On the Expressive Power of Regular Expressions with Backreferences

Taisei Nogami
Waseda University, Tokyo, Japan

Tachio Terauchi
Waseda University, Tokyo, Japan

Abstract
A rewb is a regular expression extended with a feature called backreference. It is broadly known that backreference is a practical extension of regular expressions, and is supported by most modern regular expression engines, such as those in the standard libraries of Java, Python, and more. Meanwhile, indexed languages are the languages generated by indexed grammars, a formal grammar class proposed by A.V.Aho. We show that these two models’ expressive powers are related in the following way: every language described by a rewb is an indexed language. As the smallest formal grammar class previously known to contain rewbs is the class of context sensitive languages, our result strictly improves the known upper-bound. Moreover, we prove the following two claims: there exists a rewb whose language does not belong to the class of stack languages, which is a proper subclass of indexed languages, and the language described by a rewb without a captured reference is in the class of nonerasing stack languages, which is a proper subclass of stack languages. Finally, we show that the hierarchy investigated in a prior study, which separates the expressive power of rewbs by the notion of nested levels, is within the class of nonerasing stack languages.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Regular expressions, Backreferences, Expressive power

Digital Object Identifier 10.4230/LIPIcs.MFCS.2023.71


Funding This work was supported by JSPS KAKENHI Grant Numbers JP20H04162, JP20K20625, and JP22H03570.

1 Introduction
A rewb is a regular expression empowered with a certain extension, called backreference, that allows preceding substrings to be used later. It is closer to practical regular expressions than the pure ones, and supported by the standard libraries of most modern programming languages. A typical example of a rewb follows:

Example 1. Let \( \Sigma \) be the alphabet \( \{a, b\} \). The language \( L(\alpha) \) described by the rewb \( \alpha = (1(a + b)^*)1 \backslash 1 \) is \( \{ww \mid w \in \Sigma^*\} \). Intuitively, \( \alpha \) first captures a preceding string \( w \in L((a + b)^*) \) by \( (1)1 \), and second references that \( w \) by following \( \backslash 1 \). Therefore, \( \alpha \) matches \( ww \). Because this \( L(\alpha) \) is a textbook example of a non-context-free language (and therefore non-regular), the expressive power of rewbs exceeds that of the pure ones.

In 1968, A.V.Aho discovered indexed languages with characterizations by two equivalent models: indexed grammars and (one-way\(^1\) nondeterministic, or 1N) nested stack automata (NSA) \([1, 2]\). The class of indexed languages is a proper superclass of context free languages (CFL), and a proper subclass of context sensitive languages (CSL) \([1]\).

\(^1\) “One-way” means that the input cursor will not move back to left. The antonym is “two-way.”
Berglund and van der Merwe [4], and Câmpeanu et al. [5] have shown that the class of rewbs is incomparable with the class of CFLs and is a proper subclass of CSLs. As the first main contribution of this paper, we prove that the language described by a rewb is an indexed language. Since the class of CSLs was the previously known best upper-bound of rewbs, our result gives a novel and strictly tighter upper-bound.

Meanwhile, there is a class of the languages called stack languages [8, 7]. This class corresponds to the model (1N) stack automata (SA), a restriction of NSA. Hence, it trivially follows that the class of stack languages is a subclass of indexed languages. Actually, this containment is known to be proper [2]. Furthermore, a model called nonerasing stack automata (NESA) has been studied in papers such as [8, 11, 14], and its language class is known to be a proper subclass of stack languages [14].

In this paper, we show that every rewb without a captured reference (that is, one in which no reference \( i \) appears as a subexpression of an expression of the form \((j\alpha)_j\)) describes a nonerasing stack language. Given our result, the following question is natural: does every rewb describe a (nonerasing) stack language? We show that the answer is no. Namely, we show a rewb that describes a non-stack language. Finally, Larsen [12] has proposed a notion called nested levels of a rewb and showed that they give rise to a concrete increasing hierarchy of expressive powers of rewbs by exhibiting, for each nested level \( i \in \mathbb{N} \), a language \( L_i \) that is expressible by a rewb at level \( i \) but not at any levels below \( i \). We show that this hierarchy is within the class of nonerasing stack languages, that is, there exists an NESA \( A_i \) recognizing \( L_i \) for every nested level \( i \). Below, we summarize the main contributions of the paper.

(a) Every rewb describes an indexed language. (Section 4, Corollary 16)
(b) Every rewb without a captured reference describes a nonerasing stack language. (Section 4, Corollary 17)
(c) There exists a rewb that describes a non-stack language. (Section 5, Theorem 18)
(d) The hierarchy given by Larsen [12] is within the class of nonerasing stack languages. (Section 6, Theorem 20)

Note that by (b) and (c), it follows that there is a rewb that needs capturing of references (Section 5, Corollary 19). See also Figure 2 for a summary of the results.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 defines preliminary notions used in the paper such as the syntax and semantics of rewb, SA, NESA, and NSA. Sections 4, 5, and 6 formally state and prove the paper’s main contributions listed above. Section 7 concludes the paper with a discussion on future work. For space, the proofs are in the full paper [13].

2 Related Work

First, we discuss related work on rewbs. There are several variants of the syntax and semantics of rewbs since they first appeared in the seminal work by Aho [3]. A recent study by Berglund and van der Merwe [4] summarizes the variants and the relations between them. In sum, there are two variants of the syntax, whether or not a same label may appear as the index of more than one capture (“may repeat labels”, “no label repetitions”), and two variants of the semantics, whether an unbound reference is interpreted as the empty string or an undefined factor (\( \varepsilon \)-semantics, \( \emptyset \)-semantics). As shown in [4], there is no difference in the expressive powers between these two semantics under the “may repeat labels” syntax (therefore, there are three classes with different expressive powers, namely “no label repetitions” with \( \emptyset \)-semantics, “no label repetitions” with \( \varepsilon \)-semantics, and “may repeat labels”). In this paper, we focus on the “may repeat labels” formalization, which has
the highest expressive power of the three and is often studied in formal language theory. We adopt the \(\varepsilon\)-semantics as the semantics of rewbs. Note that the pioneering formalization of rewbs given by Aho [3] has the equivalent expressive power as this class. The rewbs with “may repeat labels” with \(\varepsilon\)-semantics was recently proposed by Schmid with the notion of ref-words and dereferences [15]. Simultaneously, he proposed a class of automata called memory automata (MFA), and showed that its expressive power is equivalent to that of rewbs. Freydenberger and Schmid extended MFA to \textit{MFA with trap-state} [6]. Berglund and van der Merwe [4] showed that the class of Schmid’s rewbs is a proper subclass of CSLs, and is incomparable with the class of CFLs. Note that there is a pumping lemma for the formalization given by Câmpeanu et al. [5] but it is known not to work for Schmid’s rewbs.

As mentioned above, Larsen introduced the notion of nested levels and showed that increase in the levels increases the expressive powers of rewbs [12].

Next, we discuss related work on the three automata used throughout the paper, namely SA, NESA, and NSA. Ginsburg et al. introduced SA as a mathematical model that is more powerful than pushdown automaton (PDA), and NESA as a restricted version of SA [8]. Hopcroft and Ullman discovered a type of Turing machine corresponding to the class of two-way NESA [11]. Ogden proposed a pumping lemma for stack languages and nonerasing stack languages [14]. Also proposed NSA with a proof of the fact that (1N) NSA and indexed grammars given by himself in [1] are equivalent in their expressive powers, and recognized PDA and SA as special cases of NSA [2]. Aho also showed that the class of indexed languages is a proper superclass of CFLs, and a proper subclass of CSLs [1]. Hayashi proposed a pumping lemma for indexed languages [9].

