Information and Computation 208 (2010)385–394

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/ic

More concise representation of regular languages by automata and regular expressions☆

Viliam Geffert [a], Carlo Mereghetti [b,*], Beatrice Palano [b]

[a] Department of Computer Science, P. J. Šafárik University, Jesenná 5, 04154 Košice, Slovakia
[b] Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, via Comelico 39, 20135 Milano, Italy

ARTICLE INFO

Article history:
Received 12 February 2009
Revised 27 July 2009
Available online 18 January 2010

Keywords:
Pushdown automata
Regular expressions
Straight line programs
Descriptional complexity

ABSTRACT

We consider two formalisms for representing regular languages: *constant height pushdown automata* and *straight line programs for regular expressions*. We constructively prove that their sizes are *polynomially related*. Comparing them with the sizes of finite state automata and regular expressions, we obtain *optimal exponential and double exponential gaps*, i.e., a more concise representation of regular languages.

1. Introduction

Several systems for representing *regular languages* have been presented and studied in the literature. For instance, for the original model of finite state automaton [11], a lot of modifications have been introduced: nondeterminism [11], alternation [4], probabilistic evolution [10], two-way input head motion [12], etc. Other important formalisms for defining regular languages are, e.g., regular grammars [7] and regular expressions [8]. All these models have been proved to share the same expressive power by exhibiting simulation results.

However, representation of regular languages may be much more "economical" in one system than another. This consideration has lead to a consolidated line of research—sometimes referred to as *descriptional complexity*—aiming to compare formalisms by comparing their *size*. The oldest and most famous result in this sense is the optimal exponential gap between the size of a deterministic (DFA) and nondeterministic (NFA) finite state automaton [9,11].

In this paper, we study the size of two formalisms for specifying regular languages, namely: a *constant height pushdown automaton* and a *straight line program for a regular expression*.

First, it is well known that the languages recognized by nondeterministic pushdown automata (NPDAs) form the class of context-free languages, a proper superclass of the languages accepted by deterministic pushdown automata (DPDAs), which in turn is a proper superclass of regular languages [7]. However, if the maximum height of the pushdown store is bounded by a constant, i.e., if it does not depend on the input length, it is a routine exercise to show that such machine accepts a regular

☆ This work was partially supported by the Slovak Grant Agency for Science (VEGA) under contract "Combinatorial structures and complexity of algorithms", and by the Italian MIUR under project "Aspetti matematici e applicazioni emergenti degli automi e dei linguaggi formali: metodi probabilistici e combinatori in ambito di linguaggi formali". A preliminary version of this work was presented at the 12th International Conference Developments in Language Theory, Kyoto, Japan, September 16–19, 2008.

* Corresponding author.
E-mail addresses: viliam.geffert@upjs.sk (V. Geffert), mereghetti@dsi.unimi.it (C. Mereghetti), palano@dsi.unimi.it (B. Palano).

0890-5401/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2010.01.002

language. (In general, it is not possible to bound the pushdown height by a constant.) Nevertheless, a representation by constant height PDAs can potentially be more succinct than by the standard finite state automata, both for deterministic and nondeterministic machines. Here, we prove *optimal exponential* and *optimal double exponential simulation costs* of constant height PDAs by finite state automata. We also get an *exponential lower bound* for eliminating nondeterminism in constant height PDAs. (Some related results on PDAs accepting regular languages vs. finite state automata can be found in [13].)

Second, a natural counterpart to constant height PDAs turned out to be straight line programs (SLPs), in perfect analogy to the relation of finite state automata vs. regular expressions. An SLP is a loopless program representing a directed acyclic graph, the internal nodes of which represent the basic regular operations (i.e., union, concatenation, and star). Compared with the size of the standard regular expression represented by a binary tree, an SLP can be more succinct by using the fact that replicated subtrees are shared. Here, we prove an *optimal exponential gap* between the sizes of regular expressions and SLPs.

It is well known that the sizes of the classical NFAs and regular expressions are not polynomially related: the cost for the conversion of regular expressions to NFAs is linear [2,3], but it is exponential for the opposite conversion [5]. (For more bibliography related to these results, the reader is referred to [6].) On the contrary, here we design conversions constructing equivalent constant height PDAs from SLPs and vice versa, in such a way that *the costs are polynomial in both directions*. This clearly shows that the exponential gap between the classical NFAs and the classical regular expressions has been obtained only due to the fact that the resulting regular expression must contain several (in fact, exponentially many) replicated copies of the same subexpression. (Otherwise, such a witness regular expression could not be "compressed" into a polynomial-size SLP, as stated by our result.)

2. Preliminaries

In this section, we present the formalisms we shall be dealing with. We begin by introducing straight line programs for representing regular expressions, emphasizing also their directed acyclic graph (DAG) form. Then we turn to finite state and pushdown automata, the latter machines being presented in an equivalent but simplified dynamics. Finally, the notion of constant height pushdown automata is considered. For each formalism, a size measure is introduced, to compare their descriptive powers.

We assume the reader is familiar with the basic notions from formal language theory (see, e.g., [7]). The set of natural numbers, including zero, is denoted here by **N**.

2.1. Straight line programs for regular expressions

A *regular expression*, defined over a given alphabet Σ, is:

(i) $\emptyset$, ε, or any symbol $a \in \Sigma$,
(ii) r_1+r_2, $r_1 \cdot r_2$, or r_1^*, if r_1 and r_2 are regular expressions.

The language represented by a given regular expression r, denoted by $L(r)$, is inductively defined as: $\emptyset$, $\{\varepsilon\}$, $\{a\}$ in case (i), and $L(r_1) \cup L(r_2)$, $L(r_1) \cdot L(r_2)$, $L(r_1)^*$, respectively, in case (ii). With a slight abuse of terminology, we often identify a regular expression with the language it represents. Thus, Σ^* is the set of words over Σ, including the empty word ε. By $|w|$, we denote the length of a word $w \in \Sigma^*$ and by Σ^i the set of words of length $i \in \mathbf{N}$, with $\Sigma^0 = \{\varepsilon\}$ and $\Sigma^{\leqslant m} = \bigcup_{i=0}^{m} \Sigma^i$. By $|\Sigma|$, we denote the cardinality of the set Σ.

