Synthesizing Context-free Grammars from Recurrent Neural Networks
Extended Version*

Daniel M. Yellin[1] and Gail Weiss[2]

[1] IBM, Givatayim, Israel
dannyyellin@gmail.com
[2] Technion, Haifa, Israel
sgailw@cs.technion.ac.il

Abstract. We present an algorithm for extracting a subclass of the context free grammars (CFGs) from a trained recurrent neural network (RNN). We develop a new framework, *pattern rule sets* (PRSs), which describe sequences of deterministic finite automata (DFAs) that approximate a non-regular language. We present an algorithm for recovering the PRS behind a sequence of such automata, and apply it to the sequences of automata extracted from trained RNNs using the L^* algorithm. We then show how the PRS may converted into a CFG, enabling a familiar and useful presentation of the learned language.
Extracting the learned language of an RNN is important to facilitate understanding of the RNN and to verify its correctness. Furthermore, the extracted CFG can augment the RNN in classifying correct sentences, as the RNN's predictive accuracy decreases when the recursion depth and distance between matching delimiters of its input sequences increases.

Keywords: Model Extraction · Learning Context Free Grammars · Finite State Machines · Recurrent Neural Networks

1 Introduction

Recurrent Neural Networks (RNNs) are a class of neural networks adapted to sequential input, enjoying wide use in a variety of sequence processing tasks. Their internal process is opaque, prompting several works into extracting interpretable rules from them. Existing works focus on the extraction of deterministic or weighted finite automata (DFAs and WFAs) from trained RNNs [19,6,27,3].

However, DFAs are insufficient to fully capture the behavior of RNNs, which are known to be theoretically Turing-complete [21], and for which there exist architecture variants such as LSTMs [14] and features such as stacks [9,24] or attention [4] increasing their practical power. Several recent investigations

* This is an extended version of a paper that will appear in the 27th International Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2021)

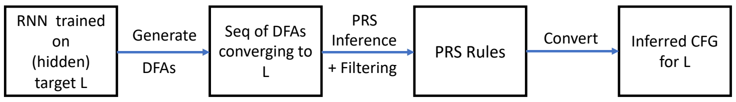
RNN trained on (hidden) target L
Generate DFAs
Seq of DFAs converging to L
PRS Inference + Filtering
PRS Rules
Convert
Inferred CFG for L

Fig. 1. Overview of steps in algorithm to synthesize the hidden language L

explore the ability of different RNN architectures to learn Dyck, counter, and other non-regular languages [20,5,28,22], with mixed results. While the data indicates that RNNs can generalize and achieve high accuracy, they do not learn hierarchical rules, and generalization deteriorates as the distance or depth between matching delimiters becomes dramatically larger[20,5,28]. Sennhauser and Berwick conjecture that "what the LSTM has in fact acquired is sequential statistical approximation to this solution" instead of "the 'perfect' rule-based solution" [20]. Similarly, Yu et. al. conclude that "the RNNs can not truly model CFGs, even when powered by the attention mechanism".

Goal of this paper We wish to extract a CFG from a trained RNN. Our motivation is two-fold: first, extracting a CFG from the RNN is important to facilitate understanding of the RNN and to verify its correctness. Second, the learned CFG may be used to augment or generalise the rules learned by the RNN, whose own predictive ability decreases as the depth of nested structures and distance between matching constructs in the input sequences increases [5,20,28]. Our technique can synthesize the CFG based upon training data with relatively short distance and small depth. As pointed out in [13], a fixed precision RNN can only learn a language of fixed depth strings (in contrast to an idealized infinite precision RNN that can recognize any Dyck language[16]). Our goal is to find the CFG that not only explains the finite language learnt by the RNN, but generalizes it to strings of unbounded depth and distance.

Our approach Our method builds on the DFA extraction work of Weiss et al. [27], which uses the L^* algorithm [2] to learn the DFA of a given RNN. The L^* algorithm operates by generating a *sequence* of DFAs, each one a hypothesis for the target language, and interacting with a teacher, in our case the RNN, to improve them. Our main insight is that we can view these DFAs as increasingly accurate approximations of the target CFL. We assume that each hypothesis improves on its predecessor by applying an unknown rule that recursively increases the distance and embedded depth of sentences accepted by the underlying CFL. In this light, synthesizing the CFG responsible for the language learnt by the RNN becomes the problem of recovering these rules. A significant issue we must also address is that the DFAs produced are often inexact or not as we expect, either due to the failure of the RNN to accurately learn the language, or as an artifact of the L^* algorithm.

We propose the framework of *pattern rule sets* (PRSs) for describing such rule applications, and present an algorithm for recovering a PRS from a sequence of DFAs. We also provide a method for converting a PRS to a CFG, translating our extracted rules into familiar territory. We test our method on RNNs trained on several PRS languages.

Pattern rule sets are expressive enough to cover several variants of the Dyck languages, which are prototypical CFLs: the Chomsky–Schützenberger representation theorem shows that any context-free language can be expressed as a homomorphic image of a Dyck language intersected with a regular language[17].

To the best of our knowledge, this is the first work on synthesizing a CFG from a general RNN [1].

Contributions The main contributions of this paper are:

– *Pattern Rule Sets* (PRSs), a framework for describing a sequence of DFAs approximating a CFL.
– An algorithm for recovering the PRS generating a sequence of DFAs, that may also be applied to noisy DFAs elicited from an RNN using L^* .
– An algorithm converting a PRS to a CFG.
– An implementation of our technique, and an evaluation of its success on recovering various CFLs from trained RNNs. [2]

The overall steps in our technique are given in Figure 1. The rest of this paper is as follows. Section 2 provides basic definitions used in the paper, and Section 3 introduces *Patterns*, a restricted form of DFAs. Section 4 defines *Pattern Rule Sets (PRS)*, the main construct of our research. Section 5 gives an algorithm to recover a PRS from a sequence of DFAs, even in the presence of noise, and Section 6 gives an algorithm to convert a PRS into a CFG. Section 7 presents our experimental results, Section 8 discusses related research and Section 9 outlines directions for future research. Appendices B and C provide proofs of the correctness of the algorithms given in the paper, as well results relating to the expressibility of a PRS.

2 Definitions and Notations

2.1 Deterministic Finite Automata

Definition 1 (Deterministic Finite Automata). *A deterministic finite automaton (DFA) over an alphabet Σ is a 5-tuple $\langle \Sigma, q_0, Q, F, \delta \rangle$ such that Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final (accepting) states and $\delta : Q \times \Sigma \to Q$ is a (possibly partial) transition function.*

[1] Though some works extract push-down automata [24,9] from RNNs with an external stack (Sec. 8), they do not apply to plain RNNs.

[2] The implementation for this paper, and a link to all trained RNNs, is available at `https://github.com/tech-srl/RNN_to_PRS_CFG`.

Unless stated otherwise, we assume each DFA's states are unique to itself, i.e., for any two DFAs A, B – including two instances of the same DFA – $Q_A \cap Q_B = \emptyset$. A DFA A is said to be *complete* if δ is complete, i.e., the value $\delta(q, \sigma)$ is defined for every $q, \sigma \in Q \times \Sigma$. Otherwise, it is *incomplete*.

We define the extended transition function $\hat{\delta} : Q \times \Sigma^* \to Q$ and the language $L(A)$ accepted by A in the typical fashion. We also associate a language with intermediate states of A: $L(A, q_1, q_2) \triangleq \{w \in \Sigma^* \mid \hat{\delta}(q_1, w) = q_2\}$. The states from which no sequence $w \in \Sigma^*$ is accepted are known as the *sink reject states*.

Definition 2. *The* sink reject states *of a DFA* $A = \langle \Sigma, q_0, Q, F, \delta \rangle$ *are the maximal set* $Q_R \subseteq Q$ *satisfying:* $Q_R \cap F = \emptyset$, *and for every* $q \in Q_R$ *and* $\sigma \in \Sigma$, *either* $\delta(q, \sigma) \in Q_R$ *or* $\delta(q, \sigma)$ *is not defined.*

Incomplete DFAs are partial representations of complete DFAs, where every unspecified transition is shorthand for a transition to a sink reject state. All definitions for complete DFAs are extended to incomplete DFAs A by considering their *completion*: the DFA A_C obtained by connecting a (possibly new) sink reject state to all its missing transitions. For each DFA, we take note of the transitions which cannot be removed even in its partial representations.

Definition 3 (Defined Tokens). *Let* $A = \langle \Sigma, q_0, Q, F, \delta \rangle$ *be a complete DFA with sink reject states* Q_R. *For every* $q \in Q$, *its* defined tokens *are* $\mathrm{def}(A, q) \triangleq \{\sigma \in \Sigma \mid \delta(q, \sigma) \notin Q_R\}$. *When the DFA A is clear from context, we write* $\mathrm{def}(q)$.

We now introduce terminology that will help us discuss merging automata states.

Definition 4 (Set Representation of δ). *A (possibly partial) transition function* $\delta : Q \times \Sigma \to Q$ *may be equivalently defined as the set* $S_\delta = \{(q, \sigma, q') \mid \delta(q, \sigma) = q'\}$. *We use* δ *and* S_δ *interchangeably.*

Definition 5 (Replacing a State). *For a transition function* $\delta : Q \times \Sigma \to Q$, *state* $q \in Q$, *and new state* $q_n \notin Q$, *we denote by* $\delta_{[q \leftarrow q_n]} : Q' \times \Sigma \to Q'$ *the transition function over* $Q' = (Q \setminus \{q\}) \cup \{q_n\}$ *and* Σ *that is identical to* δ *except that it redirects all transitions into or out of* q *to be into or out of* q_n.

2.2 Dyck Languages

A Dyck language *of order N* is expressed by the grammar `D ::=` ε `|` $\mathtt{L}_i$ `D` $\mathtt{R}_i$ `| D D` with start symbol `D`, where for each $1 \leq i \leq N$, L_i and R_i are matching left and right delimiters. A common methodology for measuring the complexity of a Dyck word is to measure its maximum *distance* (number of characters) between matching delimiters and *embedded depth* (number of unclosed delimiters) [20].

While L_i and R_i are single characters in a Dyck language, we generalize and refer to *Regular Expression Dyck (RE-Dyck)* languages as languages expressed by the same CFG, except that each $\mathtt{L}_i$ and each $\mathtt{R}_i$ derive some regular expression.

Regular Expressions: We present regular expressions as is standard, for example: $\{a|b\}\cdot c$ refers to the language consisting of one of a or b, followed by c.

3 Patterns

Patterns are DFAs with a single *exit* state q_X in place of a set of final states, and with no cycles on their initial or exit states unless $q_0 = q_X$. In this paper we express patterns in incomplete representation, i.e., they have no explicit sink-reject states.

Definition 6 (Patterns). *A* pattern $p = \langle \Sigma, q_0, Q, q_X, \delta \rangle$ *is a DFA* $A^p = \langle \Sigma, q_0, Q, \{q_X\}, \delta \rangle$ *satisfying:* $L(A^p) \neq \emptyset$*, and either* $q_0 = q_X$*, or* $\operatorname{def}(q_X) = \emptyset$ *and* $L(A, q_0, q_0) = \{\varepsilon\}$*. If* $q_0 = q_X$ *then* p *is called* circular*, otherwise, it is* non-circular*.*

Note that our definition does not rule out a cycle in the middle of an *non-circular* pattern but only one that traverses the initial or final states.

All the definitions for DFAs apply to patterns through A^p. We denote each pattern p's language $L_p \triangleq L(p)$, and if it is marked by some superscript i, we refer to all of its components with superscript i: $p^i = \langle \Sigma, q_0^i, Q^i, q_X^i, \delta^i \rangle$.

3.1 Pattern Composition

We can compose two non-circular patterns p^1, p^2 by merging the exit state of p^1 with the initial state of p^2, creating a new pattern p^3 satisfying $L_{p^3} = L_{p^1} \cdot L_{p^2}$.