\section{Preliminaries}

In this section, we formalize the syntax and the semantics of rewbs following the formalization given in [6]. We begin with the syntax. Let \(\Sigma_{\varepsilon} = \Sigma \uplus \{\varepsilon\}\) and \(\{k\} = \{1, 2, \ldots, k\}\), where the symbol \(\uplus\) denotes a disjoint union.

\begin{definition}
For each natural number \(k \geq 1\), the set of \(k\)-rewbs over \(\Sigma\), written \(\text{REW}B_k\), and the mapping \(\text{var} : \text{REW}B_k \rightarrow \mathcal{P}(\{k\})\) are defined as follows, where \(a \in \Sigma_{\varepsilon}\) and \(i \in \{k\}\):
\[
(a, \text{var}(a)) := (a, \emptyset) | (i, \{i\}) | (a_0\alpha_1, \text{var}(a_0) \cup \text{var}(\alpha_1)) | (a_0 + \alpha_1, \text{var}(a_0) \cup \text{var}(\alpha_1))
\]
\end{definition}

We also write \(\text{REW}B_0\) for the set REG of regular expressions over \(\Sigma\), and \(\text{REW}B\) for the set of all rewbs, namely \(\bigcup_{k \geq 0} \text{REW}B_k\).

\begin{example}
For example, \(a, a, \text{\varepsilon}, a^*\backslash 1, (a*a)_1, (1a)_{a_1}, (2a^*_2)(2, (a^*_2)(2a_2^*_2)(1 + \backslash 2), (2(a + b^*_2)(1)_{2\backslash 2}(2\backslash 1))_2, ((1\text{\varepsilon} a_1)(2\text{\varepsilon} 3)_2 (2a_2^*_3) (a^*_1)_{3\backslash 3})_4\) are rewbs. On the other hand, \((1a^*_2)_{1\backslash 1}, (a^*_2)_{1\backslash 1}, (2(a^*_1)_{1\backslash 2})_{1\backslash 1}\) are not rewbs.

Note that this syntax allows multiple occurrences of captures with the same label, that is, we adopt the “may repeat labels” convention. Next, we define the semantics.

\begin{definition}
Let \(B_k = \{i, i\} \in \{k\}\). The mapping \(\mathcal{R}_k : \text{REW}B_k \rightarrow \mathcal{P}(\mathcal{R}(\Sigma \uplus B_k \uplus \{k\}))\) is defined as follows, where \(a \in \Sigma_{\varepsilon}\) and \(i \in \{k\}\):
\[
\mathcal{R}_k(a) = \{a\}, \mathcal{R}_k(i, i) = \{i\}, \mathcal{R}_k(a_0\alpha_1) = \mathcal{R}_k(a_0)\mathcal{R}_k(\alpha_1),
\]
\end{definition}

\[
\mathcal{R}_k(a_0 + \alpha_1) = \mathcal{R}_k(a_0) \cup \mathcal{R}_k(\alpha_1), \mathcal{R}_k(a^*) = \mathcal{R}_k(\alpha)^*, \mathcal{R}_k((i, i)) = \{i\}, \mathcal{R}_k(a) \{i\}.
\]

We let \(\Sigma_k^{\uplus}\) denote \(\bigcup_{a \in \text{REW}B_k} \mathcal{R}_k(a)\).
Example 5. \( R_k((1\langle a + b \rangle)^*{1\backslash 1}) = \{[1] \{a, b\}^* \{1\} \{1\} = \{[1 \, w \, 1] \mid w \in \{a, b\}^*\}. \)

That is, we first regard a rew b α over \( Σ \) as a regular expression over \( Σ \cup B_k \cup [k] \), deducing the language \( R_k(b) \). The second step, described next, is to apply the dereferencing (partial) function \( D_k : (Σ \cup B_k \cup [k])^* \to Σ^* \) to each of its element.

We give an intuitive description of \( D_k \). First, \( D_k \) scans its input string from the beginning toward the end, seeking \( i \in [k] \). If such \( i \) is found, \( D_k \) replaces this \( i \) with the sub-string obtained by removing the brackets in \( v \) that comes from the preceding \([v]\), if \([v]\) exists (if this \([v]\) has no corresponding \([i]\), \( D_k \) becomes undefined). Otherwise, \( D_k \) replaces this \( i \) with \( ε \).

The dereferencing function \( D_k \) repeats this procedure until all elements of \([k]\) appearing in the string are exhausted, then removes all remaining brackets. We let \( v_{[i]} \) denote the string which \( D_k \) scans at the \( i \)-th number \( n_r \in [k] \) at the \( i \)-th loop (see the full version [13] for the formal definitions of \( D_k \) and \( v_{[i]} \)).

1. \([a \langle bb \rangle \langle 2 \rangle \langle 1 \rangle] \). In this example, \( D_k \) encounters \( n_1 = 2 \) first, and this \( 2 \) corresponds the preceding \( \langle bb \rangle \), therefore this \( 2 \) is replaced with \( v_{[1]} = b \). As a result, the input string becomes \([a \langle bb \rangle \langle 1 \rangle] \). \( D_k \) repeats this process again. Now, \( D_k \) locates \( n_2 = 1 \) corresponding the preceding \([a \langle bb \rangle \langle b \rangle] \), so this \( 1 \) is replaced with \( v_{[2]} = a \langle bb \rangle \langle b \rangle \) but with the brackets erased. Therefore we gain \([a \langle bb \rangle \langle 1 \rangle] \). Finally, \( D_k \) removes all remaining brackets and produces \( bbabb \). Here is the diagram: \([a \langle bb \rangle \langle 2 \rangle \langle 1 \rangle] \rightarrow [a \langle bb \rangle \langle b \rangle] \rightarrow [a \langle bb \rangle \langle 1 \rangle] \rightarrow [a \langle bb \rangle \langle 1 \rangle \rightarrow a \langle bb \rangle \langle 1 \rangle \rightarrow ab \) \( bbabb \).

2. \([a \langle bb \rangle \langle 1 \rangle] \). In this example, \( n_1 = n_2 = 1 \), \( v_{[1]} = a \), \( v_{[2]} = bb \), and \( a \langle bb \rangle \langle 1 \rangle \rightarrow [a \langle bb \rangle \langle 1 \rangle \rightarrow [a \langle bb \rangle \langle 1 \rangle \rightarrow a \langle bb \rangle \langle 1 \rangle \rightarrow ab \) \( bbabb \).

3. \( abc12 \). In this example, \( n_1 = n_2 = 1 \), \( v_{[1]} = v_{[2]} = ε \), and \( abc12 \rightarrow abc2 \rightarrow abc \).

Note that an unbound reference is replaced with the empty string \( ε \), that is, we adopt the \( ε \)-semantics. However, as mentioned in Section 2, this semantics’ expressive power is equivalent to that of the \( 0 \)-semantics under the “may repeat labels” convention (see [4] for the proof). We define the language \( L(α) \) denoted by a \( k \)-rewb \( α \in \text{REW}_{B_k} \) to be \( D_k(R_k(α)) = \{v \mid v \in R_k(α)\} \) (Lemmas 6 and 8 ensure that \( L(α) \) is well-defined).