Definition 1. The size of a regular expression r over Σ, denoted by size(r), is the number of occurrences of symbols of $\{\emptyset, \varepsilon\} \cup \Sigma$, plus the number of occurrences of operators $+$, $\cdot$, and $*$ inside r.

Example 1. For $r = a \cdot (a+b)^* + (a+b)^* \cdot b \cdot a^*$, we have size$(r) = 16$.

A convenient way for representing regular expressions is provided by straight line programs (see, e.g., [1]). Given a set of variables $X = \{x_1, \ldots, x_\ell\}$, a *straight line program for regular expressions* (SLP) on Σ is a finite sequence of instructions

$$P \equiv \text{instr}_1 ; \ \ldots \ \text{instr}_i ; \ \ldots \ \text{instr}_\ell,$$

where the ith instruction instr_i has one of the following forms:

(i) $x_i := \emptyset$, $x_i := \varepsilon$, or $x_i := a$ for any symbol $a \in \Sigma$,
(ii) $x_i := x_j + x_k$, $x_i := x_j \cdot x_k$, or $x_i := x_j^*$, for $1 \leqslant j, k < i$.

Such program P expands to the regular expression reg-exp$(P) = x_\ell$, obtained by nested macro-expansion of the variables $x_1, \ldots, x_{\ell-1}$, using the right parts of their instructions. Notice that a variable may be reused several times in the right parts. Such a number of occurrences is called a *fan-out* of the variable. The fan-out of x_ℓ is 0, while the fan-out of any other variable is at least 1, since, with the exception of x_ℓ, we can remove instructions defining variables not used at least once in some right part.

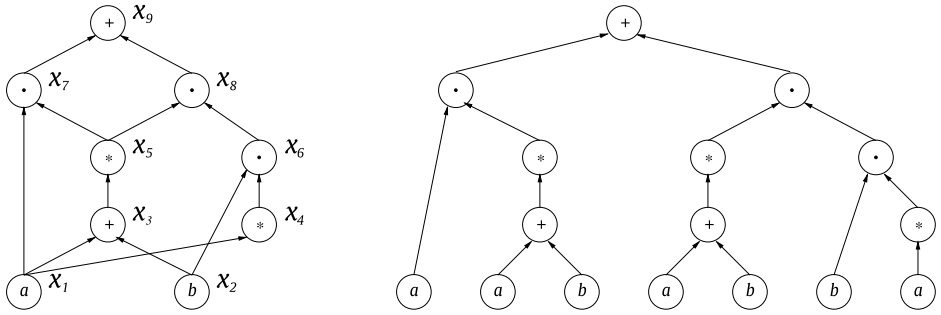

Fig. 1. Left: The DAG D_P associated with the SLP P introduced in Example 2. Labels are written inside the nodes, sources are labeled by $a, b \in \Sigma$, and the sink is the topmost vertex. The number of vertices is length$(P) = 9$, the maximum out-degree is fan-out$(P) = 3$ (given by the vertex labeled by a), and depth$(D_P) = 4$. Right: The corresponding classical representation as a binary tree of the regular expression.

Definition 2. The *size of a straight line program P* is the ordered pair

$$\text{size}(P) = (\text{length}(P), \text{fan-out}(P)),$$

where length(P) denotes the number of instructions in P, and fan-out(P) the maximum fan-out of its variables.

Example 2. Let us show an SLP for the regular expression in Example 1:

$$\begin{aligned} P \equiv\ & x_1 := a;\\ & x_2 := b;\\ & x_3 := x_1 + x_2;\\ & x_4 := x_1^*;\\ & x_5 := x_3^*;\\ & x_6 := x_2 \cdot x_4;\\ & x_7 := x_1 \cdot x_5;\\ & x_8 := x_5 \cdot x_6;\\ & x_9 := x_7 + x_8. \end{aligned}$$

Clearly, reg-exp$(P) = x_9 = a \cdot (a+b)^* + (a+b)^* \cdot b \cdot a^*$, and size$(P) = (9, 3)$.

Each SLP P can be associated with a *vertex-labeled directed acyclic graph* (DAG) $D_P = (V, E)$, where the vertices in $V = \{v_1, \ldots, v_\ell\}$ correspond to the respective variables in $X = \{x_1, \ldots, x_\ell\}$. That is, a vertex v_i is labeled by $e \in \{\emptyset, \varepsilon\} \cup \Sigma$, whenever the ith instruction is $x_i := e$, and by '$+$' or '$\cdot$', whenever this instruction is $x_i := x_j + x_k$ or $x_i := x_j \cdot x_k$, respectively. In the case of a binary operation, the directed arcs (v_j, v_i) and (v_k, v_i) are included in E, to connect v_i with its left and right sons, respectively. Similarly, v_i is labeled by '$*$', if the ith instruction is $x_i := x_j^*$, with (v_j, v_i) included in E.

This idea is illustrated by Fig. 1, where the DAG D_P corresponding to the SLP P in Example 2 is displayed, together with the usual tree-like representation for the computed regular expression.

From the definition of P, it is easy to see that D_P does not contain any directed cycle and that the fan-out of a variable establishes the out-degree of the corresponding vertex. Thus, there exists a unique *sink* v_ℓ (vertex without outgoing arcs) corresponding to the variable x_ℓ and some *sources* (vertices without ingoing arcs) labeled by $e \in \{\emptyset, \varepsilon\} \cup \Sigma$. We define the *depth* of D_P, depth(D_P), as the maximum length of a path from a source to the sink.

In what follows, we point out some relations between the sizes of an SLP and the corresponding regular expression. Clearly, an SLP with fan-out 1 is just an ordinary regular expression written down in a slightly different way:

Proposition 1. *For each* SLP P, length$(P) =$ size(reg-exp(P)) *if and only if* fan-out$(P) = 1$.

In general, however, SLPs can be exponentially more succinct than the classical regular expressions. The following example shows that, even only with fan-out 2, we get an exponential gap.