Definition 7 (Serial Composition). *Let* p^1, p^2 *be two non-circular patterns. Their* serial composite *is the pattern* $p^1 \circ p^2 = \langle \Sigma, q_0^1, Q, q_X^2, \delta \rangle$ *in which* $Q = Q^1 \cup Q^2 \setminus \{q_X^1\}$ *and* $\delta = \delta^1_{[q_X^1 \leftarrow q_0^2]} \cup \delta^2$*. We call* q_0^2 *the* join state *of this operation.*

If we additionally merge the exit state of p_2 with the initial state of p_1, we obtain a circular pattern p which we call the *circular composition* of p_1 and p_2. This composition satisfies $L_p = \{L_{p_1} \cdot L_{p_2}\}^*$.

Definition 8 (Circular Composition). *Let* p^1, p^2 *be two non-circular patterns. Their* circular composite *is the circular pattern* $p_1 \circ_c p_2 = \langle \Sigma, q_0^1, Q, q_0^1, \delta \rangle$ *in which* $Q = Q^1 \cup Q^2 \setminus \{q_X^1, q_X^2\}$ *and* $\delta = \delta^1_{[q_X^1 \leftarrow q_0^2]} \cup \delta^2_{[q_X^2 \leftarrow q_0^1]}$*. We call* q_0^2 *the* join state *of this operation.*

Figure 2 shows 3 examples of serial and circular compositions of patterns.

Patterns do not carry information about whether or not they have been composed from other patterns. We maintain such information using *pattern pairs*.

Definition 9 (Pattern Pair). *A* pattern pair *is a pair* $\langle P, P_c \rangle$ *of pattern sets, such that* $P_c \subset P$ *and for every* $p \in P_c$ *there exists exactly one pair* $p_1, p_2 \in P$ *satisfying* $p = p_1 \odot p_2$ *for some* $\odot \in \{\circ, \circ_c\}$*. We refer to the patterns* $p \in P_c$ *as the* composite patterns *of* $\langle P, P_c \rangle$*, and to the rest as its* base patterns*.*

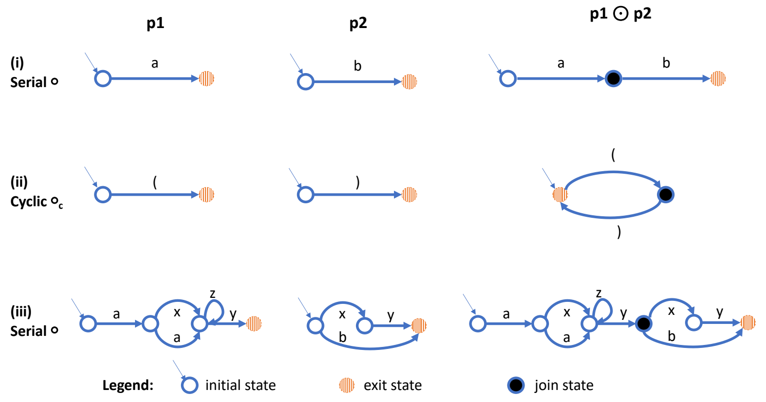

Fig. 2. Examples of the composition operator

Every instance $\hat{p}$ of a pattern p in a DFA A is uniquely defined by p, A, and $\hat{p}$'s initial state in A. If p is a composite pattern with respect to some pattern pair $\langle P, P_c \rangle$, the join state of its composition within A is also uniquely defined.

Definition 10 (Pattern Instances). *Let $A = \langle \Sigma, q_0^A, Q^A, F, \delta^A \rangle$ be a DFA, $p = \langle \Sigma, q_0, Q, q_X, \delta \rangle$ be a pattern, and $\hat{p} = \langle \Sigma, q_0', Q', q_X', \delta' \rangle$ be a pattern 'inside' A, i.e., $Q' \subseteq Q^A$ and $\delta' \subseteq \delta^A$. We say that $\hat{p}$ is an* instance of p in A *if $\hat{p}$ is isomorphic to p.*

A pattern instance $\hat{p}$ in a DFA A is uniquely determined by its structure and initial state: (p, q).

Definition 11. *For every pattern pair $\langle P, P_c \rangle$ we define the function* join *as follows: for each composite pattern $p \in P_c$, DFA A, and initial state q of an instance $\hat{p}$ of p in A,* $\text{join}(p, q, A)$ *returns the join state of $\hat{p}$ with respect to its composition in $\langle P, P_c \rangle$.*

4 Pattern Rule Sets

For any infinite sequence $S = A_1, A_2, ...$ of DFAs satisfying $L(A_i) \subset L(A_{i+1})$, for all i, we define the language of S as the union of the languages of all these DFAs: $L(S) = \cup_i L(A_i)$. Such sequences may be used to express CFLs such as the language $L = \{\mathrm{a}^n \mathrm{b}^n \mid n \in \mathbb{N}\}$ and the Dyck language of order N.

In this work we take a finite sequence $A_1, A_2, ..., A_n$ of DFAs, and assume it is a (possibly noisy) finite prefix of an infinite sequence of approximations for a language, as above. We attempt to reconstruct the language by guessing how the sequence may continue. To allow such generalization, we must make assumptions about how the sequence is generated. For this we introduce *pattern rule sets.*

Pattern rule sets (PRSs) create sequences of DFAs with a single accepting state. Each PRS is built around a pattern pair $\langle P, P_c \rangle$, and each rule application involves the connection of a new pattern instance to the current DFA A_i, at the join state of a composite-pattern inserted whole at some earlier point in the DFA's creation. In order to define where a pattern can be inserted into a DFA, we introduce an *enabled instance* set $\mathcal{I}$.

Definition 12. *An* enabled DFA *over a pattern pair* $\langle P, P_c\rangle$ *is a tuple* $\langle A, \mathcal{I}\rangle$ *such that* $A = \langle \Sigma, q_0, Q, F, \delta\rangle$ *is a DFA and* $\mathcal{I} \subseteq P_c \times Q$ *marks* enabled instances *of composite patterns in* A.

Intuitively, for every enabled DFA$\langle A, \mathcal{I}\rangle$ and $(p, q) \in \mathcal{I}$, we know: (i) there is an instance of pattern p in A starting at state q, and (ii) this instance is *enabled*; i.e., we may connect new pattern instances to its join state $\mathrm{join}(p, q, A)$.

We now formally define pattern rule sets and how they are applied to create enabled DFAs, and so sequences of DFAs.

Definition 13. *A PRS* $\mathbf{P}$ *is a tuple* $\langle \Sigma, P, P_c, R\rangle$ *where* $\langle P, P_c\rangle$ *is a pattern pair over the alphabet* Σ *and* R *is a set of* rules. *Each rule has one of the following forms, for some* $p, p^1, p^2, p^3, p^I \in P$*, with* p^1 *and* p^2 *non-circular:*

(1) $\bot \twoheadrightarrow p^I$
(2) $p \twoheadrightarrow_c (p^1 \odot p^2) \eqcirc p^3$*, where* $p = p^1 \odot p^2$ *for* $\odot \in \{\circ, \circ_c\}$*, and* p^3 *is circular*
(3) $p \twoheadrightarrow_s (p^1 \circ p^2) \eqcirc p^3$*, where* $p = p^1 \circ p^2$ *and* p^3 *is non-circular*

A PRS is used to derive sequences of enabled DFAs as follows: first, a rule of type (1) is used to create an initial enabled DFA $\mathcal{D}_1 = \langle A_1, \mathcal{I}_1\rangle$. Then, for any $\langle A_i, \mathcal{I}_i\rangle$, each of the rule types define options to graft new pattern instances onto states in A_i, with $\mathcal{I}_i$ determining which states are eligible to be expanded in this way. The first DFA is simply the p^I from a rule of type (1). If p^I is composite, then it is also enabled.

Definition 14 (Initial Composition). $\mathcal{D}_1 = \langle A_1, \mathcal{I}_1\rangle$ *is generated from a rule* $\bot \twoheadrightarrow p^I$ *as follows:* $A_1 = A^{p^I}$*, and* $\mathcal{I}_i = \{(p^I, q_0^I)\}$ *if* $p^I \in P_c$ *and otherwise* $\mathcal{I}_1 = \emptyset$.

Let $\mathcal{D}_i = \langle A_i, \mathcal{I}_i\rangle$ be an enabled DFA generated from some given PRS $\mathbf{P} = \langle \Sigma, P, P_c, R\rangle$, and denote $A_i = \langle \Sigma, q_0, Q, F, \delta\rangle$. Note that for A_1, $|F| = 1$, and we will see that F is unchanged by all further rule applications. Hence we denote $F = \{q_f\}$ for all A_i.

Rules of type (1) extend A_i by grafting a circular pattern to q_0, and then enabling that pattern if it is composite.

Definition 15 (Rules of type (1)). *A rule* $\bot \twoheadrightarrow p^I$ *with circular* p^I *may extend* $\langle A_i, \mathcal{I}_i\rangle$ *at the initial state* q_0 *of* A_i *iff* $\mathrm{def}(q_0) \cap \mathrm{def}(q_0^I) = \emptyset$. *This creates the DFA* $A_{i+1} = \langle \Sigma, q_0, Q \cup Q^I \setminus \{q_0^I\}, F, \delta \cup \delta^I_{[q_0^I \leftarrow q_0]}\rangle$. *If* $p^I \in P_c$ *then* $\mathcal{I}_{i+1} = \mathcal{I}_i \cup \{(p^I, q_0)\}$*, else* $\mathcal{I}_{i+1} = \mathcal{I}_i$.

Rules of type (2) graft a circular pattern $p^3 = \langle \Sigma, q_0^3, q_x^3, F, \delta^3\rangle$ onto the join state q_j of an enabled pattern instance $\hat{p}$ in A_i, by merging q_0^3 with q_j. In doing so, they also enable the patterns composing $\hat{p}$, provided they themselves are composite patterns.

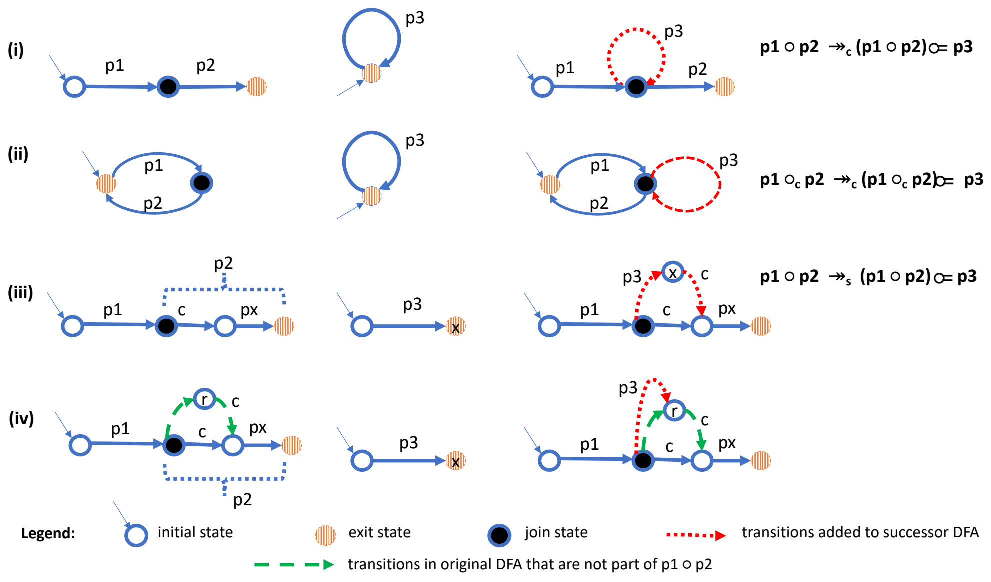
(i)
p1
p2
p3
p1 ∘ p2 ↠c (p1 ∘ p2)∝ p3
(ii)
p1 ∘c p2 ↠c (p1 ∘c p2)∝ p3
(iii)
c
px
x
p1 ∘ p2 ↠s (p1 ∘ p2)∝ p3
(iv)
r
Legend:
initial state
exit state
join state
transitions added to successor DFA
transitions in original DFA that are not part of p1 ∘ p2

Fig. 3. Structure of DFA after applying rule of type 2 or type 3

Definition 16 (Rules of type (2)). *A rule $p \twoheadrightarrow_c (p^1 \odot p^2)\propto p^3$ may extend $\langle A_i, \mathcal{I}_i\rangle$ at the join state $q_j = \text{join}(p,q,A_i)$ of any instance $(p,q) \in \mathcal{I}_i$, provided $\text{def}(q_j) \cap \text{def}(q_0^3) = \emptyset$. This creates $\langle A_{i+1}, \mathcal{I}_{i+1}\rangle$ as follows: $A_{i+1} = \langle \Sigma, q_0, Q \cup Q^3 \setminus q_0^3, F, \delta \cup \delta^3_{[q_0^3 \leftarrow q_j]}\rangle$, and $\mathcal{I}_{i+1} = \mathcal{I}_i \cup \{(p^k, q^k) \mid p^k \in P_c, k \in \{1,2,3\}\}$, where $q^1 = q$ and $q^2 = q^3 = q_j$.*

For an application of $r = p \twoheadrightarrow_c (p^1 \odot p^2)\propto p^3$, consider the languages L_L and L_R leading into and 'back from' the considered instance (p,q): $L_L = L(A_i, q_0, q)$ and $L_R = L(A_i, q_X^{(p,q)}, q_f)$, where $q_X^{(p,q)}$ is the exit state of (p,q). Where $L_L \cdot L_p \cdot L_R \subseteq L(A_i)$, then now also $L_L \cdot L_{p^1} \cdot L_{p^3} \cdot L_{p_2} \cdot L_R \subseteq L(A_{i+1})$ (and moreover, $L_L \cdot (L_{p^1} \cdot L_{p^3} \cdot L_{p_2})^* \cdot L_R \subseteq L(A_{i+1})$ if p is circular). Example applications of rule (2) are shown in Figures 3(i) and 3(ii).

