q_+$,
- $f\colon Q_- \to Q_+$ *is a partially defined function,*
- $q \notin \mathit{Im} f$,
- f *is injective,*
- f *is undefined on exactly* m *states.*

Proof. Induction on the length of the string. Let (q,f) be reachable in $\mathscr{B}$ by a string u. If $u = \varepsilon$, then (q,f) is the initial state

$$(q,f) = (\delta_\vdash(q_0), \delta_\vdash\big|_{Q_-})$$

The first two conditions are satisfied because the domain of $\delta_\vdash$ is split into q_0 and Q_-. The third and the fourth conditions follow from the injectivity of $\delta_\vdash$ and the disjointness of $\{q_0\}$ and Q_-. The states on which $\delta_\vdash\big|_{Q_-}$ is not defined are the states in $Q_-^\times$ by definition, and there are m of them.

Let (q,f) be reachable in $\mathscr{B}$ by a string u and let (q',f') be reachable from it by a transition by a. The induction assumption is true for the state (q,f), and (q',f') is defined as

$$(q',f') = (\delta_a^+(q), \delta_a^+ \circ f \circ \delta_a^-)$$

The state $q' \in Q_+$ because δ_a^+ is a total function which acts from Q_+ to Q_+. The function f' acts from Q_- to Q_+ because δ_a^- acts from Q_- and δ_a^+ acts to Q_+. To see that $\delta_a^+(q) \notin \mathrm{Im}\ \delta_a^+ \circ f \circ \delta_a^-$, consider that $\{q\}$ and $\mathrm{Im}\ f$ are disjoint by the induction assumption, and therefore their images under a bijection δ_a^+, that are, $\{\delta_a^+(q)\}$ and $\mathrm{Im}\ \delta_a^+ \circ f$, are disjoint as well.

The function $\delta_a^+ \circ f \circ \delta_a^-$ is injective as a composition of injective functions. The function $\delta_a^+ \circ f \circ \delta_a^-$ is undefined on exactly m states because f is, and functions δ_a^+ and δ_a^- are total bijections. □

Let Q be the set of all pairs (q,f) satisfying Claim 1.

Claim 2. *After reading a prefix* $s \in \Sigma^*$ *the automaton* $\mathscr{B}$ *comes to a state* (q,f)*, where*

- *if the 2PerFA starts on* $\vdash s$ *in its initial configuration, then it eventually moves from the last symbol of* $\vdash s$ *to the right in the state* q*,*
- *if the 2PerFA starts at the last symbol of* $\vdash s$ *in a state* p *from* Q_-*, then it eventually leaves* s *in the state* $f(p) \in Q_+$*. If the computation reaches an undefined transition at* $\vdash$*, then the value* $f(p)$ *is undefined.*

Proof. Induction on the length of the string. It is clear for the empty string and the initial state. Let (q,f) be the state of $\mathscr{B}$ after reading s, then $(q',f') = \widetilde{\delta}_a((q,f))$ is the state after reading sa. By the induction assumption, the state (q,f) and the string s satisfy the property. Then $\mathscr{B}$ reads the symbol a and comes to the state $(\delta_a^+(q), \delta_a^+ \circ f \circ \delta_a^-)$. The automaton eventually leaves $\vdash s$ to the right in the state q; then it comes to a in this state and makes a transition to $\delta_a^+(q)$, thus leaving $\vdash sa$ to the right. To prove the second condition, let the 2PerFA start on $\vdash sa$ at the symbol a in a state $p \in Q_-$. Then it moves left to the last symbol of $\vdash s$ in the state $\delta_a^-(p)$. Then the computation continues on the string $\vdash s$ and its outcome is given by the function f. Eventually the 2PerFA leaves $\vdash s$ to the right and comes to a in the state $f(\delta_a^-(p))$. Then the 2PerFA looks at the symbol and goes to $\delta_a^+(f(\delta_a^-(p)))$ moving to the right. If $f(\delta_a^-(p))$ is undefined, then so is $f'(p)$. So, the function $\delta_a^+ \circ f \circ \delta_a^-$ indeed satisfies the second claim. □

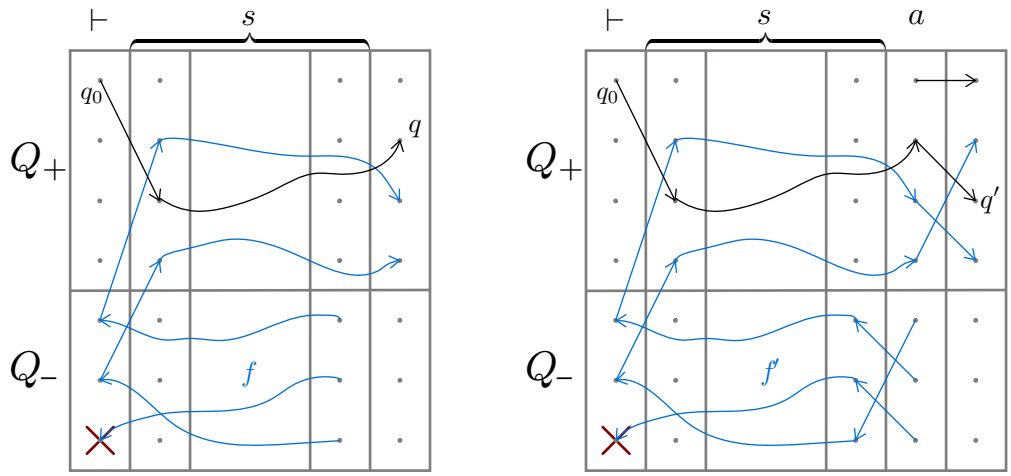


Figure 2: (left) $\mathscr{B}$ in a state (q,f); (right) transition of $\mathscr{B}$ by a.

To define (q,f) as an accepting or a rejecting state, consider the following sequence of states $\{q_i\}_{i\geqslant 1}$ with $q_i \in Q_+$. The first element is

$$q_1 = q$$

For each q_i if the 2PerFA has a transition by $\dashv$ from q_i and the function f is defined on $\delta_{\dashv}(q_i)$ then

$$q_{i+1} = f(\delta_{\dashv}(q_i))$$

Otherwise the sequence ends. The sequence $\{q_i\}_{i\geqslant 1}$ is always finite, because if it loops then some state $\widetilde{q}$ appears at least twice. Consider the earliest repeated state. If it is not q_1 then there is the previous one. The previous state for $\widetilde{q}$ is the same for all its appearances as $f(\delta_{\dashv})$ is an injective function. Therefore, $\widetilde{q}$ is not the earliest repeated state. So, $\widetilde{q}$ should be q_1. As $q_1 \notin \operatorname{Im} f$ the state q_1 cannot be repeated, a contradiction.

If this sequence ends with an accepting state $q_i \in F$, then the state (q,f) is accepting in $\mathscr{B}$. Otherwise, the state (q,f) is rejecting.

The constructed one-way automaton accepts the same language as the 2PerFA because when it comes by some string s to a state (q,f), then before accepting or rejecting on $\vdash s\dashv$ the 2PerFA passes through the sequence of states $\{q_i\}_{i\geqslant 1}$, with $q_i \in Q_+$, at $\dashv$.

Claim 3. *The resulting one-way automaton is a permutation automaton.*

Proof. We will prove that the transition function $\widetilde{\delta}_a$ is a bijection for each symbol $a \in \Sigma$. Firstly, we show its injectivity. Let

$$\widetilde{\delta}_a((q_1,f_1)) = \widetilde{\delta}_a((q_2,f_2))$$

Then, by the definition of $\widetilde{\delta}_a$,

$$(\delta_a^+(q_1), \delta_a^+ \circ f_1 \circ \delta_a^-) = (\delta_a^+(q_2), \delta_a^+ \circ f_2 \circ \delta_a^-)$$

Then $\delta_a^+(q_1) = \delta_a^+(q_2)$, which means that $q_1 = q_2$, because δ_a^+ is a bijection. Then consider the equality

$$\delta_a^+ \circ f_1 \circ \delta_a^- = \delta_a^+ \circ f_2 \circ \delta_a^-$$

Taking a composition of both sides of the equation with $(\delta_a^+)^{-1}$ on the left and $(\delta_a^-)^{-1}$ on the right yields

$$(\delta_a^+)^{-1} \circ \delta_a^+ \circ f_1 \circ \delta_a^- \circ (\delta_a^-)^{-1} = (\delta_a^+)^{-1} \circ \delta_a^+ \circ f_2 \circ \delta_a^- \circ (\delta_a^-)^{-1}$$

Then $f_1 = f_2$, and the injectivity is proved. The function $\widetilde{\delta_a}$ is total and has equal domain and range, it is therefore a bijection. □

This completes the proof of the lemma. □

Theorem 1. *For every 2PerFA* $\mathscr{A} = (\Sigma, Q_+, Q_-, q_0, \langle\delta_a^+\rangle_{a\in\Sigma}, \langle\delta_a^-\rangle_{a\in\Sigma}, \delta_\vdash, \delta_\dashv, F)$ *with*

$$|Q_+| = k, \quad |Q_-| = \ell, \quad |Q_-^\times| = m,$$

where $Q_-^\times \subseteq Q_-$ *is the set of states from which there is no transition by* $\vdash$*, there exists a 1PerFA with at most*

$$k \cdot \binom{\ell}{m} \cdot \binom{k-1}{\ell-m} \cdot (\ell-m)!$$

states that recognizes the same language.

Proof. Consider the one-way automaton $\mathscr{B}$ obtained for the 2PerFA $\mathscr{A}$ in Lemma 1. Every state (q, f) of $\mathscr{B}$ satisfies the following conditions:

- $q \in Q_+$,
- $f\colon Q_- \to Q_+$ is a partially defined function,
- $q \notin \operatorname{Im} f$,
- f is injective,
- and f is undefined on exactly m states.

For a fixed $q \in Q_+$, let us count the number of functions satisfying the conditions above: firstly, we should choose m states from Q_- on which f is not defined. Secondly, from the $k-1$ states we should choose $\ell - m$ different values for f's range. And lastly, we can choose a bijection between these two sets in $(\ell-m)!$ ways.

$$\binom{\ell}{m} \cdot \binom{k-1}{\ell-m} \cdot (\ell-m)!$$

By multiplying this number by k (the number of different states q) we will get the claimed number of states. □

If a two-way automaton has n states in total, then there is only a finite number of partitions into left-moving and right-moving states, and finitely many choices of m, and hence the following number of states is sufficient to transform this automaton to one-way.

$$F(n) = \max_{\substack{k,\ell,m \\ k>0,\ \ell\geqslant m\geqslant 0 \\ m\geqslant \ell-k+1}} k \cdot \binom{\ell}{m} \cdot \binom{k-1}{\ell-m} \cdot (\ell-m)!$$

Corollary 1. *For every n-state 2PerFA there exists a 1PerFA with* $F(n)$ *states that recognizes the same language.*

Later it will be proved that $F(n)$ is a sharp bound, that is, for some n-state 2PerFA every 1PerFA recognizing the same language has to have at least $F(n)$ states.

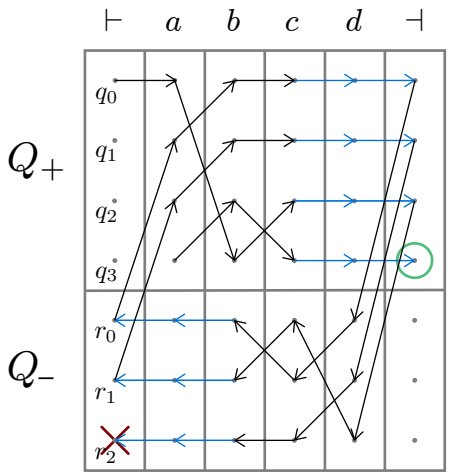


Figure 3: Symbols a,b,c,d in the construction of $\mathscr{A}$.

4 Lower bound on the number of states

In this section it will be shown that the upper bound on the number of states in a 1DFA needed to simulate a 2PerFA is sharp for each triple (k,ℓ,m), where $k>\ell$ and $\ell \geqslant m>0$. Only the case of $k>\ell$ is considered, because, as it will be shown later, the maximum over (k,ℓ,m) in $F(n)$ is reached for $k>\ell$ (in other words, a 2PerFA that requires the maximum number of states in a 1PerFA has $|Q_+|>|Q_-|$).

Theorem 2. *For all k,ℓ,m with $k>\ell>0$ and $\ell \geqslant m>0$ there exists a 2PerFA $\mathscr{A}=(\Sigma,Q_+,Q_-,q_0,\langle\delta_a^+\rangle_{a\in\Sigma},\langle\delta_a^-\rangle_{a\in\Sigma},\delta_\vdash,\delta_\dashv,F)$ such that*

$$|Q_+|=k, \quad |Q_-|=\ell,$$

the function $\delta_\vdash$ is undefined on exactly m arguments from Q_-, and every 1DFA recognizing $L(\mathscr{A})$ must have at least

$$k\cdot\binom{\ell}{m}\cdot\binom{k-1}{\ell-m}\cdot(\ell-m)!$$

states.

Proof. Fix k,ℓ,m and consider a 2PerFA $\mathscr{A}=(\Sigma,Q_+,Q_-,q_0,\langle\delta_a^+\rangle_{a\in\Sigma},\langle\delta_a^-\rangle_{a\in\Sigma},\delta_\vdash,\delta_\dashv,F)$ where

- $Q_+=\{q_0,\ldots,q_{k-1}\}, Q_-=\{r_0,\ldots,r_{\ell-1}\}$.
- The initial state is q_0 and the accepting states are $\{q_\ell,\ldots,q_{k-1}\}$.
- The functions δ_a^+ and δ_b^+ are generators of the permutation group on the set Q_+ (for example, these could be a cycle on all elements of Q_+ and an elementary transposition). Similarly, δ_c^-,δ_d^- are generators of the permutation group on the set Q_-, and $\delta_a^-,\delta_b^-,\delta_c^+,\delta_d^+$ are identity functions.
- Transitions at the left end-marker are $\delta_\vdash(q_0)=q_0$ and $\delta_\vdash(r_i)=q_{i+1}$ for $0\leqslant i<\ell-m$. There are no transitions by $\vdash$ in the remaining m states.
- Transitions at the right end-marker are $\delta_\dashv(q_i)=r_i$ for $0\leqslant i<\ell$. There are no transitions by $\dashv$ in the remaining $k-\ell$ states.

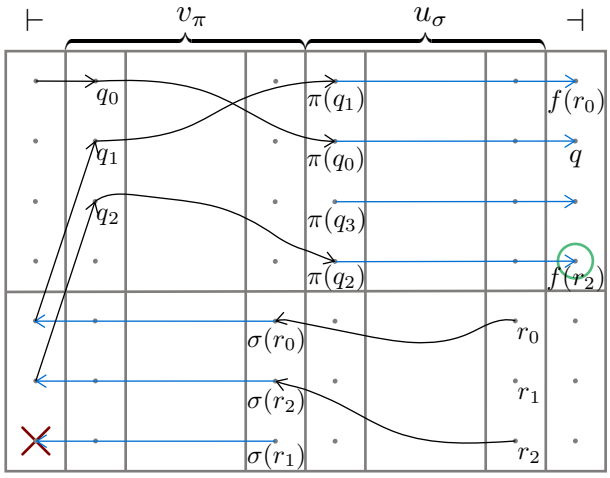


Figure 4: The action of σ and π.

The proof of the lower bound on the size of any 1DFA recognizing this language is by showing that the automaton $\mathscr{B} = (Q, \widetilde{q}_0, \widetilde{F}, \widetilde{\delta}, \Sigma)$ obtained from $\mathscr{A}$ by the transformation in Lemma 1 will be minimal. We will show that every state is reachable and for every two states there exists a separating string. Firstly prove the reachability. Consider a state (q, f). It satisfies the following conditions from Lemma 1:

- $q \in Q_+$,
- $f\colon Q_- \to Q_+$ is a partially defined function,
- $q \notin \mathrm{Im} f$,
- f is injective,
- f is undefined on exactly m states.

Let σ be a permutation on the set Q_- that maps the states on which f is not defined to the m states from Q_- without a transition by $\vdash$. Take a string $u_\sigma \in \{c,d\}^*$ that, when read from right to left, implements the permutation σ, and acts as an identity on Q_+ when read from left to right. Next, the goal is to define a permutation π on the set Q_+ that maps q_0 to q, and, for each state $q' \in Q_-$ on which f is defined, it should map the state $\delta_\vdash(\sigma(q'))$ to the state $f(q')$, as shown in Figure 4. Note that $\delta_\vdash(\sigma(q'))$ is defined for all q' in the domain of f. We can introduce such a permutation because each state $\delta_\vdash(\sigma(q'))$ is not equal to q_0 and they are all pairwise distinct (as σ is a permutation and $\delta_\vdash$ is an injection). Also each state $f(q')$ is not equal to q and they are all pairwise distinct too. Take the string $v_\pi \in \{a,b\}^*$ that, when read from left to right, implements π, and is an identity on Q_- if read from right to left. So, by the string $v_\pi u_\sigma$ we reach the state (q, f).

Next, the existence of a separating string for all pairs of states is proved. Consider different states (q_1, f_1) and (q_2, f_2). Let them be reached by strings s_1 and s_2, respectively. There are several cases.

