Unary Pushdown Automata and Straight-Line Programs

Dmitry Chistikov and Rupak Majumdar

Max Planck Institute for Software Systems (MPI-SWS)
Kaiserslautern and Saarbrücken, Germany
{dch,rupak}@mpi-sws.org

Abstract. We consider decision problems for deterministic pushdown automata over a unary alphabet (udpda, for short). Udpda are a simple computation model that accept exactly the unary regular languages, but can be exponentially more succinct than finite-state automata. We complete the complexity landscape for udpda by showing that emptiness (and thus universality) is **P**-hard, equivalence and compressed membership problems are **P**-complete, and inclusion is **coNP**-complete. Our upper bounds are based on a *translation theorem* between udpda and straight-line programs over the binary alphabet (SLPs). We show that the characteristic sequence of any udpda can be represented as a pair of SLPs—one for the prefix, one for the lasso—that have size linear in the size of the udpda and can be computed in polynomial time. Hence, decision problems on udpda are reduced to decision problems on SLPs. Conversely, any SLP can be converted in logarithmic space into a udpda, and this forms the basis for our lower bound proofs. We show **coNP**-hardness of the ordered matching problem for SLPs, from which we derive **coNP**-hardness for inclusion. In addition, we complete the complexity landscape for unary nondeterministic pushdown automata by showing that the universality problem is $\Pi_2\mathbf{P}$-hard, using a new class of integer expressions. Our techniques have applications beyond udpda. We show that our results imply $\Pi_2\mathbf{P}$-completeness for a natural fragment of Presburger arithmetic and **coNP** lower bounds for compressed matching problems with one-character wildcards.

1 Introduction

Any model of computation comes with a set of fundamental decision questions: emptiness (does a machine accept some input?), universality (does it accept all inputs?), inclusion (are all inputs accepted by one machine also accepted by another?), and equivalence (do two machines accept exactly the same inputs?). The theoretical computer science community has a fairly good understanding of the precise complexity of these problems for most "classical" models, such as finite and pushdown automata, with only a few prominent open questions (e.g., the precise complexity of equivalence for deterministic pushdown automata).

In this paper, we study a simple class of machines: deterministic pushdown automata working on unary alphabets (unary dpda, or *udpda* for short). A classic theorem of Ginsburg and Rice [7] shows that they accept exactly the unary

regular languages, albeit with potentially exponential succinctness when compared to finite automata. However, the precise complexity of most basic decision problems for udpda has remained open.

Our first and main contribution is that we close the complexity picture for these devices. We show that emptiness is already **P**-hard for udpda (even when the stack is bounded by a linear function of the number of states) and thus **P**-complete. By closure under complementation, it follows that universality is **P**-complete as well. Our main technical construction shows equivalence is in **P** (and so **P**-complete). Somewhat unexpectedly, inclusion is **coNP**-complete. In addition, we study the *compressed membership* problem: given a udpda over the alphabet $\{a\}$ and a number n in binary, is a^n in the language? We show that this problem is **P**-complete too.

A natural attempt at a decision procedure for equivalence or compressed membership would go through translations to finite automata (since udpda only accept regular languages, such a translation is possible). Unfortunately, these automata can be exponentially larger than the udpda and, as we demonstrate, such algorithms are not optimal. Instead, our approach establishes a connection to straight-line programs (*SLPs*) on binary words (see, e. g., Lohrey [20]). An SLP $\mathcal{P}$ is a context-free grammar generating a single word, denoted eval($\mathcal{P}$), over $\{0, 1\}$. Our main construction is a translation theorem: for any udpda, we construct in polynomial time two SLPs $\mathcal{P}'$ and $\mathcal{P}''$ such that the infinite sequence eval($\mathcal{P}'$) $\cdot$ eval($\mathcal{P}''$)$^\omega \in \{0, 1\}^\omega$ is the characteristic sequence of the language of the udpda (for any $i \geq 0$, its ith element is 1 iff a^i is in the language). With this construction, decision problems on udpda reduce to decision problems on compressed words. Conversely, we show that from any pair $(\mathcal{P}', \mathcal{P}'')$ of SLPs, one can compute, in logarithmic space, a udpda accepting the language with characteristic sequence eval($\mathcal{P}'$)$\cdot$eval($\mathcal{P}''$)$^\omega$. Thus, as regards the computational complexity of decision problems, lower bounds for udpda may be obtained from lower bounds for SLPs. Indeed, we show **coNP**-hardness of inclusion via **coNP**-hardness of the *ordered matching* problem for compressed words (i. e., is eval($\mathcal{P}_1$) $\leq$ eval($\mathcal{P}_2$) letter-by-letter, where the alphabet comes with an ordering $\leq$), a problem of independent interest.

As a second contribution, we complete the complexity picture for unary *nondeterministic* pushdown automata (unpda, for short). For unpda, the precise complexity of most decision problems was already known [14]. The remaining open question was the precise complexity of the universality problem, and we show that it is $\mathbf{\Pi_2 P}$-hard (membership in $\mathbf{\Pi_2 P}$ was shown earlier by Huynh [14]). An equivalent question was left open in Kopczyński and To [18] in 2010, but the question was posed as early as in 1976 by Hunt III, Rosenkrantz, and Szymanski [12, open problem 2], where it was asked whether the problem was in **NP** or **PSPACE** or outside both. Huynh's $\mathbf{\Pi_2 P}$-completeness result for equivalence [14] showed, in particular, that universality was in **PSPACE**, and our $\mathbf{\Pi_2 P}$-hardness result reveals that membership in **NP** is unlikely under usual complexity assumptions. As a corollary, we characterize the complexity of the

$\forall_{\text{bounded}} \exists^*$-fragment of Presburger arithmetic, where the universal quantifier ranges over numbers at most exponential in the size of the formula.

To show $\Pi_2\mathbf{P}$-hardness, we show hardness of the universality problem for a class of integer expressions. Several decision problems of this form, with the set of operations $\{+, \cup\}$, were studied in the classic paper of Stockmeyer and Meyer [31], and we show that checking universality of expressions over $\{+, \cup, \times 2, \times \mathbb{N}\}$ is $\Pi_2\mathbf{P}$-complete (the upper bound follows from Huynh [14]).

Related work. Table 1 provides the current complexity picture, including the results in this paper. Results on general alphabets are mostly classical and included for comparison. Note that the complexity landscape for udpda differs from those for unpda, dpda, and finite automata. Upper bounds for emptiness and universality are classical, and the lower bounds for emptiness are originally by Jones and Laaser [17] and Goldschlager [9]. In the nondeterministic unary case, **NP**-completeness of compressed membership is from Huynh [14], rediscovered later by Plandowski and Rytter [25]. The **PSPACE**-completeness of the compressed membership problem for binary pushdown automata (see definition in Section 7) is by Lohrey [22].