For non-circular patterns we also wish to insert an optional L_{p^3} between L_{p^1} and L_{p^2}, but this time we must avoid connecting the exit state q_X^3 to q_j lest we loop over p^3 multiple times. We therefore duplicate the outgoing transitions of q_j in $p^1 \circ p^2$ to the inserted state q_X^3 so that they may act as the connections back into the main DFA.

Definition 17 (Rules of type (3)). *A rule $p \twoheadrightarrow_s (p^1 \circ p^2)\propto p^3$ may extend $\langle A_i, \mathcal{I}_i\rangle$ at the join state $q_j = \text{join}(p,q,Ai)$ of any instance $(p,q) \in \mathcal{I}_i$, provided $\text{def}(q_j) \cap \text{def}(q_0^3) = \emptyset$. This creates $\langle A_{i+1}, \mathcal{I}_{i+1}\rangle$ as follows: $A_{i+1} = \langle \Sigma, q_0, Q \cup$*

$Q^3 \setminus q_0^3, F, \delta \cup \delta^3_{[q_0^3 \leftarrow q_j]} \cup C\rangle$ *where* $C = \{\ (q_X^3, \sigma, \delta(q_j, \sigma)) |\ \sigma \in \mathrm{def}(p^2, q_0^2)\}$, *and* $\mathcal{I}_{i+1} = \mathcal{I}_i \cup \{(p^k, q^k) \mid p^k \in P_c, k \in \{1, 2, 3\}\}$ *where* $q^1 = q$ *and* $q^2 = q^3 = q_j$.

We call the set C *connecting transitions*. This application of this rule is depicted in Diagram (iii) of Figure 3, where the transition labeled 'c' in this Diagram is a member of C from our definition.

Multiple applications of rules of type (3) to the same instance $\hat{p}$ will create several equivalent states in the resulting DFAs, as all of their exit states will have the same connecting transitions. These states are merged in a minimized representation, as depicted in Diagram (iv) of Figure 3.

We now formally define the language defined by a PRS. This is the language that we will assume a given finite sequence of DFAs is trying to approximate.

Definition 18 (DFAs Generated by a PRS). *We say that a PRS* $\mathbf{P}$ generates *a DFA* A*, denoted* $A \in G(\mathbf{P})$*, if there exists a finite sequence of enabled DFAs* $\langle A_1, \mathcal{I}_1\rangle, ..., \langle A_i, \mathcal{I}_i\rangle$ *obtained only by applying rules from* $\mathbf{P}$*, for which* $A = A_i$.

Definition 19 (Language of a PRS). *The language of a PRS* $\mathbf{P}$ *is the union of the languages of the DFAs it can generate:* $L(\mathbf{P}) = \cup_{A \in G(\mathbf{P})} L(A)$.

4.1 Examples

EXAMPLE 1: Let p^1 and p^2 be the patterns accepting 'a' and 'b' respectively. Consider the rule set R_{ab} with two rules, $\bot \twoheadrightarrow p^1 \circ p^2$ and $p^1 \circ p^2 \twoheadrightarrow_s (p^1 \circ p^2) \circeq (p^1 \circ p^2)$. This rule set creates only one sequence of DFAs. Once the first rule creates the initial DFA, by continuously applying the second rule, we obtain the infinite sequence of DFAs each satisfying $L(A_i) = \{a^j b^j : 1 \leq j \leq i\}$, and so $L(R_{ab}) = \{a^i b^i : i > 0\}$. Figure 2(i) presents A_1, while A_2 and A_3 appear in Figure 4(i). Note that we can substitute any non-circular patterns for p^1 and p^2, creating the language $\{x^i y^i : i > 0\}$ for any pair of non-circular pattern regular expressions x and y.

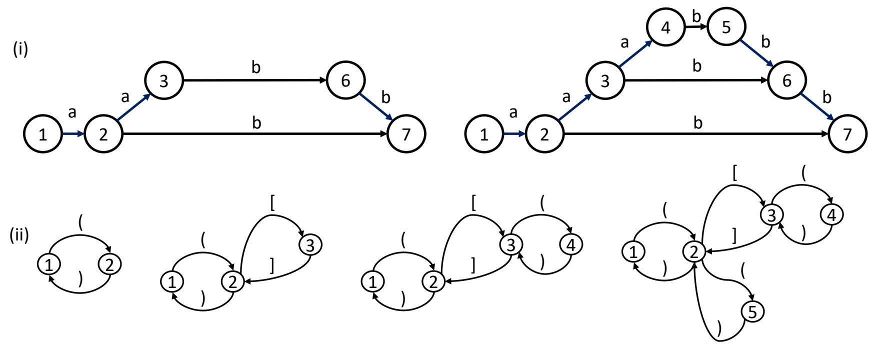

Fig. 4. DFAs sequences for R_{ab} and R_{Dyck2}

EXAMPLE 2: Let p^1,p^2,p^4, and p^5 be the non-circular patterns accepting '(', ')', '[', and ']' respectively. Let $p^3 = p^1 \circ_c p^2$ and $p^6 = p^4 \circ_c p^5$. Let R_{Dyck2} be the PRS containing rules $\bot \twoheadrightarrow p^3$, $\bot \twoheadrightarrow p^6$, $p^3 \twoheadrightarrow_c (p^1 \circ_c p^2) \multimapinv p^3$, $p^3 \twoheadrightarrow_c (p^1 \circ_c p^2) \multimapinv p^6$, $p^6 \twoheadrightarrow_c (p^4 \circ_c p^5) \multimapinv p^3$, and $p^6 \twoheadrightarrow_c (p^4 \circ_c p^5) \multimapinv p^6$. R_{Dyck2} defines the Dyck language of order 2. Figure 4 (ii) shows one of its possible DFA-sequences.

EXAMPLE 3: Let p^0 and p^1 be the patterns that accept the characters "0" and "1" respectively, $p^{00} = p^0 \circ p^0$ and $p^{11} = p^1 \circ p^1$. Let R^{pal} consist of the rules $\bot \twoheadrightarrow p^{00}$, $\bot \twoheadrightarrow p^{11}$, $p^{00} \twoheadrightarrow_s (p^0 \circ p^0) \multimapinv p^{00}$, $p^{00} \twoheadrightarrow_s (p^0 \circ p^0) \multimapinv p^{11}$, $p^{11} \twoheadrightarrow_s (p^1 \circ p^1) \circ p^{00}$, and $p^{11} \twoheadrightarrow_s (p^1 \circ p^1) \multimapinv p^{11}$. $L(R^{pal})$ is exactly the language of even-length palindromes over the alphabet $\{0, 1\}$.

Note. Consider a DFA A accepting (among others) the palindrome $s = 01100110$, derived from R^{pal}. If we were to consider A without the context of its enabled pattern instances $\mathcal{I}$, we could apply $p^{11} \twoheadrightarrow_s (p^1 \circ p^1) \multimapinv p^{11}$ to the 'first' instance of p^{11} in A, creating a DFA accepting the string 0111100110 which is not a palindrome. This illustrates the importance of the notion of *enabled* patterns in our framework.

5 PRS Inference Algorithm

We have shown how a PRS can generate a sequence of DFAs that can define, in the limit, a non-regular language. However, we are interested in the dual problem: given a sequence of DFAs generated by a PRS $\mathbf{P}$, can we reconstruct $\mathbf{P}$? Coupled with an L^* extraction of DFAs from a trained RNN, solving this problem will enable us to extract a PRS language from an RNN, provided the L^* extraction also follows a PRS pattern (as we often find it does).

We present an algorithm for this problem, and show its correctness in Section 5.1. We note that in practice the DFAs we are given are not "perfect"; they contain noise that deviates from the PRS. We therefore augment this algorithm in Section 5.2, allowing it to operate smoothly even on imperfect DFA sequences created from RNN extraction.

In the following, for each pattern instance $\hat{p}$ in A_i, we denote by p the pattern that it is an instance of. Additionally, for each consecutive DFA pair A_i and A_{i+1}, we refer by $\hat{p}^3$ to the new pattern instance in A_{i+1}.

Main steps of inference algorithm. Given a sequence of DFAs $A_1 \cdots A_n$, the algorithm infers $\mathbf{P} = \langle \Sigma, P, P_c, R \rangle$ in the following stages:

1. Discover the initial pattern instance $\hat{p}^I$ in A_1. Insert p^I into P and mark $\hat{p}^I$ as enabled. Insert the rule $\bot \rightarrow p^I$ into R.
2. For $i, 1 \leq i \leq n-1$:
   (a) Discover the new pattern instance $\hat{p}^3$ in A_{i+1} that extends A_i.
   (b) If $\hat{p}^3$ starts at the initial state q_0 of A_{i+1}, then it is an application of a rule of type (1). Insert p^3 into P and mark $\hat{p}^3$ as enabled, and add the rule $\bot \twoheadrightarrow p^3$ to R.

   (c) Otherwise ($\hat{p}^3$ does not start at q_0), find the unique enabled pattern $\hat{p} = \hat{p}^1 \odot \hat{p}^2$ in A_i s.t. $\hat{p}^3$'s initial state q is the join state of $\hat{p}$. Add p^1, p^2, and p^3 to P and p to P_c, and mark $\hat{p}^1$,$\hat{p}^2$, and $\hat{p}^3$ as enabled. If $\hat{p}^3$ is non-circular add the rule $p \twoheadrightarrow_s (p^1 \circ p^2) \propto p^3$ to R, otherwise add the rule $p \twoheadrightarrow_c (p^1 \odot p^2) \propto p^3$ to R.
3. Define Σ to be the set of symbols used by the patterns P.

Once we know the newly created pattern p^I or $\hat{p}^3$ (step 1 or 2a) and the pattern $\hat{p}$ that it is grafted onto (step 2c), creating the rule is straightforward. We elaborate below on the how the algorithm accurately finds these patterns.

Discovering new patterns $\hat{p}^I$ and $\hat{p}^3$ The first pattern p^I is easily discovered; it is A_1, the first DFA. To find those patterns added in subsequent DFAs, we need to isolate the pattern added between A_i and A_{i+1}, by identifying which states in $A_{i+1} = \langle \Sigma, q_0', Q', F', \delta' \rangle$ are 'new' relative to $A_i = \langle \Sigma, q_0, Q, F, \delta \rangle$. From the PRS definitions, we know that there is a subset of states and transitions in A_{i+1} that is isomorphic to A_i:

Definition 20. *(Existing states and transitions) For every $q' \in Q'$, we say that q'* exists *in A_i, with* parallel state *$q \in Q$, iff there exists a sequence $w \in \Sigma^*$ such that $q = \hat{\delta}(q_0, w)$, $q' = \hat{\delta}'(q_0, w)$, and neither is a sink reject state. Additionally, for every $q_1', q_2' \in Q'$ with parallel states $q_1, q_2 \in Q$, we say that $(q_1', \sigma, q_2') \in \delta'$* exists *in A_i if $(q_1, \sigma, q_2) \in \delta$.*

We refer to the states and transitions in A_{i+1} that do not exist in A_i as the *new* states and transitions of A_{i+1}, denoting them $Q_N \subseteq Q'$ and $\delta_N \subseteq \delta'$ respectively. By construction of PRSs, each state in A_{i+1} has at most one parallel state in A_i, and marking A_{i+1}'s existing states can be done in one simultaneous traversal of the two DFAs, using any exploration that covers all the states of A_i.