Example 3. Consider the SLP P_ℓ on $\Sigma = \{a\}$:

$$\begin{aligned} P_\ell \equiv\ & x_1 := a;\\ & x_2 := x_1 \cdot x_1;\\ & x_3 := x_2 \cdot x_2;\\ & \quad\vdots\\ & x_\ell := x_{\ell-1} \cdot x_{\ell-1}. \end{aligned}$$

It is easy to see that fan-out$(P_\ell) = 2$ and reg-exp$(P_\ell) = a^{2^{\ell-1}}$. Thus, for any $\ell \geqslant 1$, we obtain size(reg-exp(P_ℓ)) $= 2^{\text{length}(P_\ell)} - 1$.

The above example establishes the *optimality* of the following general result.

Proposition 2. *Let P and P' be two equivalent* SLPs *such that* fan-out$(P') = 1$. *Then,* length$(P') \leqslant 2^{\text{depth}(P)}$.

2.2. Constant height pushdown automata

It is well known that the regular expressions (hence, SLPs as well) represent the class of *regular languages* [7]. This class can also be represented by automata.

A *nondeterministic finite state automaton* (NFA) is formally defined as a 5-tuple $A = \langle Q, \Sigma, H, q_0, F\rangle$, where Q is the finite set of states, Σ the finite input alphabet, $H \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ the transition relation, $q_0 \in Q$ the initial state, and $F \subseteq Q$ the set of final (accepting) states. An input string is *accepted*, if there exists a computation beginning in the state q_0 and ending in some final state $q \in F$ after reading this input. The *language accepted by A*, denoted by $L(A)$, is the set of all inputs accepted by A. The automaton A is *deterministic* (DFA), if there are no ε-transitions in H and, for every $q \in Q$ and $a \in \Sigma$, there exists at most one $p \in Q$ such that $(q, a, p) \in H$.

In the literature, a *nondeterministic pushdown automaton* (NPDA) is usually obtained from an NFA by adding a pushdown store, containing symbols from Γ, the pushdown alphabet. At the beginning, the pushdown contains a single initial symbol $Z_0 \in \Gamma$. The transition relation is usually given in the form of δ, a mapping from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$. Let $\delta(q, x, X) \ni (p, \omega)$. Then A, being in the state q, reading x on the input and X on top of the pushdown, can reach the state p, replace X by ω and finally, if $x \neq \varepsilon$, advance the input head one symbol. Its *deterministic* version (DPDA) is obtained in the usual way. (For more details, see, e.g., [7].)

For technical reasons, we shall introduce the NPDAs in the following form, where moves manipulating the pushdown store are clearly distinguished from those reading the input tape: an NPDA is a 6-tuple $A = \langle Q, \Sigma, \Gamma, H, q_0, F\rangle$, where $Q, \Sigma, \Gamma, q_0, F$ are defined as above, while $H \subseteq Q \times (\{\varepsilon\} \cup \Sigma \cup \{+, -\} \cdot \Gamma) \times Q$ is the *transition relation* with the following meaning:

(i) $(p, \varepsilon, q) \in H$: A reaches the state q from the state p without using the input tape or the pushdown store,
(ii) $(p, a, q) \in H$: A reaches the state q from the state p by reading the symbol a from the input, not using the pushdown store,
(iii) $(p, -X, q) \in H$: if the symbol on top of the pushdown is X, A reaches the state q from the state p by popping X, not using the input tape,
(iv) $(p, +X, q) \in H$: A reaches the state q from the state p by pushing the symbol X onto the pushdown, not using the input tape.

Such machine does not use any initial pushdown symbol: an accepting computation begins in the state q_0 with the empty pushdown store and the input head at the beginning, and ends in a final state $q \in F$ after reading the entire input. A *deterministic pushdown automaton* (DPDA) is obtained from NPDA by claiming that it can never get into a situation in which more than one instruction can be executed. (As an example, a DPDA cannot have a pair of instructions of the form (q, ε, p_1) and (q, a, p_2).) As usual, $L(A)$ denotes the *language accepted by A*.

It is not hard to see that any NPDA in the classical form can be turned into this latter form and vice versa, preserving determinism in the case of DPDAs.

At the cost of one more state, we can transform our NPDAs so that they accept by entering a *unique* final state at the end of input processing, with *empty* pushdown store. Notice, however, that the following transformation does not preserve determinism.

Lemma 1. *For any* NPDA $A = \langle Q, \Sigma, \Gamma, H, q_0, F\rangle$, *there exists an equivalent* NPDA $A' = \langle Q \cup \{q_f\}, \Sigma, \Gamma, H', q_0, \{q_f\}\rangle$, *where A' accepts by entering the unique final state $q_f \notin Q$ with empty pushdown store at the end of the input*.

Proof. The new transition relation H' consists of H plus the following new transitions: for any $q \in F$, we add the ε-transition (q, ε, q_f). This leads A' to the unique final state q_f whenever A reaches any final state in F. Then, we empty the pushdown by adding transitions $(q_f, -X, q_f)$ for every $X \in \Gamma$. It is easy to verify that $L(A') = L(A)$. □

Given a constant $h \in \mathbf{N}$, we say that the NPDA A is of pushdown height h if, for any word in $L(A)$, there exists an accepting computation along which the pushdown store never contains more than h symbols.

From now on, we shall consider *constant height* NPDAs only. Such machine will be denoted by a 7-tuple $A = \langle Q, \Sigma, \Gamma, H, q_0, F, h\rangle$, where $h \in \mathbf{N}$ is a constant denoting the pushdown height, and all other elements are defined as above. By definition, the meaning of the transitions in the form (iv) is modified as follows:

(iv') $(p, +X, q) \in H$: *if the current pushdown store height is smaller than h*, then A reaches the state q from the state p by pushing the symbol X onto the pushdown, not using the input tape.

Thus, this kind of transitions is disabled if the current pushdown height is equal to h. A constant height NPDA can be replaced by an equivalent standard NPDA (without a built-in limit h on the pushdown) by storing, in the finite control states, a counter recording permitted pushdown heights (i.e., a number ranging within $\{0, \ldots, h\}$), hence, paying by a constant increase in the number of states.

Note that, for $h = 0$, the definition of constant height NPDA exactly coincides with that of an NFA, as one may easily verify. Moreover, Lemma 1 holds for constant height NPDAs as well, which enables us to consider acceptance by a *single final state* and, at the same time, with *empty pushdown store*. Therefore, from now on, a constant height NPDA will assume the form $A = \langle Q, \Sigma, \Gamma, H, q_0, \{q_f\}, h\rangle$.