The main remaining open question is the precise complexity of the equivalence problem for dpda. It was shown decidable by Sénizergues [29] and primitive recursive by Stirling [30] and Jančar [15], but only **P**-hardness (from emptiness) is currently known. Recently, the equivalence question for dpda when the stack alphabet is unary was shown to be **NL**-complete by Böhm, Göller, and Jančar [4]. From this, it is easy to show that emptiness and universality are also **NL**-complete. Compressed membership, however, remains **PSPACE**-complete (see Caussinus et al. [5] and Lohrey [21]), and inclusion is, of course, already undecidable. When we additionally restrict dpda to both unary input and unary stack alphabet, all five decision problems are **L**-complete.

We discuss corollaries of our results and other related work in Section 7. While udpda are a simple class of machines, our proofs show that reasoning about these machines can be surprisingly subtle.

Acknowledgements. We thank Joshua Dunfield for discussions.

2 Preliminaries

Pushdown automata. A *unary pushdown automaton* (unpda) over the alphabet $\{a\}$ is a finite structure $\mathcal{A} = (Q, \Gamma, \bot, q_0, F, \delta)$, with Q a set of (control) states, Γ a stack alphabet, $\bot \in \Gamma$ a bottom-of-the-stack symbol, $q_0 \in Q$ an initial state, $F \subseteq Q$ a set of final states, and $\delta \subseteq (Q \times (\{a\} \cup \{\varepsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$ a set of transitions with the property that, for every $(q_1, \sigma, \gamma, q_2, s) \in \delta$, either $\gamma \neq \bot$ and $s \in (\Gamma \setminus \{\bot\})^*$, or $\gamma = \bot$ and $s \in \{\varepsilon\} \cup (\Gamma \setminus \{\bot\})^* \bot$. Here and everywhere below ε denotes the empty word.

The semantics of unpda is defined in the following standard way. The set of *configurations* of $\mathcal{A}$ is $Q \times (\Gamma \setminus \{\bot\})^* \bot$. Suppose (q_1, s_1) and (q_2, s_2) are configurations; we write $(q_1, s_1) \vdash_\sigma (q_2, s_2)$ and say that a *move* to (q_2, s_2) is available to $\mathcal{A}$ at (q_1, s_1) iff there exists a transition $(q_1, \sigma, \gamma, q_2, s) \in \delta$ such that,

Table 1. Complexity of decision problems for pushdown automata.

	unary		binary	
	dpda	npda	dpda	npda
Emptiness	$\mathbf{P}^l$	$\mathbf{P}$	$\mathbf{P}$	$\mathbf{P}$
Universality	$\mathbf{P}^l$	$\Pi_\mathbf{2}\mathbf{P}^l$	$\mathbf{P}$	undecidable
Equivalence	$\mathbf{P}^{u,l}$	$\Pi_\mathbf{2}\mathbf{P}$	$\mathbf{P}$.. pr.rec.	undecidable
Inclusion	$\mathbf{coNP}^{u,l}$	$\Pi_\mathbf{2}\mathbf{P}$	undecidable	undecidable
Compressed membership	$\mathbf{P}^{u,l}$	$\mathbf{NP}$	$\mathbf{PSPACE}$	$\mathbf{PSPACE}$

Legend: "dpda" and "npda" stand for deterministic and *possibly* nondeterministic pushdown automata, respectively; "unary" and "binary" refer to their input alphabets. Names of complexity classes stand for completeness with respect to logarithmic-space reductions; abbreviation "pr.rec." stands for "primitive recursive". Superscripts u and l denote new upper and lower bounds shown in this paper.

if $\gamma \neq \bot$ or $s \neq \varepsilon$, then $s_1 = \gamma s'$ and $s_2 = s s'$ for some $s' \in \Gamma^*$, or, if $\gamma = \bot$ and $s = \varepsilon$, then $s_1 = s_2 = \bot$. A unary pushdown automaton is called *deterministic*, shortened to *udpda*, if at every configuration at most one move is available.

A word $w \in \{a\}^*$ is *accepted* by $\mathcal{A}$ if there exists a configuration (q_k, s_k) with $q_k \in F$ and a sequence of moves $(q_i, s_i) \vdash_{\sigma_i} (q_{i+1}, s_{i+1})$, $i = 0, \ldots, k-1$, such that $s_0 = \bot$ and $\sigma_0 \ldots \sigma_{k-1} = w$; that is, the acceptance is by final state. The *language* of $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of all words $w \in \{a\}^*$ accepted by $\mathcal{A}$.

We define the *size* of a unary pushdown automaton $\mathcal{A}$ as $|Q| \cdot |\Gamma|$, provided that for all transitions $(q_1, \sigma, \gamma, q_2, s) \in \delta$ the length of the word s is at most 2 (see also [24]). While this definition is better suited for deterministic rather than nondeterministic automata, it already suffices for the purposes of Section 6, where we handle unpda, because it is always the case that $|\delta| \leq 2 |Q|^2 |\Gamma|^4$.

Decision problems. We consider the following decision problems: emptiness $(L(\mathcal{A}) =^? \emptyset)$, universality $(L(\mathcal{A}) =^? \{a\}^*)$, equivalence $(L(\mathcal{A}_1) =^? L(\mathcal{A}_2))$, and inclusion $(L(\mathcal{A}_1) \subseteq^? L(\mathcal{A}_2))$. The *compressed membership* problem for unary pushdown automata is associated with the question $a^n \in^? L(\mathcal{A})$, with n given in binary as part of the input. In the following, hardness is with respect to logarithmic-space reductions. Our first result is that emptiness is already **P**-hard for udpda.

Proposition 1. UDPDA-Emptiness *and* UDPDA-Universality *are* **P**-*complete.*

Proof. Emptiness is in **P** for non-deterministic pushdown automata on any alphabet, and deterministic automata can be complemented in polynomial time. So, we focus on showing **P**-hardness for emptiness.

We encode the computations of an alternating logspace Turing machine using an udpda. We assume without loss of generality that each configuration c of the machine has exactly two successors, denoted c_l (left successor) and c_r (right

successor), and that each run of the machine terminates. The udpda encodes a successful run over the computation tree of the TM. The states of the udpda are configurations of the Turing machine and an additional *context*, which can be a ("accepting"), r ("rejecting"), or x ("exploring"). The stack alphabet consists of pairs (c, d), where c is a configurations of the machine and the direction $d \in \{l, r\}$, together with an additional end-of-stack symbol. The alphabet has just one symbol 0. The initial state is (c_0, x), where c_0 is the initial configuration of the machine, and the stack has the end-of-stack symbol.