The new states are a new pattern instance $\hat{p}$ in A_{i+1}, excluding its initial and possibly its exit state. The initial state of $\hat{p}$ is the existing state $q_s' \in Q' \setminus Q_N$ that has outgoing new transitions. The exit state q_X' of $\hat{p}$ is identified by the following *Exit State Discovery* algorithm:

1. If q_s' has incoming new transitions, then $\hat{p}$ is circular: $q_X' = q_s'$. (Fig. 3(i), (ii)).
2. Otherwise p is non-circular. If $\hat{p}$ is the first (with respect to the DFA sequence) non-circular pattern to have been grafted onto q_s', then q_X is the unique new state whose transitions into A_{i+1} are the *connecting* transitions from Definition 17 (Fig. 3 (iii)).
3. If there is no such state then $\hat{p}$ is not the first non-circular pattern grafted onto q_s'. In this case, q_X' is the unique existing state $q_X' \neq q_s'$ with new incoming transitions but no new outgoing transitions. (Fig. 3(iv)).

Finally, the new pattern instance is $p = \langle \Sigma, q_s', Q_p, q_X', \delta_p \rangle$, where $Q_p = Q_N \cup \{q_s', q_X'\}$ and δ_p is the restriction of δ_N to the states of Q_p.

Discovering the pattern $\hat{p}$ Once we have found the pattern $\hat{p}^3$ in step 2a, we need to find the pattern $\hat{p}$ to which it has been grafted. We begin with some observations:

1. The join state of a composite pattern is always different from its initial and exit states (*edge states*): we cannot compose circular patterns, and there are no 'null' patterns.
2. For every two enabled pattern instances $\hat{p}, \hat{p}' \in \mathcal{I}_i$, $\hat{p} \neq \hat{p}'$, exactly 2 options are possible: either (a) every state they share is an edge state of at least one of them, or (b) one (p^s) is contained entirely in the other (p^c), and the containing pattern p^c is a composite pattern with join state q_j such that q_j is either one of p^s's edge states, or q_j is not in p^s at all.

Together, these observations imply that no two enabled pattern instances in a DFA can share a join state. We prove the second observation in Appendix A.

Finding the pattern $\hat{p}$ onto which $\hat{p}^3$ has been grafted is now straightforward. Denoting q_j as the parallel of $\hat{p}^3$'s initial state in A_i, we seek the enabled composite pattern instance $(p, q) \in \mathcal{I}_i$ for which $\text{join}(p, q, A_i) = q_j$. If none is present, we seek the only enabled instance $(p, q) \in \mathcal{I}_i$ that contains q_j as a non-edge state, but is not yet marked as a composite. (Note that if two enabled instances share a non-edge state, we must already know that the containing one is a composite, otherwise we would not have found and enabled the other).

5.1 Correctness

Definition 21. *A PRS* $\mathbf{P} = \langle \Sigma, P, P_c, R \rangle$ *is a* minimal generator (MG) *of a sequence of DFAs* $S = A_1, A_2, ...A_n$ *iff it is sufficient and necessary for that sequence, i.e.: 1. it generates* S*, 2. removing any rule* $r \in R$ *would render* $\mathbf{P}$ *insufficient for generating* S*, and 3. removing any element from* Σ, P, P_c *would make* $\mathbf{P}$ *no longer a PRS.*

Lemma 1. *Given a finite sequence of DFAs, the minimal generator of that sequence, if it exists, is unique.*

Theorem 1. *Let* $A_1, A_2, ...A_n$ *be a finite sequence of DFAs that has a minimal generator* $\mathbf{P}$*. Then the PRS Inference Algorithm will discover* $\mathbf{P}$*.*

The proofs for these claims are given in Appendix B.

5.2 Deviations from the PRS framework

Given a sequence of DFAs generated by the rules of PRS $\mathbf{P}$, the inference algorithm given above will faithfully infer $\mathbf{P}$ (Section 5.1). In practice however, we will want to apply the algorithm to a sequence of DFAs extracted from a trained RNN using the L^* algorithm (as in [27]). Such a sequence may contain noise: artifacts from an imperfectly trained RNN, or from the behavior of L^* (which does not necessarily create PRS-like sequences). The major deviations are incorrect pattern creation, simultaneous rule applications, and slow initiation.

Incorrect pattern creation Either due to inaccuracies in the RNN classification, or as artifacts of the L^* process, incorrect patterns are often inserted into the DFA sequence. Fortunately, the incorrect patterns that get inserted are somewhat random and so rarely repeat, and we can discern between the 'legitimate' and 'noisy' patterns being added to the DFAs using a *voting* and *threshold* scheme.

The *vote* for each discovered pattern $p \in P$ is the number of times it has been inserted as the new pattern between a pair of DFAs A_i, A_{i+1} in S. We set a *threshold* for the minimum vote a pattern needs to be considered valid, and only build rules around the connection of valid patterns onto the join states of other valid patterns. To do this, we modify the flow of the algorithm: before discovering rules, we first filter incorrect patterns.

We modify step 2 of the algorithm, splitting it into two phases: *Phase 1:* Mark the inserted patterns between each pair of DFAs, and compute their votes. Add to P those whose vote is above the threshold. *Phase 2:* Consider each DFA pair A_i, A_{i+1} in order. If the new pattern in A_{i+1} is valid, and its initial state's parallel state in A_i also lies in a valid pattern, then synthesize the rule adding that pattern according to the original algorithm in Section 5. Whenever a pattern is discovered to be composite, we add its composing patterns as valid patterns to P.

A major obstacle to our research was producing a high quality sequence of DFAs faithful to the target language, as almost every sequence produced has some noise. The voting scheme greatly extended the reach of our algorithm.

Simultaneous rule applications In the theoretical framework, A_{i+1} differs from A_i by applying a *single* PRS rule, and therefore q'_s and q'_X are uniquely defined. L^* however does not guarantee such minimal increments between DFAs. In particular, it may apply multiple PRS rules between two subsequent DFAs, extending A_i with several patterns. To handle this, we expand the initial and exit state discovery methods given in Section 5:

1. Mark the new states and transitions Q_N and δ_N as before.
2. Identify the *set* of new pattern instance initial states (*pattern heads*): the set $H \subseteq Q' \setminus Q_N$ of states in A_{i+1} with outgoing new transitions.
3. For each pattern head $q' \in H$, compute the *relevant* sets $\delta_{N|q'} \subseteq \delta_N$ and $Q_{N|q'} \subseteq Q_N$ of new transitions and states: the members of δ_N and Q_N that are reachable from q' *without passing through any existing transitions.*
4. For each $q' \in H$, restrict to $Q_{N|q'}$ and $\delta_{N|q'}$ and compute q'_X and p as before.

If A_{i+1}'s new patterns have no overlap and do not create an ambiguity around join states (e.g., do not both connect into instances of a single pattern whose join state has not yet been determined), then they may be handled independently and in arbitrary order. They are used to discover rules and then enabled, as in the original algorithm.

Simultaneous but dependent rule applications – such as inserting a pattern and then grafting another onto its join state – are more difficult to handle, as it is not always possible to determine which pattern was grafted onto which. However, there is a special case which appeared in several of our experiments (examples L13 ad L14 of Section 7) for which we developed a technique as follows:

Suppose we discover a rule $r_1 : p_0 \twoheadrightarrow_s (p_l \circ p_r) \propto p$, and p contains a cycle c around some internal state q_j. If later another rule inserts a pattern p_n at the state q_j, we understand that p is in fact a composite pattern, with $p = p_1 \circ p_2$ and join state q_j. However, as patterns do not contain cycles at their edge states, c cannot be a part of either p_1 or p_2. We conclude that the addition of p was in fact a simultaneous application of two rules: $r'_1 : p_0 \twoheadrightarrow_s (p_l \circ p_r) \propto p'$ and $r_2 : p' \twoheadrightarrow_c (p_1 \circ p_2) \propto c$, where p' is p without the cycle c, and update our PRS and our DFAs' enabled pattern instances accordingly. The case when p is circular is handled similarly.

Slow initiation Ideally, A_1 would directly supply an initial rule $\bot \twoheadrightarrow p^I$ to our PRS. In practice, we found that the first couple of DFAs generated by L^* – which deal with extremely short sequences – have completely incorrect structure, and it takes the algorithm some time to stabilise. Ultimately we solve this by leaving discovery of the initial rules to the *end* of the algorithm, at which point we have a set of 'valid' patterns that we are sure are part of the PRS. From there we examine the *last* DFA A_n generated in the sequence, note all the enabled instances (p^I, q_0) at its initial state, and generate a rule $\bot \twoheadrightarrow p^I$ for each of them. Note however that this technique will not recognise patterns p^I that do not also appear as an extending pattern p_3 elsewhere in the sequence (and therefore do not meet the threshold).

6 Converting a PRS to a CFG

We present an algorithm to convert a given PRS to a context free grammar (CFG), making the rules extracted by our algorithm more accessible.

A restriction: Let $\mathbf{P} = \langle \Sigma, P, P_c, R \rangle$ be a PRS. For simplicity, we restrict the PRS so that every pattern p can only appear on the LHS of rules of type (2) or only on the LHS of rules of type (3) but cannot only appear on the LHS of both types of rules. Similarly, we assume that for each rule $\bot \rightarrow p_I$, the RHS patterns p_I are all circular or non-circular[3]. In Appendix C.1 we show how to create a CFG without this restriction.

We will create a CFG $G = \langle \Sigma, N, S, Prod \rangle$, where Σ, N, S, and $Prod$ are the terminals (alphabet), non-terminals, start symbol and productions of the grammar. Σ is the same alphabet of $\mathbf{P}$, and we take S as a special start symbol. We now describe how we obtain N and $Prod$.

For every pattern $p \in P$, let $G_p = \langle \Sigma_p, N_p, Z_p, Prod_p \rangle$ be a CFG describing $L(p)$. Recall that P_C are composite patterns. Let $P_Y \subseteq P_C$ be those patterns that appear on the LHS of a rule of type (2) ($\twoheadrightarrow_c$). Create the non-terminal

[3] This restriction is natural: Dyck grammars and all of the examples in Sections 4.1 and 7.3 conform to this restriction.

C_S and for each $p \in P_Y$, create an additional non-terminal C_p. We set $N = \{S, C_S\} \bigcup\limits_{p \in P} \{N_p\} \bigcup\limits_{p \in P_Y} \{C_p\}$.

Let $\bot \twoheadrightarrow p_I$ be a rule in $\mathbf{P}$. If p_I is non-circular, create a production $S ::= Z_{p_I}$. If p_I is circular, create the productions $S ::= S_C$, $S_C ::= S_C S_C$ and $S_C ::= Z_{p_I}$.

For each rule $p \twoheadrightarrow_s (p_1 \circ p_2) \propto p_3$ create a production $Z_p ::= Z_{p_1} Z_{p_3} Z_{p_2}$. For each rule $p \twoheadrightarrow_c (p_1 \circ p_2) \propto p_3$ create the productions $Z_p ::= Z_{p_1} C_p Z_{p_2}$, $C_p ::= C_p C_p$, and $C_p ::= Z_{p_3}$. Let $Prod'$ be the all the productions defined by the above process. We set $Prod = \{\bigcup\limits_{p \in P} Prod_p\} \cup Prod'$.

Theorem 2. *Let G be the CFG constructed from $\mathbf{P}$ by the procedure given above. Then $L(\mathbf{P}) = L(G)$.*

The proof is given in Appendix C.