Definition 3. The *size of a constant height* NPDA $A = \langle Q, \Sigma, \Gamma, H, q_0, \{q_f\}, h\rangle$ is the ordered triple

$$\text{size}(A) = (|Q|, |\Gamma|, h).$$

Observe that this definition immediately gives that the size of an NFA is completely determined by the number of its states, since it is $(|Q|, 0, 0)$.

3. From a constant height NPDA to an SLP

In this section, we show how to convert a constant height NPDA into an equivalent SLP. Yet, we focus on the cost of such a conversion and prove that the size of the resulting SLP is polynomial in the size of the original NPDA.

In what follows, in order to simplify our notation, a "long" regular expression $r_1 + r_2 + \cdots + r_n$ will also be written as $\sum_{i=1}^{n} r_i$. We define an "empty sum" as the regular expression $\emptyset$.

Let $A = \langle\{q_1, \ldots, q_k\}, \Sigma, \Gamma, H, q_1, \{q_k\}, h\rangle$ be a constant height NPDA. For each $i, j \in \{1, \ldots, k\}$, $s \in \{0, \ldots, k\}$, and $t \in \{0, \ldots, h\}$, we define $[q_i, s, t, q_j]$ to be the set of strings $x \in \Sigma^*$ such that, for each of them, there exists at least one computation with the following properties.

- The computation begins in the state q_i, with the pushdown empty.
- After reading the entire string x from the input, the computation ends in the state q_j, with the pushdown empty again.
- Any time the pushdown is empty during this computation, the current finite control state is from the set $\{q_1, \ldots, q_s\}$. (This restriction does not apply to q_i, q_j themselves.) For $s = 0$, the pushdown is never empty in the meantime.
- During this computation, the pushdown height never exceeds t.

We are now going to give an algorithm, consisting of two PHASES, which dynamically constructs regular expressions describing all sets $[q_i, s, t, q_j]$. The ultimate goal is to obtain $[q_1, k, h, q_k]$, the regular expression for the language $L(A)$. First, we easily construct $[q_i, 0, 0, q_j]$ for each q_i, q_j, basically by a direct inspection of the transitions in H. After that, we gradually increment the parameter s from 1 to k, thus obtaining $[q_i, k, 0, q_j]$ for each q_i, q_j. Second, we show how to upgrade from the parameters $k, t-1$ to parameters $0, t$, and then from parameters s, t to $s+1, t$ leading up to $[q_i, k, h, q_j]$ for each q_i, q_j.

For any $1 \leqslant i, j \leqslant k$, we let

$$\Psi_0(q_i, q_j) = \{\alpha \in \Sigma \cup \{\varepsilon\} : (q_i, \alpha, q_j) \in H\} \cup \Delta_{i,j}, \quad \text{where}$$
$$\Delta_{i,j} = \begin{cases} \emptyset, & \text{if } i \neq j, \\ \{\varepsilon\}, & \text{if } i = j. \end{cases}$$

Thus, $\Psi_0(q_i, q_j)$ consists of the input symbols, possibly with ε, taking A from q_i to q_j in at most one computation step, without involving the pushdown.

PHASE I:
for each $1 \leqslant i, j \leqslant k$ **do**
  $[q_i, 0, 0, q_j] = \sum_{\alpha \in \Psi_0(q_i, q_j)} \alpha$;
for $s = 0$ **to** $k-1$ **do**
 for each $1 \leqslant i, j \leqslant k$ **do**
  $[q_i, s+1, 0, q_j] = [q_i, s, 0, q_j] + [q_i, s, 0, q_{s+1}] \cdot [q_{s+1}, s, 0, q_{s+1}]^* \cdot [q_{s+1}, s, 0, q_j]$;

Actually, $[q_i, 0, 0, q_j]$ is the regular expression representing the set $\Psi_0(q_i, q_j)$ by a formal sum of its elements, if any, otherwise by $\emptyset$. After that, this phase computes the regular expression $[q_i, s+1, 0, q_j]$ by adding to $[q_i, s, 0, q_j]$ the strings which enable A to use also the state q_{s+1}, without using the pushdown. This is exactly the Kleene's recursion for computing regular expressions from NFAs [7].

Then the second phase starts, for which we need some more notation. For any $1 \leqslant i, j \leqslant k$ and $X \in \Gamma$, we let

$$\Psi_+(q_i, X) = \{q \in Q : (q_i, +X, q) \in H\},$$
$$\Psi_-(X, q_j) = \{q \in Q : (q, -X, q_j) \in H\}.$$

Thus, $\Psi_+(q_i, X)$ is the set of states reachable by A from q_i while pushing the symbol X onto the stack, and $\Psi_-(X, q_j)$ is the set of states from which A reaches q_j while popping X.

PHASE II:
for $t = 1$ **to** h **do begin**
  for each $1 \leqslant i, j \leqslant k$ **do**
$(\diamond)[q_i, 0, t, q_j] = [q_i, 0, t-1, q_j] + \sum_{X\in\Gamma} \sum_{p\in\Psi_+(q_i,X)} \sum_{q\in\Psi_-(X,q_j)} [p, k, t-1, q]$;
  for $s = 0$ **to** $k-1$ **do**
    for each $1 \leqslant i, j \leqslant k$ **do**
      $[q_i, s+1, t, q_j] = [q_i, s, t, q_j] + [q_i, s, t, q_{s+1}] \cdot [q_{s+1}, s, t, q_{s+1}]^* \cdot [q_{s+1}, s, t, q_j]$;
end;
return $[q_1, k, h, q_k]$

For this phase, the construction marked by $(\diamond)$ is worth explaining. By definition, $[q_i, 0, t, q_j]$ consists of $[q_i, 0, t-1, q_j]$ plus the set of all strings x such that:

- starting in the state q_i with empty pushdown, A pushes some symbol X on the stack in the first move—the first two summands in $(\diamond)$,
- pops this symbol X in the last move only, by entering the state q_j with empty pushdown—third summand in $(\diamond)$—and,
- during the computation processing the input string x, A pushes no more than $t-1$ symbols over X, so that globally the pushdown height is at least 1 and never exceeds t—variables $[p, k, t-1, q]$ in $(\diamond)$.