- The states q_1, q_2 are different, as shown in Figure 5. Fix a state $r \in Q_-$ with no transition on the left end-marker defined. Since $\mathscr{A}$ is a permutation automaton, there exists a state $\widetilde{r} \in Q_-$ such that after reading s_1 from right to left starting in the state $\widetilde{r}$, the automaton is in the state r. Also let $q_1' \in Q_+$ be the state from which there is a transition to $\widetilde{r}$ by the right end-marker: such a state exists, because in the 2PerFA there are transitions to all states in Q_- by the right end-marker. Then

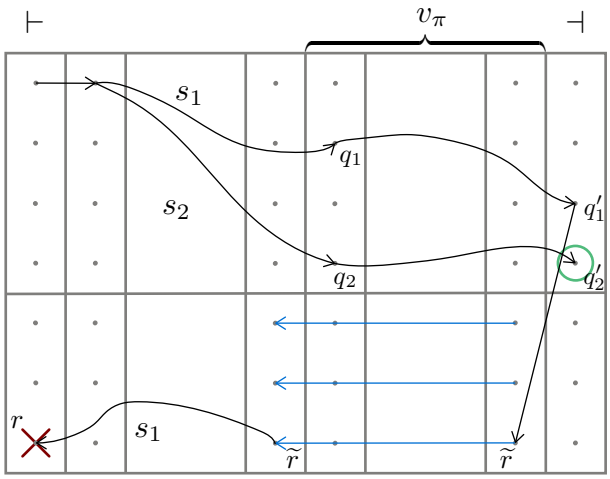


Figure 5: A separating string for states (q_1, f_1) and (q_2, f_2), with $q_1 \neq q_2$.

let π be such a permutation on the set Q_+ that maps q_1 to q_1' and q_2 to an accepting state $q_2' \in Q_+$. Let $v_\pi \in \{a,b\}^*$ be the string that implements the permutation π when read from left to right. So, the string $s_1 v_\pi$ will be rejected by 2PerFA and the string $s_2 v_\pi$ will be accepted. That is, v_π is a separating string.

- The states q_1, q_2 are the same, but $f_1 \neq f_2$. Because $f_1 \neq f_2$ there exists a state $r \in Q_-$ on which these functions differ, that is, either one of $f_1(r), f_2(r)$ is defined and the other is not, or both are defined and are different states.

  1. First, assume that f_1 is defined on r, but f_2 is not (the case of $f_2(r)$ defined and $f_1(r)$ undefined is symmetric). It is true that $f_1(r) \neq q_1$, because $q_1 \notin \mathrm{Im} f_1$ in every state of $\mathscr{B}$. Fix the state q with a transition from it to r by the right end-marker. Then let π be such a permutation of the set Q_+ that maps q_1 to q and $f_1(r)$ to an accepting state. And let the string $v_\pi \in \{a,b\}^*$ implement π when the 2PerFA reads it from left to right as shown in Figure 6. The string $s_1 v_\pi$ will be accepted by the 2PerFA and the string $s_2 v_\pi$ will be rejected. So, v_π is a separating string.

  2. Both functions f_1 and f_2 are defined on r. It is true that $f_1(r) \neq q_1$ and $f_2(r) \neq q_1$. Again, fix the state q from which there is a transition to r by the right end-marker. Then fix a state $\widetilde{r}$ from Q_- without a transition by the left end-marker (it exists because $m \geqslant 1$). Let $r^* \in Q_-$ be the state from which the 2PerFA reads s_1 and finishes in $\widetilde{r}$. We can choose such a state because the 2PerFA is a permutation automaton, and this state cannot coincide with r because in this case $f_1(r)$ would be undefined. Then let $q^* \in Q_+$ be the state, from which there is a transition by the right end-marker to r^*: it exists because there are such transitions to all states in Q_-. Let $v_\pi \in \{a,b\}^*$ implement a permutation π on Q_+ that maps q_1 to q, $f_1(r)$ to q^* and $f_2(r)$ to an accepting state, as illustrated in Figure 7. The string $s_1 v_\pi$ will be rejected by the 2PerFA, and the string $s_2 v_\pi$ will be accepted, so, v_π is a separating string.

□

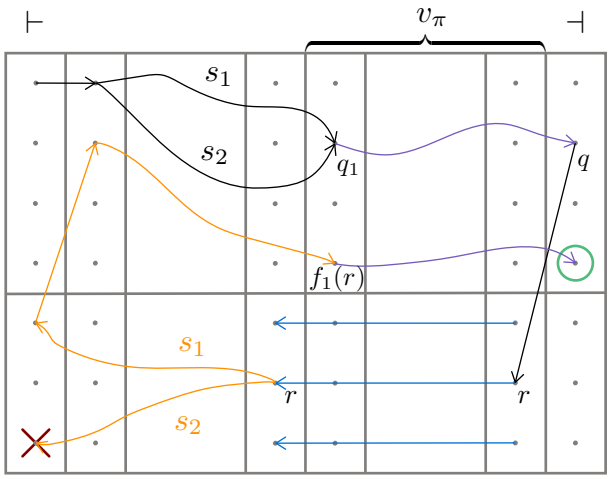


Figure 6: A separating string for states (q_1, f_1) and (q_2, f_2), with $q_1 = q_2$, $f_1(r)$ defined, $f_2(r)$ undefined.

5 Optimal partition of n in $F(n)$ and the logarithmic asymptotics of $F(n)$

It has been proved above that every n-state 2PerFA can be transformed to an equivalent 1PerFA with

$$F(n) = \max_{\substack{k,\ell,m \\ k>0,\ \ell \geqslant m \geqslant 0 \\ m \geqslant \ell-k+1}} G(k,\ell,m)$$

states, where

$$G(k,\ell,m) = k \cdot \binom{\ell}{m} \cdot \binom{k-1}{\ell-m} \cdot (\ell-m)!$$

What is the optimal partition of n states into k right-moving and ℓ left-moving states, and what is the optimal number m of unused states at the left end-marker? This question is answered in the following lemma.

Lemma 2. *For every fixed $n = k+\ell$ the function*

$$G(k,\ell,m) = k \cdot \binom{\ell}{m} \cdot \binom{k-1}{\ell-m} \cdot (\ell-m)!$$

is defined for $k > 0$, $\ell \geqslant m \geqslant 0$, $m \geqslant \ell-k+1$, and reaches its maximum value on a triple (k,ℓ,m), where $k > \ell$. If $n \geqslant 8$, then the optimal values of the arguments are:

$$k = \left\lfloor \frac{n+2}{2} \right\rfloor, \quad \ell = \left\lceil \frac{n-2}{2} \right\rceil, \quad m = \begin{cases} \left\lceil \frac{\sqrt{3+2n}-3}{2} \right\rceil, & n \text{ is odd} \\ \left\lceil \frac{\sqrt{4+2n}-4}{2} \right\rceil, & n \text{ is even} \end{cases}$$

Sketch of a proof. To prove this, firstly, find an optimal value of m for a fixed pair (k,ℓ). Denote it by $m_{\text{opt}} = m_{\text{opt}}(k,\ell)$. Then analyse the next ratio

$$\frac{G(k,\ell,m_{\text{opt}}(k,\ell))}{G(k+1,\ell-1,m_{\text{opt}}(k+1,\ell-1))}$$

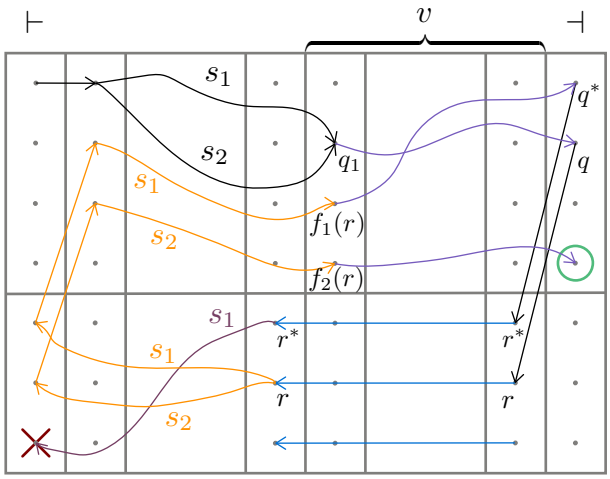


Figure 7: A separating string for states (q_1, f_1) and (q_2, f_2), with $q_1 = q_2$, $f_1(r) \neq f_2(r)$.

It is proved that this ratio is at least 1 if $k > \ell$ and $n \geqslant 8$, and at most 1 if $k \leqslant \ell$. Therefore, the optimal partition $n = k + \ell$ has $k > \ell$. If $n \geqslant 8$, then it has $k = \ell + 1$ or $k = \ell + 2$, depending on the parity of n, and the optimal values of k and ℓ are

$$k = \left\lfloor \frac{n+2}{2} \right\rfloor, \quad \ell = \left\lceil \frac{n-2}{2} \right\rceil$$

There is a formula for $m_{\text{opt}}(k, \ell)$, and its value for approximately equal k and ℓ is

$$m_{\text{opt}}(k, \ell) = \left\lceil \frac{\sqrt{D} + \ell - k - 2}{2} \right\rceil, \quad \text{where } D = (k-\ell)^2 + 4(\ell+1)$$

Then, for a given $n \geqslant 8$, the claimed optimal value of m can be found by substituting the optimal values of k and ℓ into the formula for m_{opt}. □

With the optimal values of k, ℓ and m determined, the main result of this paper can now be finally stated.

Theorem 3. *Let $n \geqslant 1$. For every n-state 2PerFA there exists a 1PerFA with $F(n)$ states that recognizes the same language, and in the worst case $F(n)$ states in a 1PerFA are necessary.*

Proof. The upper bound is given in Corollary 1.

For the lower bound, for every $n \geqslant 8$, let k, ℓ and m be as in Lemma 2. Then, since $k > \ell$, Theorem 2 presents the desired n-state 2PerFA, for which every 1PerFA recognizing the same language must have at least $G(k, \ell, m) = F(n)$ states.

For $n = 5, 6, 7$, a calculation of possible values k, ℓ, m shows that the maximum of $G(k, \ell, m)$ is reached for $m = 1$. Then, Theorem 2 is still applicable and provides the witness languages.

For $n = 4$, the optimal values given by a calculation are $k = 3$, $\ell = 1$ and $m = 0$. Nevertheless, the same automaton as in Theorem 2 still provides the desired lower bound, which was checked by a computer calculation.

Finally, $F(n) = n$ for $n = 1, 2, 3$, and (trivial) witness languages are $(a^n)^*$. □

So, $F(n) = G(k,\ell,m)$ for the specified values of k,ℓ,m. As

$$G(k,\ell,m) = k \cdot \binom{\ell}{m} \cdot \binom{k-1}{\ell-m} \cdot (\ell-m)! = \frac{k!\ell!}{(k-1-\ell+m)!m!(\ell-m)!}$$

the asymptotics of $F(n)$ can be determined by using Stirling's approximation of factorials for k,ℓ,m from the optimal partition. The final result is given in the following theorem.

Theorem 4. $F(n) = n^{\frac{n}{2} - \frac{1+\ln 2}{2} \frac{n}{\ln n}(1+o(1))}$.

To compare, transformation of 2DFA of the general form to 1DFA has the sharp bound proved by Kapoutsis [6].

$$n(n^n - (n-1)^n) + 1$$

The transformation of sweeping 2DFA to 1DFA [2] requires slightly fewer states, yet still of the order $n^{n(1+o(1))}$.

$$\varphi(n) = \max_{k=1}^{n} k^{n-k+1} + 1 = n^{n - \frac{n \ln\ln n}{\ln n} + O(\frac{n}{\ln n})}$$

Evidently, in the case of 2PerFA, the cost of transformation to one-way is substantially reduced (with the exponent divided by two).

The transformation complexity in these three cases is compared for small values of n in Table 1.

n	$F(n)$ **(2PerFA to 1DFA)**	$\max_{k=1}^{n} k^{n-k+1} + 1$ (sweeping to 1DFA)	$n(n^n - (n-1)^n) + 1$ (2DFA to 1DFA)
1	1	2	2
2	2	3	7
3	3	5	58
4	6	10	701
5	12	28	10506
6	24	82	186187
7	72	257	3805250
8	180	1025	88099321
9	480	4097	2278824850
10	1440	16385	65132155991
11	3600	78126	2038428376722
12	12600	390626	69332064858421

Table 1: The value of $F(n)$ compared to the known transformations for irreversible 2DFA, for small values of n.

6 A more general definition

Consider a variant of the definition of a 2PerFA, in which acceptance is also allowed at the left end-marker in states from Q_-. It entails that in a transformation from 2PerFA to 1PerFA in each state (q,f) the function f operates from Q_- to $Q_+ \cup \{\mathrm{ACC}, \mathrm{REJ}\}$. Upon reading a string $u \in \Sigma^*$, the 1PerFA comes to a state (q,f), where $q \in Q_+$ is the state in which the 2PerFA first moves to the right from the last symbol of $\vdash u$, and for every state $r \in Q_-$ and $p \in Q_+$, if $f(r) = p$, then the 2PerFA after reading $\vdash u$

starting at its last symbol in the state r finishes in the state p. If $f(r) = \text{REJ}$ then, after reading $\vdash u$ from right to left starting in r, the 2PerFA rejects at the left end-marker. And if $f(r) = \text{ACC}$, then, after reading $\vdash u$ from right to left starting in r the 2PerFA accepts at the left end-marker. The transition function δ, the initial state and the set of accepting states will be defined similarly to Lemma 1. The automaton constructed by this transformation will be a permutation automaton. To prove this claim, δ is first shown to be injective, and then bijectivity follows from the equality of its domain and range.

As in the proof of Lemma 1, suppose that δ is not injective, and has the same value on two different states:

$$\delta((q_1, f_1)) = \delta((q_2, f_2))$$

$$(\delta_a^+(q_1), \delta_a^+ \circ f_1 \circ \delta_a^-) = (\delta_a^+(q_2), \delta_a^+ \circ f_2 \circ \delta_a^-)$$

From

$$\delta_a^+(q_1) = \delta_a^+(q_2)$$

follows

$$q_1 = q_2$$

as δ_a^+ is a bijection. And from

$$\delta_a^+ \circ f_1 \circ \delta_a^- = \delta_a^+ \circ f_2 \circ \delta_a^-$$

by multiplying by inverse functions of $(\delta_a^+)^{-1}, (\delta_a^-)^{-1}$ from the left side and from the right side respectively, the next equation follows

$$(\delta_a^+)^{-1} \circ \delta_a^+ \circ f_1 \circ \delta_a^- \circ (\delta_a^-)^{-1} = (\delta_a^+)^{-1} \circ \delta_a^+ \circ f_2 \circ \delta_a^- \circ (\delta_a^-)^{-1}$$

$$f_1 = f_2$$

So, $(q_1, f_1) = (q_2, f_2)$, therefore δ is a bijection and the constructed automaton is a permutation automaton.

Denote the number of accepting states in Q_- by e. The exact number of states in the constructed 1PerFA is given in the following theorem.

Theorem 5. *For every 2PerFA $\mathscr{A} = (\Sigma, Q_+, Q_-, q_0, \langle \delta_a^+ \rangle_{a \in \Sigma}, \langle \delta_a^- \rangle_{a \in \Sigma}, \delta_\vdash, \delta_\dashv, F)$ with $F \subseteq Q_+ \cup Q_-$ and*

$$|Q_+| = k, \quad |Q_-| = \ell, \quad |Q_-^\times| = m, \quad |F \cap Q_-| = e,$$

where $Q_-^\times \subseteq Q_-$ is the set of rejecting states from which there is no transition by $\vdash$, there exists a 1PerFA with at most

$$k \cdot \binom{\ell}{m} \cdot \binom{m}{e} \cdot \binom{k-1}{\ell-m} \cdot (\ell - m)!$$

states that recognizes the same language.

7 Conclusion

The complexity of transforming sweeping permutation automata (2PerFA) to classical one-way permutation automata (1PerFA) has been determined precisely. A suggested question for future research is the state complexity of operations on 2PerFA. Indeed, state complexity of operations on 1PerFA has recently been investigated [3, 12], state complexity of operations on 2DFA of the general form was studied as well [4], and it would be interesting to know how the case of 2PerFA compares to these related models.

References

[1] D. Angluin, "Inference of Reversible Languages", *Journal of the ACM*, 29:3 (1982), 741–765. https://doi.org/10.1145/322326.322334

[2] V. Geffert, A. Okhotin, "Deterministic one-way simulation of two-way deterministic finite automata over small alphabets", *Descriptional Complexity of Formal Systems 2021*, LNCS 13037, 26–37. https://doi.org/10.1007/978-3-030-93489-7_3

[3] M. Hospodár, P. Mlynárčik, "Operations on Permutation automata", DLT 2020, LNCS 12086, 122–136. https://doi.org/10.1007/978-3-030-48516-0_10

[4] G. Jirásková, A. Okhotin, "On the state complexity of operations on two-way finite automata", *Information and Computation*, 253:1 (2017), 36–63. http://dx.doi.org/10.1016/j.ic.2016.12.007

[5] C. A. Kapoutsis, "Removing bidirectionality from nondeterministic finite automata", *Mathematical Foundations of Computer Science* (MFCS 2005, Gdansk, Poland, 29 August–2 September 2005), LNCS 3618, 544–555. http://dx.doi.org/10.1007/11549345_47

[6] C. A. Kapoutsis, *Algorithms and Lower Bounds in Finite Automata Size Complexity*, Ph. D. thesis, Massachusetts Institute of Technology, 2006.