The class of languages expressible by a PRS Every RE-Dyck language (Section 2.2) can be expressed by a PRS. But the converse is not true; an RE-Dyck language requires that any delimiter pair can be embedded in any other delimiter pair while a PRS grammar provides more control over which delimiters can be embedded in which other delimiters. For instance, the language L12 of Section 7.3 contains 2 pairs of delimiters and only includes strings in which the first delimiter pair is embedded in the second delimiter pair and vice versa. L12 is expressible by a PRS but is not a Dyck language. Hence the class of PRS languages are more expressive than Dyck languages and are contained in the class of CFLs. But not every CFL can be expressed by a PRS. See Appendix C.3.

Succinctness The construction above does not necessarily yield a minimal CFG G equivalent to $\mathbf{P}$. For a PRS defining the Dyck language of order 2 - which can be expressed by a CFG with 4 productions and one non-terminal - our construction yields a CFG with 10 non-terminals and 12 productions.

In general, the extra productions can be necessary to provide more control over what delimiter pairs can be nested in other delimiter pairs as described above. However, when these productions are not necessary, we can often post-process the generated CFG to remove unnecessary productions. See Appendix C.2 for the CFGs generated for the Dyck language of order 2 and for the language of alternating delimiters.

7 Experimental results

7.1 Methodology

We test the algorithm on several PRS-expressible context free languages, attempting to extract them from trained RNNs using the process outlined in Figure 1. For each language, we create a probabilistic CFG generating it, train an RNN on samples from this grammar, extract a sequence of DFAs from the RNN, and apply our PRS inference algorithm[4]. Finally, we convert the extracted PRS back to a CFG, and compare it to our target CFG.

In all of our experiments, we use a vote-threshold s.t. patterns with less than 2 votes are not used to form any PRS rules (Section 5.2). Using no threshold significantly degraded the results by including too much noise, while higher thresholds often caused us to overlook correct patterns and rules.

7.2 Generating a sequence of DFAs

We obtain a sequence of DFAs for a given CFG using only positive samples[11,1] by training a *language-model RNN* (LM-RNN) on these samples and then extracting DFAs from it with the aid of the L^* algorithm [2], as described in [27]. To apply L^* we must treat the LM-RNN as a binary classifier. We set an 'acceptance threshold' t and define the RNN's language as the set of sequences s satisfying: 1. the RNN's probability for an end-of-sequence token after s is greater than t, and 2. at no point during s does the RNN pass through a token with probability $< t$. This is identical to the concept of *locally t-truncated support* defined in [13]. (Using the LM-RNN's probability for the entire sequence has the flaw that this decreases for longer sequences.)

To create the samples for the RNNs, we write a weighted version of the CFG, in which each non-terminal is given a probability over its rules. We then take N samples from the weighted CFG according to its distribution, split them into train and validation sets, and train an RNN on the train set until the validation loss stops improving. In our experiments, we used $N = 10,000$. For our languages, we used very small 2-layer LSTMs: hidden dimension 10 and input dimension 4.

In some cases, especially when all of the patterns in the rules are several tokens long, the extraction of [27] terminates too soon: neither L^* nor the RNN abstraction consider long sequences, and equivalence is reached between the L^* hypothesis and the RNN abstraction despite neither being equivalent to the 'true' language of the RNN. In these cases we push the extraction a little further using two methods: first, if the RNN abstraction contains only a single state, we make an arbitrary initial refinement by splitting 10 hidden dimensions, and restart the extraction. If this is also not enough, we sample the RNN according to its distribution, in the hope of finding a counterexample to return to L^* . The latter approach is not ideal: sampling the RNN may return very long sequences, effectively increasing the next DFA by many rule applications.

In other cases, the extraction is long, and slows down as the extracted DFAs grow. We place a time limit of $1,000$ seconds (~ 17 minutes) on the extraction.

7.3 Languages

We experiment on 15 PRS-expressible languages $L_1 - L_{15}$, grouped into 3 classes:

[4] The implementation needs some expansions to fully apply to complex multi-composed patterns, but is otherwise complete and works on all languages described here.

LG	DFAs	Init Pats	Final Pats	Min/Max Votes	CFG Correct	LG	DFAs	Init Pats	Final Pats	Min/Max Votes	CFG Correct
L_1	18	1	1	16/16	Correct	L_9	30	6	4	5/8	Correct
L_2	16	1	1	14/14	Correct	L_{10}	6	2	1	3/3	Correct
L_3	14	6	4	2/4	Incorrect	L_{11}	24	6	3	5/12	Incorrect
L_4	8	2	1	5/5	Correct	L_{12}	28	2	2	13/13	Correct
L_5	10	2	1	7/7	Correct	L_{13}	9	6	1	2/2	Correct
L_6	22	9	4	3/16	Incorrect	L_{14}	17	5	2	5/7	Correct
L_7	24	2	2	11/11	Correct	L_{15}	13	6	4	3/6	Incorrect
L_8	22	5	4	2/9	Partial						

Table 1. Results of experiments on DFAs extracted from RNNs

1. Languages of the form $\mathtt{X}^n\mathtt{Y}^n$, for various regular expressions `X` and `Y`. In particular, the languages L_1 through L_6 are $\mathtt{X}_i^n\mathtt{Y}_i^n$ for: $(\mathtt{X}_1,\mathtt{Y}_1)$`=(a,b)`, $(\mathtt{X}_2,\mathtt{Y}_2)$`=(a|b,c|d)`, $(\mathtt{X}_3,\mathtt{Y}_3)$`=(ab|cd,ef|gh)`, $(\mathtt{X}_4,\mathtt{Y}_4)$`=(ab,cd)`, $(\mathtt{X}_5,\mathtt{Y}_5)$`=(abc,def)`, and $(\mathtt{X}_6,\mathtt{Y}_6)$`=(ab|c,de|f)`.
2. Dyck and RE-Dyck languages, excluding the empty sequence. In particular, languages L_7 through L_9 are the Dyck languages (excluding ε) of order 2 through 4, and L_{10} and L_{11} are RE-Dyck languages of order 1 with the delimiters $(\mathtt{L}_{10},\mathtt{R}_{10})$`=(abcde,vwxyz)` and $(\mathtt{L}_{11},\mathtt{R}_{11})$`=(ab|c,de|f)`.
3. Variations of the Dyck languages, again excluding the empty sequence. L_{12} is the language of alternating single-nested delimiters, generating only sequences of the sort `([([])])` or `[([])]`. L_{13} and L_{14} are Dyck-1 and Dyck-2 with additional neutral tokens `a,b,c` that may appear multiple times anywhere in the sequence. L_{15} is like L_{13} except that the neutral additions are the token `d` and the sequence `abc`, eg: `(abc()())d` is in L_{15}, but `a(bc()())d` is not.

7.4 Results

Table 1 shows the results. The 2nd column shows the number of DFAs extracted from the RNN. The 3rd and 4th columns present the number of patterns found by the algorithm before and after applying vote-thresholding to remove noise. The 5th column gives the minimum and maximum votes received by the final patterns[5]. The 6th column notes whether the algorithm found a correct CFG, according to our manual inspection. For languages where our algorithm only missed or included 1 or 2 valid/invalid productions, we label it as partially correct.

Alternating Patterns Our algorithm struggled on the languages L_3, L_6, and L_{11}, which contained patterns whose regular expressions had alternations (such as `ab|cd` in L_3, and `ab|c` in L_6 and L_{11}). Investigating their DFA sequences uncovered the that the L^* extraction had 'split' the alternating expressions, adding their parts to the DFAs over multiple iterations. For example, in the

[5] We count only patterns introduced as a new pattern p^3 in some A_{i+1}; if $p^3 = p^4 \circ p^5$, but p^4 is not introduced independently as a new pattern, we do not count it.

sequence generated for L_3, `ef` appeared in A_7 without `gh` alongside it. The next DFA corrected this mistake but the inference algorithm could not piece together these two separate steps into a single rule. It will be valuable to expand the algorithm to these cases.

Simultaneous Applications Originally our algorithm failed to accurately generate L_{13} and L_{14} due to simultaneous rule applications. However, using the technique described in Section 5.2 we were able to correctly infer these grammars. However, more work is needed to handle simultaneous rule applications in general.

Additionally, sometimes a very large counterexample was returned to L^* , creating a large increase in the DFAs: the 9thiteration of the extraction on L_3 introduced almost 30 new states. The algorithm does not manage to infer anything meaningful from these nested, simultaneous applications.

Missing Rules For the Dyck languages $L_7 - L_9$, the inference algorithm was mostly successful. However, due to the large number of possible delimiter combinations, some patterns and nesting relations did not appear often enough in the DFA sequences. As a result, for L_8, some productions were missing in the generated grammar. L_8 also created one incorrect production due to noise in the sequence (one erroneous pattern was generated two times). When we raised the threshold to require more than 2 occurrences to be considered a valid pattern we no longer generated this incorrect production.

RNN Noise In L_{15}, the extracted DFAs for some reason always forced that a single character `d` be included between every pair of delimiters. Our inference algorithm of course maintained this peculiarity. It correctly allowed the allowed optional embedding of "abc" strings. But due to noisy (incorrect) generated DFAs, the patterns generated did not maintain balanced parenthesis.

8 Related work

Training RNNs to recognize Dyck Grammars. Recently there has been a surge of interest in whether RNNs can learn Dyck languages [5,20,22,28]. While these works report very good results on learning the language for sentences of similar distance and depth as the training set, with the exception of [22], they report significantly less accuracy for out-of-sample sentences.

Sennhauser and Berwick [20] use LSTMs, and show that in order to keep the error rate with a 5 percent tolerance, the number of hidden units must grow exponentially with the distance or depth of the sequences[6]. They also found that out-of-sample results were not very good. They conclude that LSTMs cannot learn rules, but rather use statistical approximation. Bernardy [5] experimented

[6] However, see [13] where they show that a Dyck grammar of k pairs of delimiters that generates sentences of maximum depth m, only $3m\lceil\log k\rceil - m$ hidden memory units are required, and show experimental results that confirm this theoretical bound in practice.

with various RNN architectures. When they test their RNNs on strings that are at most double in length of the training set, they found that for out-of-sample strings, the accuracy varies from about 60 to above 90 percent. The fact that the LSTM has more difficulty in predicting closing delimiters in the middle of a sentence than at the end leads Bernardy to conjecture that for closing parenthesis the RNN is using a counting mechanism, but has not truly learnt the Dyck language (its CFG). Skachkova, Trost and Klakow [22] experiment with Ellman-RNN, GRU and LSTM architectures. They provide a mathematical model for the probability of a particular symbol in the i^{th} position of a Dyck sentence. They experiment with how well the models predict the closing delimiter, which they find varying results per architecture. However, for LSTMs, they find nearly perfect accuracy across words with large distances and embedded depth.

Yu, Vu and Kuhn [28] compares the three works above and argue that the task of predicting a closing bracket of a balanced Dyck word, as performed in [22], is a poor test for checking if the RNN learnt the language, as it can be simply computed by a counter. In contrast, their carefully constructed experiments give a prefix of a Dyck word and train the RNN to predict the next valid closing bracket. They experiment with an LSTM using 4 different models, and show that the generator-attention model [18] performs the best, and is able to generalize quite well at the tagging task . However, when using RNNs to complete the entire Dyck word, while the generator-attention model does quite well with in-domain tests, it degrades rapidly with out-of-domain tests. They also conclude that RNNs do not really learn the CFG underlying the Dyck language. These experimental results are reinforced by the theoretical work in [13]. They remark that no finite precision RNN can learn a Dyck language of unbounded depth, and give precise bounds on the memory required to learn a Dyck language of bounded depth.

In contrast to these works, our research tries to extract the CFG from the RNN. We discover these rules based upon DFAs synthesized from the RNN using the algorithm in [27]. Because we can use a short sequence of DFAs to extract the rules, and because the first DFAs in the sequence describe Dyck words with increasing but limited distance and depth, we are able to extract the CFG perfectly, even when the RNN does not generalize well. Moreover, we show that our approach generalizes to more complex types of delimiters, and to Dyck languages with expressions between delimiters.

Extracting DFAs from RNNs. There have been many approaches to extract higher level representations from a neural network (NN) to facilitate comprehension and verify correctness. One of the oldest approaches is to extract rules from a NN [25,12]. In order to model state, there have been various approaches to extract FSA from RNNs [19,15,26]. We base our work on [27]. Its ability to generate sequences of DFAs that increasingly better approximate the CFL is critical to our method.