Suppose the current state is (c, x), where c is not an accepting or rejecting configuration. The udpda pushes (c, l) on to the stack and updates its state to (c_l, x), where c_l is the left successor of c. The invariant is that in the exploring phase, the stack maintains the current path in the computation tree, and if the top of the stack is (c, l) (resp. (c, r)) then the current state is the left (resp. right) successor of c.

Suppose now the current state is (c, x) where c is an accepting configuration. The context is set to a, and the udpda backtracks up the computation tree using the stack. If the top of the stack is the end-of-stack symbol, the machine accepts. If the top of the stack (c', d) consists of an existential configuration c', then the new state is (c', a) and recursively the machine moves up the stack. If the top of the stack (c', d) consists of a universal configuration c', and $d = l$, then the new state is (c'_r, x), the right successor of c', and the top of stack is replaced by (c', r). If the top of the stack (c', d) consists of a universal configuration c', and $d = r$, then the new state is (c', a), the stack is popped, and the machine continues to move up the stack.

Suppose now the current state is (c, x) where c is a rejecting configuration. The context is set to r, and the udpda backtracks up the computation tree using the stack. If the top of the stack is the end-of-stack symbol, the machine rejects. If the top of the stack (c', d) consists of an existential configuration c', and $d = l$, then the new state is (c'_r, x) and the top of stack is replaced with (c', r). If the top of the stack (c', d) consists of an existential configuration c', and $d = r$, then the new state is (c', r), the stack is popped, and the machine continues to move up the stack. The top of stack is replaced with (c', r). If the top of the stack (c', d) consists of a universal configuration c', then the new state is (c', r), the stack is popped, and the machine continues to move up the stack.

It is easy to see that the reduction can be performed in logarithmic space. If the TM has an accepting computation tree, the udpda has an accepting run following the computation tree. On the other hand, any accepting computation of the udpda is a depth-first traversal of an accepting computation tree of the TM.

Finally, since udpda can be complemented in logarithmic space, we get the corresponding results for universality. This completes the proof. $\qquad\square$

Straight-line programs. A *straight-line program* [20], or an *SLP*, over an alphabet Σ is a context-free grammar that generates a single word; in other words, it is a tuple $\mathcal{P} = (S, \Sigma, \Delta, \pi)$, where Σ and Δ are disjoint sets of *terminal* and *nonterminal* symbols (*terminals* and *nonterminals*), $S \in \Delta$ is the *axiom*,

and the function $\pi\colon \Delta \to (\Sigma \cup \Delta)^*$ defines a set of *productions* written as "$N \to w$", $w = \pi(N)$, and satisfies the property that the relation $\{(N, D) \mid N \to w \text{ and } D \text{ occurs in } w\}$ is acyclic. An SLP $\mathcal{P}$ is said to *generate* a (unique) word $w \in \Sigma^*$, denoted $\mathrm{eval}(\mathcal{P})$, which is the result of applying substitutions π to S.

An SLP is said to be in *Chomsky normal form* if for all productions $N \to w$ it holds that either $w \in \Sigma$ or $w \in \Delta^2$. The *size* of an SLP is the number of nonterminals in its Chomsky normal form.

3 Indicator pairs and the translation theorem

We say that a pair of SLPs $(\mathcal{P}', \mathcal{P}'')$ over an alphabet Σ *generates* a sequence $c \in \Sigma^\omega$ if $\mathrm{eval}(\mathcal{P}') \cdot (\mathrm{eval}(\mathcal{P}''))^\omega = c$. We call an infinite sequence $c \in \{0, 1\}^\omega$, $c = c_0 c_1 c_2 \ldots$, the *characteristic sequence* of a unary language $L \subseteq \{a\}^*$ if, for all $i \geq 0$, it holds that c_i is 1 if $a^i \in L$ and 0 otherwise. One may note that the characteristic sequence is eventually periodic if and only if L is regular.

Definition 1. *A pair of straight-line programs $(\mathcal{P}', \mathcal{P}'')$ over $\{0, 1\}$ is called an* indicator pair *for a unary language $L \subseteq \{a\}^*$ if it generates the characteristic sequence of L.*

A unary language can have several different indicator pairs. Indicator pairs form a descriptional system for unary languages, with the *size* of $(\mathcal{P}', \mathcal{P}'')$ defined as the sum of sizes of $\mathcal{P}'$ and $\mathcal{P}''$. The following translation theorem shows that udpda and indicator pairs are polynomially-equivalent representations for unary regular languages. We remark that Theorem 1 does not give a normal form for udpda because of the non-uniqueness of indicator pairs.

Theorem 1 (translation theorem). *For a unary language $L \subseteq \{a\}^*$:*
(1) *if there exists a udpda $\mathcal{A}$ of size m with $L(\mathcal{A}) = L$, then there exists an indicator pair for L of size $O(m)$;*
(2) *if there exists an indicator pair for L of size m, then there exists a udpda $\mathcal{A}$ of size $O(m)$ with $L(\mathcal{A}) = L$.*
Both statements are supported by polynomial-time algorithms, the second of which works in logarithmic space.

Proof idea. We only discuss part 1, which presents the main technical challenge. The starting point is the simple observation that a udpda $\mathcal{A}$ has a single infinite computation, provided that the input tape supplies $\mathcal{A}$ with as many input symbols a as it needs to consume. Along this computation, *events* of two types are encountered: $\mathcal{A}$ can consume a symbol from the input and can enter a final state.

The crucial technical task is to construct inductively, using dynamic programming, straight-line programs that record these events along finite computational segments. These segments are of two types: first, between matching push and pop moves ("procedure calls") and, second, from some starting point until a move pops the symbol that has been on top of the stack at that point ("exits

from current context"). Loops are detected, and infinite computations are associated with pairs of SLPs: in such a pair, one SLP records the initial segment, or prefix of the computation, and the other SLP records events within the loop.

After constructing these SLPs, it remains to transform the computational "history", or *transcript*, associated with the initial configuration of $\mathcal{A}$ into the characteristic sequence. This transformation can easily be performed in polynomial time, without expanding SLPs into the words that they generate. The result is the indicator pair for $\mathcal{A}$. □

The full proof of Theorem 1 is given is Section 5. Note that going from indicator pairs to udpda is useful for obtaining lower bounds on the computational complexity of decision problems for udpda (Theorems 2 and 5). For this purpose, it suffices to model just a single SLP, but taking into account the whole pair is interesting from the point of view of descriptional complexity (see also Section 7).