[7] A. Kondacs, J. Watrous, "On the power of quantum finite state automata", *38th Annual Symposium on Foundations of Computer Science* (FOCS 1997, Miami Beach, Florida, USA, 19–22 October 1997), IEEE, 66–75. http:dx.doi.org/10.1109/SFCS.1997.646094

[8] M. Kunc, A. Okhotin, "Reversibility of computations in graph-walking automata", *Information and Computation*, 275 (2020), article 104631. https://doi.org/10.1016/j.ic.2020.104631

[9] S. W. Margolis, J.-É. Pin, "Products of group languages", FCT 1985, 285–299. https://doi.org/10.1007/BFb0028813

[10] J.-É. Pin, "On the Language Accepted by Finite Reversible automata", *Automata, Languages and Programming, 14th International Colloquium*, (ICALP 1987, Karlsruhe, Germany, July 13–17, 1987), LNCS 267, 237–249. https://doi.org/10.1007/3-540-18088-5_19

[11] J.-É. Pin, "On Reversible automata", *LATIN '92, 1st Latin American Symposium on Theoretical Informatics* (São Paulo, Brazil, April 6–10, 1992), LNCS 583, 401–416. https://doi.org/10.1007/BFb0023844

[12] C. Rauch, M. Holzer, "On the Accepting State Complexity of Operations on Permutation Automata", *Proceedings 12th International Workshop on Non-Classical Models of Automata and Applications* (NCMA 2022, Debrecen, Hungary, August 26–27, 2022), EPTCS 367, 2022, 177–189. https://doi.org/10.4204/EPTCS.367.12

[13] J. C. Shepherdson, "The reduction of two-way automata to one-way automata", *IBM Journal of Research and Development*, 3 (1959), 198–200. http://dx.doi.org/10.1147/rd.32.0198

[14] M. Sipser, "Lower bounds on the size of sweeping automata", *Journal of Computer and System Sciences*, 21:2 (1980), 195–202. https://doi.org/10.1016/0022-0000(80)90034-3

[15] G. Thierrin, "Permutation automata", *Mathematical Systems Theory*, 2:1 (1968), 83–90. https://doi.org/10.1007/BF01691347

Merging two Hierarchies of Internal Contextual Grammars with Subregular Selection

Bianca Truthe
Institut für Informatik, Universität Giessen, Arndtstr. 2, 35392 Giessen, Germany
bianca.truthe@informatik.uni-giessen.de

In this paper, we continue the research on the power of contextual grammars with selection languages from subfamilies of the family of regular languages. In the past, two independent hierarchies have been obtained for external and internal contextual grammars, one based on selection languages defined by structural properties (finite, monoidal, nilpotent, combinational, definite, ordered, non-counting, power-separating, suffix-closed, commutative, circular, or union-free languages), the other one based on selection languages defined by resources (number of non-terminal symbols, production rules, or states needed for generating or accepting them). In a previous paper, the language families of these hierarchies for external contextual grammars were compared and the hierarchies merged. In the present paper, we compare the language families of these hierarchies for internal contextual grammars and merge these hierarchies.

1 Introduction

Contextual grammars were introduced by S. Marcus in [17] as a formal model that might be used in the generation of natural languages. The derivation steps consist in adding contexts to given well formed sentences, starting from an initial finite basis. Formally, a context is given by a pair (u,v) of words and inserting it externally into a word x gives the word uxv whereas inserting it internally gives all words $x_1ux_2vx_3$ when $x = x_1x_2x_3$. In order to control the derivation process, contextual grammars with selection were defined. In such contextual grammars, a context (u,v) may be added only if the surrounded word x or x_2 belongs to a language which is associated with the context. Language families were defined where all selection languages in a contextual grammar belong to some language family $\mathscr{F}$. Such contextual grammars are said to be 'with selection in the family $\mathscr{F}$'. Contextual grammars have been studied where the family $\mathscr{F}$ is taken from the Chomsky hierarchy (see [15, 20, 21] and references therein).

In [4], the study of external contextual grammars with selection in special regular sets was started. Finite, combinational, definite, nilpotent, regular suffix-closed, regular commutative languages and languages of the form V^* for some alphabet V were considered. The research was continued in [8, 9, 10, 16] where further subregular families of selection languages were considered and the effect of subregular selection languages on the generative power of external and internal contextual grammars was investigated. A recent survey can be found in [27] which presents for each type of contextual grammars (external and internal ones) two hierarchies, one based on selection languages defined by structural properties (finite, monoidal, nilpotent, combinational, definite, ordered, non-counting, power-separating, suffix-closed, commutative, circular, or union-free languages), the other one based on selection languages defined by resources (number of non-terminal symbols, production rules, or states needed for generating or accepting them). In [28], the language families of these hierarchies for external contextual grammars were compared and the hierarchies merged. In the present paper, we compare the language families of these hierarchies for internal contextual grammars and merge the hierarchies.

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 125–139, doi:10.4204/EPTCS.388.12

The internal case is different from the case of external contextual grammars, as there are two main differences between the ways in which words are derived. In the case of internal contextual grammars, it is possible that the insertion of a context into a sentential form can be done at more than one place, such that the derivation becomes in some sense non-deterministic; in the case of external grammars, once a context was selected, there is at most one way to insert it: wrapped around the sentential form, when this word is in the selection language of the context. On the other hand, the outermost ends of a word derived externally have been added at the end of the derivation, whereas derived internally the ends could have been at the ends of the word already from the beginning since some inner part can be 'pumped'. If a context can be added internally, then it can be added arbitrarily often (because the subword where the context is wrapped around does not change) which does not necessarily hold for external grammars.

In Section 2, we give the definitions and notation of the concepts used in this paper (languages, grammars, automata, subregular language families, inclusion relations between these families, contextual grammars, and inclusion relations between the families generated by internal contextual grammars where the selection languages belong to various subregular language families). In Section 3, we present our results where, first, several languages are presented which later serve as witness languages for proper inclusions or the incomparability of two language families and, later, these languages are used to prove relations between the various language families generated by internal contextual grammars with different types of selection. Finally, in Section 4, we state some problems which are left open and give some ideas for future research.

2 Preliminaries

Throughout the paper, we assume that the reader is familiar with the basic concepts of the theory of automata and formal languages. For details, we refer to [21]. Here we only recall some notation and the definition of contextual grammars with selection which form the central notion of the paper.

2.1 Languages, grammars, automata

Given an alphabet V, we denote by V^* and V^+ the set of all words and the set of all non-empty words over V, respectively. The empty word is denoted by λ. By V^k and $V^{\leq k}$ for some natural number k, we denote the set of all words of the alphabet V with exactly k letters and the set of all words over V with at most k letters, respectively. For a word w and a letter a, we denote the length of w by $|w|$ and the number of occurrences of the letter a in the word w by $|w|_a$. For a set A, we denote its cardinality by $|A|$.

A right-linear grammar is a quadruple

$$G = (N, T, P, S)$$

where N is a finite set of non-terminal symbols, T is a finite set of terminal symbols, P is a finite set of production rules of the form $A \to wB$ or $A \to w$ with $A, B \in N$ and $w \in T^*$, and $S \in N$ is the start symbol. Such a grammar is called regular, if all the rules are of the form $A \to xB$ or $A \to x$ with $A, B \in N$ and $x \in T$ or $S \to \lambda$. The language generated by a right-linear or regular grammar is the set of all words over the terminal alphabet which are obtained from the start symbol S by a successive replacement of the non-terminal symbols according to the rules in the set P. Every language generated by a right-linear grammar can also be generated by a regular grammar.

A deterministic finite automaton is a quintuple

$$\mathscr{A} = (V, Z, z_0, F, \delta)$$

where V is a finite set of input symbols, Z is a finite set of states, $z_0 \in Z$ is the initial state, $F \subseteq Z$ is a set of accepting states, and δ is a transition function $\delta : Z \times V \to Z$. The language accepted by such an automaton is the set of all input words over the alphabet V which lead letterwise by the transition function from the initial state to an accepting state.

The set of all languages generated by some right-linear grammar coincides with the set of all languages accepted by a deterministic finite automaton. All these languages are called regular and form a family denoted by *REG*. Any subfamily of this set is called a subregular language family.

2.2 Resources restricted languages

We define subregular families by restricting the resources needed for generating or accepting their elements:

$$RL_n^V = \{\, L \mid L \text{ is generated by a right-linear grammar with at most } n \text{ non-terminal symbols} \,\},$$
$$RL_n^P = \{\, L \mid L \text{ is generated by a right-linear grammar with at most } n \text{ production rules} \,\},$$
$$REG_n^Z = \{\, L \mid L \text{ is accepted by a deterministic finite automaton with at most } n \text{ states} \,\}.$$

2.3 Subregular language families based on the structure

We consider the following restrictions for regular languages. Let L be a language over an alphabet V. With respect to the alphabet V, the language L is said to be

- *monoidal* if and only if $L = V^*$,
- *nilpotent* if and only if it is finite or its complement $V^* \setminus L$ is finite,
- *combinational* if and only if it has the form $L = V^*X$ for some subset $X \subseteq V$,
- *definite* if and only if it can be represented in the form $L = A \cup V^*B$ where A and B are finite subsets of V^*,
- *suffix-closed* (or *fully initial* or *multiple-entry* language) if and only if, for any two words $x \in V^*$ and $y \in V^*$, the relation $xy \in L$ implies the relation $y \in L$,
- *ordered* if and only if the language is accepted by some deterministic finite automaton

  $$\mathscr{A} = (V, Z, z_0, F, \delta)$$

  with an input alphabet V, a finite set Z of states, a start state $z_0 \in Z$, a set $F \subseteq Z$ of accepting states and a transition mapping δ where $(Z, \preceq)$ is a totally ordered set and, for any input symbol $a \in V$, the relation $z \preceq z'$ implies $\delta(z,a) \preceq \delta(z',a)$,
- *commutative* if and only if it contains with each word also all permutations of this word,
- *circular* if and only if it contains with each word also all circular shifts of this word,
- *non-counting* (or *star-free*) if and only if there is a natural number $k \geq 1$ such that, for any three words $x \in V^*$, $y \in V^*$, and $z \in V^*$, it holds $xy^kz \in L$ if and only if $xy^{k+1}z \in L$,
- *power-separating* if and only if, there is a natural number $m \geq 1$ such that for any word $x \in V^*$, either $J_x^m \cap L = \emptyset$ or $J_x^m \subseteq L$ where $J_x^m = \{\, x^n \mid n \geq m \,\}$,
- *union-free* if and only if L can be described by a regular expression which is only built by product and star.

We remark that monoidal, nilpotent, combinational, definite, ordered, and union-free languages are regular, whereas non-regular languages of the other types mentioned above exist. Here, we consider among the commutative, circular, suffix-closed, non-counting, and power-separating languages only those which are also regular.

Some properties of the languages of the classes mentioned above can be found in [22] (monoids), [12] (nilpotent languages), [14] (combinational and commutative languages), [19] (definite languages), [13] and [2] (suffix-closed languages), [23] (ordered languages), [3] (circular languages), [18] (non-counting languages), [24] (power-separating languages), [1] (union-free languages).

By FIN, MON, NIL, $COMB$, DEF, SUF, ORD, $COMM$, $CIRC$, NC, PS, UF, and REG, we denote the families of all finite, monoidal, nilpotent, combinational, definite, regular suffix-closed, ordered, regular commutative, regular circular, regular non-counting, regular power-separating, union-free, and regular, languages, respectively.

As the set of all families under consideration, we set

$$\mathfrak{F} = \{FIN, MON, NIL, COMB, DEF, SUF, ORD, COMM, CIRC, NC, PS, UF\} \cup \{\, RL_n^V \mid n \geq 1 \,\} \cup \{\, RL_n^P \mid n \geq 1 \,\} \cup \{\, REG_n^Z \mid n \geq 1 \,\}.$$

2.4 Hierarchy of subregular families of languages

We present here a hierarchy of the families of the aforementioned set $\mathfrak{F}$ with respect to the set theoretic inclusion relation.

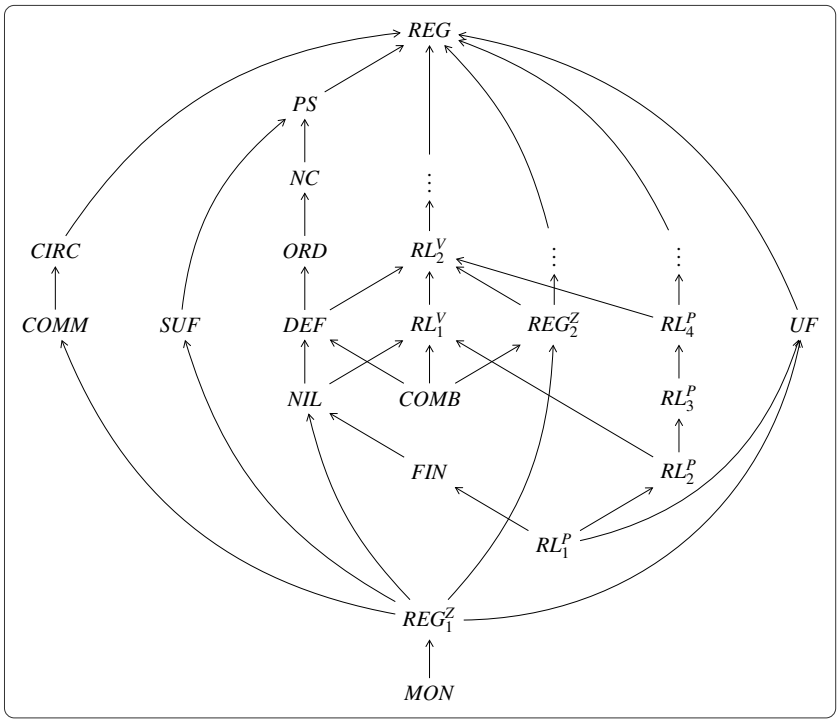


Figure 1: Hierarchy of subregular language families

Theorem 2.1 *The inclusion relations presented in Figure 1 hold. An arrow from an entry X to an entry Y depicts the proper inclusion $X \subset Y$; if two families are not connected by a directed path, then they are incomparable.*

For proofs and references to proofs of the relations, we refer to [26].

2.5 Contextual grammars

Let $\mathscr{F}$ be a family of languages. A contextual grammar with selection in $\mathscr{F}$ is a triple

$$G = (V, \mathscr{S}, A)$$

where

- V is an alphabet,
- $\mathscr{S}$ is a finite set of selection pairs (S,C) where S is a selection language over some subset U of the alphabet V which belongs to the family $\mathscr{F}$ with respect to the alphabet U and where $C \subset V^* \times V^*$ is a finite set of contexts with the condition, for each context $(u,v) \in C$, at least one side is not empty: $uv \neq \lambda$,
- A is a finite subset of V^* (its elements are called axioms).

Let $G = (V, \mathscr{S}, A)$ be a contextual grammar with selection. A direct internal derivation step in G is defined as follows: a word x derives a word y (written as $x \Longrightarrow y$) if and only if there are words x_1, x_2, x_3 with $x_1x_2x_3 = x$ and there is a selection pair $(S,C) \in \mathscr{S}$ such that $x_2 \in S$ and $y = x_1ux_2vx_3$ for some pair $(u,v) \in C$. Intuitively, we can only wrap a context $(u,v) \in C$ around a subword x_2 of x if x_2 belongs to the corresponding selection language S.

By $\Longrightarrow^*$, we denote the reflexive and transitive closure of the relation $\Longrightarrow$. The language generated by G is defined as

$$L = \{\, z \mid x \Longrightarrow^* z \text{ for some } x \in A \,\}.$$

By $\mathscr{IC}(\mathscr{F})$, we denote the family of all languages generated internally by contextual grammars with selection in $\mathscr{F}$. When a contextual grammar works in the internal mode, we call it an internal contextual grammar.

From previous research, we have the two hierarchies depicted in Figure 2. An arrow from an entry X to an entry Y depicts the proper inclusion $X \subset Y$; a solid arrow indicates that the inclusion is proper, the dashed arrow from $\mathscr{IC}(ORD)$ to $\mathscr{IC}(NC)$ indicates that it is not known so far whether this inclusion is proper or whether equality holds. The label at an edge shows in which paper the relation was proved.

If two families X and Y are not connected by a directed path, then X and Y are in most cases incomparable. The only exceptions are the relations of the family $\mathscr{IC}(SUF)$ to the families $\mathscr{IC}(ORD)$ and $\mathscr{IC}(NC)$ where it is not known whether they are incomparable or whether $\mathscr{IC}(SUF)$ is a subset of the other and the relation of the family $\mathscr{IC}(REG^Z_{n+1})$ to $\mathscr{IC}(RL^V_n)$ for $n \geq 1$ where it is not known whether they are incomparable or whether $\mathscr{IC}(REG^Z_{n+1})$ is a subset of $\mathscr{IC}(RL^V_n)$.