Notice that our algorithm can easily be regarded to as an SLP P_A for $L(A)$. Informally, we can consider all $[q_i, s, t, q_j]$'s as the variables of P_A. Before PHASE I, we need to define some input variables, by instructions $x_\alpha := \alpha$ of the form (i), at most one per each $\alpha \in \Sigma \cup \{\varepsilon, \emptyset\}$. (See the definition of SLPs in Section 2.1.) Then, we easily unroll the for-cycles and sums, which translates the two phases into a finite sequence of instructions. To keep all such instructions in the form (ii), we only have to introduce some new auxiliary variables. Clearly, the output variable is $[q_1, k, h, q_k]$. The correctness can formally be proved by a double-induction on parameters s, t in $[q_i, s, t, q_j]$.

Concerning the length of P_A, there exist no more than $|\Sigma| + 2$ input instructions, while those of PHASE I do not exceed $k^2 \cdot |\Sigma| + k^3 \cdot 4$. Finally, PHASE II requires no more than $h \cdot k^3 \cdot (k \cdot |\Gamma| + 4)$ instructions, using also the fact that the number of variables involved in the triple sum of $(\diamond)$ is bounded by $|\Gamma| \cdot k^2$. Summing up, we get length$(P_A) \leq \mathcal{O}(h \cdot k^4 \cdot |\Gamma| + k^2 \cdot |\Sigma|)$. Moreover, the fan-out of any variable in P_A does not exceed $k^2 + 1$. Formally, we have shown:

Theorem 1. *Let $A = \langle Q, \Sigma, \Gamma, H, q_0, \{q_f\}, h\rangle$ be a constant height* NPDA. *Then there exists an* SLP P_A *such that* reg-exp(P_A) *denotes $L(A)$, with*

$$\text{length}(P_A) \leq \mathcal{O}(h \cdot |Q|^4 \cdot |\Gamma| + |Q|^2 \cdot |\Sigma|) \text{ and } \text{fan-out}(P_A) \leqslant |Q|^2 + 1.$$

That is, for regular languages over a fixed alphabet, the size of P_A is polynomial in the size of A.

4. From an SLP to a constant height NPDA

Let us now show the converse result, namely, that any SLP can be turned into an equivalent constant height NPDA whose size is linear in the size of the SLP.

Let P be an SLP with variables $\{x_1, \ldots, x_\ell\}$ on Σ. To help intuition, we present our construction of an equivalent constant height NPDA by referring to the associated DAG D_P, as described in Section 2.1, where the vertex v_i corresponds to the variable x_i. The enumeration of the variables in P induces a topological ordering on the vertices of D_P. Now we proceed as follows.

For $i = 1, \ldots, \ell$, we construct an NPDA $A_i = \langle Q_i, \Sigma, \Gamma_i, H_i, q_{0,i}, \{q_{f,i}\}\rangle$ such that $L(A_i)$ is exactly the language denoted by reg-exp(x_i), a regular expression obtained by expanding the DAG rooted in v_i. For a source node v_i, we define an "elementary" NPDA without a pushdown store—actually an NFA. An NPDA for an inner node is constructed inductively, using, as subprograms, NPDAs for vertices that are topologically smaller. The desired NPDA is A_ℓ. We start with the construction for sources.

SOURCES: Let the source node v_i be labeled by $\alpha \in \Sigma \cup \{\varepsilon, \emptyset\}$. If $\alpha \neq \emptyset$, the single-transition NPDA recognizing α is defined as

$$A_i = \langle \{q_{0,i}, q_{f,i}\}, \Sigma, \emptyset, \{(q_{0,i}, \alpha, q_{f,i})\}, q_{0,i}, \{q_{f,i}\}\rangle.$$

For $\alpha = \emptyset$, we define the transition-free NPDA

$$A_i = \langle \{q_{0,i}, q_{f,i}\}, \Sigma, \emptyset, \emptyset, q_{0,i}, \{q_{f,i}\}\rangle.$$

In the latter case, the final state $q_{f,i}$ is settled only for technical reasons, but actually it cannot be reached. This completes the basis of our inductive construction.

Now, let us define the inductive step. Let v_i be an internal node in the DAG D_P. The construction of A_i depends on the label of v_i, and so we have the following cases:

LABEL '+': v_i is a vertex labeled by '+', with two ingoing arcs from vertices v_a and v_b, for $1 \leqslant a, b < i$, representing the instruction $x_i := x_a + x_b$. By the inductive hypothesis, we assume the NPDAs A_a and A_b are given for the vertices v_a and v_b, respectively. Then we define

$$A_i = \langle Q_a \cup Q_b \cup \{q_{0,i}, q_{f,i}\},\ \Sigma,\ \Gamma_a \cup \Gamma_b \cup \{X_i\},\ H_i,\ q_{0,i},\ \{q_{f,i}\}\rangle, \text{ with}$$
$$H_i = H_a \cup H_b \cup \{(q_{0,i}, +X_i, q_{0,a}), (q_{f,a}, -X_i, q_{f,i}), (q_{0,i}, +X_i, q_{0,b}), (q_{f,b}, -X_i, q_{f,i})\}.$$

Basically, A_i nondeterministically chooses to activate either the NPDA A_a or the NPDA A_b. Before activation, A_i pushes the symbol X_i onto the pushdown, and pops it right at the end of the processing of the activated NPDA.

LABEL '·': v_i is a vertex labeled by '·', with two ingoing arcs from vertices v_a and v_b, for $1 \leqslant a, b < i$, representing the instruction $x_i := x_a \cdot x_b$. For the respective vertices v_a and v_b, we assume inductively again that the NPDAs A_a and A_b are given. Then we define

$$A_i = \langle Q_a \cup Q_b \cup \{q_{0,i}, q_{m,i}, q_{f,i}\},\ \Sigma,\ \Gamma_a \cup \Gamma_b \cup \{L_i, R_i\},\ H_i,\ q_{0,i},\ \{q_{f,i}\}\rangle, \text{ with}$$
$$H_i = H_a \cup H_b \cup \{(q_{0,i}, +L_i, q_{0,a}), (q_{f,a}, -L_i, q_{m,i}), (q_{m,i}, +R_i, q_{0,b}), (q_{f,b}, -R_i, q_{f,i})\}.$$

Here, A_i sequentially activates A_a and A_b. Before activating A_a, it pushes the symbol L_i onto the pushdown, and pops it out at the end of A_a-processing, by reaching the state $q_{m,i}$. From this state, A_i pushes another symbol R_i onto the pushdown, thus activating A_b, and pops it out at the end of A_b-processing.