4   Decision problems for udpda

4.1   Compressed membership and equivalence

For an SLP $\mathcal{P}$, by $|\mathcal{P}|$ we denote the length of the word eval($\mathcal{P}$), and by $\mathcal{P}[n]$ the nth symbol of eval($\mathcal{P}$), counting from 0 (that is, $0 \le n \le |\mathcal{P}| - 1$). We write $\mathcal{P}_1 \equiv \mathcal{P}_2$ if and only if eval($\mathcal{P}_1$) = eval($\mathcal{P}_2$).

The following SLP-QUERY problem is known to be **P**-complete (see Lifshits and Lohrey [19]): given an SLP $\mathcal{P}$ over $\{0, 1\}$ and a number n in binary, decide whether $\mathcal{P}[n] = 1$. The problem SLP-EQUIVALENCE is only known to be in **P** (see, e. g., Lohrey [20]): given two SLPs $\mathcal{P}_1$, $\mathcal{P}_2$, decide whether $\mathcal{P}_1 \equiv \mathcal{P}_2$.

Theorem 2. UDPDA-COMPRESSED-MEMBERSHIP *is* **P**-*complete.*

Proof. The upper bound follows from Theorem 1. Indeed, given a udpda $\mathcal{A}$ and a number n, first construct an indicator pair $(\mathcal{P}', \mathcal{P}'')$ for $L(\mathcal{A})$. Now compute $|\mathcal{P}'|$ and $|\mathcal{P}''|$ and then decide if $n \le |\mathcal{P}'| - 1$. If so, the answer is given by $\mathcal{P}'[n]$, otherwise by $\mathcal{P}''[r]$, where $r = (n - |\mathcal{P}'|) \bmod |\mathcal{P}''|$ and in both cases 1 is interpreted as "yes" and 0 as "no".

To prove the lower bound, we reduce from the SLP-QUERY problem. Take an instance with an SLP $\mathcal{P}$ and a number n in binary. By transforming the pair $(\mathcal{P}, \mathcal{P}_0)$, with $\mathcal{P}_0$ any fixed SLP over $\{0, 1\}$, into a udpda $\mathcal{A}$ using part 2 of Theorem 1, this problem is reduced, in logspace, to whether $a^n \in L(\mathcal{A})$. □

Recall that emptiness and universality of udpda are **P**-complete by Proposition 1. Our next theorem extends this result to the general equivalence problem for udpda.

Theorem 3. UDPDA-EQUIVALENCE *is* **P**-*complete.*

Proof. Hardness follows from Proposition 1. We show how Theorem 1 can be used to prove the upper bound: given udpda $\mathcal{A}_1$ and $\mathcal{A}_2$, first construct indicator pairs $(\mathcal{P}'_1, \mathcal{P}''_1)$ and $(\mathcal{P}'_2, \mathcal{P}''_2)$ for $L(\mathcal{A}_1)$ and $L(\mathcal{A}_2)$, respectively. Now reduce the problem of whether $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ to SLP-EQUIVALENCE. The key observation is that an eventually periodic sequence that has periods $|\mathcal{P}''_1|$ and $|\mathcal{P}''_2|$ also has period $t = \gcd(|\mathcal{P}''_1|, |\mathcal{P}''_2|)$. Therefore, it suffices to check that, first, the initial segments of the generated sequences match and, second, that $\mathcal{P}''_1$ and $\mathcal{P}''_2$ generate powers of the same word up to a certain circular shift.

In more detail, let us first introduce some auxiliary operations for SLP. For SLPs $\mathcal{P}_1$ and $\mathcal{P}_2$, by $\mathcal{P}_1 \cdot \mathcal{P}_2$ we denote an SLP that generates $\mathrm{eval}(\mathcal{P}_1) \cdot \mathrm{eval}(\mathcal{P}_2)$, obtained by "concatenating" $\mathcal{P}_1$ and $\mathcal{P}_2$. Now suppose that $\mathcal{P}$ generates $w = w[0] \ldots w[|\mathcal{P}| - 1]$. Then the SLP $\mathcal{P}[a \mathinner{.\,.} b)$ generates the word $w[a \mathinner{.\,.} b) = w[a] \ldots w[b-1]$, of length $b - a$ (as in $\mathcal{P}[n]$, indexing starts from 0). The SLP $\mathcal{P}^{\alpha}$ generates w^{α}, with the meaning clear for $\alpha = 0, 1, 2, \ldots$, also extended to $\alpha \in \mathbb{Q}$ with $\alpha \cdot |\mathcal{P}| \in \mathbb{Z}_{\geq 0}$ by setting $w^{k+n/|w|} = w^k \cdot w[0 \mathinner{.\,.} n)$, $n < |w|$. Finally, $\mathcal{P} \curvearrowright s$ denotes *cyclic shift* and evaluates to $w[s \mathinner{.\,.} |w|) \cdot w[0 \mathinner{.\,.} s)$. One can easily demonstrate that all these operations can be implemented in polynomial time.

So, assume that $|\mathcal{P}'_1| \geq |\mathcal{P}'_2|$. First, one needs to check whether $\mathcal{P}'_1 \equiv \mathcal{P}'_2 \cdot (\mathcal{P}''_2)^{\alpha}$, where $\alpha = (|\mathcal{P}'_1| - |\mathcal{P}'_2|)/|\mathcal{P}''_2|$. Second, note that an eventually periodic sequence that has periods $|\mathcal{P}''_1|$ and $|\mathcal{P}''_2|$ also has period $t = \gcd(|\mathcal{P}''_1|, |\mathcal{P}''_2|)$. Compute t and an auxiliary SLP $\mathcal{P}'' = \mathcal{P}''_1[0 \mathinner{.\,.} t)$, and then check whether $\mathcal{P}''_1 \equiv (\mathcal{P}'')^{|\mathcal{P}''_1|/t}$ and $\mathcal{P}''_2 \curvearrowright s \equiv (\mathcal{P}'')^{|\mathcal{P}''_2|/t}$ with $s = (|\mathcal{P}'_1| - |\mathcal{P}'_2|) \bmod |\mathcal{P}''_2|$. It is an easy exercise to show that $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ iff all the checks are successful. This completes the proof. □

4.2 Inclusion

A natural idea for handling the inclusion problem for udpda would be to extend the result of Theorem 3, that is, to tackle inclusion similarly to equivalence. This raises the problem of comparing the words generated by two SLPs in the componentwise sense with respect to the order $0 \leq 1$. To the best of our knowledge, this problem has not been studied previously, so we deal with it separately. As it turns out, here one cannot hope for an efficient algorithm unless $\mathbf{P} = \mathbf{NP}$.