Let \( g : (Σ \cup B_k)^* \to Σ^* \) denote the free monoid homomorphism where \( g(x) \) is for each \( x \in Σ \), and \( ε \) for each \( x \in B_k \). Every \( v \in (Σ \cup B_k \cup [k])^* \) can be written uniquely in the form \( v = v_0n_1 \cdot \ldots \cdot m_n v_m \), where \( m \geq 0 \) (denoted by \( \text{cnt} v \)), and \( v_r \in (Σ \cup B_k)^* \) and \( n_r \in [k] \) for each \( r \in \{0, \ldots, m\} \). Here, let \( y_0 \triangleq v_0 \) and for each \( r \in \{1, \ldots, m\} \), \( y_r \triangleq v_0n_1 \cdot \ldots \cdot n_r v_r \). A string \( v = v_0n_1 \cdot \ldots \cdot m_n v_m \) over \( Σ \cup B_k \cup [k] \) is said to be matching if

\[
\forall r \in \{1, \ldots, m\}, \forall x_1, x_2, y_{r-1} = x_1[n_r x_2 = x_2[n_r x_r \wedge x_r' \notin \{n_r\}]\]

holds. Intuitively, a string \( v \) being matching means that for all \( n_r \in [k] \) in \( v \), if there exists a left bracket \([n_r \rangle \) in the string immediately before \( n_r \), then there is a right bracket \( ]n_r \langle \) in between this \( n_r \) and \( n_r \). The following three lemmas follow.

Lemma 6. Given a matching string \( v \), \( D_k(v) = g(v_0) g(v_1) \ldots g(v_m) \).

Lemma 7. A prefix of a matching string is matching. That is, if we decompose a string \( v \) into \( v = xy \), \( x \) is matching. Moreover, \( x_{[r]} = v_{[r]} \) holds for each \( r = 1, \ldots, \text{cnt} x \leq \text{cnt} v \).

Lemma 8. Every \( v \in Σ_k^* \) is matching.
A nondeterministic finite automaton $N$ is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where $Q$ is a finite set of states, $\Sigma$ a finite set of input symbols (also called alphabet), $q_0 \in Q$ a start state, $F \subseteq Q$ a set of final states, and $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ a transition function.

As well known, the transition function $\delta$ can be extended to $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ where $\hat{\delta}(q, w)$ represents the set of all states reachable from $q$ via $w$. Let $q \xrightarrow{a} N \bar{q}'$ denote $q' \in \hat{\delta}(q, a)$, and $q \xrightarrow{w} N \bar{q}'$ denote $q' \in \hat{\delta}(q, w)$. With this notation, the language of an NFA $N$ can be written as follows: $L(N) = \{ w \in \Sigma^* \mid \exists q_f \in F \ q_0 \xrightarrow{w} N q_f \}$.  

A pushdown automaton (PDA) is an NFA equipped with a stack such that the PDA may write and read its stack top with a transition. A stack automaton (SA) is “an extended PDA”, which can reference not only the top but inner content of the stack. That is, while the stack pointer of a PDA is fixed to the top, an SA allows its pointer to move left and right and read a stack symbol pointed to by the pointer. However, the only place on the stack that can be rewritten is the top, as in PDA. Formally, a (1N)SA $A$ is a 9-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, \#, \$, $S, F)$ satisfying the following conditions: the components $Q, \Sigma, q_0$ and $F$ are the same as those of NFA. $\Gamma (\neq \emptyset)$ is a finite set of stack symbols, and $Z_0 \in \Gamma$ is an initial stack symbol. The stack symbol $\#$ is not in $\Sigma \cup \Gamma$ and $\$ is always and only written at the leftmost (bottom) (resp. the right most (top)) of the stack.\(^2\) The transition function $\delta$ has the following two modes, where $L, S, R \notin (\Sigma \cup \Gamma) \cup \{\#, \$\}$, $\Delta_i \triangleq \{S, R\}$, and $\Delta_s \triangleq \{L, S, R\}$:

(i) (pushdown mode) $Q \times \Sigma \times \Gamma S \to \mathcal{P}(Q \times \Delta_i \times \Gamma S)$,
(ii) (stack reading mode) (a) $Q \times \Sigma \times I \Sigma \rightarrow \mathcal{P}(Q \times \Delta_i \times \{L\})$, (b) $Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Delta_i \times \Delta_s)$,
(c) $Q \times \Sigma \times \{\#\} \rightarrow \mathcal{P}(Q \times \Delta_i \times \{\} \cup \{R\})$.

Intuitively, $\delta$ works as follows (Definition 10 provides the formal semantics). (i) The statement $(q', d, w$) $\in \delta(q, a, Z\$)$ says that whenever the current state is $q$, the input symbol is $a$, and the pointer references the top symbol $Z$, the machine can move to the state $q'$, move the input cursor along $d$, and replace $Z$ with the string $w$. (ii) The statement $(q', d, e) \in \delta(q, a, Z)$ says that whenever the current state is $q$, the input symbol is $a$, and the pointer references the symbol $Z$, the machine can move to the state $q'$, move the input cursor along $d$, and move the pointer along $e$. The statements (a) and (c) are similar to (b) except that the direction in which the pointer can move is restricted lest the pointer go out of the stack. In particular, an SA that cannot erase a symbol once written on the stack is called a nonerasing stack automaton (NESA). That is, a (1N) nonerasing stack automaton is an SA whose transition function $\delta$ satisfies the condition that, in (i) (pushdown mode), $(q', d, w$) $\in \delta(q, a, Z\$)$ implies $w \in \Sigma^*$. To formally describe how SA works, we define a tuple called instantaneous description (ID), which consists of a state, an input string, and a string representation of the stack, and define the binary relation $\vdash_A$ over the set of these tuples. Let $L = -1, S = 0$, and $R = 1$.

Definition 10. Let $A$ be an SA $(Q, \Sigma, \Gamma, \delta, q_0, \#, \$, $F)$. An element of the set $I = Q \times \Sigma^* \times \{\#\}$ $\cup \{\} \cup \{1\} \text{ or } \{\} \cup \{\}$ is called instantaneous description, where the stack symbol $\{\} \notin \Gamma$ stands for the position of stack pointer. Moreover, let $\vdash_A$ (or $\vdash$ when $A$ is clear) be the smallest binary relation over $I$ satisfying the following conditions:

---

\(^2\) These special symbols $\#, \$ representing “bottom” and “top” of the stack respectively do not appear in [7] and are introduced anew in this paper to define NESA and NSA, which will be defined later, in the style of [2]. In fact, SA defined in [7] is not capable of directly discerning whether the stack pointer is at the top or not. Although it is not difficult to see that directly adding the ability does not increase the expressive power of SA, the ability is directly in NESA as seen in [11, 14]. Therefore, to make it easy to see that NESA is a restriction of SA, we define SA to also directly have the ability.
On the Expressive Power of Regular Expressions with Backreferences

(i) \((q, a_1 \cdots a_k, \#yZ \uparrow S) \vdash_A (q', a_{i+2} \cdots a_k, \#yw \uparrow S)\) if \((q', d, wS) \in \delta(q, a_i, ZS)\).

(ii) (a) \((q, a_1 \cdots a_k, \#yZ \uparrow S) \vdash_A (q', a_{i+2} \cdots a_k, \#yZ \uparrow S)\) if \((q', d, L) \in \delta(q, a_i, ZS)\).