LABEL '$*$': v_i is a vertex labeled by '$*$', with a single ingoing arc from a vertex v_a, for $1 \leqslant a < i$, representing the instruction $x_i := x_a^*$. By inductive hypothesis, we assume the NPDA A_a is given for the vertex v_a. Then we define

$$A_i = \langle Q_a \cup \{q_{0,i}, q_{f,i}\},\ \Sigma,\ \Gamma_a \cup \{X_i\},\ H_i,\ q_{0,i},\ \{q_{f,i}\}\rangle, \text{ with}$$
$$H_i = H_a \cup \{(q_{0,i}, \varepsilon, q_{f,i}), (q_{0,i}, +X_i, q_{0,a}), (q_{f,a}, -X_i, q_{0,i})\}.$$

Here, A_i nondeterministically chooses to activate A_a a certain number of times, including zero. Each time A_a is going to be activated, A_i pushes the symbol X_i onto the pushdown, and pops it out at the end of A_a-processing by returning to the state $q_{0,i}$. In the state $q_{0,i}$, A_i can also terminate the iteration by reaching the state $q_{f,i}$ with an ε-move. Notice that the final state $q_{f,i}$ is used only for technical reasons, but it can actually be eliminated, together with the transition $(q_{0,i}, \varepsilon, q_{f,i})$, by setting $q_{0,i} = q_{f,i}$.

Informally, for parsing an input string, A_ℓ verifies matching with reg-exp(P) by starting from the source node of D_P and traveling along the arcs. When traveling towards sources, one symbol per each visited vertex is pushed onto the pushdown. Vice versa, when traveling back towards the sink, the pushdown symbols are popped. These operations are needed to record the sequence of visited vertices, since some of them are shared (i.e., their fan-out is greater than 1). By induction on the depth of D_P, one may formally prove that $L(A_\ell)$ is the language denoted by reg-exp(P).

Let us measure the size of the NPDA A_ℓ. In the construction of each A_i, we use at most three new states and two new pushdown symbols, if v_i is an inner node, but only two states with no pushdown symbols if it is a source node. Hence, $|Q_\ell| < 3\ell$ and $|\Gamma_\ell| < 2\ell$. Finally, the pushdown height of A_ℓ is easily seen to be equal to depth$(D_P) < \ell$.

Actually, some improvements on the size of A_ℓ can be obtained. Given the dynamics of A_ℓ above described, the use of the pushdown turns out to be necessary only for shared vertices, so that the machine can identify the proper ancestor by popping a symbol from the pushdown. Thus, by renaming the pushdown symbols, we can reduce the size of the pushdown alphabet to fan-out(P). Moreover, for vertices with fan-out equal to 1, the moves involving the pushdown can be transformed into ε-moves, thus reducing the pushdown height. Clearly, one may also eliminate ε-moves, possibly reducing the number of states. In conclusion:

Theorem 2. *Let P be an* SLP. *Then there exists a constant height* NPDA $A_P = \langle Q, \Sigma, \Gamma, H, q_0, \{q_f\}, h\rangle$ *such that $L(A_P)$ is denoted by* reg-exp(P) *and the size of A_P is linear in the size of P. In particular*

$$|Q| < 3 \cdot \text{length}(P),\ \ |\Gamma| = \text{fan-out}(P),\ \ \textit{and}\ \ h < \text{length}(P).$$

More precisely, h equals to the maximum number of vertices with fan-out greater than 1 *along paths from sources to the sink*.

5. Constant height PDAs vs. finite state automata

Here, we compare the sizes of constant height PDAs and the standard finite state automata. In what follows, NPDAs (but not DPDAs) are in the form stated in Lemma 1, i.e., they accept by entering a unique final state with empty pushdown. First, we prove an exponential upper bound on the size of NFAs (DFAs) simulating constant height NPDAs (DPDAs, respectively).

Proposition 3. *For each constant height* NPDA $A = \langle Q, \Sigma, \Gamma, H, q_0, \{q_f\}, h\rangle$, *there exists an equivalent* NFA $A' = (Q', \Sigma, H', q_0', \{q_f'\})$ *with* $|Q'| \leq |Q| \cdot |\Gamma^{\leqslant h}|$ *states. If* $B = \langle Q, \Sigma, \Gamma, H, q_0, F, h\rangle$ *is a constant height* DPDA, *we can construct an equivalent* DFA *with no more than* $|Q| \cdot |\Gamma^{\leqslant h}|$ *states*.

Proof. We define the set of states in A' as $Q' = Q \times \Gamma^{\leqslant h}$. During the simulation, A' records the current state q of A together with the current pushdown content γ. (Here, γ grows to right.) We set $q_0' = (q_0, \varepsilon)$ and $q_f' = (q_f, \varepsilon)$. The transitions in H' are defined as follows, for each $p, q \in Q$, $\alpha \in \Sigma \cup \{\varepsilon\}$, $\gamma \in \Gamma^{\leqslant h}$, and $X \in \Gamma$:

- If $(p, \alpha, q) \in H$, then $((p, \gamma), \alpha, (q, \gamma)) \in H'$.
- If $(p, +X, q) \in H$ and $|\gamma| < h$, then $((p, \gamma), \varepsilon, (q, \gamma X)) \in H'$.
- If $(p, -X, q) \in H$ and $|\gamma| < h$, then $((p, \gamma X), \varepsilon, (q, \gamma)) \in H'$.

It is an easy task to verify that $L(A') = L(A)$.