We note here that in [4, 8, 9, 10, 16, 25, 5] a slightly different definition was used than in [27, 11] and the present paper. This difference consists in the alphabet of the selection languages. In the early papers, the selection languages belong to some subfamily $\mathscr{F}$ with respect to the whole alphabet V of the contextual grammar whereas in later papers, the selection languages belong to some subfamily $\mathscr{F}$ with respect to some subalphabet $U \subseteq V$ of the contextual grammar. The language $\{a\}^*\{a\}^5$, for instance, is nilpotent with respect to the alphabet $\{a\}$ but not with respect to the alphabet $\{a,b\}$. For almost all

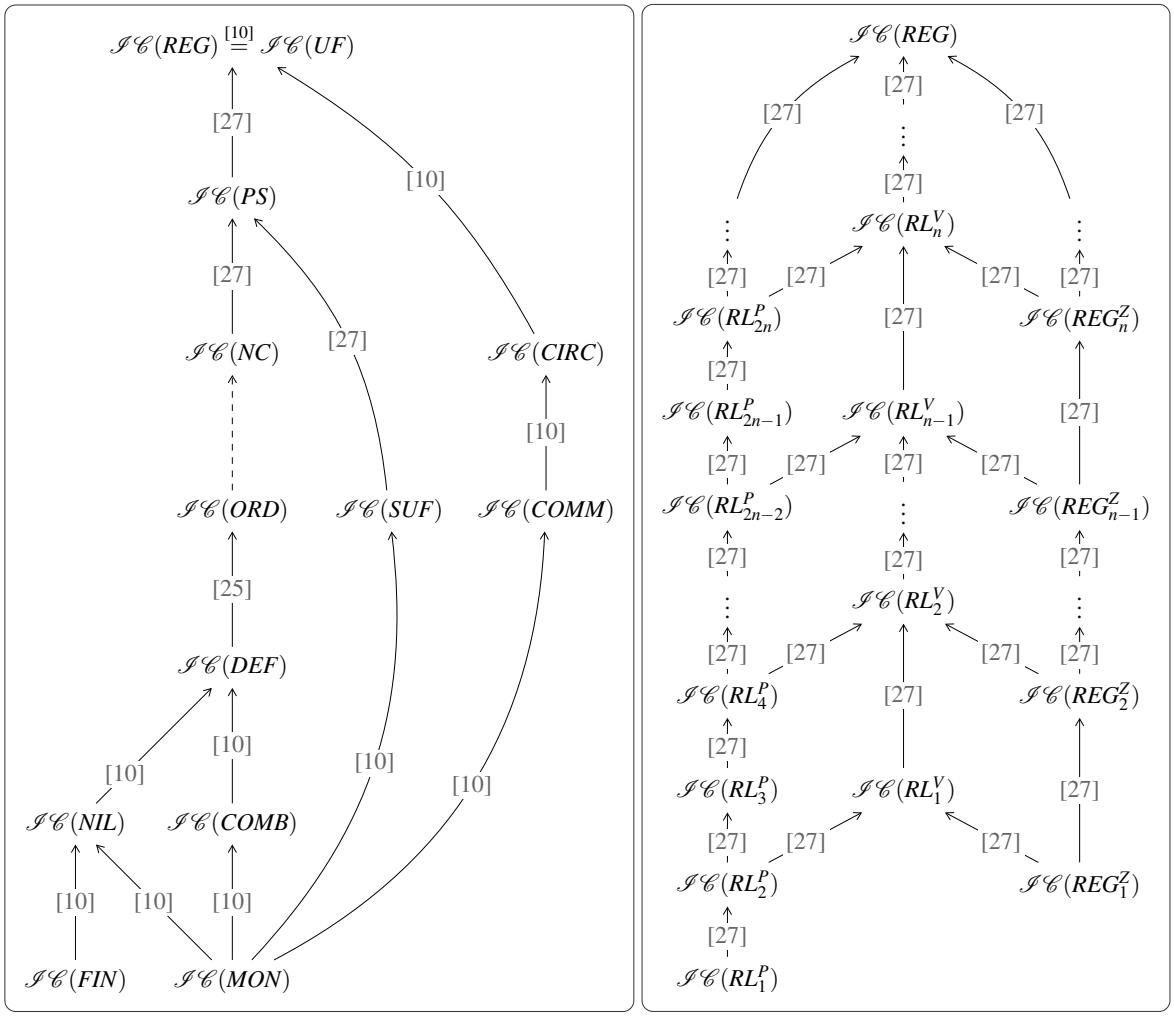


Figure 2: Hierarchies of the language families by internal contextual grammars with selection languages defined by structural properties (left) or restricted resources (right). An edge label refers to the paper where the respective inclusion is proved.

proofs in the mentioned papers, there is no difference between using one or the other definition. The only proof which relies on the definition is that of the relation $L(G) \notin \mathscr{IC}(DEF)$ for

$$G = (V, \{(Suf(\{d\}^*\{b\}), \{(a,b)\}), (\{a, \lambda\}, \{(c,d)\})\}, \{ecadb\})$$

from [10, Lemma 21] (also used in [25, Theorem 3.5]). However, the proof is valid also with the subalphabet definition if one changes the axiom *ecadb* to the word *dcadb*.

From the definition follows that the subset relation is preserved under the use of contextual grammars: if we allow more, we do not obtain less.

Lemma 2.2 *For any two language classes X and Y with $X \subseteq Y$, we have the inclusion*

$$\mathscr{IC}(X) \subseteq \mathscr{IC}(Y).$$

In the following section, we relate the families of the two hierarchies mentioned above.

3 Results

When we speak about contextual grammars in this section, we mean internal contextual grammars (whose languages are generated in the internal mode).

First, we present languages which will serve later as witness languages for proper inclusions or incomparabilities.

Lemma 3.1 *Let $V = \{a,b,c,d,e\}$ be an alphabet, $G = (V, \{(S_1,C_1),(S_2,C_2)\}, \{c\})$ be a contextual grammar with*

$$\begin{aligned} S_1 &= \{b\}^*\{c\}, \quad & C_1 &= \{(ab,ab)\}, \\ S_2 &= \{aa\}^*, & C_2 &= \{(d,e)\}, \end{aligned}$$

and $L = L(G)$ be the laguage generated. Then

$$L \in (\mathscr{IC}(RL_1^V) \cap \mathscr{IC}(RL_2^P) \cap \mathscr{IC}(REG_2^Z)) \setminus \mathscr{IC}(PS).$$

Proof. The selection languages are generated by right-linear grammars with the following rules (and start symbol S):

$$\begin{aligned} S_1 &: \ S \to bS, \ S \to c, \\ S_2 &: \ S \to aaS, \ S \to \lambda. \end{aligned}$$

Since these rules contain one non-terminal symbol only and two rules each, we obtain

$$L \in \mathscr{IC}(RL_1^V) \cap \mathscr{IC}(RL_2^P).$$

Since the words of the language L contain only one letter c (the axiom has no more and the contexts do not contain c), the language L is also generated if S_1 is replaced by the language $S_1' = (\{b\}^*\{c\})^+$ (the additional words cannot be used for selection).

The selection languages S_1' and S_2 are accepted by automata with two states each whose transition functions are given in the following diagram:

Hence, $L \in \mathscr{IC}(REG_2^Z)$.

Now we prove that L cannot be generated by a contextual grammar where all selection languages are power-separating. Assume the contrary. Then there is a contextual grammar $G' = (V, \mathscr{S}, A)$ which also generates the language L and all selection languages belong to the class PS. For each selection language S occurring in $\mathscr{S}$, there exists a natural number $m_S \geq 1$ such that for all words $x \in V^*$ it holds

$$\text{either } J_x^{m_S} \cap S = \emptyset \text{ or } J_x^{m_S} \subseteq S \text{ with } J_x^{m_S} = \{\, x^n \mid n \geq m_S \,\}.$$

Since $\mathscr{S}$ is finite, there is also a natural number $m \geq 1$ such that

$$\text{either } J_x^{m} \cap S = \emptyset \text{ or } J_x^{m} \subseteq S \text{ with } J_x^{m} = \{\, x^n \mid n \geq m \,\}$$

holds for every selection language S. Now let m be such a value.

Further, let k be the maximal length of the axioms and contexts plus m:

$$k = \max\{\max\{\ |w| \mid w \in A\ \}, \max\{\ |uv| \mid (u,v) \in C, (S,C) \in \mathscr{S}\ \}\} + m.$$

Consider the word $w = da^{2k}eb^{2k}c(ab)^{2k}$ which belongs to the language L but not to the set A of axioms due to its length. Therefore, it is derived from another word $w' \in L$ by insertion of a context (u,v) from a selection pair (S,C). We now study the possibilities for u and, depending from this, also for v. Let w'_1, w'_2, and w'_3 be the subwords of w' which are separated by the insertion of (u,v):

$$w' = w'_1w'_2w'_3 \Longrightarrow w'_1uw'_2vw'_3 = w.$$

If $u = d$, then $v = e$. This case will be continued later.

If $u = da^n$ for some n with $1 \leq n \leq k$, then v contains the letter e and has to bear also n letters b to be inserted before the letter c but also n letters of a and b to be created after the c which is not possible.

If $u = a^n$ for some number n with $1 \leq n \leq k$ and $w'_1 = da^p$ for some p with $0 \leq p \leq 2k-n$, then v has to bear also n letters b to be inserted before the letter c and also n letters of a and b to be created after the c which is not possible.

It is not possible that u contains the letter e because d and e are inserted at the same time but d cannot be present in u together with e due to the length of u.

If w'_1 starts with $da^{2k}e$ (if u as a subword of w starts after the letter e), then the word w' does not have the correct form (does not belong to the language L which is a contradiction), since the number of letters a before c is already $2k$ whereas the number of occurrences of b before c or the number of occurrences of ab after c is less (since $|uv| > 0$).

Thus, the only possibility is that $(u,v) = (d,e)$ and $w'_2 = a^{2k}$. We have $2k > m$ and, therefore, $a^{2k} \in J_a^m$. Hence, $J_a^m \cap S \neq \emptyset$ and $J_a^m \subseteq S$. Therefore, the word a^{2k+1} (which belongs to the set J_a^m) also belongs to the selection language S. The language L also contains the word $a^{2k+1}b^{2k+1}c(ab)^{2k+1}$. With the same selection pair (S,C), the word $da^{2k+1}eb^{2k+1}c(ab)^{2k+1}$ could be derived. But this does not belong to the language L. This contradiction shows that our assumption was wrong and that $L \notin \mathscr{IC}(PS)$ holds. □

Lemma 3.2 *Let* $L = \{\ c^nac^mbc^{n+m} \mid n \geq 0, m \geq 0\ \} \cup \{\ c^nbc^na \mid n \geq 0\ \}$. *Then the relation*

$$L \in (\mathscr{IC}(RL_1^V) \cap \mathscr{IC}(RL_2^P) \cap \mathscr{IC}(REG_2^Z)) \setminus (\mathscr{IC}(CIRC) \cup \mathscr{IC}(SUF))$$

holds.

Proof. Let $V = \{\ a,b,c\ \}$. The language L is generated by the contextual grammar

$$G = (V, \{(\{ab,b\}, \{(c,c)\})\}, \{ab,ba\}).$$

Since the selection language is finite with two words, it can be generated by a right-linear grammar with one non-terminal symbol and two rules only. Hence, $L \in \mathscr{IC}(RL_1^V) \cap \mathscr{IC}(RL_2^P)$.

The language L is also generated by the contextual grammar

$$G = (V, \{(V^*\{b\}, \{(c,c)\})\}, \{ab,ba\}).$$

with a combinational selection language only. Every combinational language is accepted by a deterministic finite automaton with two states (see Theorem 2.1 and Figure 1). Hence, $L \in \mathscr{IC}(REG_2^Z)$.

In [10, Lemma 18], it was shown that the language L can neither be generated by a contextual grammar with circular filters nor by one with suffix-closed filters. Hence, $L \notin \mathscr{IC}(CIRC) \cup \mathscr{IC}(SUF)$.
□

Lemma 3.3 *Let $n \geq 1$ be a natural number and let*

$$A_n = \{a_1, \ldots, a_n\}, \quad B_n = \{b_1, \ldots, b_n\}, \quad C_n = \{c_1, \ldots, c_n\}, \quad D_n = \{d_1, \ldots, d_n\},$$

as well as

$$\begin{aligned} V_n &= A_n \cup B_n \cup C_n \cup D_n, \\ P_n &= \{\ (a_i, c_j) \mid 1 \leq i \leq n, 1 \leq j \leq n\ \}, \\ Q_n &= \{\ (b_i, d_j) \mid 1 \leq i \leq n, 1 \leq j \leq n\ \}, \\ G_n &= (V_n, \{(B_n^*, P_n), (C_n^*, Q_n)\}, \{\ a_{i_a} b_{i_b} c_{i_c} d_{i_d} \mid 1 \leq i_x \leq n,\ x \in \{a,b,c,d\}\ \}), \end{aligned}$$

and $L_n = L(G_n)$. Then the relation $L_n \in \mathscr{IC}(MON) \setminus \mathscr{IC}(RL_n^P)$ holds.

Proof. Let $n \geq 1$. The selection languages of G_n are monoidal. Thus, $L_n \in \mathscr{IC}(MON)$.
From [27, Lemma 3.30], we know that $L \notin \mathscr{IC}(RL_n^P)$. □

Lemma 3.4 *Let $V = \{a,b\}$ and $L_n = \{\ a^{p_0} b a^{p_1} b a^{p_2} b \cdots a^{p_n} b a^{p_0} b a^{p_1} b a^{p_2} b \cdots a^{p_n} \mid p_i \geq 1,\ 0 \leq i \leq n\ \}$ for $n \geq 1$. Then*

$$L_n \in (\mathscr{IC}(COMM) \cap \mathscr{IC}(ORD)) \setminus \mathscr{IC}(RL_n^V).$$

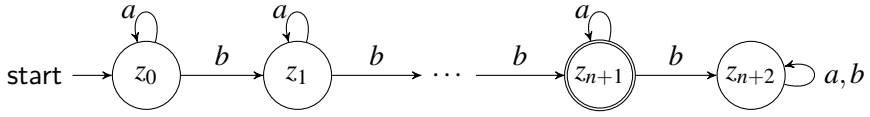
Proof. Let n be a natural number with $n \geq 1$.

The language L_n is generated by the contextual grammar

$$G_n = (V, \{(S_n, \{(a,a)\})\}, \{(ab)^{2n+1} a\})$$

with the selection language $S_n = (\{a\}^* \{b\} \{a\}^*)^{n+1}$. This selection language is commutative; hence, we have $L_n \in \mathscr{IC}(COMM)$.

The selection language is accepted by an automaton whose transition function is shown in the following diagram:



This shows that the automaton is ordered (with $z_0 \prec z_1 \prec \cdots \prec z_{n+2}$, it holds $\delta(z_i, x) \preceq \delta(z_j, x)$ for any two states z_i and z_j with $z_i \prec z_j$ and any $x \in \{a,b\}$). Hence, $L_n \in \mathscr{IC}(ORD)$.
In [27, Lemma 3.29], the relation $L_n \notin \mathscr{IC}(RL_n^V)$ was proved. □

Lemma 3.5 *Let $n \geq 2$ be a natural number, $V_n = \{a_1, a_2, \ldots, a_n\}$ be an alphabet, and L_n be the language $L_n = \{a_1 a_2 \ldots a_n\}^+ \cup V_n^{n-1}$. Then the relation $L_n \in \mathscr{IC}(FIN) \setminus \mathscr{IC}(REG_n^Z)$ holds.*

Proof. Let $n \geq 2$. The language L_n is generated by the contextual grammar

$$G_n = (V_n, \{(\{a_1 a_2 \ldots a_n\}, \{(\lambda, a_1 a_2 \ldots a_n)\})\}, V_n^{n-1} \cup \{a_1 a_2 \ldots a_n\})$$

with a finite selection language only. Thus, $L_n \in \mathscr{IC}(FIN)$.
In [27, Lemma 3.31], it was shown that $L_n \notin \mathscr{IC}(REG_n^Z)$. □

In a similar way, the following result is proved.

Lemma 3.6 *Let $n \geq 2$ be a natural number, $V_n = \{a_1, a_2, \ldots, a_n\}$ be an alphabet, and L_n be the language*

$$L_n = V_n^{\leq n-1} \cup \bigcup_{k \geq 1} V_n^{kn}.$$

Then the relation $L_n \in \mathscr{IC}(COMM) \setminus \mathscr{IC}(REG_n^Z)$ holds.

Proof. Let $n \geq 2$. The language L_n is generated by the contextual grammar

$$G_n = (V_n, \{(V_n^n, \{\ (\lambda, w) \mid w \in V_n^n\ \})\}, V_n^{\leq n-1} \cup V_n^n)$$

with a commutative selection language only. Thus, $L_n \in \mathscr{IC}(COMM)$.

In any contextual grammar generating the language L_n, every context has a length which is divisible by n and can only be added to subwords of words of the language which have a length of at least n. Since every subword of length less than n occurs in the language, the selected subwords must have a length of at least n. This cannot be checked by a deterministic finite automaton with n states only. □

We now prove the relations between the language families of contextual grammars where the selection languages are taken from subregular families of languages which have common structural properties and from families of regular languages defined by restricting the resources needed for generating or accepting them. We start with families which are defined by the number of production rules necessary for generating the selection languages.

Lemma 3.7 *The language families $\mathscr{IC}(RL_1^P)$ and $\mathscr{IC}(FIN)$ coincide.*

Proof. The inclusion $\mathscr{IC}(RL_1^P) \subseteq \mathscr{IC}(FIN)$ follows by Lemma 2.2 from the inclusion $RL_1^P \subseteq FIN$ (see Theorem 2.1 and also Figure 1).

For the converse inclusion, let $m \geq 1$ and

$$G = (V, \{\ (S_i, C_i) \mid 1 \leq i \leq m\ \}, A)$$

be a contextual grammar where all selection languages S_i ($1 \leq i \leq m$) are finite. Then we split up the selection languages into singleton sets and obtain the contextual grammar

$$G' = (V, \{\ (\{w\}, C_i) \mid 1 \leq i \leq m,\ w \in S_i\ \}, A)$$

which generates the same language as G and all selection languages belong to the family RL_1^P. Hence, also the inclusion $\mathscr{IC}(FIN) \subseteq \mathscr{IC}(RL_1^P)$ holds and together we obtain $\mathscr{IC}(FIN) = \mathscr{IC}(RL_1^P)$ □

Lemma 3.8 *The language families $\mathscr{IC}(RL_n^P)$ for $n \geq 2$ are incomparable to the families*

$$\mathscr{IC}(MON),\ \mathscr{IC}(NIL),\ \mathscr{IC}(COMB),\ \mathscr{IC}(DEF),\ \mathscr{IC}(ORD),\ \mathscr{IC}(NC),\ \mathscr{IC}(PS),$$
$$\mathscr{IC}(SUF),\ \mathscr{IC}(COMM),\ \text{and}\ \mathscr{IC}(CIRC).$$

Proof. Due to the inclusion relations stated in Theorem 2.1, depicted in Figure 1, proofs of the following relations are sufficient:

1. $\mathscr{IC}(RL_2^P) \setminus \mathscr{IC}(PS) \neq \emptyset$,
2. $\mathscr{IC}(RL_2^P) \setminus \mathscr{IC}(CIRC) \neq \emptyset$,
3. $\mathscr{IC}(MON) \setminus \mathscr{IC}(RL_n^P) \neq \emptyset$ for every natural number n with $n \geq 2$.

The first relation was proved in Lemma 3.1, the second relation in Lemma 3.2, and the third relation in Lemma 3.3. □

Regarding the families which are defined by the number of states necessary for accepting the selection languages, we obtain the following results.