(b) if \((q', d, c) \in \delta(q, a_i, Z)\) and \(Z = Z_j\), 1 \(\leq j < n\), then
\[(q, a_1 \cdots a_k, \#Z_1 \cdots Z_{n-1}Z_n \uparrow S) \vdash_A (q', a_{i+2} \cdots a_k, \#Z_1 \cdots Z_{n-1}Z_n \uparrow S)\].

(c) \((q, a_1 \cdots a_k, \#Z \uparrow y \uparrow S) \vdash_A (q', a_{i+2} \cdots a_k, \#Z \uparrow y \uparrow S)\) if \((q', d, R) \in \delta(q, a_i, \#)\). Note that \(L \not\in L\), which means the input cursor will not move back to left. We say that \(A\) accepts \(w \in \Sigma^*\) if there exist \(y_1, y_2 \in \Gamma^*\) and \(q_f \in F\) such that \((q_0, w, \#Z_0 \uparrow S) \vdash_A (q_f, \varepsilon, \#y_1 \#y_2 \uparrow S)\). Let \(L(A)\) denote the set of all strings accepted by \(A\).

We next define nested stack automaton (NSA) which is SA extended with the capability to create and remove substacks. For instance, suppose that the stack is \(#a_1a_2 \uparrow a_3S\) and we are to create a new substack containing \(#b_1b_2 S\):

\[\#a_1 \uparrow b_1 b_2 \#S \leftrightarrow \#a_1 \uparrow b_1 \#a_2 \uparrow b_2 \#a_3 \#S.\] (1)

Note that the new substack \(b_1 b_2 \#S\) is embedded below the symbol \(a_2\) indicated by the stack pointer, and the pointer moves to the top of the created substack. The creation of the inner substack narrows the range within which the stack pointer can move as indicated by the underlined part \(#a_1 \uparrow b_1 b_2 \#S\). While the bottom of the entire stack is always fixed by the leftmost symbol \(\#\), the top of the embedded substack is regarded as the top of the entire stack. The inner substacks are allowed to be embedded endlessly and everywhere, whereas the writing in the pushdown mode is still restricted to the top of the stack:

\[\#a_1 \uparrow b_1 b_2 \#S \leftrightarrow \#a_1 \uparrow b_1 \#a_2 \uparrow b_2 \#a_3 \#S;\] (2)

\[\#a_1 \uparrow b_1 b_2 \#a_3 \#S \leftrightarrow \#a_1 \uparrow b_1 \#a_2 \uparrow b_2 \#a_3 \#S;\] (3)

We must empty the inner substack and then remove itself in advance whenever we want to reference the right side of the inner substack such as \(a_2, a_3\). For example, let us empty the inner substack by popping twice from (1) and then removing it:

\[\text{pop} \#a_1 \uparrow b_1 b_2 \#S \leftrightarrow \#a_1 \uparrow b_1 \#a_2 \uparrow a_3 \#S;\] (4)

Notice that the stack pointer moves to the right after removing the inner substack. We now define NSA formally. A (1N) nested stack automaton \(A\) is a 10-tuple \((Q, \Sigma, \Gamma, \delta, q_0, Z_0, \#, \epsilon, F, S)\) satisfying the following conditions: the components \(Q, \Sigma, \Gamma, q_0, Z_0, \#\), \(\epsilon\), \(F\) and \(S\) are the same as those of SA. The stack symbol \(\epsilon \notin \Sigma \cup \Gamma\) represents the bottom of a substack. The transition function \(\delta\) has the following four modes, where \(\Gamma^* \triangleq \Gamma \uplus \{\epsilon\}:

(i) (pushdown mode) \(Q \times \Sigma \times \Gamma^* \rightarrow P(Q \times \Delta_1 \times \Gamma^*)\).

(ii) (stack reading mode) (a) \(Q \times \Sigma \times \Gamma^* \rightarrow P(Q \times \Delta_1 \times \{L\})\), (b) \(Q \times \Sigma \times \Gamma^* \rightarrow P(Q \times \Delta_1 \times \Delta_1)\),

(c) \(Q \times \Sigma \times \{\#\} \rightarrow P(Q \times \Delta_1 \times \{R\})\).

(iii) (stack creation mode) \(Q \times \Sigma \times (\Gamma^* \uplus \Gamma^* \uplus \{\#\}) \rightarrow P(Q \times \Delta_1 \times \{\epsilon\} \uplus \Gamma^*)\).

(iv) (stack destruction mode) \(Q \times \Sigma \times \{\epsilon\} \rightarrow P(Q \times \Delta_1)\).

Moreover, we define how NSA works with ID and \(\vdash\) in the same manner as SA. Given an NSA \(A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \#\), \(\epsilon\), \(F\), \(S)\), we define \(ID, \vdash_A\), and \(L(A)\) in the same way as Definition 10 (however, we let \(I\) be \(Q \times \Sigma^* \times \{\#\} \uplus \Gamma \uplus \{\epsilon, \\uplus \{\$\}\} \uplus \{\$\}\)). Here, we only give the rules corresponding to (ii) and (iv) in the definition of \(\delta\) (the others are essentially the same as those of SA):

\[\text{We regard } a_{i+1} \cdots a_k \text{ as } \epsilon.\]

\[\text{Note that the bottom of the entire stack is always represented by } \# \text{ and not } \epsilon, \text{ as mentioned above.}\]
As described above, to obtain the language $L(\alpha)$ described by a $k$-rewb $\alpha$, we derive the regular language $R_k(\alpha)$ over the alphabet $\Sigma \cup B_k \cup [k]$ first, then apply the dereferencing function $D_k$ to every element of $R_k(\alpha)$. Using this observation, we construct an NSA $A_{\alpha}$ recognizing the language $L(\alpha)$ as follows.

The NSA $A_{\alpha}$ is based on an NFA $N$ recognizing the language $R_k(\alpha)$, in the sense that each transition in $A_{\alpha}$ comes from a corresponding transition of $N$. The NFA $N$ has the alphabet $\Sigma \cup B_k \cup [k]$, and so handles three types of characters. For each transition $q \xrightarrow{a} q'$ with $a \in \Sigma$, i.e., moving from $q$ to $q'$ by an input symbol $a$, $A_{\alpha}$ also has the same transition except pushing $a$ to the stack, denoted by $q \xrightarrow{a/s} q'$. For each transition $q \xrightarrow{b} q'$ with $b \in B_k$, i.e., moving by a bracket $b$, $A_{\alpha}$ has the transition pushing $b$ without consuming input symbols, denoted by $q \xrightarrow{c/b} q'$. For each transition $q \xrightarrow{i} q'$ with $i \in [k]$, $A_{\alpha}$ has a large “transition” that consists of several transitions. In this “transition,” $A_{\alpha}$ first seeks the left bracket $[i$ of the bracketed string $[i \cdot v]$, within the stack, and checks if the input from the cursor position matches $v$ character by character while consuming the input, and finally moves to $q'$ if all characters of $v$ matched.

A difficult yet interesting point is that NSA cannot check $v$ against the stack and push $v$ onto the stack at the same time, that is, after checking a character $c$ of $v$, if $A_{\alpha}$ wants to push $c$ to the stack, $A_{\alpha}$ must leave from $v$, climb up the stack toward the top, and write $c$. However, after the push, $A_{\alpha}$ becomes lost by not knowing where to go back to. How about marking the place where $A_{\alpha}$ should return in advance? Unfortunately, that does not work; NSA can insert such marks anywhere by creating substacks, but due to the restriction of NSA, it cannot go above the position of the mark, much less climb up to the top. Therefore, NSA cannot directly push the result of a dereference onto the stack.