Let us define the following family of problems, parameterized by partial order R on the alphabet of size at least 2, and denoted SLP-COMPONENTWISE-R. The input is a pair of SLPs $\mathcal{P}_1, \mathcal{P}_2$ over an alphabet partially ordered by R, generating words of equal length. The output is "yes" iff for all i, $0 \leq i < |\mathcal{P}_1|$, the relation $R(\mathcal{P}_1[i], \mathcal{P}_2[i])$ holds. By SLP-COMPONENTWISE-$(0 \leq 1)$ we mean a special case of this problem where R is the partial order on $\{0, 1\}$ given by $0 \leq 0$, $0 \leq 1$, $1 \leq 1$.

Theorem 4. SLP-COMPONENTWISE-R *is* **coNP***-complete if R is not the equality relation (that is, if $R(a, b)$ holds for some $a \neq b$), and in* **P** *otherwise.*

Proof. We first prove that SLP-COMPONENTWISE-$(0 \leq 1)$ is **coNP**-hard. We show a reduction from the complement of SUBSET-SUM: suppose we start with

an instance of Subset-Sum containing a vector of naturals $w = (w_1, \ldots, w_n)$ and a natural t, and the question is whether there exists a vector $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ such that $x \cdot w = t$, where $x \cdot w$ is defined as the inner product $\sum_{i=1}^n x_i w_i$. Let $s = (1, \ldots, 1) \cdot w$ be the sum of all components of w.

We use the construction of so-called Lohrey words. Lohrey shows [22, Theorem 5.2] that given such an instance, it is possible to construct in logarithmic space two SLPs that generate words $W_1 = \prod_{x \in \{0,1\}^n} a^{x \cdot w} b a^{s - x \cdot w}$ and $W_2 = (a^t b a^{s-t})^{2^n}$, where the product in W_1 enumerates the $x$es in the lexicographic order. Now W_1 and W_2 share a symbol b in some position iff the original instance of Subset-Sum is a yes-instance. Substitute 0 for a and 1 for b in the first SLP, and 0 for b and 1 for a in the second SLP. The new SLPs $\mathcal{P}_1$ and $\mathcal{P}_2$ obtained in this way form a no-instance of SLP-Componentwise-$(0 \leq 1)$ iff the original instance of Subset-Sum is a yes-instance, because now the "distinguished" pair of symbols consists of a 1 in $\mathcal{P}_1$ and 0 in $\mathcal{P}_2$. Therefore, SLP-Componentwise-$(0 \leq 1)$ is **coNP**-hard.

Now observe that, for any R, membership of SLP-Componentwise-R in **coNP** is obvious, and the hardness is by a simple reduction from SLP-Componentwise-$(0 \leq 1)$: just substitute a for 0 and b for 1 (recall that by the definition of partial order, $R(b, a)$ would entail $a = b$, which is false). In the last special case in the statement, R is just the equality relation, so deciding SLP-Componentwise-R is the same as deciding SLP-Equivalence, which is in **P** (see Section 4). This concludes the proof. □

A corollary of Theorem 4 on a problem of matching for compressed partial words is demonstrated in Section 7.

Remark. An alternative reduction showing hardness of SLP-Componentwise-$(0 \leq 1)$, this time from Circuit-SAT, but also making use of Subset-Sum and Lohrey words, can be derived from Bertoni, Choffrut, and Radicioni [3, Lemma 3]. They show that for any Boolean circuit with NAND-gates there exists a pair of straight-line programs $\mathcal{P}_1, \mathcal{P}_2$ generating words over $\{0, 1\}$ of the same length with the following property: the function computed by the circuit takes on the value 1 on at least one input combination iff both words share a 1 at some position. Moreover, these two SLPs can be constructed in polynomial time. As a result, after flipping all terminal symbols in the second of these SLPs, the resulting pair is a no-instance of SLP-Componentwise-$(0 \leq 1)$ iff the original circuit is satisfiable.

Theorem 5. UDPDA-Inclusion *is* **coNP**-*complete.*

Proof. First combine Theorem 4 with part 2 of Theorem 1 to prove hardness. Indeed, Theorem 4 shows that SLP-Componentwise-$(0 \leq 1)$ is **coNP**-hard. Take an instance with two SLPs $\mathcal{P}_1, \mathcal{P}_2$ and transform indicator pairs $(\mathcal{P}_1, \mathcal{P}_0)$ and $(\mathcal{P}_2, \mathcal{P}_0)$, with $\mathcal{P}_0$ any fixed SLP over $\{0, 1\}$, into udpda $\mathcal{A}_1, \mathcal{A}_2$ with the help of part 2 of Theorem 1. Now the characteristic sequence of $L(\mathcal{A}_i)$, $i = 1, 2$, is equal to $\mathrm{eval}(\mathcal{P}_i) \cdot (\mathrm{eval}(\mathcal{P}_0))^\omega$. As a result, it holds that $\mathrm{eval}(\mathcal{P}_1) \leq \mathrm{eval}(\mathcal{P}_2)$

in the componentwise sense if and only if $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$. This concludes the hardness proof.

It remains to show that UDPDA-INCLUSION is in **coNP**. First note that for any udpda $\mathcal{A}$ there exists a deterministic pushdown automaton (DFA) that accepts $L(\mathcal{A})$ and has size at most $2^{O(m)}$, where m is the size of $\mathcal{A}$ (see discussion in Section 7 or Pighizzini [24, Theorem 8]). Therefore, if $L(\mathcal{A}_1) \not\subseteq L(\mathcal{A}_2)$, then there exists a witness $a^n \in L(\mathcal{A}_2) \setminus L(\mathcal{A}_1)$ with n at most exponential in the size of $\mathcal{A}_1$ and $\mathcal{A}_2$. By Theorem 2, compressed membership is in **P**, so this completes the proof. □

5   Proof of Theorem 1

Let us first recall some standard definitions and fix notation. In a udpda $\mathcal{A}$, if $(q_1, s_1) \vdash_\sigma (q_2, s_2)$ for some σ, we also write $(q_1, s_1) \vdash (q_2, s_2)$. A *computation* of a udpda $\mathcal{A}$ starting at a configuration (q, s) is defined as a (finite or infinite) sequence of configurations (q_i, s_i) with $(q_1, s_1) = (q, s)$ and, for all i, $(q_i, s_i) \vdash_{\sigma_i} (q_{i+1}, s_{i+1})$ for some σ_i. If the sequence is finite and ends with (q_k, s_k), we also write $(q_1, s_1) \vdash_w^* (q_k, s_k)$, where $w = \sigma_1 \ldots \sigma_{k-1} \in \{a\}^*$. We can also omit the word w when it is not important and say that (q_k, s_k) is *reachable* from (q_1, s_1); in other words, the *reachability* relation $\vdash^*$ is the reflexive and transitive closure of the move relation $\vdash$.