Unlike DFA extraction, there has been relatively little research on extracting a CFG from an RNN. One exception is [24], where they develop a Neural Network Pushdown Automata (NNPDA) framework, a hybrid system augmenting an RNN with external stack memory. The RNN also reads the top of the stack as added input, optionally pushes to or pops the stack after each new input symbol. They show how to extract a Push-down Automaton from a NNPDA, however, their technique relies on the PDA-like structure of the inspected architecture. In contrast, we extract CFGs from RNNs without stack augmentation.

Learning CFGs from samples. There is a wide body of work on learning CFGs from samples. An overview is given in [10] and a survey of work for grammatical inference applied to software engineering tasks can be found in [23].

Clark et. al. studies algorithms for learning CFLs given only positive examples [11]. In [7], Clark and Eyraud show how one can learn a subclass of CFLs called *CF substitutable* languages. There are many languages that can be expressed by a PRS but are not substitutable, such as $x^n b^n$. However, there are also substitutable languages that cannot be expressed by a PRS (wxw^R - see Appendix C.3). In [8], Clark, Eyraud and Habrard present Contextual Binary Feature Grammars. However, it does not include Dyck languages of arbitrary order. None of these techniques deal with noise in the data, essential to learning a language from an RNN. While we have focused on practical learning of CFLs, theoretical limits on learning based upon positive examples is well known; see[11,1].

9 Future Directions

Currently, for each experiment, we train the RNN on that language and then apply the PRS inference algorithm on a single DFA sequence generated from that RNN. Perhaps the most substantial improvement we can make is to extend our technique to learn from multiple DFA sequences. We can train multiple RNNs (each one based upon a different architecture if desired) and generate DFA sequences for each one. We can then run the PRS inference algorithm on each of these sequences, and generate a CFG based upon rules that are found in a significant number of the runs. This would require care to guarantee that the final rules form a cohesive CFG. It would also address the issue that not all rules are expressed in a single DFA sequence, and that some grammars may have rules that are executed only once per word of the language.

Our work generates CFGs for generalized Dyck languages, but it is possible to generalize PRSs to express a greater range of languages. Work will be needed to extend the PRS inference algorithm to reconstruct grammars for all context-free and perhaps even some context-sensitive languages.

A Observation on PRS-Generated Sequences

We present and prove an observation on PRS-generated sequences used for deriving the PRS-inference algorithm (Section 5).

Lemma 2. *Let $\langle A_i, \mathcal{I}_i \rangle$ be a PRS-generated enabled DFA. Then for every two enabled pattern instances $\hat{p}, \hat{p}' \in \mathcal{I}_i, \hat{p} \neq \hat{p}'$, exactly 2 options are possible: 1. every state they share is the initial or exit state (*edge state*) of at least one of them, or*

2. one ($\hat{p}^s$) is contained entirely in the other ($\hat{p}^c$), and $\hat{p}^c$ is a composite pattern with join state q_j such that either q_j is one of $\hat{p}^s$'s edge states, or q_j is not in $\hat{p}^s$ at all.

Proof. We prove by induction. For $\langle A_1, \mathcal{I}_1 \rangle$, $|\mathcal{I}_1| \leq 1$ and the lemma holds vacuously. We now assume it is true for $\langle A_i, \mathcal{I}_i \rangle$.

Applying a rule of type (1) adds only one new instance $\hat{p}^I$ to $\mathcal{I}_{i+1}$, which shares only its initial state with the existing patterns, and so option 1 holds.

Rules of type (2) and (3) add up to three new enabled instances, $\hat{p}^1, \hat{p}^2$, and $\hat{p}^3$, to $\mathcal{I}_{i+1}$. $\hat{p}^3$ only shares its edge states with A_i, and so option (1) holds between $\hat{p}^3$ and all existing instances $\hat{p}' \in \mathcal{I}_i$, as well as the new ones $\hat{p}^1$ and $\hat{p}^2$ if they are added (as their states are already contained in A_i).

We now consider the case where $\hat{p}^1$ and $\hat{p}^2$ are also newly added (i.e. $\hat{p}^1, \hat{p}^2 \notin \mathcal{I}_i$). We consider a pair $\hat{p}^i, \hat{p}'$ where $i \in \{1, 2\}$. As $\hat{p}^1$ and $\hat{p}^2$ only share their join states with each other, and both are completely contained in $\hat{p}$ such that $\hat{p}$'s join state is one of their edge states, the lemma holds for each of $\hat{p}' \in \{\hat{p}^1, \hat{p}^2, \hat{p}\}$. We move to $\hat{p}' \neq \hat{p}^1, \hat{p}^2, \hat{p}$. Note that (i) $\hat{p}'$ cannot be contained in $\hat{p}$, as we are only now splitting $\hat{p}$ into its composing instances, and (ii), if $\hat{p}$ shares any of its edge states with $\hat{p}^i$, then it must also be an edge state of $\hat{p}^i$ (by construction of composition).

As $\hat{p}^i$ is contained in $\hat{p}$, the only states that can be shared by $\hat{p}^i$ and $\hat{p}'$ are those shared by $\hat{p}$ and $\hat{p}'$. If $\hat{p}, \hat{p}'$ satisfy option 1, i.e., they only share edge states, then this means any states shared by $\hat{p}'$ and $\hat{p}^i$ are edge states of $\hat{p}'$ or $\hat{p}$. Clearly, $\hat{p}'$ edge states continue to be $\hat{p}'$ edge states. As for each of $\hat{p}$'s edge states, by (ii), it is either not in $\hat{p}^i$, or necessarily an edge state of $\hat{p}^i$. Hence, if $\hat{p}, \hat{p}'$ satisfy option 1, then $\hat{p}^i, \hat{p}'$ do too.

Otherwise, by the assumption on $\langle A_i, \mathcal{I}_i \rangle$, option 2 holds between $\hat{p}'$ and $\hat{p}$, and from (i) $\hat{p}'$ is the containing instance. As $\hat{p}^i$ composes $\hat{p}$, then $\hat{p}'$ also contains $\hat{p}^i$. Moreover, by definition of option 2, the join state of $\hat{p}'$ is either one of $\hat{p}$'s edge states or not in $\hat{p}$ at all, and so from (ii) the same holds for $\hat{p}^i$. □

B Correctness of the Inference Algorithm

Lemma 1. *Given a finite sequence of DFAs, the minimal generator of that sequence, if it exists, is unique.*

Proof: Say that there exists two MGs, $\mathbf{P}_1 = \langle \Sigma^1, P^1, P_c^1, R^1 \rangle$ and $\mathbf{P}_2 = \langle \Sigma^2, P^2, P_c^2, R^2 \rangle$ that generate the sequence $A_1, A_2, \cdots, A_n$. Certainly $\Sigma^1 = \Sigma^2 = \bigcup_{i \in [n]} \Sigma^{A_i}$.

We show that $R^1 = R^2$. Say that the first time MG1 and MG2 differ from one another is in explaining which rule is used when expanding from A_i to A_{i+1}. Since MG1 and MG2 agree on all rules used to expand the sequence prior to A_{i+1}, they agree on the set of patterns enabled in A_i. If this expansion is adding a pattern p_3 originating at the initial state of the DFA, then it can only be explained by a single rule $\bot \twoheadrightarrow p_3$, and so the explanation of MG1 and MG2 is identical. Hence the expansion must be created by a rule of type (2) or (3). Since the newly added pattern instance $\hat{p}^3$ is is uniquely identifiable in A_{i+1}, $\mathbf{P}_1$ and $\mathbf{P}_2$ must agree on the pattern p^3 that appears on the RHS of the rule explaining this expansion. $\hat{p}^3$ is inserted at some state q_j of A_i. q_j must be the join state of an enabled pattern instance $\hat{p}$ in A_i. But this join state uniquely identifies that pattern: as noted in Section 5, no two enabled patterns in a enabled DFA share a join state. Hence $\mathbf{P}_1$ and $\mathbf{P}_2$ must agree that the pattern $p = p^1 \circ p^2$ is the LHS of the rule, and they therefore agree that the rule is $p \twoheadrightarrow_s (p^1 \circ p^2) \propto p^3$, if p^3 is non-circular, or $p \twoheadrightarrow_c (p^1 \odot p^2) \propto p^3$ if p_3 is circular. Hence $R^1 = R^2$.

Since $\mathbf{P}_1$ ($\mathbf{P}_2$) is an MG, it must be that $p \in P^1$ ($p \in P^2$) iff p appears in a rule in R^1 (R^2). Since $R^1 = R^2$, $P^1 = P^2$. Furthermore, a pattern $p \in P_c$ iff it appears on the LHS of a rule. Therefore $P_c^1 = P_c^2$. □

Theorem 1. *Let $A_1, A_2, ...A_n$ be a finite sequence of DFAs that has a minimal generator $\mathbf{P}$. Then the PRS Inference Algorithm will discover $\mathbf{P}$.*

Proof: This proof mimics the proof in the Lemma above. In this case $\mathbf{P}_1 = \langle \Sigma^1, P^1, P_c^1, R^1 \rangle$ is the MG for this sequence and $\mathbf{P}_2 = \langle \Sigma^2, P^2, P_c^2, R^2 \rangle$ is the PRS discovered by the PRS inference algorithm.

We need to show that the PRS inference algorithm faithfully follows the steps above for $\mathbf{P}_2$. This straightforward by comparing the steps of the inference algorithm with the steps for $\mathbf{P}_2$. One subtlety is to show that the PRS inference algorithm correctly identifies the new pattern $\hat{p}^3$ in A_{i+1} extending A_i. The algorithm easily finds all the newly inserted states and transitions in A_{i+1}. All of the states, together with the initial state, must belong to the new pattern. However not all transitions necessarily belong to the pattern. The Exit State Discovery algorithm of Section 5 correctly differentiates between new transitions that are part of the inserted pattern and those that are *connecting transitions* (The set C of Definition 17). Hence the algorithm correctly finds the new pattern in A_{i+1}. □

C The expressibility of a PRS

We present a proof to Theorem 2 showing that the CFG created from a PRS expresses the same language.

Theorem 2. *Let G be the CFG constructed from $\mathbf{P}$ by the procedure given in Section 6. Then $L(\mathbf{P}) = L(G)$.*

Proof: Let $s \in L(\mathbf{P})$. Then there exists a sequence of DFAs $A_1 \cdots A_m$ generated by $\mathbf{P}$ s.t. $s \in L(A_m)$. We will show that $s \in L(G)$. W.l.g. we assume that each DFA in the sequence is necessary; i.e., if the rule application to A_i creating A_{i+1} were absent, then $s \notin L(A_m)$. We will use the notation $\hat{p}$ to refer to a specific instance of a pattern p in A_i for some i ($1 \leq i \leq m$), and we adopt from Section 4 the notion of *enabled* pattern instances. So, for instance, if we apply a rule $p \twoheadrightarrow_s (p^1 \circ p^2) \propto p^3$, where $p = p_1 \circ p_2$, to an instance of $\hat{p}$ in A_i, then A_{i+1} will contain a new path through the enabled pattern instances $\hat{p}^1, \hat{p}^2$ and $\hat{p}^3$.

A *p-path* (short for *pattern-path*) through a DFA A_i is a path $\rho = q_0 \rightarrow^{p_1} q_1 \rightarrow^{p_2} \cdots q_{t-1} \rightarrow^{p_t} q_t$, where q_0 and q_t are the initial and final states of A_i respectively, and for each transition $q_j \rightarrow^{p_{j+1}} q_{j+1}$, q_j $(0 \leq j \leq t-1)$ is the initial state of an enabled pattern instance of type p_{j+1} and q_{j+1} is the final state of that pattern instance. A state may appear multiple times in the path if there is a cycle in the DFA and that state is traversed multiple times. If $\hat{p}$ is an enabled circular pattern and the path contains a cycle that traverses that instance of p, and only that instance, multiple times consecutively, it is only represented once in the path, since that cycle is completely contained within that pattern; a p-path cannot contain consecutive self-loops $q_j \rightarrow^{p} q_j \rightarrow^{p} q_j$. $Pats(\rho) = p_1 p_2 \cdots p_t$, the instances of the patterns traversed along the path ρ.