Lemma 3.9 *The language families $\mathscr{IC}(MON)$ and $\mathscr{IC}(REG_1^Z)$ coincide.*

Proof. This follows from the fact that $REG_1^Z = MON \cup \{\emptyset\}$ and that the empty set has no influence as a selection language. □

Lemma 3.10 *The relation $\mathscr{IC}(COMB) \subset \mathscr{IC}(REG_2^Z)$ holds.*

Proof. From Theorem 2.1 (see Figure 1), we know that $COMB \subset REG_2^Z$. By Lemma 2.2, we obtain that $\mathscr{IC}(COMB) \subseteq \mathscr{IC}(REG_2^Z)$ holds. By Lemma 3.1, this inclusion is proper. □

Lemma 3.11 *Every language family $\mathscr{IC}(REG_n^Z)$ where $n \geq 2$ is incomparable to each of the families*

$$\mathscr{IC}(FIN),\ \mathscr{IC}(NIL),\ \mathscr{IC}(DEF),\ \mathscr{IC}(ORD),\ \mathscr{IC}(NC),\ \text{and}\ \mathscr{IC}(PS).$$

Proof. Due to the inclusion relations stated in Theorem 2.1, depicted in Figure 1, proofs of the following relations are sufficient:

1. $\mathscr{IC}(REG_2^Z) \setminus \mathscr{IC}(PS) \neq \emptyset$,
2. $\mathscr{IC}(FIN) \setminus \mathscr{IC}(REG_n^Z) \neq \emptyset$ for every $n \geq 2$.

The first relation was proved with Lemma 3.1, the second one with Lemma 3.5. □

Lemma 3.12 *Every language family $\mathscr{IC}(REG_n^Z)$ where $n \geq 2$ is incomparable to each of the families $\mathscr{IC}(COMM)$ and $\mathscr{IC}(CIRC)$.*

Proof. Due to the inclusion relations stated in Theorem 2.1, depicted in Figure 1, proofs of the following relations are sufficient:

1. $\mathscr{IC}(REG_2^Z) \setminus \mathscr{IC}(CIRC) \neq \emptyset$,
2. $\mathscr{IC}(COMM) \setminus \mathscr{IC}(REG_n^Z) \neq \emptyset$ for every $n \geq 2$.

The first relation was proved with Lemma 3.2, the second one with Lemma 3.6. □

Regarding the families which are defined by the number of non-terminal symbols necessary for generating the selection languages, we obtain the following results.

Lemma 3.13 *The relation $\mathscr{IC}(DEF) \subset \mathscr{IC}(RL_1^V)$ holds.*

Proof. We first prove the inclusion $\mathscr{IC}(DEF) \subseteq \mathscr{IC}(RL_1^V)$.

Let $n \geq 1$ and

$$G = (V, \{\ (S_i, C_i) \mid 1 \leq i \leq n\ \}, A)$$

be a contextual grammar where every selection language can be represented in the form $S_i = A_i \cup V^* B_i$ with $1 \leq i \leq n$ for finite subsets A_i and B_i of V^*. The same language $L(G)$ is also generated by the contextual grammar

$$G' = (V, \{\ (A_i, C_i) \mid 1 \leq i \leq n\ \} \cup \{\ (V^* B_i, C_i) \mid 1 \leq i \leq n\ \}, A).$$

Every such selection language A_i and V^*B_i for $1 \leq i \leq n$ can be generated by a right-linear grammar with one non-terminal symbol only:

$$G_{A_i} = (\{S\}, V, \{\ S \to w \mid w \in A_i\ \}, S)$$

for generating the language A_i and

$$G_{B_i} = (\{S\}, V, \{\ S \to xS \mid x \in V\ \} \cup \{\ S \to w \mid w \in B_i\ \}, S)$$

for generating the language V^*B_i. Hence, $\mathscr{IC}(DEF) \subseteq \mathscr{IC}(RL_1^V)$.

With Lemma 3.1, it is proved that a language exists in the set $\mathscr{IC}(RL_1^V) \setminus \mathscr{IC}(PS)$. This language is also a witness language for the properness of the inclusion $\mathscr{IC}(DEF) \subset \mathscr{IC}(RL_1^V)$. □

Lemma 3.14 *Every language family $\mathscr{IC}(RL_n^V)$ where $n \geq 1$ is incomparable to the families*

$$\mathscr{IC}(ORD),\ \mathscr{IC}(NC),\ \mathscr{IC}(PS),\ \mathscr{IC}(COMM),\ \text{and}\ \mathscr{IC}(CIRC).$$

Proof. Due to the inclusion relations stated in Theorem 2.1, depicted in Figure 1, proofs of the following relations are sufficient:

1. $\mathscr{IC}(RL_1^V) \setminus \mathscr{IC}(PS) \neq \emptyset$,
2. $\mathscr{IC}(RL_1^V) \setminus \mathscr{IC}(CIRC) \neq \emptyset$,
3. $\mathscr{IC}(COMM) \setminus \mathscr{IC}(RL_n^V) \neq \emptyset$ for every $n \geq 1$,
4. $\mathscr{IC}(ORD) \setminus \mathscr{IC}(RL_n^V) \neq \emptyset$ for every $n \geq 1$.

The first relation is proved in Lemma 3.1, the second in Lemma 3.2, and the other two in Lemma 3.4. □

The following theorem summarizes the results.

Theorem 3.15 *The relations depicted in Figure 3 hold. An arrow from an entry X to an entry Y denotes the proper inclusion $X \subset Y$. If two families are not connected by a directed path then they are not necessarily incomparable.*

If two families X and Y are not connected by a directed path, then X and Y are in most cases incomparable. The only exceptions are the relations of the family $\mathscr{IC}(SUF)$ to the families $\mathscr{IC}(ORD)$ and $\mathscr{IC}(NC)$, to the families $\mathscr{IC}(RL_n^V)$ for $n \geq 1$, to the families $\mathscr{IC}(REG_n^Z)$ for $n \geq 2$ where it is not known whether they are incomparable or whether $\mathscr{IC}(SUF)$ is a subset of the other and the relation of the family $\mathscr{IC}(REG_{n+1}^Z)$ to $\mathscr{IC}(RL_n^V)$ for $n \geq 1$ where it is not known whether they are incomparable or whether $\mathscr{IC}(REG_{n+1}^Z)$ is a subset of $\mathscr{IC}(RL_n^V)$.

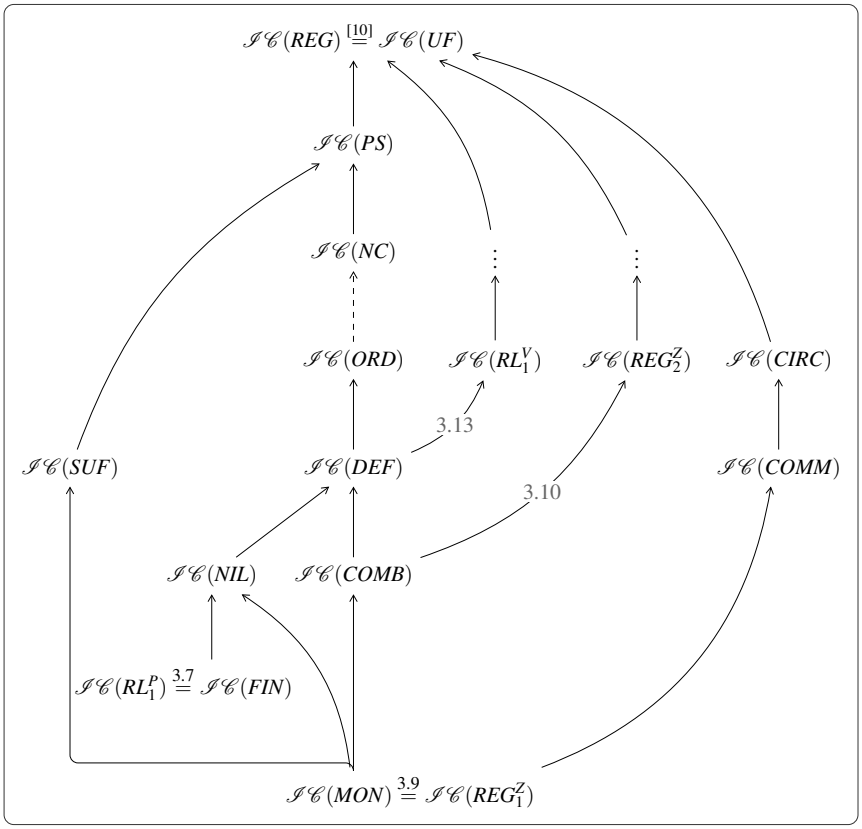


Figure 3: Hierarchy of language families by contextual grammars; an edge label refers to the corresponding lemma (where the relation was not already shown in Figure 2). The incomparabilities were proved in the Lemmas 3.8, 3.11, 3.12 and 3.14.

4 Conclusions and Further Work

In [27], two independent hierarchies have been obtained for each type of contextual grammars, one based on selection languages defined by structural properties, the other one based on resources. In the present paper, these hierarchies have been merged for internal contextual grammars.

Some questions remain open:

- Let $n \geq 1$. Is there a language $L_n \in \mathscr{IC}(SUF) \setminus \mathscr{IC}(RL_n^V)$?
- Let $n \geq 2$. Is there a language $L_n \in \mathscr{IC}(SUF) \setminus \mathscr{IC}(REG_n^Z)$ for $n \geq 2$?

If the first question is answered affirmatively, then these languages L_n satisfy also $L_n \notin \mathscr{IC}(REG_n^Z)$ since $\mathscr{IC}(REG_n^Z) \subset \mathscr{IC}(RL_n^V)$ for $n \geq 1$ (Theorem 2.1, see Figure 1).

If such languages are found, then it is clear that every language family $\mathscr{IC}(RL_n^V)$ for $n \geq 1$ and every languages family $\mathscr{IC}(REG_n^Z)$ for $n \geq 2$ is incomparable to the family $\mathscr{IC}(SUF)$. So far, we only know that $\mathscr{IC}(RL_n^V) \not\subseteq \mathscr{IC}(SUF)$ for $n \geq 1$ and that $\mathscr{IC}(REG_n^Z) \not\subseteq \mathscr{IC}(SUF)$ for $n \geq 2$ (both shown in Lemma 3.1)

Recently, in [5, 11], strictly locally k-testable languages have been investigated as selection languages for contextual grammars. Also for the language families defined by those selection languages, it should be investigated where they are located in the presented hierarchy.

Additionally, other subfamilies of regular languages could be taken into consideration. Recently, in [6, 7], external contextual grammars have been investigated where the selection languages are ideals or codes. This reseach could be extended to internal contextual grammars with ideals or codes as selection languages.

References

[1] J. A. Brzozowski (1962): *Regular Expression Techniques for Sequential Circuits*. Ph.D. thesis, Princeton University, Princeton, NJ, USA.

[2] J. A. Brzozowski, G. Jirásková & C. Zou (2014): *Quotient complexity of closed languages*. *Theory of Computing Systems* 54, pp. 277–292, doi:10.1007/s00224-013-9515-7.

[3] Jürgen Dassow (1979): *On the Circular Closure of Languages*. *Elektronische Informationsverarbeitung und Kybernetik/Journal of Information Processing and Cybernetics* 15(1–2), pp. 87–94.

[4] Jürgen Dassow (2005): *Contextual grammars with subregular choice*. *Fundamenta Informaticae* 64(1–4), pp. 109–118.

[5] Jürgen Dassow (2015): *Contextual languages with strictly locally testable and star free selection languages*. *Analele Universității București* 62, pp. 25–36.

[6] Jürgen Dassow (2018): *Grammars with control by ideals and codes*. *Journal of Automata, Languages and Combinatorics* 23(1–3), pp. 143–164, doi:10.25596/jalc-2018-143.

[7] Jürgen Dassow (2021): *Remarks on external contextual grammars with selection*. *Theoretical Computer Science* 862, pp. 119–129, doi:10.1016/j.tcs.2020.07.028.

[8] Jürgen Dassow, Florin Manea & Bianca Truthe (2011): *On Contextual Grammars with Subregular Selection Languages*. In Markus Holzer, Martin Kutrib & Giovanni Pighizzini, editors: *Descriptional Complexity of Formal Systems – 13th International Workshop, DCFS 2011, Gießen/Limburg, Germany, July 25 – 27, 2011. Proceedings*, *LNCS* 6808, Springer-Verlag, pp. 135–146, doi:10.1007/978-3-642-22600-7_11.

[9] Jürgen Dassow, Florin Manea & Bianca Truthe (2012): *On External Contextual Grammars with Subregular Selection Languages*. *Theoretical Computer Science* 449(1), pp. 64–73, doi:10.1016/j.tcs.2012.04.008.

[10] Jürgen Dassow, Florin Manea & Bianca Truthe (2012): *On Subregular Selection Languages in Internal Contextual Grammars*. *Journal of Automata, Languages, and Combinatorics* 17(2–4), pp. 145–164, doi:10.25596/jalc-2012-145.

[11] Jürgen Dassow & Bianca Truthe (2022): *On the Generative Capacity of Contextual Grammars with Strictly Locally Testable Selection Languages*. In Henning Bordihn, Géza Horváth & György Vaszil, editors: *12th International Workshop on Non-Classical Models of Automata and Applications (NCMA 2022), Debrecen, Hungary, August 26 – 27, 2022. Proceedings*, *EPTCS* 367, Open Publishing Association, pp. 65–80, doi:10.4204/EPTCS.367.5.

[12] F. Gécseg & I. Peak (1972): *Algebraic Theory of Automata*. Academiai Kiado, Budapest.

[13] A. Gill & L. T. Kou (1974): *Multiple-entry finite automata*. *Journal of Computer and System Sciences* 9(1), pp. 1–19, doi:10.1016/S0022-0000(74)80034-6.

[14] Ivan M. Havel (1969): *The theory of regular events II*. *Kybernetika* 5(6), pp. 520–544.

[15] Sorin Istrail (1978): *Gramatici contextuale cu selectiva regulata*. *Stud. Cerc. Mat.* 30, pp. 287–294.

[16] Florin Manea & Bianca Truthe (2012): *On Internal Contextual Grammars with Subregular Selection Languages*. In Martin Kutrib, Nelma Moreira & Rogério Reis, editors: *Descriptional Complexity of Formal Systems – 14th International Workshop, DCFS 2012, Braga, Portugal, July 23 – 25, 2012. Proceedings*, *LNCS* 7386, Springer-Verlag, pp. 222–235, doi:10.1007/978-3-642-31623-4_17.

[17] Solomon Marcus (1969): *Contextual grammars*. *Revue Roum. Math. Pures Appl.* 14, pp. 1525–1534.

[18] Robert McNaughton & Seymour Papert (1971): *Counter-free Automata*. MIT Press, Cambridge, USA.

[19] M. Perles, M. M. Rabin & E. Shamir (1963): *The theory of definite automata*. *IEEE Trans. Electronic Computers* 12, pp. 233–243, doi:10.1109/PGEC.1963.263534.

[20] Gheorghe Păun (1998): *Marcus Contextual Grammars*. Kluwer Publ. House, Doordrecht.

[21] Grzegorz Rozenberg & Arto Salomaa, editors (1997): *Handbook of Formal Languages*. Springer-Verlag, Berlin.

[22] H. J. Shyr (1991): *Free Monoids and Languages*. Hon Min Book Co., Taichung, Taiwan.

[23] H. J. Shyr & G. Thierrin (1974): *Ordered Automata and Associated Languages*. *Tankang Journal of Mathematics* 5(1), pp. 9–20.

[24] H. J. Shyr & G. Thierrin (1974): *Power-Separating Regular Languages*. *Mathematical Systems Theory* 8(1), pp. 90–95, doi:10.1007/BF01761710.

[25] Bianca Truthe (2014): *A Relation Between Definite and Ordered Finite Automata*. In Suna Bensch, Rudolf Freund & Friedrich Otto, editors: *Sixth Workshop on Non-Classical Models of Automata and Applications (NCMA), Kassel, Germany, July 28–29, 2014, Proceedings*, *books@ocg.at* 304, Österreichische Computer Gesellschaft, pp. 235–247.

[26] Bianca Truthe (2018): *Hierarchy of Subregular Language Families*. Technical Report, Justus-Liebig-Universität Giessen, Institut für Informatik, IFIG Research Report 1801.

[27] Bianca Truthe (2021): *Generative Capacity of Contextual Grammars with Subregular Selection Languages*. *Fundamenta Informaticae* 180, pp. 1–28, doi:10.3233/FI-2021-2037.

[28] Bianca Truthe (2023): *Merging two Hierarchies of External Contextual Grammars with Subregular Selection*. In Henning Bordihn, Nicholas Tran & György Vaszil, editors: *25th International Conference on Descriptional Complexity of Formal Systems, DCFS 2023, Potsdam, Germany, July 4–6, 2023, Proceedings*, *LNCS* 13918, Springer, pp. 169–180, doi:10.1007/978-3-031-34326-1_13.

Ordered Context-Free Grammars Revisited

Brink van der Merwe
Department of Computer Science
Stellenbosch University
Stellenbosch, South Africa
abvdm@cs.sun.ac.za

We continue our study of ordered context-free grammars, a grammar formalism that places an order on the parse trees produced by the corresponding context-free grammar. In particular, we simplify our previous definition of a derivation of a string for a given ordered context-free grammar, and present a parsing algorithm, using shared packed parse forests, with time complexity $O(n^4)$, where n is the length of the input string being parsed.

Keywords— Ordered context-free grammars, Unambiguous grammar formalisms, Shared packed parse forests

1 Introduction

Ordered context-free grammars (oCFGs), a grammar formalism introduced in [11], provides an alternative to parsing expression grammars (PEGs), when requiring an unambiguous grammar formalism. This formalism has much easier to understand matching semantics compared to PEGs, but this comes at the price of much worse parsing time complexity. Indeed, the complexity is $O(n^4)$ compared to linear, where n is the length of the input string being parsed. It should be noted that this is not worse than the adaptive LL($*$) algorithm, used in the popular parser generator ANTLR [12]. Ordered context-free grammars are unambiguous, since we select the least parse tree for a given input string (if possible), based on the order induced on parse trees by the oCFG formalism.