5.1   From indicator pairs to udpda

Going from indicator pairs to udpda is the easier direction in Theorem 1. We start with an auxiliary lemma that enables one to model a single SLP with a udpda. This lemma on its own is already sufficient for lower bounds of Theorem 2 and Theorem 5 in Section 4.

Lemma 1. *There exists an algorithm that works in logarithmic space and transforms an arbitrary SLP $\mathcal{P}$ of size m over $\{0, 1\}$ into a udpda $\mathcal{A}$ of size $O(m)$ over $\{a\}$ such that the characteristic sequence of $L(\mathcal{A})$ is $0 \cdot \mathrm{eval}(\mathcal{P}) \cdot 0^\omega$. In $\mathcal{A}$, it holds that $(q_0, \bot) \vdash_w^* (\bar{q}_0, \bot)$ for $w = a^{|\mathrm{eval}(\mathcal{P})|}$, q_0 the initial state, and $\bar{q}_0$ a non-final state without outgoing transitions.*

Proof. The main part of the algorithm works as follows. Assume that $\mathcal{P}$ is given in Chomsky normal form. With each nonterminal N we associate a gadget in the udpda $\mathcal{A}$, whose *interface* is by definition the entry state q_N and the exit state $\bar{q}_N$, which will only have outgoing pop transitions. With a production of the form $N \to \sigma$, $\sigma \in \{0, 1\}$, we associate a single internal transition from q_N to $\bar{q}_N$ reading an a from the input tape. The state q_N is always non-final, and the state $\bar{q}_N$ is final if and only if $\sigma = 1$. With a production of the form $N \to AB$ we associate two stack symbols γ_N^1, γ_N^2 and the following gadget. At a state q_N, the udpda pushes a symbol γ_N^1 onto the stack and goes to the state q_A. We add a pop transition from $\bar{q}_A$ that reads γ_N^1 from the stack and leads to an auxiliary state q_N'. The only transition from this state pushes γ_N^2 and leads to q_B, and

another transition from $\bar{q}_B$ pops γ_N^2 and goes to $\bar{q}_N$. Here all three states q_N, q_N', and $\bar{q}_N$ are non-final, and the four introduced incident transitions do not read from the input. Finally, if a nonterminal N is the axiom of $\mathcal{P}$, make the state q_N initial and non-final and make $\bar{q}_N$ a non-accepting sink that reads a from the input and pops $\bot$. The reader can easily check that the characteristic sequence of the udpda $\mathcal{A}$ constructed in this way is indeed $0 \cdot \mathrm{eval}(\mathcal{P}) \cdot 0^\omega$, and the construction can be performed in logarithmic space.

Now note that while the udpda $\mathcal{A}$ satisfies $|Q| = O(m)$, we may have also introduced up to 2 stack symbols per nonterminal. Therefore, the size of $\mathcal{A}$ can be as large as $\Omega(m^2)$. However, we can use a standard trick from circuit complexity to avoid this blowup and make this size linear in m. Indeed, first observe that the number of stack symbols, not counting $\bot$, in the construction above can be reduced to k, the maximum, over all nonterminals N, of the number of occurrences of N in the right-hand sides of productions of $\mathcal{P}$. Second, recall that a straight-line program naturally defines a circuit where productions of the form $N \to AB$ correspond to gates performing concatenation. The value of k is the maximum fan-out of gates in this circuit, and it is well-known how to reduce it to $O(1)$ with just a constant-factor increase in the number of gates (see, e. g., Savage [26, Theorem 9.2.1]). The construction can be easily performed in logarithmic space, and the only building block is the identity gate, which in our case translates to a production of the form $N \to A$. Although such productions are not allowed in Chomsky normal form, the construction above can be adjusted accordingly, in a straightforward fashion. This completes the proof.   □

Now, to model an entire indicator pair, we apply Lemma 1 twice and combine the results.

Lemma 2. *There exists an algorithm that works in logarithmic space and, given an indicator pair $(\mathcal{P}', \mathcal{P}'')$ of size m for some unary language $L \subseteq \{a\}^*$, outputs a udpda $\mathcal{A}$ of size $O(m)$ such that $L(\mathcal{A}) = L$.*

Proof. We shall use the same notation as in Subsection 4.1 of Section 4. First compute the bit $b = \mathcal{P}'[0]$ and construct an SLP $\mathcal{P}_1'$ of size $O(m)$ such that $\mathrm{eval}(\mathcal{P}') = b \cdot \mathrm{eval}(\mathcal{P}_1')$. Note that this can be done in logarithmic space, even though the general SLP-QUERY problem is **P**-complete. Now construct, according to Lemma 1, two udpda $\mathcal{A}'$ and $\mathcal{A}''$ for $\mathcal{P}_1'$ and $\mathcal{P}''$, respectively. Assume that their sets of control states are disjoint and connect them in the following way. Add internal ε-transitions from the "last" states of both to the initial state of $\mathcal{A}''$. Now make the initial state of $\mathcal{A}'$ the initial state of $\mathcal{A}$; make it also a final state if $b = 1$. It is easily checked that the language of the udpda $\mathcal{A}$ constructed in this way has characteristic sequence $\mathrm{eval}(\mathcal{P}') \cdot (\mathrm{eval}(\mathcal{P}''))^\omega$ and, hence, is equal to L.   □

Lemma 2 proves part 2 in Theorem 1.

5.2   From udpda to indicator pairs

Going from udpda to indicator pairs is the main part of Theorem 1, and in this subsection we describe our construction in detail. The proof of the key technical lemma is deferred until the following Subsection 5.3.

Assumptions and notation. We assume without loss of generality that the given udpda $\mathcal{A}$ satisfies the following conditions. First, its set of control states, Q, is partitioned into three subsets according to the type of available moves. More precisely, we assume[1] $Q = Q_0 \sqcup Q_{+1} \sqcup Q_{-1}$ with the property that all transitions $(q, \sigma, \gamma, q', s)$ with states q from Q_d, $d \in \{0, -1, +1\}$, have $|s| = 1 + d$; moreover, we assume that $s = \gamma$ whenever $d = 0$, and $s = \gamma'\gamma$ for some $\gamma' \in \Gamma$ whenever $d = 1$.

Second, for convenience of notation we assume that there exists a subset $R \subseteq Q$ such that all transitions departing from states from R read a symbol from the input tape, and transitions departing from $Q \setminus R$ do not.

Third, we assume that δ is specified by means of total functions $\delta_0 \colon Q_0 \to Q$, $\delta_{+1} \colon Q_{+1} \to Q \times \Gamma$, and $\delta_{-1} \colon Q_{-1} \times \Gamma \to Q$. We write $\delta_0(q) = q'$, $\delta_{+1}(q) = (q', \gamma)$, and $\delta_{-1}(q, \gamma) = q'$ accordingly; associated transitions and states are called *internal*, *push*, and *pop* transitions and states, respectively. Note that this assumption implies that only pop transitions can "look" at the top of the stack.