In the deterministic case, we convert B into the NFA A' as above, with the only exception: instead of $\{q_f'\}$, we take $F' = F \times \Gamma^{\leqslant h}$ as the set of final states. It is easy to see that, for each $q' \in Q'$ and $a \in \Sigma \cup \{\varepsilon\}$, there exists at most one transition in A'. Moreover, if there exists an ε-move from $q' \in Q'$, then there is no ε-free move from q', for any $a \in \Sigma$. For this type of NFA, the standard technique for ε-move elimination is easily seen to yield an equivalent DFA without increasing the number of states. □

By Proposition 3 and the usual subset construction, one immediately gets:

Corollary 1. *Let* $A = \langle Q, \Sigma, \Gamma, H, q_0, \{q_f\}, h\rangle$ *be a constant height* NPDA. *Then there exists an equivalent* DFA *with no more than* $2^{|Q|\cdot|\Gamma^{\leqslant h}|}$ *states*.

We are now going to show that the simulation costs in Proposition 3 and Corollary 1 are *optimal* by exhibiting two witness languages with matching exponential and double exponential gaps.

For a string $x = x_1 \cdots x_n$, let $x^R = x_n \cdots x_1$ denote its reverse. Given an $h > 0$, an alphabet Γ, and two separator symbols $\sharp, \$ \notin \Gamma$, we define the language

$$L_{\Gamma,h} = \{\sharp w_1 \sharp w_2 \sharp \cdots \sharp w_m \$ w : w_1, \ldots, w_m \in \Gamma^*, w \in \Gamma^{\leqslant h}, \text{ and } w \in \textstyle\bigcup_{i=1}^{m} \{w_i^R\}\}.$$

We begin by providing upper bounds on the size of machines accepting $L_{\Gamma,h}$:

Lemma 2. *For each* $h > 0$ *and each alphabet* Γ*:*

(i) *The language* $L_{\Gamma,h}$ *can be accepted by an* NPDA *with* $\mathcal{O}(1)$ *states, pushdown alphabet* Γ, *and constant height* h.

(ii) *The language* $L_{\Gamma,h}$ *can also be accepted by an* NFA (*or* DFA) *with* $\mathcal{O}(|\Gamma^{\leqslant h}|)$ *states* (*or* $2^{\mathcal{O}(|\Gamma^{\leqslant h}|)}$ *states, respectively*).

Proof. We informally describe accepting devices for $L_{\Gamma,h}$:

(i) An NPDA A of constant height h runs as follows: first, it nondeterministically chooses a block w_i and pushes this block onto the pushdown store. If $|w_i| > h$, then A is blocked as soon as it tries to push the $(h+1)$st symbol onto the pushdown. Otherwise, A moves its input head on the first symbol of w, and then checks whether $w = w_i^R$ by comparing the symbols of w with those in the pushdown store. With the pushdown of height h and the alphabet Γ, $\mathcal{O}(1)$ states are sufficient to load w_i and then to match w against w_i^R.

(ii) An NFA N for $L_{\Gamma,h}$ nondeterministically chooses a block w_i and stores it in its finite control, using no more than $\mathcal{O}(|\Gamma^{\leqslant h}|)$ states. If the length of the chosen block exceeds h, then N rejects. Next, N reaches the first symbol of w, and checks, symbol by symbol, whether $w = w_i^R$. It is easy to see that $\mathcal{O}(|\Gamma^{\leqslant h}|)$ states are sufficient to store all possible blocks of length up to h, and to match a stored block against w. From this result and by applying the standard subset construction, one immediately gets a $2^{\mathcal{O}(|\Gamma^{\leqslant h}|)}$-state DFA for $L_{\Gamma,h}$. □

Now we show that the sizes of an NFA or a DFA for $L_{\Gamma,h}$ stated in Lemma 2 (ii) are optimal.

Lemma 3. *For each* $h > 0$ *and each alphabet* Γ, *any* DFA (*or* NFA) *accepting the language* $L_{\Gamma,h}$ *must use at least* $2^{|\Gamma^{\leqslant h}|}$ *states* (*or* $|\Gamma^{\leqslant h}|$ *states, respectively*).

Proof. Let us start with the DFA. Using the lexicographical ordering on the set $\Gamma^{\leqslant h}$, we can associate, with each ordered subset $S = \{v_1, v_2, \ldots, v_k\} \subseteq \Gamma^{\leqslant h}$, the word $v_S = \sharp v_1 \sharp v_2 \sharp \cdots \sharp v_k$. Let L be the set of all such words v_S. Clearly, $|L| = 2^{|\Gamma^{\leqslant h}|}$. Now,

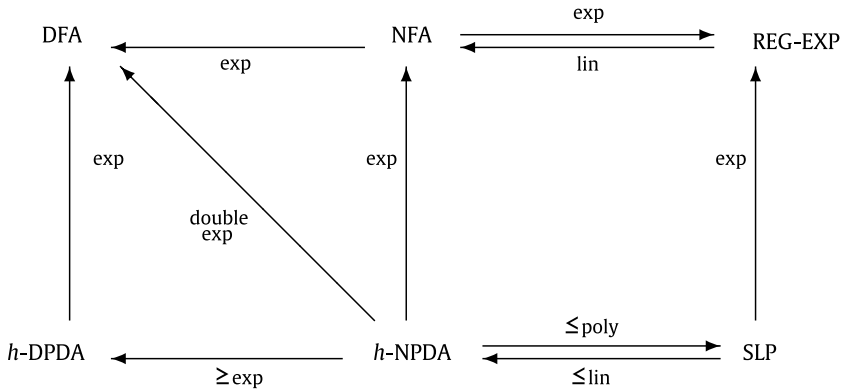

Fig. 2. Costs of simulations among different types of formalisms defining regular languages. An arc labeled by lin (poly, exp, double exp) from a vertex A to a vertex B means that, given a representation of type A, we can construct an equivalent representation of type B, paying by a linear (polynomial, exponential, double exponential, respectively) increase in the size. For clarity, some trivial linear conversions are omitted.

suppose by contradiction that $L_{\Gamma,h}$ is accepted by a DFA B with fewer states than $2^{|\Gamma^{\leqslant h}|}$. By counting arguments, there exist two *different* words $v_{S_1}, v_{S_2} \in L$ on which the computations of B end in the same state. Since $v_{S_1} \neq v_{S_2}$, we have that S_1 differs from S_2 in at least one element, say v. So, without loss of generality, assume that $v \in S_1 \setminus S_2$. This gives two different inputs $x = v_{S_1}\$v^R$ and $y = v_{S_2}\$v^R$ satisfying $x \in L_{\Gamma,h}$ and $y \notin L_{\Gamma,h}$. However, for both these inputs, the DFA B ends its computations in the same state, yielding that $x \in L_{\Gamma,h}$ if and only if $y \in L_{\Gamma,h}$. This gives a contradiction.