We say that a p-path $\rho = q_0 \rightarrow^{p_1} q_1 \rightarrow^{p_2} \cdots q_{t-1} \rightarrow^{p_t} q_t$ through A_m is an *acceptor* (of s) iff $s = s_1 \cdots s_t$ and $s_i \in L(p_i)$ for all i $(1 \leq i \leq t)$. DFAs earlier in the sequence are not acceptors as they contain patterns that have not yet been expanded. But we can "project" the final p-path onto a p-path in an earlier DFA. We do so with the following definition of a *p-cover*:

- If a path ρ is an acceptor, then it is a p-cover.
- Let p be a pattern and let A_{i+1} be obtained from A_i by application of the rule $p \twoheadrightarrow_s (p^1 \circ p^2) \propto p^3$ or $p \twoheadrightarrow_c (p^1 \odot p^2) \propto p^3$ to $\hat{p}$ in A_i obtaining a sub-path $q_1 \rightarrow^{p_1} q_3 \rightarrow^{p_3} q_4 \rightarrow^{p_2} q_2$ through instances $\hat{p}^1, \hat{p}^2$ and $\hat{p}^3$. Furthermore, say that the p-path $\rho^{(i+1)}$ through A_{i+1} is a p-cover. Then the path $\rho^{(i)}$ through A_i is p-cover, where $\rho^{(i)}$ is obtained from $\rho^{(i+1)}$ by replacing each occurrence of $q_1 \rightarrow^{p_1} q_3 \rightarrow^{p_3} q_4 \rightarrow^{p_2} q_2$ in $\rho^{(i+1)}$ traversing $\hat{p}^1, \hat{p}^3$ and $\hat{p}^2$ by the single transition $q_1 \rightarrow^{p} q_2$ traversing $\hat{p}$ in $\rho^{(i)}$. (If p is circular then $q_1 = q_2$). If this results in consecutive self loops $q_1 \rightarrow^{p} q_1 \rightarrow^{p} q_1$ we collapse them into a single cycle, $q_1 \rightarrow^{p} q_1$.
- Let A_{i+1} be obtained by applying a rule $\perp \twoheadrightarrow p^I$ to A_i obtaining an instance of $\hat{p}^I$, where p^I is a circular pattern (Defn. 15). Furthermore, say that the p-path $\rho^{(i+1)}$ through A_{i+1} is a p-cover. Then the path $\rho^{(i)}$ through A_i is p-cover, where $\rho^{(i)}$ is obtained from $\rho^{(i+1)}$ by replacing each occurrence of $q_0 \rightarrow^{p^I} q_0$ traversing $\hat{p}^I$ by the single state q_0.

Hence we can associate with each $A_i, 1 \leq i \leq m$ a unique p-cover $\rho^{(i)}$.

Let $\mathcal{T}$ be a partial derivation tree for the CFG G, where every branch of the tree terminates with a non-terminal Z_p for some pattern p. We write $\hat{Z}_p$ for a particular instance of Z_p in $\mathcal{T}$. $Leaves(\mathcal{T})$ is the list of patterns obtained by concatenating all the leaves (left-to-right) in $\mathcal{T}$ and replacing each leaf Z_{p_k} by the pattern p_k.

We claim that for each A_i with p-cover $\rho^{(i)}$ there exists a partial derivation tree $\mathcal{T}^{(i)}$ such that $Pats(\rho^{(i)}) = Leaves(\mathcal{T}^{(i)})$. We show this by induction.

For the base case, consider A_1, which is formed by application of a rule $\perp \twoheadrightarrow p^I$. By construction of G, there exists a production $S ::= Z_{p^I}$. $\rho^{(1)} = s_0 \rightarrow^{p^I} s_f$, where S_0 and s_f are the initial and final states of p^I respectively, and let $\mathcal{T}^{(1)}$ be the tree formed by application of the production $S ::= Z_{p^I}$ creating the instance $\hat{Z_{p^I}}$. Hence $Pats(\rho^{(1)}) = p^I = Leaves(\mathcal{T}^{(1)})$.

For the inductive step assume that for A_i there exists $\mathcal{T}^{(i)}$ s.t. $Pats(\rho^{(i)}) = Leaves(\mathcal{T}^{(i)})$. Say that A_{i+1} is formed from A_i by applying the rule $p \twoheadrightarrow_c (p^1 \odot p^2) \propto p^3$ (of type (2)) or $p \twoheadrightarrow_s (p^1 \circ p^2) \propto p^3$ (of type (3)) to an instance $\hat{p}$ of p in A_i, where the initial state of $\hat{p}$ is q_1 and its final state is q_2 ($q_1 = q_2$ if p is circular) and there is a sub-path in A_i of the form $q_1 \rightarrow^p q_2$. After applying this rule there is an additional sub-path $q_1 \rightarrow^{p_1} q_3 \rightarrow^{p_3} q_4 \rightarrow^{p_2} q_2$ in A_{i+1} traversing $\hat{p}^1, \hat{p}^3$ and $\hat{p}^2$. We consider two cases:

Case 1. p is non-circular. The sub-path $q_1 \rightarrow^p q_2$ may appear multiple times in $\rho^{(i)}$ even though p is non-circular, since it may be part of a larger cycle. Consider one of these instances where $q_1 \rightarrow^p q_2$ gets replaced by $q_1 \rightarrow^{p_1} q_3 \rightarrow^{p_3} q_4 \rightarrow^{p_2} q_2$ in $\rho^{(i+1)}$. Say that this instance of $\hat{p}$ is represented by pattern p at position u in $Pats(\rho^{(i)})$. In $\rho^{(i+1)}$, the sub-list of patterns p_1, p_3, p_3 will replace p at that position (position u). By induction there is a pattern p in $Leaves(\mathcal{T}^{(i)})$ at position u and let $\hat{Z}_p$ be the non-terminal instance in $\mathcal{T}^{(i)}$ corresponding to that pattern p. If the rule being applied is of type (3) then, by construction of G, there exists a production $Z_p ::= Z_{p_1} Z_{p_3} Z_{p_2}$. We produce $\mathcal{T}^{(i+1)}$ by extending $\mathcal{T}^{(i)}$ at that instance of Z_p by applying that production to $\hat{Z}_p$. If the rule is of type (2), then we produce $\mathcal{T}^{(i+1)}$ by extending $\mathcal{T}^{(i)}$ at that instance of Z_p by applying the productions $Z_p ::= Z_{p_1} C_p Z_{p_2}$ and $C_p ::= Z_{p_3}$, which exist by the construction of G. Hence both $Pats(\rho^{(i+1)})$ and $Leaves(\mathcal{T}^{(i+1)})$ will replace p at position u by p_1, p_3, p_3. We do this for each traversal of $\hat{p}$ in $\rho^{(i)}$ that gets replaced in $\rho^{(i+1)}$ by the traversal of $\hat{p}^1, \hat{p}^3$, and $\hat{p}^2$. By doing so, $Pats(\rho^{(i+1)}) = Leaves(\mathcal{T}^{(i+1)})$.

Case 2: p is circular. This is similar to the previous case except this time, since p is circular, we may need to replace a single sub-path $q_1 \rightarrow^p q_1$ corresponding to an instance of $\hat{p}$ in $\rho^{(i)}$ by multiple explicit cycles as defined by $\rho^{(i+1)}$. Each cycle will either traverse $q_1 \rightarrow^p q_1$ or the longer sub-path $q_1 \rightarrow^{p_1} q_3 \rightarrow^{p_3} q_4 \rightarrow^{p_2} q_1$.

Say that there exists an instance $\hat{p}$ represented by pattern p at position u in $Pats(\rho^{(i)})$ that gets replaced in $\rho^{(i+1)}$ by explicit cycles; i.e., $\rho^{(i+1)}$ replaces $q_1 \rightarrow^p q_1$ traversing $\hat{p}$ in $\rho^{(i)}$ with a new sub-path σ in $\rho^{(i+1)}$ containing x cycles $q_1 \rightarrow^{p_1} q_3 \rightarrow^{p_3} q_4 \rightarrow^{p_2} q_1$ interspersed with y cycles $q_1 \rightarrow^p q_1$, where $p = p_1 \circ_c p_2$. (Per definition of a p-path, there cannot be two consecutive instances of these latter cycles). Hence in total σ may enter and leave q_1 a total of $z = x + y$ times. By induction there is a pattern p in $Leaves(\mathcal{T}^{(i)})$ at position u and let $\hat{Z}_p$ be the non-terminal instance in $\mathcal{T}^{(i)}$ corresponding to that pattern p. By construction of G, since p is circular, the parent of $\hat{Z}_p$ is an instance $\hat{C_{p'}}$ of the non-terminal $C_{p'}$ for some pattern p' and there exists productions $C_{p'} ::= C_{p'} C_{p'}$, and $C_{p'} ::= Z_p$. Using these productions we replace this single instance $\hat{C_{p'}}$ by z copies of $C_{p'}$. If the j^{th} cycle of σ is $q_1 \rightarrow^p q_1$ then we have the j^{th} instance of $C_{p'}$ derive Z_p without any further derivations. If the j^{th} cycle is $q_1 \rightarrow^{p_1} q_3 \rightarrow^{p_3} q_4 \rightarrow^{p_2} q_1$, then we also have the j^{th} instance of $C_{p'}$ derive Z_p. However, if the rule being applied is of type (3) then that instance of Z_p derives $Z_{p_1} Z_{p_3} Z_{p_2}$. If it is of type (2) then that instance of Z_p derives $Z_{p_1} C_p Z_{p_2}$ and C_p derives Z_{p_3}. Hence both $Pats(\rho^{(i)})$ and $Leaves(\mathcal{T}^{(i)})$ will replace p at position u by x copies of p_1, p_3, p_3 intermixed with y copies of p. We do this for each traversal of $\hat{p}$ in $\rho^{(i)}$ that gets expanded in $\rho^{(i+1)}$ by application of this rule. By doing so, $Pats(\rho^{(i+1)}) = Leaves(\mathcal{T}^{(i+1)})$.

To complete the inductive step, we need to consider the case when A_{i+1} is formed from A_i by applying a rule $\bot \twoheadrightarrow p^{I'}$, where $p^{I'}$ is circular, per Defn. 15. This will insert $p^{I'}$ into $Pats(\rho^{(i+1)})$ at a point when $\rho^{(i)}$ is at the initial state q_0. Say that there exists a sub-path $\sigma = q_0 \rightarrow^{p_1} q_1 \rightarrow^{p_2} \cdots q_e \rightarrow^{p_e} q_0$ in $\rho^{(i)}$. Then the application of this rule may add the sub-path $q_0 \rightarrow^{p^{I'}} q_0$ either at the beginning or end of σ in $\rho^{(i+1)}$. W.l.g. assume it gets asserted at the end of this sub-path, and p_e occurs at position u. Then $Pats(\rho^{(i+1)})$ will extend $Pats(\rho^{(i)})$ by inserting $p^{I'}$ at position $u+1$ in $\rho^{(i)}$. Since σ is a cycle, starting and ending at q_0, there must be an instance $\hat{C_S}$ of C_S in $\mathcal{T}^{(i)}$ where C_S is derived by one or more productions of the form $S ::= C_S$ and $C_S ::= C_S\ C_S$. Furthermore, $\hat{C_S}$ derives a sub-tree T s.t. $Leaves(T) = Pats(\sigma)$. By construction of G, there exists a production $C_S ::= C_{p'_I}$. We add the production $C_S ::= C_S\ C_S$ to $\hat{C_S}$ so that the first child C_S derives T as in $\mathcal{T}^{(i)}$. At the second instance we apply the production $C_S ::= C_{p'_I}$. Hence p'_I will appear at position $u+1$ in $\mathcal{T}^{(i+1)}$. We repeat this for each cycle involving q_0 in $\rho^{(i)}$ that gets extended by the pattern $p^{I'}$ in $\rho^{(i+1)}$. By doing so, $Pats(\rho^{(i+1)}) = Leaves(\mathcal{T}^{(i+1)})$. A similar argument holds if $p^{I'}$ is added to the first position in $Pats(\rho^{(i+1)})$.