Claim 1. An arbitrary udpda $\mathcal{A}' = (Q', \Gamma, \bot, q_0', F', \delta')$ of size m can be transformed into a udpda $\mathcal{A} = (Q, \Gamma, \bot, q_0, F, \delta)$ that accepts $L(\mathcal{A}')$, satisfies the assumptions of this subsubsection, and has $|Q| = O(m)$ control states.

The proof is easy and left to the reader.

Note that since $\mathcal{A}$ is deterministic, it holds that for any configuration (q, s) of $\mathcal{A}$ there exists a unique infinite computation $(q_i, s_i)_{i=0}^{\infty}$ starting at (q, s), referred to as *the* computation in the text below. This computation can be thought of as a run of $\mathcal{A}$ on an input tape with an infinite sequence a^ω. *The* computation of $\mathcal{A}$ is, naturally, the computation starting from $(q_0, \bot)$. Note that it is due to the fact that $\mathcal{A}$ is unary that we are able to feed it a single infinite word instead of countably many finite words.

In the text below we shall use the following notation and conventions. To refer to an SLP (S, Σ, Δ, π), we sometimes just use its axiom, S. The generated word, w, is denoted by $\mathrm{eval}(S)$ as usual. Note that the set of terminals is often understood from the context and the set of nonterminals is always the set of left-hand sides of productions. This enables us to use the notation $\mathrm{eval}(S)$ to refer to the word generated by the implicitly defined SLP, whenever the set of productions is clear from the context.

Transcripts of computations and overview of the algorithm. Recall that our goal is to describe an algorithm that, given a udpda $\mathcal{A}$, produces an indicator pair for $L(\mathcal{A})$. We first assemble some tools that will allow us to handle the

---

[1] Here and further in the text we use the symbol $\sqcup$ to denote the union of disjoint sets.

computation of $\mathcal{A}$ per se. To this end, we introduce transcripts of computations, which record "events" that determine whether certain input words are accepted or rejected.

Consider a (finite or infinite) computation that consists of moves $(q_i, s_i) \vdash_{\sigma_i} (q_{i+1}, s_{i+1})$, for $1 \le i \le k$ or for $i \ge 1$, respectively. We define the *transcript* of such a computation as a (finite or infinite) sequence

$$\mu(q_1)\,\sigma_1\,\mu(q_2)\,\sigma_2\,\ldots\,\mu(q_k)\,\sigma_k \quad \text{or} \quad \mu(q_1)\,\sigma_1\,\mu(q_2)\,\sigma_2\,\ldots, \quad \text{respectively,}$$

where, for any q_i, $\mu(q_i) = f$ if $q_i \in F$ and $\mu(q_i) = \varepsilon$ if $q_i \in Q \setminus F$. Note that in the finite case the transcript does *not* include $\mu(q_{k+1})$ where q_{k+1} is the control state in the last configuration. In particular, if a computation consists of a single configuration, then its transcript is ε. In general, transcripts are finite words and infinite sequences over the auxiliary alphabet $\{a, f\}$.

The reader may notice that our definition for the finite case basically treats finite computations as left-closed, right-open intervals and lets us perform their concatenation in a natural way. We note, however, that from a technical point of view, a definition treating them as closed intervals would actually do just as well.

Note that any sequence $s \in \{a, f\}^\omega$ containing infinitely many occurrences of a naturally *defines* a unique characteristic sequence $c \in \{0, 1\}^\omega$ such that if s is the transcript of a udpda computation, then c is the characteristic sequence of this udpda's language. The following lemma shows that this correspondence is efficient if the sequences are represented by pairs of SLPs.

Lemma 3. *There exists a polynomial-time algorithm that, given a pair of straight-line programs $(\mathcal{T}', \mathcal{T}'')$ of size m that generates a sequence $s \in \{a, f\}^\omega$ and such that the symbol a occurs in $\mathrm{eval}(\mathcal{T}'')$, produces a pair of straight-line programs $(\mathcal{P}', \mathcal{P}'')$ of size $O(m)$ that generates the characteristic sequence defined by s.*

Proof. Observe that it suffices to apply to the sequence generated by $(\mathcal{T}', \mathcal{T}'')$ the composition of the following substitutions: $h_1 \colon af \mapsto 1$, $h_2 \colon a \mapsto 0$, and $h_3 \colon f \mapsto \varepsilon$. One can easily see that applying h_2 and h_3 reduces to applying them to terminal symbols in SLPs, so it suffices to show that the application of h_1 can also be done in polynomial time and increases the number of productions in Chomsky normal form by at most a constant factor.

We first show how to apply h_1 to a single SLP. Assume Chomsky normal form and process the productions of the SLP inductively in the bottom-up direction. Productions with terminal symbols remain unchanged, and productions of the form $N \to AB$ are handled as follows: if $\mathrm{eval}(A)$ ends with an a and $\mathrm{eval}(B)$ begins with an f, then replace the production with $N \to (Aa^{-1}) \cdot 1 \cdot (f^{-1}B)$, otherwise leave it unchanged as well. Here we use auxiliary nonterminals of the form Na^{-1} and $f^{-1}N$ with the property that $\mathrm{eval}(Na^{-1}) \cdot a = \mathrm{eval}(N)$ and $f \cdot \mathrm{eval}(f^{-1}N) = \mathrm{eval}(N)$. These nonterminals are easily defined inductively in a straightforward manner, just after processing N. At the end of this process one obtains an SLP that generates the result of applying h_1 to the word generated by the original SLP.

We now show how to handle the fact that we need to apply h_1 to the entire sequence $\text{eval}(\mathcal{T}') \cdot (\text{eval}(\mathcal{T}''))^\omega$. Process the SLPs $\mathcal{T}'$ and $\mathcal{T}''$ as described above; for convenience, we shall use the same two names for the obtained programs. Then deal with the junction points in the sequence $\text{eval}(\mathcal{T}') \cdot (\text{eval}(\mathcal{T}''))^\omega$ as follows. If $\text{eval}(\mathcal{T}'')$ does not start with an f, then there is nothing to do. Now suppose it does; then there are two options. The first option is that $\text{eval}(\mathcal{T}'')$ ends with an a. In this case replace $\mathcal{T}''$ with $(f^{-1}\mathcal{T}'') \cdot 1$ and $\mathcal{T}'$ with $(\mathcal{T}'a^{-1}) \cdot 1$ or with $(\mathcal{T}'f)$ according to whether it ends with an a or not. The second option is that $\mathcal{T}''$ does not end with an a. In this case, if $\mathcal{T}'$ ends with an a, replace it with $(\mathcal{T}'a^{-1}) \cdot 1 \cdot (f^{-1}\mathcal{T}'')$, otherwise do nothing. One can easily see that the pair of SLPs obtained on this step will generate the image of the original sequence $\text{eval}(\mathcal{T}') \cdot (\text{eval}(\mathcal{T}''))^\omega$ under h_1. This completes the proof. $\qquad\square$

Note that we could use a result by Bertoni, Choffrut, and Radicioni [3] and apply a four-state transducer (however, the underlying automaton needs to be ε-free, which would make us figure out the last position "manually").