The matching semantics of oCFGs are more intuitive than PEGs, since an oCFG matches exactly the same string language as the corresponding context-free grammar (CFG), in contrast to PEGs. We obtain PEGs from context-free grammars by replacing the choice operator, typically denoted by the pipe character ' | ', by an ordered choice operator, i.e. the choice operator becomes non-commutative. The semantics of the ordered choice operator is such that if the first alternative succeeds locally, i.e. if the ordered choice lets the current nonterminal consume some substring starting at the current position without regard for the overall match, the second alternative is never attempted. Specifying when a rule succeeds locally should be stated with more care – more on this later in the introduction. The oCFG formalism also makes use of an ordered choice operator, but the emphasis is on overall instead of local success. Despite the popularity of PEGs as unambiguous grammar formalism, there are some downsides, for example, proving that a given PEG matches an intended string language is often complicated. As pointed out in [9], the influence of PEGs can be illustrated by the fact that despite having been introduced only twenty years ago, the number of PEG-based parser generators exceeds the number of parser generators based on any other parsing method.

The unexpected behaviour of PEGs can for example be seen when considering the PEG with $S \to aSa \,/\, a$ as the only production, describing the regular language $\{a^{2n+1} \mid n \geq 0\}$ when replacing the PEG with the corresponding CFG. Normally, PEGs use the symbol '$\leftarrow$' in productions, and not '$\to$', although we will deviate from this convention. Next, we explain (informally) why this PEG does not match a^5, while matching, for example, a^3 and a^7. We also discuss this example more formally in Example 2.

Rudolf Freund, Benedek Nagy (Eds.): 13th International Workshop on
Non-Classical Models of Automata and Applications (NCMA 2023)
EPTCS 388, 2023, pp. 140–153, doi:10.4204/EPTCS.388.13

Given that $S \to aSa \mid a$ is an example of an unambiguous CFG, we note that the PEG formalism not only makes a CFG unambiguous, but might also reduce the set of strings being matched. Let's also consider $S \to aSa \mid a$ as an oCFG. In both the PEG and oCFG case, derivations begin by applying the rule $S \to aSa$ twice, but the PEG then applies $S \to aSa$ a *third time,* as it considers this rule as being "locally successful", since the right-hand side of the rule consumes the 3rd to the 5th 'a' (after replacing the S in aSa with a). But this will cause the 2nd application of $S \to aSa$ to fail. In comparison to PEGs, the oCFG would select $S \to a$ as the 3rd rule to apply. This ensures that applying $S \to aSa$ is successful as the 2nd derivation step, and in this way we obtain a successful oCFG derivation of a^5. That is, PEGs select the first locally successful rule, whereas with oCFGs, the first rule which enables overall derivation success, is selected. As stated before, applying a rule r to rewrite a nonterminal A, is regarded as locally successful, if by applying r, and keeping on rewriting the nonterminals produced by r, we obtain a string of terminals which is a prefix of the remainder of the input string. But, in PEGs, the selection later of locally successful rules in a derivation, applied to nonterminals produced by an earlier rule application step, has precedence over the success of the earlier selected rule. Thus, a selected rule might fail in PEGs, since local success preference is given to later applied rules.

The non-commutativity of the choice operator in PEGs can be seen when changing the above example to $S \to a \mid aSa$, and noting that in this case only the input string a is matched. In oCFGs, the operator '|' is also non-commutative when considering the order on the parse trees produced by an oCFG, but not when only considering the strings being matched. Of course, given that $S \to aSa \mid a$ is an unambiguous CFG, it makes no difference whether this example grammar is considered as a CFG or as an oCFG.

The oCFG formalism is a natural way to generalize Perl-compatible regular expression (PCRE) matching, to context-free parsing. PCRE matching semantics is used in almost all regular expression matching libraries. See for example [3] for a discussion on how real-world regex matching semantics are deeply intertwined with a depth-first backtracking parsing technique. In both PCRE regex matching and oCFGs, ambiguity is removed in perhaps the most natural generic way, i.e. when having multiple transition or rule choices, we place and preference on which one should be used, by ordering transitions and rules respectively.

When considering regular expressions, PEGs correspond to the atomic operator (see [4]), as illustrated in the following example. Consider the regular expression $r := a^*a$, which we translate into a CFG G_r with productions $S \to Aa$ and $A \to aA \mid \varepsilon$. When using the atomic operator in r to obtain r', with $r' := (\rhd a^*)a$, we obtain the corresponding PEG $G_{r'}$ with productions $S \to Aa$ and $A \to aA \mathbin{/} \varepsilon$. In this case, $(\rhd a^*)$ consumes locally as many characters as possible, and thus r' and $G'_{r'}$ describe the empty language. In both regexes and grammars, atomic operators and parsing expressions grammars provides respectively improved efficiency in matching or parsing, but at the cost of often difficult to understand or unexpected matching behaviour.

In [12], it is pointed out that the parser generator ANTLR, a top-down parser generator developed by Terence Parr, uses the order in which rules are specified, as one way of resolving ambiguities. The parser generator YACC (see [1]) also uses the order of rules to resolve reduce-reduce conflicts. This observation provides additional motivation for why the oCFG formalism is of interest.

Strictly speaking, we should rather refer to oCFGs, as ordered parse tree context-free grammars, given, as will be shown in the next section, the order of rules in an oCFG is used to obtain an order on the parse trees. The terminology "ordered context-free grammars" is also used in the regulated rewriting community for a related formalism (see for example [6, 8]). In this related formalism, a partial order is placed on the grammar rules, and a rule is not allowed to be applied to a sentential form if a larger rule is also applicable to the sentential form. In contrast to PEGs (or oCFGs), this regulated rewriting formalism determines if a rule is applicable to a sentential form (and that there are no larger applicable rules), and

not if a rule is both applicable and succeeds locally (or respectively, guantees overall success).

In this paper, we simplify the notion of an oCFG derivation in Section 4, compared to [11], by not explicitly modelling backtracking. In Section 5, we also consider the complexity of parsing oCFGs, a question not considered before. Results from [11] required to follow the exposition in this paper, are stated without proof. The outline of this paper is as follows. The next two sections provide definitions and elementary results on oCFGs and on PEGs. Then, oCFG derivations are considered, after which we discuss oCFG parsing by using shared packed parse forests. Finally, we present our conclusions and a discussion on envisioned future work.

2 Definitions and elementary properties of oCFG

Next, we define oCFGs. In an oCFG, we order all rules with the same nonterminal on the left-hand side, and then number each of these collections of rules, from one onward. We consider only the leftmost derivations, and associate a list of integers with each derivation, based on rules used in the derivation, from left to right. Derivations (and parse trees) can thus be compared and ordered, using the lexicographic ordering of the list of integers associated with a derivation. We also consider a subclass of oCFGs, where for each string w in the language of the grammar, there is a least derivation (and thus parse tree) for w. Thus, oCFGs extend CFGs in such a way that the strings accepted, and their corresponding parse trees are the same, but we also have an order on the parse trees.

In the following definition, we define trees, which will mostly be used as parse trees in this paper.

Definition 1. The set of *ordered, rooted and ranked trees,* over a finite ranked alphabet $\Gamma = \cup_{i=0}^{\infty}\Gamma_i$, denoted by $\mathcal{T}_\Gamma$, where Γ_i is the set of alphabet symbols of rank i, is defined inductively as follows:

- if $a \in \Gamma_0$, then $a \in \mathcal{T}_\Gamma$;
- if $a \in \Gamma_k$ and $t_i \in \mathcal{T}_\Gamma$ for $1 \leq i \leq k$, then $a[t_1, \ldots, t_k] \in \mathcal{T}_\Gamma$.

The height of $t \in \mathcal{T}_\Gamma$, denoted $ht(t)$, is defined inductively as follows. We let $ht(t) = 0$ if $t = a \in \Gamma_0$, otherwise, if $t = a[t_1, \ldots, t_k]$, then $ht(t) = 1 + max(ht(t_1), \ldots, ht(t_k))$.

Next, we define trees referred to as contexts. Using contexts, we can construct a larger tree by substituting the special symbol $\Box$, by another tree.

Definition 2. Assume $\Box$ is a symbol of rank 0 that is not in the ranked alphabet Γ. Denote by $\mathcal{C}_\Gamma$ the set of trees over the ranked alphabet $\Gamma \cup \{\Box\}$, where each tree has precisely one leaf node labelled by $\Box$. A tree in $\mathcal{C}_\Gamma$ is referred to as a *context*.

For $t \in \mathcal{C}_\Gamma$ and $t' \in \mathcal{C}_\Gamma \cup \mathcal{T}_\Gamma$, denote by $t[\![t']\!] \in \mathcal{C}_\Gamma \cup \mathcal{T}_\Gamma$ the tree obtained by replacing the instance of $\Box$ in t, by t'.

Now, we are ready to define ordered context-free grammars, which at this stage, looks the same as CFGs. The way in which we extend CFGs to obtain oCFGs, will become clear once explain how to order parse trees.

Definition 3. An *ordered context-free grammar* G is a tuple (N, Σ, P, S), where:

(i) N is a finite set of nonterminals;

(ii) Σ the input alphabet;

(iii) P is the production function and for $A \in N$, we have $P(A) = (r_1^A, \ldots, r_{n_A}^A)$, with $r_i^A \in (N \cup \Sigma)^*$;

(iv) $S \in N$ is the start nonterminal.

When $P(A) = (r_1^A, \ldots, r_{n_A}^A)$, we also use the notation $A \to r_1^A \mid \cdots \mid r_{n_A}^A$. The order of the r_i^A, in $(r_1^A, \ldots, r_{n_A}^A)$, will play a role in the order of the parse trees, defined later. In results where order is not important, we will mostly use the terminology CFG, instead of oCFG.

We refer to $A \to r_1^A \mid \cdots \mid r_{n_A}^A$ as a production, and to $A \to r_i^A$, for some $1 \leq i \leq n_A$, as a rule. As is usual in CFGs, we say that for $u, v \in (N \cup \Sigma)^*$, that u directly yields v, written as $u \Rightarrow v$, if $u = u_1 A u_2$ and $v = u_1 r_i^A u_2$, for some $1 \leq i \leq n_A$. Also, we denote by $\Rightarrow^*$ the reflexive transitive closure of $\Rightarrow$, and by $\Rightarrow^+$ the transitive closure of $\Rightarrow$. If $S \Rightarrow^* u$, for $u \in (N \cup \Sigma)^*$, we refer to u as a sentential form.

A ranked alphabet Γ_G (which we will use in parse trees) is associated with an oCFG G as follows. Denote by $|v|$ the length of a string v, with the length of the empty string ε taken to be 0. We let $\Sigma \cup \{\varepsilon\}$ be the elements of rank 0 in Γ_G, since these will label the leafs of the parse trees. If $P(A) = (r_1^A, \ldots, r_{n_A}^A)$, then define A_i, for $1 \leq i \leq n_A$, to be a symbol of rank $\max\{1, |r_i^A|\}$ in Γ_G. We use the symbols with subscripts, A_i, in parse trees to encode the production choice $A \to r_i^A$. Since r_i^A might be equal to ε, we take the rank of A_i to be $\max\{1, |r_i^A|\}$, since a node in a parse tree labelled by A_i, will still have a child labelled by ε when $r_i^A = \varepsilon$.

For a tree t, the notation $y(t)$ is used for the yield of t, i.e. the string of non-ε leaf symbols in t, considered left to right. Thus, to obtain $y(t)$, we delete ε and all symbols of rank greater than zero and also '[', ']' and ',' in t. In the special case where all leaf symbols are ε, we define $y(t)$ to be ε as well.

Definition 4. For an oCFG G and string $w \in \Sigma^*$, we define the set of *parse trees of* w, denoted by $\mathcal{P}_G(w)$, as all trees over the ranked alphabet Γ_G, satisfying the following criteria:

(i) The root is labelled by some S_i, $1 \leq i \leq n_S$, where S is the start nonterminal of G;

(ii) $y(t) = w$;

(iii) The children of a node labelled by A_i, ignoring subscripts of nonterminals, are labelled, in order, by the symbols in r_i^A. As a special case, when $|r_i^A| = 0$, a node labelled by A_i will have a single child leaf labelled by ε.

The *string language defined by* G, denoted by $\mathcal{L}(G)$, is the set of strings w for which $\mathcal{P}_G(w) \neq \varnothing$.

By $\mathcal{L}_\mathcal{T}(G)$ we denote the set of parse trees of G, which is the set $\bigcup_{w \in \Sigma^*} \mathcal{P}_G(w)$. We modified the usual definition of parse trees to make it possible to directly read off the productions used to obtain the parse tree, by considering the indices of the nonterminal labels used in the parse tree. More precisely, when doing a pre-order traversal of the non-leaf nodes of a parse tree, the integer subscripts of the nonterminals describe uniquely (with the subscript of a nonterminal indicating which rule choice, from a given production, was made for a given nonterminal) the productions used in a left-most derivation to produce the respective parse tree. Since we know that derivations start with the initial nonterminal S, it is not required to know both the nonterminals and their respective indices to deduce the productions used, i.e. the indices are sufficient.

For $t \in \mathcal{L}_\mathcal{T}(G)$, let $n(t)$ denote the sequence of integers obtained by replacing all symbols A_i in the representation of t, as used in Definition 1, by i, and deleting all other symbols (i.e. '[', ']', ',' and terminal leaves) in the representation of t.

Definition 5 (Total order on parse trees)**.** A total order $\prec_G$ is defined on $\mathcal{L}_\mathcal{T}(G)$ by letting $t_1 \prec_G t_2$ when $n(t_1)$ is smaller than $n(t_2)$ lexicographically.

When having unit or empty rules, oCFGs might not have well-ordered sets of parse trees for each given input string, and since this is relevant to ensure that oCFGs are unambiguous grammar formalisms, we focus on the following two classes of oCFGs.

Definition 6. Let G be any oCFG.

- We define G to have *least parse trees* or simply *least trees*, if for all strings w, $\mathcal{P}_G(w)$ is either empty or has a least parse tree.
- We define G to be *well-ordered*, if for all strings w, the set of trees $\mathcal{P}_G(w)$ is well-ordered (i.e. every subset of $\mathcal{P}_G(w)$ has a least parse tree).

An oCFG having least trees is sufficient to turn oCFGs into an unambiguous grammar formalism by for each w selecting the least tree in $\mathcal{P}_G(w)$. The well-ordered property is stronger, but it is decidable as shown in Theorem 1, in contrast to determining if an oCFG has least trees, which is not decidable (see [11]).

We can use the order $\prec_G$ to define a *filter* on the set of parse trees of the oCFG G (see [7] for more on using filters for disambiguation). For a set A, denote by $\Pi(A)$ the power set of A. Then a function $\mathcal{F} : \Pi(\mathcal{L}_\mathcal{T}(G)) \to \Pi(\mathcal{L}_\mathcal{T}(G))$ is a filter, if for $\Phi \in \Pi(\mathcal{L}_\mathcal{T}(G))$, we have $\mathcal{F}(\Phi) \subseteq \Phi$. We define the filter $\mathcal{F}_G$ such that $\mathcal{F}_G(\Phi)$ consists of the trees $t \in \Phi$, such that for no tree $t' \in \Phi$ (with $t' \neq t$), we have $t' \prec_G t$. Then G having least trees is equivalent to the filter $\mathcal{F}_G$ being *complete*, where a filter is complete if it selects one tree from each non-empty set $\mathcal{P}_G(w)$.

Instead of using the positive natural numbers, i.e. a totally ordered set, to index each of the rules in a given production, from 1 onwards, we can index the rules by a partially ordered set. These indices can then be used in a lexicographic way, to define a partial order on parse trees. In this way, one can support ordered and unordered choice between rules in a production. Again, we obtain a filter on the set of parse trees, as before, but not necessarily a complete filter. More than one filter can of course be used to remove ambiguity, for example in the LR parser YACC, one could have shift-reduce and reduce-reduce conflicts, where shift-reduce conflicts are resolved by preferring shift over reduce, and only reduce-reduce conflicts are resolved by using the order in which rules are specified.

Next, we provide a sufficient condition for a grammar G to be well-ordered. In particular, we provide a necessary and sufficient condition so that all strings w will have finitely many parse trees. We in fact give a necessary and sufficient condition for the opposite, i.e. a condition to ensure that some strings will have infinitely many parse trees, which can then be negated. We assume all nonterminals in G are useful. We define a nonterminal A in G to be *useful* if a sentential form can be derived from S containing A, and if a string of terminal symbols can be derived when starting from A. We say a grammar G is *cyclic* if for some nonterminal A in G, we have $A \Rightarrow^+ A$, with $\Rightarrow^+$ being the transitive closure of $\Rightarrow$. Being cyclic is a necessary condition for some strings to have infinitely many parse trees, and conversely, if each nonterminal in G is useful, then G being cyclic is sufficient for some strings w to have infinitely many parse trees. We thus obtain the following result, generalizing Lemma 1 in [11]. If in an oCFG G we have $A_1 \Rightarrow A_2 \Rightarrow \ldots \Rightarrow A_n$, for nonterminals $A_1, \ldots, A_n$ where $A_1 = A_n$, we say G has a *cycle of unit rules*.

Lemma 1. *Let G be a CFG with all nonterminals being useful.*

1. *If G is not cyclic, then $\mathcal{P}_G(w)$ is finite for all $w \in \Sigma^*$.*
2. *If G is cyclic, then some strings will have infinitely many parse trees.*
3. *If G neither has any ε-rules nor cycles of unit rules, then it is not cyclic.*
4. *If G neither has any ε-rules nor cycles of unit rules, then it is well-ordered.*

Proof. Observe that the only way a given string can have parse trees of unbounded size (and thus infinitely many parse trees) is if G is cyclic. Also, conversely, if all nonterminals are useful, then when we have nonterminals involved in cycles, these nonterminals must appear in some parse trees, and we

can repeat these cycles as many times as we want in parse trees, without changing the strings being parsed. From these observations we obtain (1) and (2). Statement (3) follows from the definition of a grammar being cyclic, and (4) follows from (1), (3), and the observation that finite ordered sets are in fact well-ordered. □

The previous lemma implies that an oCFG in Chomsky normal form is well-ordered. Thus, the class of string languages recognized by well-ordered oCFGs, or oCFGs with least parse trees, is equal to the class of context-free languages.