Hence we have shown that $Pats(\rho^{(m)}) = Leaves(\mathcal{T}^{(m)})$. Let $Pats(\rho^{(m)}) = p_1 \cdots p_t$. Since $\rho^{(m)}$ is an acceptor for s, it must be that there exists $s_j \in \Sigma^+$ $(1 \leq j \leq t)$ s.t. $s_j \in L(p_j)$ and $s = s_1 \cdots s_t$. But since $Leaves(\mathcal{T}^{(m)}) = Z_{p_1} \cdots Z_{p_t}$ and each Z_{p_j} can derive s_j, we can complete the derivation of $\mathcal{T}^{(m)}$ to derive s. This shows that $s \in L(PR) \implies s \in L(G)$. The converse is also true and can be shown by similar technique so we leave the proof to the reader. □

C.1 Constructing a CFG from an unrestricted PRS

The construction of Section 6 assumed a restriction that a pattern p cannot appear on the LHS of rules of type (2) and of type (3). I.e., we cannot have two rules of the form $p \twoheadrightarrow_c (p^1 \odot p^2) \fatsemi p^3$ and $p \twoheadrightarrow_s (p^1 \circ p^2) \fatsemi p^3$. If we were to allow both of these rules then one could construct a path through a DFA instance that first traverses an instance of p_1, then traverses instance of the circular pattern p'_3 any number of times, then traverses an instance of p_3, and then traverses p_2. However the current grammar does not allow such constructions; the non-terminal Z_p can *either* derive Z_{p_1} followed by Z_{p_3} followed by Z_{p_2} or, in place of Z_{p_3}, any number of instances of C_p that in turn derives $Z_{p'_3}$.

Hence to remove this restriction, we modify the constructed CFG. Following Section 6, for every pattern $p \in P$, G_p is the CFG with Start symbol Z_p and non-terminals N_p. P_Y are the patterns appearing on the LHS of some rule of type (2). Given the PRS $\mathbf{P} = \langle \Sigma, P, P_C, R \rangle$ we create a CFG $G = (\Sigma, N, S, Prod)$, where $N = \{S, C_S, C'_S\} \bigcup_{p \in P} \{N_p\} \bigcup_{p \in P_Y} \{C_p, C'_p\}$.

Create the productions $S ::= C'_S$, $S ::= C_S C'_S$ and $C_S ::= C_S C_S$. Let $\bot \twoheadrightarrow p^I$ be a rule in $\mathbf{P}$. Create the production $C'_S ::= Z_{p^I}$. If p^I is circular, create the additional production $C_S ::= Z_{p^I}$.

For each rule $p \twoheadrightarrow_c (p^1 \odot p^2) \fatsemi p^3$ or $p \twoheadrightarrow_s (p^1 \circ p^2) \fatsemi p^3$ create the productions $Z_p \twoheadrightarrow Z_{p_1} C'_p Z_{p_2}$ and $C'_p ::= Z_{p_3}$. For each rule $p \twoheadrightarrow_c (p^1 \odot p^2) \fatsemi p^3$ create the additional productions $Z_p ::= Z_{p_1} C_p C'_p Z_{p_2}$, $C_p ::= C_p C_p$, and $C_p ::= Z_{p_3}$. Let $Prod'$ be the all the productions defined by the process just given. $Prod = \{ \bigcup_{p \in P} Prod_p \} \cup Prod'$.

C.2 Example of a CFG generated from a PRS

The following is the CFG generated for the Dyck Language of order 2 (L_7 of Section 7.3)[7].

```
S ::= SC
SC ::= SC SC | P1 | P2
P1::= ( P1C )
P1C ::= P1C  P1C | P1 | P2
P2::= [ P2C ]
P2C ::= P2C  P2C | P1 | P2
```

To illustrate that sometimes the extra non-terminals generated by the algorithm are necessary, the following is the generated CFG for alternating delimiters, (L_{12} of Section 7.3).

```
S ::= P1 | P2
P1::= ( P2 )
P2::= [ P1 ]
```

C.3 Limitations on the expressibility of a PRS

Not every CFL is expressible by a PRS. In particular, let Σ be some alphabet, $w \in \Sigma^*$, w^R be the reverse of w, x a symbol not in Σ and $L^R = \{wxw^R : w \in \Sigma^*\}$, the infinite language of palindromes of odd length. L^R is a CFL but is not expressible by a PRS. Every word in L^R contains a single x.

Assume there exists a PRS $\mathbf{P}$ s.t. $L(\mathbf{P}) = L^R$. $\mathbf{P}$ contains a finite number of initial rules $\bot \to p^I$. Every word recognized by $A_1 = A^{P^I}$ must be of the form wxw^R and therefore traverses a straight path ρ_w from q_0 to q_f in A_1. Hence only a finite subset of L^R is recognized from these initial rules and there must be at least one rule that has an initial pattern p^I on its LHS. Applying this rule to A_1 will create a new DFA A_2 with a new pattern p grafted onto some state in A_1. This creates the new path from q_0 to q_f in A_2 of the form $\rho = \rho_1 p \rho_2$ for some w, where $\rho_w = \rho_1 \rho_2$. Since ρ_w recognizes wxw^R, x is a symbol recognized along the path ρ_1 or ρ_2. Assume x is recognized along the path ρ_1; i.e., ρ_1 recognizes the string wxu, ρ_2 recognizes the string v and $uv = w^R$. Then $wxu\alpha v \in L(A_2) \subseteq L(\mathbf{P})$, where $\alpha \in L(p)$ and $|\alpha| \geq 1$. But $|w| < |u\alpha v|$ and therefore $wxu\alpha v \notin L^R$. A similar argument holds if x is recognized along the path ρ_2. We therefore conclude that no such $\mathbf{P}$ recognizing L^R exists.

It is interesting to note that the language $L^{pal} = \{ww^R : w \in \Sigma^*\}$ is expressible by a PRS (Section 4.1) as is $L^R \cup L^{pal}$.

[7] As previously noted, the terminals, in this case "(",")", "[","]", are actually represented as base patterns.

References

1. Angluin, D.: Inductive inference of formal languages from positive data. Inf. Control. **45**(2), 117–135 (1980), https://doi.org/10.1016/S0019-9958(80)90285-5
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
3. Ayache, S., Eyraud, R., Goudian, N.: Explaining black boxes on sequential data using weighted automata. In: Unold, O., Dyrka, W., Wieczorek, W. (eds.) Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018. Proceedings of Machine Learning Research, vol. 93, pp. 81–103. PMLR (2018), http://proceedings.mlr.press/v93/ayache19a.html
4. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015 (2015), http://arxiv.org/abs/1409.0473
5. Bernardy, J.P.: Can recurrent neural networks learn nested recursion? In: Linguistic Issues in Language Technology, Volume 16, 2018. CSLI Publications (2018), https://www.aclweb.org/anthology/2018.lilt-16.1
6. Cechin, A.L., Simon, D.R.P., Stertz, K.: State automata extraction from recurrent neural nets using k-means and fuzzy clustering. In: 23rd International Conference of the Chilean Computer Science Society (SCCC 2003). pp. 73–78. IEEE Computer Society (2003). https://doi.org/10.1109/SCCC.2003.1245447
7. Clark, A., Eyraud, R.: Polynomial identification in the limit of substitutable context-free languages. J. Mach. Learn. Res. **8**, 1725–1745 (2007), http://dl.acm.org/citation.cfm?id=1314556
8. Clark, A., Eyraud, R., Habrard, A.: A polynomial algorithm for the inference of context free languages. In: Clark, A., Coste, F., Miclet, L. (eds.) Grammatical Inference: Algorithms and Applications, 9th International Colloquium, ICGI 2008, Proceedings. Lecture Notes in Computer Science, vol. 5278, pp. 29–42. Springer (2008). https://doi.org/10.1007/978-3-540-88009-7_3
9. Das, S., Giles, C.L., Sun, G.: Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. In: Conference of the Cognitive Science Society. pp. 791–795. Morgan Kaufmann Publishers (1992)
10. D'Ulizia, A., Ferri, F., Grifoni, P.: A survey of grammatical inference methods for natural language learning. Artif. Intell. Rev. **36**(1), 1–27 (2011). https://doi.org/10.1007/s10462-010-9199-1
11. Gold, E.M.: Language identification in the limit. Information and Control **10**(5), 447–474 (May 1967), https://doi.org/10.1016/S0019-9958(67)91165-5
12. Hailesilassie, T.: Rule extraction algorithm for deep neural networks: A review. International Journal of Computer Science and Information Security (IJCSIS) **14**(7) (July 2016)
13. Hewitt, J., Hahn, M., Ganguli, S., Liang, P., Manning, C.D.: RNNs can generate bounded hierarchical languages with optimal memory. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 1978–2010. Association for Computational Linguistics (2020), https://www.aclweb.org/anthology/2020.emnlp-main.156
14. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735
15. Jacobsson, H.: Rule extraction from recurrent neural networks: A taxonomy and review. Neural Computation **17**(6), 1223–1263 (2005). https://doi.org/10.1162/0899766053630350

16. Korsky, S.A., Berwick, R.C.: On the Computational Power of RNNs. CoRR **abs/1906.06349** (2019), http://arxiv.org/abs/1906.06349
17. Kozen, D.C.: The Chomsky—Schützenberger theorem. In: Automata and Computability. pp. 198–200. Springer Berlin Heidelberg, Berlin, Heidelberg (1977)
18. Luong, T., Pham, H., Manning, C.D.: Effective approaches to attention-based neural machine translation. In: Màrquez, L., Callison-Burch, C., Su, J., Pighin, D., Marton, Y. (eds.) Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015. pp. 1412–1421. The Association for Computational Linguistics (2015). https://doi.org/10.18653/v1/d15-1166
19. Omlin, C.W., Giles, C.L.: Extraction of rules from discrete-time recurrent neural networks. Neural Networks **9**(1), 41–52 (1996). https://doi.org/10.1016/0893-6080(95)00086-0
20. Sennhauser, L., Berwick, R.: Evaluating the ability of LSTMs to learn context-free grammars. In: Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP. pp. 115–124. Association for Computational Linguistics (Nov 2018). https://doi.org/10.18653/v1/W18-5414
21. Siegelmann, H.T., Sontag, E.D.: On the Computational Power of Neural Nets. J. Comput. Syst. Sci. **50**(1), 132–150 (1995). https://doi.org/10.1006/jcss.1995.1013
22. Skachkova, N., Trost, T., Klakow, D.: Closing brackets with recurrent neural networks. In: Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP. pp. 232–239. Association for Computational Linguistics (Nov 2018). https://doi.org/10.18653/v1/W18-5425
23. Stevenson, A., Cordy, J.R.: A survey of grammatical inference in software engineering. Sci. Comput. Program. **96**(P4), 444–459 (Dec 2014). https://doi.org/10.1016/j.scico.2014.05.008
24. Sun, G., Giles, C.L., Chen, H.: The neural network pushdown automaton: Architecture, dynamics and training. In: Giles, C.L., Gori, M. (eds.) Adaptive Processing of Sequences and Data Structures, International Summer School on Neural Networks. Lecture Notes in Computer Science, vol. 1387, pp. 296–345. Springer (1997). https://doi.org/10.1007/BFb0054003
25. Thrun, S.: Extracting rules from artifical neural networks with distributed representations. In: Tesauro, G., Touretzky, D.S., Leen, T.K. (eds.) Advances in Neural Information Processing Systems 7, NIPS Conference, 1994. pp. 505–512. MIT Press (1994), http://papers.nips.cc/paper/924-extracting-rules-from-artificial-neural-networks-with-distributed-representations
26. Wang, Q., Zhang, K., Liu, X., Giles, C.L.: Connecting first and second order recurrent networks with deterministic finite automata. CoRR **abs/1911.04644** (2019), http://arxiv.org/abs/1911.04644
27. Weiss, G., Goldberg, Y., Yahav, E.: Extracting automata from recurrent neural networks using queries and counterexamples. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018. Proceedings of Machine Learning Research, vol. 80, pp. 5244–5253. PMLR (2018), http://proceedings.mlr.press/v80/weiss18a.html
28. Yu, X., Vu, N.T., Kuhn, J.: Learning the Dyck language with attention-based Seq2Seq models. In: Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP. pp. 138–146. Association for Computational Linguistics (2019), https://www.aclweb.org/anthology/W19-4815