From this lower bound, one immediately gets that any NFA accepting $L_{\Gamma,h}$ must use *at least* $|\Gamma^{\leqslant h}|$ states. Otherwise, by the usual subset construction, we could obtain an equivalent DFA with fewer states than $2^{|\Gamma^{\leqslant h}|}$, thus contradicting our lower bound on the size of DFAs. □

Let us now show the optimality of the exponential simulation cost of constant height DPDAs by DFAs presented in Proposition 3. Consider the following witness language: given an $h > 0$, an alphabet Γ, and a separator symbol $\sharp \notin \Gamma$, let

$$D_{\Gamma,h} = \{w\sharp w^R : w \in \Gamma^{\leqslant h}\}.$$

Lemma 4. *For each $h > 0$ and each alphabet Γ:*

(i) *The language $D_{\Gamma,h}$ is accepted by a* DPDA *with $\mathcal{O}(1)$ states, pushdown alphabet Γ and constant height h, and also by a* DFA *with $2 \cdot |\Gamma^{\leqslant h}| + 1$ states.*

(ii) *Any* DFA *accepting the language $D_{\Gamma,h}$ must have at least $|\Gamma^{\leqslant h}|$ states.*

Proof. Point (i) can easily be checked by the reader. To prove the lower bound at point (ii), assume a DFA B for $D_{\Gamma,h}$ with less than $|\Gamma^{\leqslant h}|$ states. Then, by counting arguments, there exist two different words $v, u \in \Gamma^{\leqslant h}$ taking B to the same state. This means that B accepts $v\sharp v^R \in D_{\Gamma,h}$ if and only if it accepts $u\sharp v^R \notin D_{\Gamma,h}$, which is clearly a contradiction. □

6. The final picture

In conclusion, in Fig. 2, we sum up the main relations on the sizes of the different types of formalisms defining regular languages we considered in this paper. By h-DPDA (h-NPDA), we denote the constant height DPDA (NPDA, respectively). Let us briefly discuss the simulation costs displayed in this figure.

The costs of the following simulations are asymptotically optimal:

- h-DPDA → DFA: the exponential cost comes from Proposition 3, while its optimality follows from Lemma 4.
- h-NPDA → NFA: the exponential cost comes from Proposition 3, while its optimality follows from Lemmas 2(i) and 3.
- SLP → REG-EXP: the exponential cost comes from Proposition 2, while its optimality follows, e.g., from Example 3 in Section 2.1.
- h-NPDA → DFA: the double exponential cost is presented by Corollary 1, its optimality follows from Lemmas 2(i) and 3.
- NFA → DFA: the exponential cost is known from [11], its optimality from [9].
- NFA ↔ REG-EXP: the linear cost for the "←" conversion comes directly from the Kleene's Theorem (see also [2,3] for more sophisticated translations), while its optimality follows trivially by considering, e.g., the regular expression a^n, for a fixed $n > 0$. The exponential cost for the converse direction and its optimality is from [5].

- DFA $\to$ NFA, DFA $\to$ h-DPDA, DFA $\to$ h-NPDA, NFA $\to$ h-NPDA, REG-EXP $\to$ SLP, h-DPDA $\to$ h-NPDA: all these costs are trivially linear. Their asymptotic optimality can be obtained, e.g., by considering a single word language $\{a_1 a_2 \ldots a_n\}$, where the symbols $a_1, a_2, \ldots, a_n$ are all distinct.

The costs of the following simulations are not yet known to be optimal:

- h-NPDA $\leftrightarrow$ SLP: Theorems 1 and 2 prove polynomial and linear, respectively, upper bounds.
- h-NPDA $\to$ h-DPDA: the exponential lower bound comes from the following consideration: a sub-exponential cost of h-NPDA $\to$ h-DPDA conversion together with the optimal exponential cost for h-DPDA $\to$ DFA would lead to a sub-double exponential cost of h-NPDA $\to$ DFA, thus contradicting the optimality of the double exponential cost. In general, for h-NPDA $\to$ h-DPDA conversion, we conjecture a double exponential optimal cost.

Acknowledgments

The authors thank the anonymous referee for kind and helpful comments.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974..
[2] A. Brüggemann-Klein, Regular expressions into finite automata, Theor. Comput. Sci. 120 (1993) 197–213.
[3] P. Caron, D. Ziadi, Characterization of Glushkov automata, Theor. Comput. Sci. 233 (2000) 75–90.
[4] A. Chandra, D. Kozen, L. Stockmeyer, Alternation, J. ACM 28 (1981) 114–133.
[5] A. Ehrenfeucht, P. Zieger, Complexity measures for regular expressions, J. Comput. Syst. Sci. 12 (1976) 134–146.
[6] H. Gruber, M. Holzer, Provably shorter regular expressions from deterministic finite automata, in: Proceedings of the Developments in Language Theory, Lecture Notes in Computer Science, vol. 5257, Springer, Berlin, 2008, pp. 383–395.
[7] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 2001.
[8] S. Kleene, Representation of events in nerve nets and finite automata, in: C. Shannon, J. McCarthy (Eds.), Automata Studies, Princeton University Press, Princeton, NJ, 1956, pp. 3–42.
[9] A.R. Meyer, M.J. Fischer, Economy of description by automata, grammars, and formal systems, in: Proceedings of the IEEE 12th Symposium on Switching and Automata Theory, 1971, pp. 188–191.
[10] M. Rabin, Probabilistic automata, Inform. Control 6 (1963) 230–245.
[11] M. Rabin, D. Scott, Finite automata and their decision problems, IBM J. Res. Develop. 3 (1959) 114–125.
[12] J.C. Shepherdson, The reduction of two-way automata to one-way automata, IBM J. Res. Develop. 3 (1959) 198–200.
[13] L.G. Valiant, Regularity and related problems for deterministic pushdown automata, J. ACM 22 (1975) 1–10.