We cope with this problem as follows. We allow $j \in [k]$ to appear in $v$, and for each appearance of $j$ in the checking of $v$, $A_{\alpha}$ pauses the checking and puts a substack containing the current state as a marker at the stack pointer position. Then, $A_{\alpha}$ searches down the stack for the corresponding bracketed string $[j \cdot v']$, and begins checking $v'$ if it is found. By repeating this process, $A_{\alpha}$ eventually reaches a string $v'' \in (\Sigma \cup B_k)^* \kern-1em$ containing no characters of $[k]$. Once done with the check of $v''$, $A_{\alpha}$ climbs up toward the stack top, finds a marker $p$ denoting the state to return to, and resumes from $p$ after deleting the substack containing the marker. By repeating this, if $A_{\alpha}$ returns to the position where it initially found $j$, it has successfully consumed the substring of the input string corresponding to the dereference of $j$.

The following lemma is immediate.

\begin{lemma}
Let $k \geq 1$ and $\alpha \in \text{REWB}_k$. There exists an NFA $(Q, \Sigma \cup B_k \cup [k], \delta, q_0, F)$ over $\Sigma \cup B_k \cup [k]$ recognizing $R_k(\alpha)$ all of whose states can reach some final state, that is, $\forall q \in Q, \exists w \in (\Sigma \cup B_k \cup [k])^* \kern-1em \exists q_f \in F.q \xrightarrow{w} q_f$.
\end{lemma}

Strictly speaking, our NFA (cf. Definition 9) does not allow consuming the empty string $\varepsilon$. However, we can realize the transition $q \xrightarrow{\varepsilon/s} q'$ alternatively by adding $q \xrightarrow{c/b} q'$ for each $c \in \Sigma$, i.e., moving by $c$ with the input cursor fixed.

\textbf{4} Every rewrb describes an indexed language

As described above, to obtain the language $L(\alpha)$ described by a $k$-rewrb $\alpha$, we derive the regular language $R_k(\alpha)$ over the alphabet $\Sigma \cup B_k \cup [k]$ first, then apply the dereferencing function $D_k$ to every element of $R_k(\alpha)$. Using this observation, we construct an NSA $A_{\alpha}$ recognizing the language $L(\alpha)$ as follows.
Corollary 12. Let $N$ be the NFA in Lemma 11. For all $q \in Q$ and for all $w \in (\Sigma \cup B_k \cup \{k\})^*$, if $q_0 \xrightarrow{N} q$ then $w$ is matching (see the full version [13] for the proof).

We show the main theorem (the proof sketch is coming later):

Theorem 13. For every rew $\alpha \in \text{REW}$, there exists an NSA that recognizes $L(\alpha)$.

The claim obviously holds when $\alpha$ is a pure regular expression (i.e., $\alpha \in \text{REW}_0$). Suppose that $\alpha \in \text{REW}_k$ with $k \geq 1$. By Lemma 11, there is an NFA $N = (Q_N, \Sigma \cup B_k \cup \{k\}, \delta_N, q_0, F)$ that recognizes $R_k(\alpha)$ and satisfies Corollary 12. We construct an NSA $A_\alpha = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \#)$ as follows. Let $Q \triangleq Q_N \cup \{e_i \mid i \in [k]\} \cup \{W_q \mid q \in Q_N \cup \{e_i \mid i \in [k]\}, i \in [k]\}$, $\Gamma \triangleq \Sigma \cup B_k \cup \{k\} \cup Q \cup \{Z_0\}$, and let $\delta$ be the smallest relation that, for all $a \in \Sigma, b \in B_k, c \in \Sigma, i, j \in [k], q, q' \in Q_N, Z \in \Gamma$ and $p \in Q_N \cup \{e_i \mid i \in [k]\}$, satisfies the following conditions:

1. $\delta_N(q, a) \supseteq q' \Rightarrow \delta(q, a, ZS) \ni (q', R, Z \alpha S)$
2. $\delta_N(q, b) \ni q' \Rightarrow \delta(q, c, ZS) \ni (q', S, Z \beta S)$
3. $\delta_N(q, i) \ni q' \Rightarrow \delta(q, c, ZS) \ni (W_{q'}, c, Z \alpha S)$
4. $\delta(W_{q'}, c, i) = \{(c, i, S, \alpha \Xi\epsilon\Xi)\}$
5. $\delta(e_i, c, pS) = \{(c, i, S, L)\}$
6. $\delta(e_i, c, Z) = \{(c, i, S, L)\}$ where $Z \neq \# Z_0$
7. $\delta(e_i, c, Z_0) = \{(c, i, S, R)\}$
8. $\delta(e_i, c, i) = \{(e_i, S, R)\}$
9. $\delta(e_i, c, a) = \{(e_i, R, R)\}$

Rule (1) translates $q \xrightarrow{a} q'$ into $q \xrightarrow{a/S \rightarrow \alpha S} q'$, (2) translates $q \xrightarrow{b} q'$ into $q \xrightarrow{c/S \rightarrow \beta S} q'$, and rules (3)–(17) translate $q \xrightarrow{N} q'$ into a large “transition” to consume the string that corresponds to the dereferencing of $i$. The details of the “transition” are as follows. By looking at the underlying $N$ with rule (3), $A_\alpha$ finds a state $q'$ that it should go back to after going throughout the “transition,” and goes to the state $W_{q'}$ by pushing $i$ to the stack. At $W_{q'}$, by rule (4), $A_\alpha$ inserts $\epsilon \alpha \Xi\epsilon\Xi\beta$ just below $i$, and goes to the state $e_i$. The state $e_i$ represents the call mode in which $A_\alpha$ looks for the left-nearest $[i]$ by rules (5) and (6) and proceeds to the state $e_i$ (execution mode) by (8) if it finds $[i]$. Otherwise (i.e., the case when $A_\alpha$ arrives at the bottom of the stack), it proceeds to the state $r_i$ (return mode) by rule (7). At $e_i$, $A_\alpha$ consumes input symbols by checking them against the symbols on the stack (rules (9)–(12)). In particular, rule (9) handles the case when the symbols match. Rules (10) and (11) handle the cases when brackets are read from the stack. The first case of (11) handles the case when the right bracket $\]$, is read, and the rules handle the other brackets (i.e., $\[$ or $\]$ with $i \neq j$) by simply skipping them (note that $[i = [i$ cannot happen since we started from the left-nearest $[i$). Reading $j \in [k]$, by rule (12), $A_\alpha$ inserts $\epsilon \alpha \Xi\epsilon\Xi\beta$ just below $j$ and goes to $c_j$ to locate the corresponding $[j$ (here, $j \neq i$ holds by the definition of the syntax). At $r_i$, $A_\alpha$ proceeds to return to the state $p$ that passed the control to $c_i$ (rules (13)–(17)). Since this $p$ was pushed at the stack top, $A_\alpha$ first climbs up to the stack top by rule (13), transits to the state $E_{p,i}$ popping $p$ by (14), then goes to $L_{p,i}$ removing the embedded substack by (15), and finally goes back to $p$ by (16) and (17). A subtle point in the last step is that where the stack pointer should be placed depends on whether $p$ is a state $c_j$ (for some $j \in [k]$) or in $Q_N$. In the former case, after (15) removes the embedded substack $\epsilon \alpha \Xi\epsilon\Xi\beta$ that was created just below the call to $i$, the stack pointer points to $i$. However, the stack pointer should shift one more to the right, lest $A_\alpha$ begins to repeat the call reading $i$ again by (12). Therefore, (16) correctly handles the case by doing the shift. In the latter case, as stipulated by (17), the stack pointer should point to the stack top symbol $i$ since $p$ is the state stored at (3).
We state two lemmas used to prove Theorem 13. Let ⊢_{(n)} denote the subrelation of ⊢ derived from the rule (n). The following lemma is immediate from the definition of ⊢_{(n)}.