Example 1. In this example, we give a well-ordered oCFG for arithmetic expressions, with parenthesis used as usual to indicate precedence. It is also considered how an equivalent grammar could be specified in the popular parser generator ANTLR (see [13]). We allow addition ($+$), subtraction ($-$), multiplication ($*$), division ($\div$) and exponentiation ($\hat{}$), and the oCFG is constructed in a way to indicate precedence and associativity of these operators in the parse trees. Left associativity (for $+, -, *, \div$) is encoded as $S \to SPS \mid x$, $P \to + \mid -$, and $S \to STS \mid x$, $T \to * \mid \div$, and right associativity (for $\hat{}$) as $S \to x \mid S\hat{}S$. To reflect precedence in the parse trees, operators with lower precedence are specified first. Putting these observations together, we obtain the following oCFG:

$$S \to SPS \mid STS \mid x \mid (S) \mid S\hat{}S, \;\; P \to + \mid -, \;\; T \to * \mid \div$$

ANTLR can handle (only) direct left recursion by making use of grammar rewriting, and will by default assume that operators are left associative, unless specified otherwise. In contrast to oCFGs, the choice between left and right associativity can not be enforced by making use of the order in which rules are specified, and the order of the placement of a rule having only a terminal (or terminals) in the right-hand side (for example $S \to x$), has no influence on the parse tree produced. Also, ANTLR assumes that rules are specified in the reverse order as used in oCFGs. Thus, the ANTLR equivalent of this grammar will be:

$$S \to <\text{assoc=right}> S\hat{}S \mid (S) \mid STS \mid SPS \mid x\,, \;\; P \to - \mid +\,, \;\; T \to \div \mid *$$

◇

Observation 1. *The arithmetic operator oCFG in Example 1 does produce the correct (to be defined in the motivation below) least parse trees, but no grammar with single nonterminal does. More broadly, having various required combinations of precedence and associativity will still require significant grammar rewriting to produce a correct abstract syntax tree (AST).*

Motivation. Intuitively, we are seeking grammars which produce least trees which do not *misrepresent* the priority and associativity of the operators. More precisely, when replacing the rule $S \to (S)$ with $S \to y$, and keeping the other rules as is, we want this new oCFG to produce parse trees reflecting the correct priority and associativity of operators. When comparing the oCFG without the rule $S \to y$, with the new oCFG having this rule, we regard the terminal y in the new oCFG as representing recursively (note, parenthesized subexpressions might themselves contain more parenthesized subexpressions) the parse tree of a parenthesized expression (when considering smallest parse trees). Also, in the new oCFG, we convert the parse trees to ASTs, by replacing $S[SP[+]S]$ with $+[SS]$, and similarly for $-, *, \div$, and repeating this replacement on the two inner S's in $+[SS]$, and also replacing $S[x]$ with x and $S[y]$ by y. Also, we replace the y's inductively by the ASTs of the parenthesized subexpressions they represent. In these ASTs we now do not allow $+$ or $-$ as the right child of a $+$ or $-$ node, and similar for $*$ and $\div$. We also do not allow $\hat{}$ as a left child of a node labelled by $\hat{}$. Additionally, we do not want $+$ or $-$ nodes below $*$, $\div$ or $\hat{}$ nodes in the AST, and similarly for $*$ and $\div$ nodes below $\hat{}$ nodes.

The grammar in Example 1 can be shown to be correct by induction. Observe that a least tree will never contain the subtree pattern $S_1[\alpha,\beta,S_1[\gamma_1,\gamma_2,\gamma_3]]$, for any subtrees $\alpha,\beta,\gamma_1,\gamma_2,\gamma_3$, as the tree $S_1[S_1[\alpha,\beta,\gamma_1],\gamma_2,\gamma_3]$ will necessarily be smaller. This establishes the left-associativity of addition and subtraction, and correct associativity for multiplication, division and exponentiation can be shown similarly. Precedence is obtained by noting that rules for lower priority operators are specified first, and this ensures that they then appear higher up in the parse trees and ASTs.

For the second part, observe the role P and T play in the grammar: they make it possible for operators to have the same precedence. That is, $x+x-x+x$ should be parsed as $((x+x)-x)+x$, treating + and − as interchangeable from a syntactic structure perspective. Simply inlining the operators, as in $S \to S+S \mid S-S \mid \cdots \mid x \mid \cdots$, does not work, as $x+x-x+x$ would produce a least tree describing $(x+(x-x))+x$. Reversing + and −, similarly, gives an incorrect tree for $x-x+x-x$. Although this is not the only grammar rewriting to consider, we will not provide exhaustively all arguments required. ◇

Observation 2. *From the last paragraph in the motivation of the previous observation, we see that one needs to be cautious when applying some otherwise natural-seeming grammar rewriting. Specifically, replacing $X \to \gamma Y \delta$ and $Y \to \alpha \mid \beta$, by $X \to \gamma\alpha\delta \mid \gamma\beta\delta$, with $\alpha,\beta \in \Sigma^*$, might not preserve the ordering. More precisely, it is not the case that when taking the smallest parse trees when using the original grammar $X \to \gamma Y \delta$, that one can now replace $Y[\alpha]$ and $Y[\beta]$, by α and β respectively, and then obtain the smallest parse trees when using the grammar $X \to \gamma\alpha\delta \mid \gamma\beta\delta$.*

The next theorem also appears as Theorem 2 in [11], but the proof that follows is significantly more readable and provides more insight, and also specifies the time complexity of deciding if an oCFG is well-ordered. One can regard the argument in the proof as analysing the potential cycles that might appear in the shared parse forests of input strings. If there are no cycles in the parse forest of an input string, then there are only finitely many parse trees for the given string, but if the parse forest contains a cycle that creates smaller trees when followed, there will be an infinite set of decreasing parse trees. Shared packed parse forests are defined and used in Section 5, but the proof of the following theorem can be followed without any knowledge about parse forests.

Theorem 1. *It is decidable, in time $\mathcal{O}(p|N|)$, where p is the sum of the lengths of right-hand sides of the productions in P, whether an oCFG $G = (N,\Sigma,P,S)$ is well-ordered.*

Proof. Since we can determine in time $\mathcal{O}(p|N|)$ which nonterminals are useful, and then discard rules involving these, we may assume that all nonterminals in G are useful. Recall, we refer to a nonterminal in G as being cyclic if $A \Rightarrow^+ A$. Also, we define a rule $A \to r$ to be cyclic if $A \Rightarrow r \Rightarrow^* A$. Now, observe that G is well-ordered if and only if all cyclic rules have the highest possible index (i.e. appear last) in the production in which they occur, i.e. if $A \to r_1^A \mid \ldots \mid r_{n_A}^A$, then there is at most one possible cyclic rule amongst the rules $A \to r_i^A$, and if there is one, it is the rule $A \to r_{n_A}^A$. To see this, first note that if there are no cyclic rules, then G is well-ordered, since then all strings will have only finitely many parse trees. Also, if all cyclic rules appear last, i.e. as $A \to r_{n_A}^A$, then smaller trees are obtained when removing these cycles, and there are only finitely many parse trees, when not using cycles. Next, note that if we have a cyclic rule $A \to r_i^A$, with $i < n_A$, then G is not well-ordered. This follows from a pumping argument: observe that some parse tree containing a node labelled A_{n_A} (at least one exists as A is useful) can in that case be modified into a smaller (under $\prec_G$) parse tree by instead applying the cyclic rule $A \to r_i^A$ in that position, rather than using a rule from the production for A with a larger index. We then use the cyclic derivation to produce a new A_{n_A} lower down in the tree. Iterating this process gives rise to an infinite sequence of smaller trees, violating well-orderedness.

Thus, we can decide whether G is well-ordered by:

(i) Computing the nullable nonterminals; the nonterminals in the smallest set $M \subseteq N^*$, such that $A \in M$ if and only if there is a rule $A \to r$ with $r \in M^*$ (i.e. the Kleene closure of M), and as base case to this inductive definition, we use $M = \varnothing$ and $\varepsilon \in M^*$;

(ii) Computing the set of cyclic rules; search rules participating in cycles in the graph induced by having an edge from $A \in N$ to $B \in N$ if there is a rule $A \to r$ with $r = r'Br''$ for $r', r'' \in M^*$;

(iii) Checking that cyclic rules only occur last in their respective productions.

Suitably implemented, each of these three steps can be done in time $\mathcal{O}(p|N|)$, where p is the sum of the lengths of right-hand sides in P. □

We conclude this section by providing a bound on the length of derivations producing least oCFG trees, assuming no ε-rules. First, we recall a related result for CFGs.

Theorem 2 (Thm. 1 in [15])**.** *For a CFG $G = (N, \Sigma, P, S)$ with no ε-rules, the length of a* shortest *CFG derivation for $w \in \mathcal{L}(G)$ is at most $(2|w| - 1)|N|$.*

Corollary 1. *Let G be an oCFG without ε-rules. Then the bound in Theorem 2 also holds for a CFG derivation of a* least tree *in G (if a least tree exists for the string w).*

Proof. Refer to the proof in [15], and observe that the bound is achieved by eliminating cycles. To see that the result also applies to *all* oCFGs with no ε-rules, observe that if a least parse tree exists, it cannot "contain" a cycle. That is, the least parse tree cannot be such that $t = c[\![c'[\![c'']\!]]\!]$, where (i) $c' \neq \Box$, (ii) c' and c'' have the same root label, and (iii) $c[\![c'']\!]$ is also a parse tree for the same string, since then, either $c[\![c'']\!]$ or $c[\![c'[\![c'[\![c'']\!]]\!]]\!]$ must be smaller. If $c[\![c'[\![c'[\![c'']\!]]\!]]\!]$ is smaller, then we can keep on repeating the context c', and in this way, each time obtain a smaller tree. □

Remark 1. If we allow ε-rules in the previous theorem and corollary in the grammar G, then we need to replace the length of the derivation by the height of a parse tree obtained from a shortest derivation, and also replace the bound $(2|w| - 1)|N|$ by:

$$\max\{(2|w| - 1)|N| + |N|, |N|\} = \max\{(2|w||N|, |N|\} \tag{1}$$

To see this, first note that we may assume that we consider parse trees obtained from leftmost derivations of a CFG, not having any cycles in the derivation. Also, it is enough to obtain a result similar to Theorem 2, since from this theorem, we obtain the corresponding corollary. Next, note that a nonterminal is not repeated in any node to leaf path in a parse tree (obtained from a shortest derivation), from a nonterminal deriving ε. Thus, in particular, the bound given in (1) holds when $w = \varepsilon$. Next, let G' be the grammar obtained from G by applying ε-rule removal (to G) in the standard way, i.e. we replace a rule of the form $A \to r$ by all possible rules $A \to r'$, where r' is obtained from r by deleting some (or none) of the nonterminals in r from which ε can derived (and we also remove all ε-rules). Now, consider a parse tree t for a string $w \neq \varepsilon$, when using G, and remove from t all subtrees deriving ε, to obtain a tree t'. Thus, t' is a parse tree for w when using G'. Now use the previous theorem on G' and w (note G and G' have the same number of nonterminals). We obtain the bound in (1) by noting that the length bound on a derivation (in Theorem 2 applied to G') is a height bound on the corresponding parse tree t' (which gives us the height bound $(2|w| - 1)|N|$ on t'), and then we add back the subtrees deriving ε to t' to obtain t. Thus, we obtain the height bound $(2|w| - 1)|N| + |N|$ for t by adding $|N|$ to the height bound for t'. ⋄

3 Parsing expression grammars

In this section, we formally introduce parsing expression grammars, following [5], but restricting what we allow as parsing expressions, and also assuming that the nonterminal S is the starting expression, instead of making use of a general parsing expression as starting expression.

Definition 7 (Parsing expressions)**.** A *parsing expression* is a string of the form $e_1/e_2/\ldots/e_n$, with $n \geq 1$, and $e_i \in (N \cup \Sigma)^*$, where N and Σ are finite sets of nonterminal and terminal symbols respectively.

We refer to "/" as the prioritized choice operator. The set of parsing expressions over N and Σ is denoted by $\mathcal{PE}(N,\Sigma)$.

Definition 8 (Parsing expression grammars)**.** A *parsing expression grammar* (PEG) is a tuple $G = (N,\Sigma,P,S)$, where N and Σ are finite sets of nonterminal and terminal symbols respectively, $P(A) = (e_1^A,\ldots,e_{n_A}^A)$, with $e_i^A \in (N \cup \Sigma)^*$, is the production function, and S the starting expression.

We write $A \to e_1^A/\ldots/e_{n_A}^A$, if $P(A) = (e_1^A,\ldots,e_{n_A}^A)$, i.e. we interpret P as being a function from N to $\mathcal{PE}(N,\Sigma)$. Note, we do not use the typical convention for PEGs, where $A \leftarrow e_1^A/\ldots/e_{n_A}^A$ denotes $P(A) = (e_1^A,\ldots,e_{n_A}^A)$. As in the case of oCFGs, if we have $A \to e_1^A/\ldots/e_n^A$, we refer to $A \to e_i^A$ and $A \to e_1^A/\ldots/e_n^A$ as a rule and production respectively.

Definition 9 (Matching semantics of PEGs)**.** For a PEG $G = (N,\Sigma,R,S)$, we define a function $\rightsquigarrow_G$: $\mathcal{PE}(G) \times \Sigma^* \to \Sigma^* \cup \{f\}$, where $f \notin (N \cup \Sigma)$ denotes failure ($\rightsquigarrow_G$ will also be used as an infix operator). If $(e,x) \rightsquigarrow_G y$, with $y \in \Sigma^*$, then parsing succeeds by parsing the prefix y of x, while if $(e,x) \rightsquigarrow_G f$, then parsing fails. For $a,b \in \Sigma, a \neq b$, with $e,e_1,e_2 \in \mathcal{PE}(G)$, and $x,x_1,x_2,y \in \Sigma^*$, and $o \in \Sigma^* \cup \{f\}$, we define $\rightsquigarrow_G$ inductively as follows.

- Empty rule: $(\varepsilon,x) \rightsquigarrow_G \varepsilon$;
- Terminal success: $(a,ax) \rightsquigarrow_G a$;
- Terminal failure: $(a,bx) \rightsquigarrow_G f$;
- Nonterminal rule: if $A \to e$ and $(e,x) \rightsquigarrow_G o$, then $(A,x) \rightsquigarrow_G o$;
- Sequence success: if $(e_1,x_1x_2y) \rightsquigarrow_G x_1$ and $(e_2,x_2y) \rightsquigarrow_G x_2$, then $(e_1e_2,x_1x_2y) \rightsquigarrow_G x_1x_2$;
- Sequence failure: if $(e_1,x_1x_2) \rightsquigarrow_G f$, or $(e_1,x_1x_2) \rightsquigarrow_G x_1$ and $(e_2,x_2) \rightsquigarrow_G f$, then $(e_1e_2,x_1x_2) \rightsquigarrow_G f$;
- Alternation case 1: if $(e_1,xy) \rightsquigarrow_G x$, then $(e_1/e_2,xy) \rightsquigarrow_G x$;
- Alternation case 2: if $(e_1,x) \rightsquigarrow_G f$ and $(e_2,x) \rightsquigarrow_G o$, then $(e_1/e_2,x) \rightsquigarrow_G o$.

Next, we translate $(S,w) \rightsquigarrow_G w'$ into a deterministic derivation, using derivation steps denoted by $\Rightarrow_w$, similar to $\Rightarrow$, in the case of CFGs. All derivation steps $\Rightarrow_w$ for PEGs will also be derivation steps $\Rightarrow$ for the corresponding CFG (where we change prioritized choice, i.e. "/", into non-deterministic choice, i.e. "|", to go from a PEG to the corresponding CFG), but not necessarily conversely. We have that $(S,w) \rightsquigarrow_G w'$ if and only if $S \Rightarrow_w^+ w'$ ($\Rightarrow_w^+$ denotes the transitive closure of $\Rightarrow_w$), and $S \Rightarrow_w^+ w'$ implies $S \Rightarrow^+ w'$.

Definition 10 (Derivations, languages and parse trees defined by PEGs)**.** A *PEG derivation step* $u \Rightarrow_w v$ (for a PEG G), w.r.t. $w \in \Sigma^*$, denotes that it is possible to use a rule $A \to r_i^A$ (in G) from the production $A \to r_1^A/\ldots/r_{n_A}^A$, to replace the left-most nonterminal in the string $u \in (N \cup \Sigma)^*$ (which thus must be an A), by r_i^A, to produce the string v. Also, if u' is the prefix of u in Σ^* to the left of A, we require that $(u'r_i^A,w) \rightsquigarrow_G u'v'$, for some $v' \in \Sigma^*$, i.e. $(u'r_i^A,w) \not\rightsquigarrow_G f$. Additionally (in contrast to left-most derivation

steps $u \Rightarrow v$ in CFGs), we require that $(u' r_j^A, w) \rightsquigarrow_G f$, for $j < i$. If $S \Rightarrow_w^+ w'$, with $w' \in \Sigma^*$, for some w, then w' is in the *language defined by G*, and the (left-most) rule applications in the steps of this derivation (in order), is used to construct a *parse tree for w*.

Note that if $S \Rightarrow_w^+ w'$, then w' is a prefix of w, and $S \Rightarrow_{w''}^+ w'$ for any w'' that contains w' as prefix and is a prefix of w, in particular, $S \Rightarrow_{w'}^+ w'$.

Example 2. In this example, we consider the PEG discussed in the introduction with production $S \to aSa/a$, and show that a^5 is not accepted.