**Lemma 14.** For all q, q′ ∈ Q_N, w, w′ ∈ Σ*, γ, γ′ ∈ Γ*,
(a) 1. for each a ∈ Σ, (q, aw, #Z_0γ | $) ⊢_{(1)} (q′, w, #Z_0γa | $) if q \xrightarrow{a}_N \rightarrow q′,
2. ∃a ∈ Σ, q \xrightarrow{a}_N \rightarrow q′ ∧ w = aw′ ∧ β = Z_0γa | if (q, w, #Z_0γ | $) ⊢_{(1)} (q′, w′, #β$).
(b) 1. for each b ∈ B_k, (q, w, #Z_0γ | $) ⊢_{(2)} (q′, w, #Z_0γb | $) if q \xrightarrow{b}_N \rightarrow q′,
2. ∃b ∈ B_k, q \xrightarrow{b}_N \rightarrow q′ ∧ w = w′ ∧ β = Z_0γb | if (q, w, #Z_0γ | $) ⊢_{(2)} (q′, w′, #β$).

In particular, letting ⊢_{(1)} and ⊢_{(2)} zero or more times: For all v ∈ (Σ ⊨ B_k)*, (q, g(v) w, #Z_0γ | $) ⊢_{(1),(2)} (q′, w′, #β$), where no ID with a state in Q_N appears in the calculation ⋯.

**Proof of Theorem 13 (sketch).** For proving L(α) ⊆ L(A_α), we take w ∈ L(α) and v ∈ R_k(α) such that w = D_k(v). Decomposing v into v_0v_1⋯v_nv_m (where m = ctt v), we obtain a transition sequence in the underlying NFA N, denoted by q_0 \xrightarrow{v_0}_N q_0(q_0 \xrightarrow{v_1}_N q_1(⋯ \xrightarrow{v_m}_N q_m) ∈ F. We prove by induction on r = 0,⋯,m that A_α can reach q_{r(0)} while consuming z_r = g(v_0)g(v_1)⋯g(v_r) from the input and pushing y_r = v_0v_1⋯v_r to the stack. Conversely, we suppose a calculation in A_α, denoted by C_{(r)} = (q_0, w, #Z_0 | $) ⊢ ⋯ ⊢ C_{(r)} ⊢ ⋯ ⊢ C_{(m)} = (p_m, ε, #β_m$), where p_m ∈ F and C_{(r)} = (p_r, w_r, #β_r) for each r ∈ {1,⋯,m}. By induction on r = 1,⋯,m, we extract an underlying transition q_0 \xrightarrow{v_r}_N p_r step by step while maintaining the invariants γ_r ∈ (Σ ⊨ B_k ⊨ k)* and w = D_k(γ_r) w_r, as long as p_r ∈ Q_N (the formal proof is available in the full version [13]).

**Corollary 16.** Every rew describes an indexed language, but not vice versa.

**Proof.** The first half follows by Theorem 13 since 1N NSA and indexed grammars are equivalent [2]. The second half also follows since the class of CFLs is a subclass of indexed languages [1], and the class of rewbs and that of CFLs are incomparable [4].

In the case of a rew α without a captured reference (that is, in which no reference \gamma_i appears as a subexpression of an expression of the form (⋯)), we can transform A_α into an NESA A'_α, recognizing L(α), i.e., one that neither uses substacks nor pops its stack. First, we transform A_α to an NSA without substacks (i.e., SA) A'_α. Inspecting how substacks are used in A_α, we can drop rules (12) and (16) in A'_α because there is no captured reference in α. We also remove the uses of substacks from rules (3) and (4), which correspond to calling, and rules (14), (15) and (17), which correspond to returning. Namely, while A_α, upon a call, stores the substack \gamma_S$ that consists of just the state q where the control should return, A'_α simply pushes q’ to the stack top. That is, we remove (4), (15) and (17), and change (3) and (14) to the following (3’ and 14’), respectively:

(3’) δ_N(q, i) ⊢ q’ =⇒ δ(q, c, Z$) ⊢ (ε, S, Ziq)$,
(14’) δ(r, c, q$) = \{(q, S, 0)\}.  


Furthermore, we transform $A_0'$ to an SA without stack popping (i.e., NESA) $A_0''$. Observe that $A_0''$ pops only when returning via (14') and popping a state that was pushed in a preceding call. Thus, $A_0''$, rather than popping $q'$, leaves it on the stack, and has the modes $c_1$, $e_1$ and $r_1$ skip all state symbols on the stack except the ones at the top. Here, we only need to modify $e_1$ since $A_0$ already skips them at $c_1$ and $r_1$ (rules (6) and (13)). In short, we add the new rule (9*) and change (14') to (14''), as follows:

$$(9*) \delta(c_i, c, q) = \{(c_i, S, R)\}, \quad (14'') \delta(r_i, c, q\$) = \{(q, S, q\$)\}.$$ 

This NESA $A_0''$ whose transition function consists of the rules (1),(2),(3'),(5)–(9),(9*),(10), (11), (13) and (14'') recognizes $L(a)$. Therefore,

- **Corollary 17.** Every rewb without a captured reference describes a nonerasing stack language, but not vice versa.\(^6\)

Note that the converse of Corollary 17 fails to hold. In other words, there is a rewb with a captured reference that describes a nonerasing stack language. The rewb $(a_1a_1\{2\}2|2a_2\{2\}2)$ is a simple counterexample. In addition, as shown later in Section 6, NESA can recognize nontrivial language (hierarchy) with a captured reference such as Larsen’s hierarchy [12].

### 5 A rewb that describes a non-stack language

We just showed that every rewb describes an indexed language and in particular every rewb without a captured reference describes a nonerasing stack language. So, a natural question is whether every rewb describes a (nonerasing) stack language. We show that the answer is no. That is, there is a rewb that describes a non-stack language.

Ogden has proposed a pumping lemma for stack languages and shown that the language \(\{a^n n \in \mathbb{N}\}\) is a non-stack language as an application (see [14], Theorem 2). A key point in the proof is that the exponential $n^3$ of $a$ is a cubic polynomial, and we can show that for every cubic polynomial $f : \mathbb{N} \rightarrow \mathbb{N}$, the language \(\{a^{f(n)} n \in \mathbb{N}\}\) is also a non-stack language by the same proof. Thus, a rewb that describes a language in this form is a counterexample.

We borrow the technique in [6] (Example 1) which shows that the rewb $\alpha = ((1\{2\}12\{2\}1a_2)^n)$ describes $L(\alpha) = \{a^n n \in \mathbb{N}\}$. This follows since $D_k(|[2]|2a|^n|2a^2|) = a^n$ holds by recording the iteration count of the Kleene star, $n$, in the capture $(2)_2$ as $a^n$, and extending the length by $2n + 1$, as shown below:

$$D_k(|[2]|2a|^n+1|2a^2|) = D_k(|[2]|2a|^n|2a^2|2|2a^2|2) = D_k([\cdots [a^n\|2|2a^2|2a^n\]2[2a^2|2a^n\]2] = a^n a^{2n+1} = a^{n+1^2}.$$ 

The rewb $(|1\{4\}a|12\{3\}2|a\{2\}a_3|4\{1\}3\{2\})^*$ describes \(\{a^{n+7}(2n+1)/6 n \in \mathbb{N}\}\) and extends the length by a quadratic in $n$ instead (see the full version [13] for the calculation). Thus,

- **Theorem 18.** There exists a rewb that describes a non-stack language.

From this and Corollary 17, this rewb needs a captured reference, in the sense that:

- **Corollary 19.** There exists a rewb that describes a language that no rewb without a captured reference can describe.

\(^6\) For the latter part, we can take the language \(\{a^n b^n n \in \mathbb{N}\}\) that can be described by an NESA (see the full version [13]) but not by any rewb [4].
6 Larsen’s hierarchy is within the class of nonerasing stack language

In this section, we construct an NESA $A_i$ that describes $L(x_i)$, where the rewb $x_i$ over the alphabet $\Sigma = \{a_0, a_0^m, a_0^m, a_1^m, a_1^m, \ldots\}$ is given by Larsen [12] and defined as follows: $x_0 \equiv (a_0^m a_0^m a_0^m)^*, x_{i+1} \equiv (a_1^i a_1^i a_1^i \ldots a_1^i)^\ast (i \geq 0)$. Our result implies that Larsen’s hierarchy is within the class of nonerasing stack languages. Since Larsen showed that no rewb with its nested level less than $i$ can describe $L(x_i)$ [12], it also implies that for every $i \in \mathbb{N}$, there is a nonerasing stack language that needs a rewb of nested level at least $i$.

![Figure 1](image)

The NESA $A_i$ has the start state $q_0$ which is also its only final state. Figure 1 depicts $A_0$, $A_1$, and $A_2$. $A_0$ is easy. $A_1$ is obtained by connecting the eight states to $q_0$ and making $q_0$ the start/final state, as shown in the figure. The five states on the right handle the dereference of $\backslash 0$ in $x_1$. That is, at $c_0$, $A_1$ first seeks the left-nearest $[0]$; passes the control to $c_1$, checks the input string against the stack at $c_1$, passes the control to $r_0$, and at $r_0$, finally goes back to the right-nearest $0$ which must be written on the stack top. In much the same way, $A_2$ is obtained from $A_1$ but we must be sensitive to the handling of the dereference of $\backslash 1$ because $A_2$ must handle the dereference of not only $\backslash 1$ but also $\backslash 0$ that appears in a string captured by $[1]_1$ whereas no backreference appears in a string captured by $[0]_0$ in the case of $A_1$. To deal with this issue, we connect the three new states $c_0^2$, $c_0^3$, and $c_0^4$ to $c_0^3$. At $c_0^2$, if $A_2$ encounters 0 in a checking, $A_2$ suspends the checking and first goes to $c_0^3$ to check the input against the stack by reading out a $[0]$ (no number appears in $x_1$).

---

7 Technically, Larsen [12] adopts a syntax that excludes unbound references, and so this implied result applies only to rews with no unbound references.
On the Expressive Power of Regular Expressions with Backreferences

in this checking), and finally goes to \( r^2 \) to go back to 0 which passed the control to \( c^2 \). We repeat this modification until \( A_i \) is obtained. (Thus, \( A_i \) has such states \( e^j_i, e^j_i, r^j_i \) for each \( j \in \{0, \ldots, i - 1\} \).) Therefore,

\[ \textbf{Theorem 20.} \text{ There exists an NESA } A_i \text{ that recognizes } L(x_i). \]

7 Conclusions

In this paper, we have shown the following five results: (1) that every rewrb describes an indexed language (Corollary 16), (2) in particular that every rewrb without a captured reference describes a nonerasing stack language (Corollary 17), (3) however that there exists a rewrb that describes a non-stack language (Theorem 18), (4) therefore that there exists a rewrb that needs a captured reference (Corollary 19), and (5) finally that Larsen’s hierarchy \( \{ L(x_i) \mid i \in \mathbb{N} \} \) given in [12] is within the class of nonerasing stack languages (Theorem 20).

We have obtained the results by using three automata models, namely NESA, SA, and NSA, and using the semantics of rewbs given in [15, 6] that treats a rewrb as a regular expression allowing us to obtain the underlying NFA. Figure 2 depicts the inclusion relations between the classes mentioned in the paper. Here, \( A \rightarrow B \) stands for \( A \subseteq B \), \( A \twoheadrightarrow B \) for \( A \supseteq B \), and \( A \twoleftarrow B \) for \( A \not\subseteq B \), respectively. A label on an arrow refers to the evidence. A red dashed arrow indicates a novel result proved in this paper, where for a strict inclusion, we show for the first time the inclusion itself in addition to the fact that it is strict.

As future work, we would like to investigate the use of the pumping lemma for rewbs without a captured reference that can be derived from the contraposition of our Corollary 17 and a pumping lemma for NESA [14]. We expect it to be a useful tool for discerning which rewbs need captured references. Additionally, we suspect that our construction of NESA in Theorem 20 is useful for not just \( x_i \) of [12] but also for more general rewbs that have only one \( \setminus i \) for each \( (i) \), and we would like to investigate further uses of the construction.

References

Proof. By the definition of $\vdash'$, there is $(C(a_1 \cdots a_k), C'(a_{i+\overline{\gamma} -1} \cdots a_k)) \vdash (a_1 \cdots a_{i+\overline{\gamma} -1}, w')$ by the converse partially holds, in the sense that: $w' = w/(a_1 \cdots a_{i+\overline{\gamma} -1})$. By the definition of $\vdash$, $C(w) = C(a_1 \cdots a_{i+\overline{\gamma} -1} w') \vdash C'(w')$ holds.
Definition 22. Given $C, C' \in L$, we write $C \models_{(n)} C'$ if $C \vdash_{(n)} C'$ and $\forall j, C^n. C \vdash'_{(j)} C'' \Rightarrow j = n \land C'' = C'$. We often omit the subscript $(n)$ and simply write $C \models C'$. Note that $C \models C'$ implies not only $C \vdash C'$ but also determinism: $\forall C'' \in L, C \vdash' C'' \Rightarrow C = C''$.

Lemma 23. Suppose that $\gamma \in (\Sigma \cup B_k \cup [k])^*$, $i \in [k]$, $w \in \Sigma^*$, $\beta \in (\Gamma \cup \{\varepsilon, \$\})^*$ and $p \in Q_N \cup \{c_i | i \in [k]\}$. Let $m = \text{cunt}(\gamma i) (\geq 1)$. If $\gamma i$ is matching,

$$(c_i, w, \#Z_0 \gamma \varepsilon \varepsilon | \$i\beta\$) \models \cdots \models (r_i, w/g((\gamma i)[m]), \#Z_0 \gamma \varepsilon \varepsilon | \$i\beta\$)$$

holds, where no ID with a state in $Q_N$ appears in the calculation $\cdots$.