We use Definition 10. To see that $S \Rightarrow_{a^5} aSa \Rightarrow_{a^5} a^3$, we need to show that $(aaSa, a^5) \rightsquigarrow_G f$, which is the case since $(a^2S, a^5) \rightsquigarrow_G a^5$. This follows by verifying that $(S, a^3) \rightsquigarrow_G a^3$. ◇

4 oCFG derivations

We define in this section oCFG derivations and also show the close relationship between PEG and oCFG derivations. We obtain oCFG derivations by reformulating left-most CFG derivations to be deterministic by selecting the first rule choice, from a given production (for a given nonterminal), in the order they are specified in the oCFG, that will ensure a successful derivation. In our setting, derivations will be left-most, but by definition also deterministic, in contrast to how CFG derivations are typically defined. Also, only strings with smallest trees will have finite derivations. This can be seen by using the definition of a derivation, and also from the definition a smallest parse tree.

Derivations will be done in one of two modes: prefix mode, where parsing a prefix of the input string is regarded as a success, and full mode, where the complete input string must be parsed. The situation is similar to typical PCRE-style regular expression matchers, where the matcher can either be forced to determine if a full match is possible, or be asked to return the first prefix match.

Strictly speaking, we should use a symbol other than '$\Rightarrow$' in oCFG derivations, to distinguish between CFG and oCFG derivations, and we should also indicate, for which string a derivation is computed, just as the notation '$\Rightarrow_w$' used for PEG derivations, but to keep our notation simple, we will still use '$\Rightarrow$' in oCFG derivations.

PEGs with left recursion lead to infinite derivations, without producing a parse tree, in contrast to non-cyclic oCFGs. This is for example the case with the PEG having the production $S \to Sa/a$. But in contrast, in non-cyclic oCFGs we have finite derivations. Various ways of extending PEGs to support left-recursion have been proposed, for example in [16], which is used in the Pegen implementation [2], but these approaches often lead to unexpected parsing results in corner cases, as is pointed out in the section on related work in [10]. The time complexity of parsing also becomes quadratic in the length of the input string being parsed.

Next, we discuss distinctions between parsing with oCFGs, in contrast to when parsing with PEGs. For PEGs, we can memoize the value *False*, for pairs (A, i), with A a nonterminal and i a position in the input string, if parsing a prefix of the remainder of the input string from i, with A, is not possible, and recomputing this, is never necessary. Also, for PEGs, if $t_{i,A}$ is the parse tree when using A as root, and starting at a position i in the input string, then if $t_{0,S}$ makes use of A at position i, then $t_{0,S}$ will have $t_{i,A}$ as substree, and this subtree will be a parse tree for a prefix of the string starting at position i. Thus, for PEGs, we can also memoize parsing related to successful parsing starting from a given position in the input string with a given nonterminal. These memoization observations are not applicable to oCFGs, and they are the main reason why parsing with PEGs (when not having left recursion), can be done in linear time, in contrast to when parsing with oCFGs. Conceptually, we can regard PEGs as ignoring the overall sentential form when making rule selections during derivation steps, and only focussing on

producing locally successful parse trees, when starting from a given position with a given nonterminal, with preference given to later subderivations being locally successful.

Next, we note, as one would expect, that oCFG derivations produce least parse trees.

Theorem 3. *The rules in a derivation of a string w with a least tree over an oCFGs G, applied in order, in a left-most way, produce the least parse tree of w.*

Proof. The result follows directly from the definition of derivations in oCFGs. □

In the next section, we consider the complexity of determining an oCFG derivation of a complete input string, by making use of the shared packed parse forest for the input string.

5 oCFG parsing with shared packed parse forests

In this section, we show how to use shared packed parse forests (SPPFs) to compute oCFG derivations. First, we argue why considering only the case where the complete input string is parsed, is sufficient to also handle parsing in prefix mode. To turn prefix mode into a special case of parsing the complete input string, we note that to simulate prefix mode with full mode, we simply add a new start nonterminal S' with a rule $S' \to SA$, where S is the old start nonterminal, and A a new nonterminal not used elsewhere in the oCFG productions, and for A we add a production to ensure that A can parse any length input string.

In terms of our presentation of SPPFs, we follow [14] closely. An SPPF encodes all parse trees of a string w, derived from a CFG G, in a graph P, with the root node labelled by $(S,0,|w|)$, the number of nodes in P at worst cubic in $|w|$, and the height of a path not following cycles, bounded by $O(|w|)$, with the constant determined by G.

To define the SPPF P for a string w, derived from the CFG G, we first introduce indexed binary derivation trees. An indexed binary derivation tree (BDT) is constructed from a derivation tree by first introducing intermediate nodes, so that the tree is binarised from the right. Thus, when a node in a parse tree has more than two children, we keep the leftmost child as is, but concatenate the labels of all the other children, to obtain the label of the new right child. In contrast to how BDTs (and SPPFs) are typically presented, we binarise from the right, instead of from the left, since this corresponds more closely to how top-down, left-to-right parsing works. As is usually the case, we add to the labels of nodes, in the BDT, two integers, i and j, which are the left and right positions, in w, of the substring at their leaves. Also, if $(x\alpha,i,j)$ is the label of a node n in a BDT, with $|x\alpha| \geq 2$, where $x \in (N \cup \Sigma)$, then the left child of n is labelled by (x,i,k), and the right child by (α,k,j). Consider for example the CFG with rules, $S \to BAa \mid bAa$, $A \to a$, $B \to b$, and the input string baa. Then we have two BDTs, one in which the root node, labelled by $(S,0,3)$, has a left child $(B,0,1)$, and a right child $(Aa,1,3)$. In the other BDT, $(S,0,3)$ has left child $(b,0,1)$, and right child $(Aa,1,3)$. Nodes in the BDT labelled by (X,i,j), with $X \in (N \cup \Sigma \cup \varepsilon)$, will be referred to as symbol nodes, and those labelled by (α,i,j), with $\alpha \in (N \cup \Sigma)^*$, where $|\alpha| \geq 2$, as intermediate nodes.

A binarised SPPF is obtained from the set of indexed BDTs for w, by taking all nodes from the BDTs of w, identifying nodes with the same label, and by adding packed nodes. Non-leaf symbols nodes and intermediate nodes have one or more packed node children. A symbol node (X,i,j), with X a nonterminal, has a rule-packed child $(X \to x\beta,i,k,j)$, with $x \in (\Sigma \cup N)$ and $\beta \in (\Sigma \cup N)^*$, if:

(i) $X \to x\beta$ is a rule in G;

(ii) There is a symbol node labelled by (x,i,k);

(iii) Either $\beta \neq \varepsilon$ and there is a symbol or intermediate node labelled (β,k,j), or $\beta = \varepsilon$ and $k = j$.

The nodes (x,i,k), and (β,k,j), if $\beta \neq \varepsilon$, are the children of $(X \to x\beta,i,k,j)$. An intermediate node $(x\beta,i,j)$ with $x \in (\Sigma \cup N)$ and $\beta \in (\Sigma \cup N)^*$, where $|\beta| \geq 1$, has an intermediate packed node child labelled $(x\beta,i,k,j)$, if there are nodes labelled (x,i,k) and (β,k,j), which are then the children of the intermediate packed node.

In the example grammar $S \to BAa \mid bAa$, $A \to a$, $B \to b$, with input string baa, mentioned above, the root node $(S,0,3)$ in the SPPF, has two rule-packed nodes, namely $(S \to BAa,0,1,3)$ and also $(S \to bAa,0,1,3)$. The node $(S \to BAa,0,1,3)$ has children $(B,0,1)$ and $(Aa,1,3)$, and $(S \to bAa,0,1,3)$ has a left child $(b,0,1)$, and share its right child, $(Aa,1,3)$, with the node $(S \to BAa,0,1,3)$.

The SPPF for the input string w, can be constructed in time $O(|w|^3)$, using for example a generalized LL parsing algorithm. Also, the packed nodes on their own uniquely determine the symbol and intermediate nodes and if we only keep them, we have what is known as the binary-subtree representation (BSR) of a SPPF.

The string w has infinitely many parse trees, precisely when the SPPF has a cycle. When no cycle is present, each selection choice of rule-packed nodes, where a unique rule-packed node is selected from all rule packed node children of a given non-leaf symbol node, and all selected nodes are reachable from the root node, after removing those not selected, corresponds to a unique parse tree. Once we have made such a selection of rule-packed nodes, we obtain a parse tree, in which the selected rule-packed nodes, arranged in the order obtained by doing a pre-order traversal of the SPPF, provide the rules used in the left-most derivation of the parse tree described by SPPF with selected rule-packed nodes.

If we interpret the grammar $S \to BAa \mid bAa$, $A \to a$, $B \to b$, as an oCFG, then the parse tree for the input string baa is obtained by selecting the rule-packed node $(S \to BAa,0,1,3)$ from the SPPF, and discarding $(S \to bAa,0,1,3)$.

Example 3. Consider the CFG $S \to SS \mid b$, with bbb as input. Then the root node of the SPPF, is $(S,0,3)$, and this node has the rule-packed nodes $(S \to SS,0,1,3)$ and $(S \to SS,0,2,3)$ as children, which reflect the fact that we have two parse trees for the input string bbb. Note, in this case, when interpreting the CFG as an oCFG, it is not immediately clear that the rule-packed node $(S \to SS,0,2,3)$ should be selected, and $(S \to SS,0,1,3)$ discarded, in order to obtain the oCFG parse tree $S_1[S_1[S_2[b],S_2[b]],S_2[b]]$, for bbb, from the parse forest.

Next, consider the SPPF for $S \to SS \mid b \mid \varepsilon$, and input b. In this case, the root node $(S,0,1)$ has $(S \to SS,0,0,1)$, $(S \to SS,0,1,1)$ and $(S \to b,0,1,1)$, as packed node children. Note that $(S \to SS,0,0,1)$ has the symbol node children $(S,0,0)$ and $(S,0,1)$, and thus in this case we have a cycle in the SPPF, since $(S,0,0)$ has children $(S \to SS,0,0,0)$ and $(S \to \varepsilon,0,0,0)$, and $(S \to SS,0,0,0)$ has two edges back to $(S,0,0)$. This reflects the fact that b has infinitely many parse trees. ◇

Theorem 4. *Assume G is an oCFG. Then an oCFG derivation for a string w can be computed in time* $O(|w|^4)$.

Proof. First, we assume we have no cycles in the corresponding SPPF. We do a bottom up traversal of the SPPF, labelling along the way a node with the concatenation of indices of rules used in a left-most derivation, of the smallest parse tree below it. Thus, when encountering a symbol node with multiple packed node children, these nodes will have different labels, describing the rules used in a derivation of the smallest parse tree below them, and then amongst these, we select the packed node, with label being lexicographically the least. The traversal takes cubic time, and comparing two integer labels to find lexicographically the required SPPF node to construct the smallest parse tree (in cases where a node has multiple rule-packed node children), takes time linear in the length of the labels, which is bounded by the height of the SPPF. This provides a $O(h|w|^3)$ complexity bound, with h denoting the height of the

SPPF. The result, for the case when no cycles are present, now follows by observing that h is of order $O(|w|)$.

Now we consider the complication caused by the removed back edges. If any of these add cycles to the selected parse tree, the argument used in the proof of Theorem 1 can be applied to determine if taking any of these will lead to a larger parse tree in the order induced by the oCFG, or will lead to an infinite decreasing sequence of trees, if the cycle is repeatedly taken. This will inform us if the selected tree is minimal, or if no minimal parse tree exists for the given input string.

□

Example 4. In this example, we consider $S \to SS \mid b$, with input bbb. The root node has packed node children $(S \to SS, 0, 1, 3)$ and $(S \to SS, 0, 2, 3)$. The node $(S \to SS, 0, 1, 3)$ is labelled by 12122, and $(S \to SS, 0, 2, 3)$ by 11222, with 1 encoding the use of $S \to SS$ and 2, the use of $S \to b$, in a left-most derivation. Thus, with $(S \to SS, 0, 1, 3)$ we associate the parse tree obtained with the left-most derivation $S \Rightarrow SS \Rightarrow bS \Rightarrow bSS \Rightarrow bbS \Rightarrow bbb$. Similarly, with $(S \to SS, 0, 2, 3)$ we associate the parse tree obtained with the left-most derivation $S \Rightarrow SS \Rightarrow SSS \Rightarrow bSS \Rightarrow bbS \Rightarrow bbb$. Given that 11222 is lexicographically less than 12122, we select the parse tree with left-most derivation $S \Rightarrow SS \Rightarrow SSS \Rightarrow bSS \Rightarrow bbS \Rightarrow bbb$. ◇

6 Conclusions and Future Work

We have shown that oCFGs provide a good way to understand the relationship between PEGs and CFGs, and it has more natural matching semantics than PEGs, but this comes at the price of worse parsing complexity. Ordered context-free grammars is a natural way in which to extend PCRE regex matching to an ordered context-free grammar formalism, in which it is possible to talk about the first match or least parse tree. The natural next step is to build an oCFG parsing tool, which will make it possible to analyse the effort involved for grammar writers to use the oCFG grammar formalism rather than some of the other well-known grammar formalisms. This will also make it possible to determine experimentally if oCFG parsing is fast enough for practical use on large grammars. We are also interested in adding lookahead predicates, as used in PEGs [5], to oCFGs, and to study the properties of oCFGs with these extensions, similarly to how Bryan Ford investigated PEGs with these extensions (see [5]). Once this is added to oCFGs, it is no longer necessary to distinguish between the two modes of parsing, i.e. prefix and full mode, since full mode can be obtained from prefix mode by using a predicate to specify that the part of the input string being parsed by the start nonterminal, should not be followed by any character. Future work also includes a thorough study of which disambiguation can be done with oCFGs and which not, and a study of which disambiguation mechanisms are available in popular compiler generators, and their use in sample grammars. We would also like to investigate interesting and useful subclasses of oCFGs for which parsing can be done in much better time complexity than $O(n^4)$, where n is the length of the input string being parsed.

Acknowledgement

I would like to thank Martin Berglund for reading various versions of this document, and suggesting improvements.

References

[1] *Generating a parser using yacc*. https://www.ibm.com/docs/en/zos/2.2.0?topic=tools-generating-parser-using-yacc. Accessed: 2023-04-08.

[2] *Pegen*. https://github.com/we-like-parsers/pegen. Accessed: 2022-02-28.

[3] Martin Berglund & Brink van der Merwe (2017): *On the semantics of regular expression parsing in the wild*. *Theor. Comput. Sci.* 679, pp. 69–82, doi:10.1016/j.tcs.2016.09.006.

[4] Martin Berglund, Brink van der Merwe, Bruce W. Watson & Nicolaas Weideman (2017): *On the Semantics of Atomic Subgroups in Practical Regular Expressions*. In Arnaud Carayol & Cyril Nicaud, editors: *Implementation and Application of Automata - 22nd International Conference, CIAA 2017, Marne-la-Vallée, France, June 27-30, 2017, Proceedings*, *Lecture Notes in Computer Science* 10329, Springer, pp. 14–26, doi:10.1007/978-3-319-60134-2_2.

[5] Bryan Ford (2004): *Parsing expression grammars: a recognition-based syntactic foundation*. In Neil D. Jones & Xavier Leroy, editors: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, ACM, pp. 111–122, doi:10.1145/964001.964011.

[6] Ivan Fris (1968): *Grammars with Partial Ordering of the Rules*. *Inf. Control.* 12(5/6), pp. 415–425, doi:10.1016/S0019-9958(68)90439-7.

[7] Paul Klint & Eelco Visser (1994): *Using Filters for the Disambiguation of Context-free Grammars*. In: *Proceedings of the ASMICS Workshop on Parsing Theory*, Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano, Milano, Italy, pp. 1–20.

[8] Timo Lepistö (1973): *On Ordered Context-Free Grammars*. *Inf. Control.* 22(1), pp. 56–68, doi:10.1016/S0019-9958(73)90478-6.

[9] Bruno Loff, Nelma Moreira & Rogério Reis (2020): *The computational power of parsing expression grammars*. *J. Comput. Syst. Sci.* 111, pp. 1–21, doi:10.1016/j.jcss.2020.01.001.

[10] Sérgio Medeiros, Fabio Mascarenhas & Roberto Ierusalimschy (2014): *Left recursion in Parsing Expression Grammars*. *Sci. Comput. Program.* 96, pp. 177–190, doi:10.1016/j.scico.2014.01.013.

[11] Brink van der Merwe & Martin Berglund (2022): *Ordered Context-Free Grammars*. In Pascal Caron & Ludovic Mignot, editors: *Implementation and Application of Automata - 26th International Conference, CIAA 2022*, *Lecture Notes in Computer Science* 13266, Springer, pp. 53–66, doi:10.1007/978-3-031-07469-1_4.

[12] Terence Parr & Kathleen Fisher (2011): *LL(*): the foundation of the ANTLR parser generator*. In Mary W. Hall & David A. Padua, editors: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 425–436, doi:10.1145/1993498.1993548.

[13] Terence Parr, Sam Harwell & Kathleen Fisher (2014): *Adaptive LL(*) parsing: the power of dynamic analysis*. In Andrew P. Black & Todd D. Millstein, editors: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ACM, pp. 579–598, doi:10.1145/2660193.2660202.

[14] Elizabeth Scott, Adrian Johnstone & L. Thomas van Binsbergen (2019): *Derivation representation using binary subtree sets*. *Sci. Comput. Program.* 175, pp. 63–84, doi:10.1016/j.scico.2019.01.008.

[15] Seppo Sippu (1982): *Derivational Complexity of Context-Free Grammars*. *Inf. Control.* 53(1/2), pp. 52–65, doi:10.1016/S0019-9958(82)91111-1.

[16] Alessandro Warth, James R. Douglass & Todd D. Millstein (2008): *Packrat parsers can support left recursion*. In Robert Glück & Oege de Moor, editors: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp. 103–110, doi:10.1145/1328408.1328424.