Proof. In this proof, we sometimes write the stack representation $\# \cdots Z \cdots \#$ as $\# \cdots Z \# \cdots \$ with the head-reversed arrow $\rhd$. First, if $\gamma \not
\models [i]$, it holds that

$$(c_i, w, \#Z_0 \gamma \varepsilon \varepsilon | \$i\beta\$) \models (c_i, w, \#Z_0 | \gamma \varepsilon \varepsilon | \$i\beta\$)$$

and by $(\gamma i)[m] = \varepsilon$, we have $w = w/g((\gamma i)[m])$, as required. Henceforth, we assume that $\gamma \models [i]$ and the decomposition $\gamma = \gamma_0 | \gamma_1 (\gamma_1 \not
\models [i])$. Moreover, we can further decompose $\gamma_1 = \gamma_2 | \gamma_3 (\gamma_2 \not
\models [i], \gamma_3 \not
\models [i])$ because $\gamma i$ is matching. We prove by induction on $m$.

Case $m = 1$: By $\text{cunt} \gamma = 0$, $\gamma_2 \in (\Sigma \cup B_k)^*$ follows. Letting $w' \triangleq w/g(\gamma_2)$, we have

$$(c_i, w, \#Z_0 \gamma_0 [\gamma_1 \varepsilon \varepsilon | \$i\beta\$] \models (c_i, w, \#Z_0 \varepsilon \varepsilon | \$i\beta\$)$$

Therefore, the claim holds since no ID with a state in $Q_N$ appears in this calculation and $\gamma_2 = (\gamma i)[m] \gamma_2 \models (\gamma i)[0] = \gamma i = \gamma_0 | \gamma_2 | \gamma_3 | \gamma_3 \not
\models [i]$.

Case $\{1, \ldots, m\} \models m + 1$: Let $m_0 \triangleq \text{cunt} \gamma_0$ and $l \triangleq \text{cunt} \gamma_2 (\geq 0)$. Now, $m_0 + l \leq m = \text{cunt} \gamma$ holds and we write $\gamma_2 = \lambda_0 n_{m_0 + 1} \lambda_1 \cdots n_{m_0 + l} \lambda_l$. We also define $\eta_r \triangleq \gamma_0 | \lambda_0 n_{m_0 + 1} \cdots \lambda_{r - 1} n_{m_0 + r}$ for each $r \in \{1, \ldots, l\}$. By $\eta_r$ being a prefix of $\gamma i$ and Lemma 7, $\eta_r$ is matching and $(\eta_r)[m_{m_0 + r}] = (\gamma i)[m_{m_0 + r}], r \in \{1, \ldots, l\}$ holds. In particular, it follows that $n_{m_0 + r} \neq i$ for every $r$ (if there is $r$ such that $n_{m_0 + r} = i$, $\gamma_2 \not
\models [i]$ holds but this contradicts $\gamma_2 \not
\models [i]$). Thus, letting $w_0 \triangleq w$, $w_r \triangleq w_{r - 1}/g(\lambda_{r - 1})$, $w_r \triangleq w_r/g((\eta_r)[m_{m_0 + r}])$ and $w' = w_1/g(\lambda_i)$, we have

$$(c_i, w, \#Z_0 \gamma \varepsilon \varepsilon | \$i\beta\$)$$

(b by $\eta_i$ being matching and induction hypothesis)

$$= (c_i, w_0, \#Z_0 \gamma [\lambda_0 n_{m_0 + 1} \lambda_1 \cdots n_{m_0 + l} \lambda_l] \gamma_3 \varepsilon \varepsilon \varepsilon | \$i\beta\$)$$

$$= (c_{m_0 + 1}, w_{m_0 + 1}, \#Z_0 \gamma_0 [\lambda_0 n_{m_0 + 1} \lambda_1 \cdots n_{m_0 + l} \lambda_l] \gamma_3 \varepsilon \varepsilon \varepsilon | \$i\beta\$)$$

$$= (c_{m_0 + 1}, w_{m_0 + 1}, \#Z_0 \gamma_0 [\lambda_0 n_{m_0 + 1} \lambda_1 \cdots n_{m_0 + l} \lambda_l] \gamma_3 \varepsilon \varepsilon \varepsilon | \$i\beta\$)$$

(by similar calculation and induction hypothesis)
Then, by equation (\ref{eq:inductive_step}), we obtain
\[\gamma = \gamma_0 | \lambda_0 n_{m_0+1} \lambda_1 \cdots n_{m_t+1} \lambda_t | \gamma_3 \vDash v_0 v_1 \cdots v_n v_m\]
and decompose its substrings as
\[v_{m_0} = \chi_0 | \lambda_0, \quad v_{m_0+i} = \lambda_i | \chi_1, \quad \text{and} \quad \gamma_3 = \chi_1 n_{m_0+i+1} v_{m_0+i+1} \cdots n_m v_m.\]

Then, by equation (\ref{eq:inductive_step}), we can write \((\gamma_i)(m)\) as
\[v_0 \cdots \chi_0 | \lambda_0 g((\gamma_i)(m+1)) v_{m_0+1} \cdots g((\gamma_i)(m+1)) \lambda_1 | \chi_1 g((\gamma_i)(m_0+i+1)) v_{m_0+i+1} \cdots g((\gamma_i)(m)) v_m.\]

That is, it holds that \(\gamma'_3 \triangleq \chi_1 g((\gamma_i)(m+1)) v_{m_0+i+1} \cdots g((\gamma_i)(m)) v_m \not\vDash [i] \) by \(\gamma_3 \not\vDash [i] \), and we obtain \((\gamma_i)(m+1) = \lambda_0 g((\gamma_i)(m_0+1)) \lambda_1 \cdots g((\gamma_i)(m_0+t)) \lambda_t \), as shown above. Therefore, the claim holds for \(m+1\) since \(w' = w/g((\gamma_i)(m+1)).\)

**Proof of Lemma 15.** For arbitrary \(w \in \Sigma^*\), by Lemma 23,
\[(q, w, \#Z_0 \gamma | \$) \vdash_{(3)} (W_{q'}, w, \#Z_0 \gamma | \$) \vdash_{(4)} (e_i, w, \#Z_0 \gamma | \$) \vdash_{(5)} (q', w, \#Z_0 \gamma | \$) \vdash_{(15)} (L_{q', i}, w, \#Z_0 \gamma | \$) \vdash_{(17)} (q', w, \#Z_0 \gamma | \$)\]
(\ref{eq:proof_lemma_15}) holds. Assuming (a), we can replace \(\vdash\) in equation (\ref{eq:proof_lemma_15}) with \(\vDash\) by Lemma 21 because \(w/g((\gamma_i)(m)) = w' \in \Sigma^*\) holds, and therefore, (b) follows. Supposing (b) conversely, we have \((q, w, \#Z_0 \gamma | \$) \vdash_{(3)} (W_{q'}, w, \#Z_0 \gamma | \$) \vdash_{(5)} (q', w, \#Z_0 \gamma | \$)\), where no ID with a state in \(Q_N\) appears in either this calculation or (\ref{eq:proof_lemma_15}) except in their leftmost and rightmost IDs. Therefore, their two calculations coincide by the determinism of \(\vDash\). In particular, we obtain \(p = q', w' = w/g((\gamma_i)(m))\) and \(\beta = Z_0 \gamma | i\) by the equality of their rightmost IDs, and thus, (a) follows because \(w' \in \Sigma^*\).