Now it remains to show how to efficiently produce, given a udpda $\mathcal{A}$, a pair of SLPs $(\mathcal{T}', \mathcal{T}'')$ generating the transcript of the computation of $\mathcal{A}$. This is the key part of the entire algorithm, captured by the following lemma.

Lemma 4. *There exists a polynomial-time algorithm that, given a udpda $\mathcal{A}$ of size m, produces a pair of straight-line programs $(\mathcal{T}', \mathcal{T}'')$ of size $O(m)$ that generates the transcript of the computation of $\mathcal{A}$.*

The proof of Lemma 4 is given in the next subsection. Put together, Lemmas 3 and 4 prove the harder direction (that is, part 1) of Theorem 1. The only caveat is that if $\text{eval}(\mathcal{T}'') \in \{f\}^*$, then one needs to replace $\mathcal{T}''$ with a simple SLP that generates a and possibly adjust $\mathcal{T}'$ so that f be appended to the generated word. This corresponds to the case where $\mathcal{A}$ does not read the entire input and enters an infinite loop of ε-moves (that is, moves that do not consume a from the input).

5.3 Details: proof of Lemma 4

Returning and non-returning states. The main difficulty in proving Lemma 4 lies in capturing the structure of a unary deterministic computation. To reason about such computations in a convenient manner, we introduce the following definitions.

We say that a state q is *returning* if it holds that $(q, \bot) \vdash^* (q', \bot)$ for some state $q' \in Q_{-1}$ (recall that states from Q_{-1} are pop states). In such a case the control state q' of the first configuration of the form $(q', \bot)$, $q' \in Q_{-1}$, occurring in the infinite computation starting from $(q, \bot)$ is called the *exit point* of q, and the computation between $(q, \bot)$ and this $(q', \bot)$ the *return segment* from q. For example, if $q \in Q_{-1}$, then q is its own exit point, and the return segment from q contains no moves.

Intuitively, the exit point is the first control state in the computation where the bottom-of-the-stack symbol in the configuration $(q, \bot)$ may matter. One can

formally show that if q' is the exit point of q, then for any configuration (q, s) it holds that $(q, s) \vdash^* (q', s)$ and, moreover, the transcript of the return segment from q is equal, for any s, to the transcript of the shortest computation from (q, s) to (q', s).

If a control state is not returning, it is called *non-returning*. For such a state q, it holds that for every configuration (q', s') reachable from $(q, \perp)$ either $s' \neq \perp$ or $q' \notin Q_{-1}$. One can show formally that infinite computations starting from configurations (q, s) with a fixed non-returning state q and arbitrary s have identical transcripts and, therefore, identical characteristic sequences associated with them. As a result, we can talk about infinite computations starting at a non-returning control state q, rather than in a specific configuration (q, s).

Now consider a state $q \notin Q_{-1}$, an arbitrary configuration (q, s) and the infinite computation starting from (q, s). Suppose that this computation enters a configuration of the form $(\bar{q}, s)$ after at least one move. Then the *horizontal successor* of q is defined as the control state $\bar{q}$ of the first such configuration, and the computation between these configurations is called the *horizontal segment* from q. In other cases, horizontal successor and horizontal segment are undefined. It is easily seen that the horizontal successor, whenever it exists, is well-defined in the sense that it does not depend upon the choice of $s \in (\Gamma \setminus \{\perp\})^* \perp$. Similarly, the choice of s only determines the "lower" part of the stack in the configurations of the horizontal segment; since we shall only be interested in the transcripts, this abuse of terminology is harmless.

Equivalently, suppose that $q \notin Q_{-1}$ and $(q, s) \vdash (q', s')$. If $s' = s$ then the horizontal successor of q is q'. Otherwise it holds that $\delta_{+1}(q) = (q', \gamma)$ for some $\gamma \in \Gamma$, so that $s' = \gamma s$. Now if q' is returning, q'' is the exit point of q', and $\delta_{-1}(q'', \gamma) = \bar{q}$ for the same γ, then $\bar{q}$ is the horizontal successor of q. The horizontal segment is in both cases defined as the shortest non-empty computation of the form $(q, s) \vdash^* (\bar{q}, s)$.

General approach and data structures. Recall that our goal in this subsection is to define an algorithm that constructs a pair of straight-line programs $(\mathcal{T}', \mathcal{T}'')$ generating the transcript of the infinite computation of $\mathcal{A}$. The approach that we take is dynamic programming. We separate out intermediate goals of several kinds and construct, for an arbitrary control state $q \in Q$, SLPs and pairs of SLPs that generate transcripts of the infinite computation starting at q (if q is non-returning), of the return segment from q (if q is returning), and of the horizontal segment from q (whenever it is defined).

Our algorithm will *write* productions as it runs, always using, on their right-hand side, only terminal symbols from $\{a, f\}$ and nonterminals defined by productions written earlier. This enables us to use the notation eval(A) for nonterminals A without referring to a specific SLP. Once written, a production is never modified.

The main data structures of the algorithm, apart from the productions it writes, are as follows: three partial functions $\mathcal{E}, \mathcal{H}, \mathcal{W} : Q \to Q$ and a subset NonRet $\subseteq Q$. Associated with $\mathcal{E}$ and $\mathcal{H}$ are nonterminals E_q and H_q, and with NonRet nonterminals N_q' and N_q''.

Note that the partial functions from Q to Q can be thought of as digraphs on the set of vertices Q. In such digraphs the outdegree of every vertex is at most 1. The algorithm will subsequently modify these partial functions, that is, add new edges and/or remove existing ones (however, the outdegree of no vertex will ever be increased to above 1). We can also promise that $\mathcal{E}$ will only increase, i.e., its graph will only get new edges, $\mathcal{W}$ will only decrease, and $\mathcal{H}$ can go both ways.

During its run the algorithm will maintain the following invariants:

(I1) $Q = \operatorname{dom}\mathcal{E} \sqcup \operatorname{dom}\mathcal{H} \sqcup \operatorname{dom}\mathcal{W} \sqcup \mathrm{NonRet}$, where $\sqcup$ denotes union of disjoint sets.

(I2) Whenever $\mathcal{E}(q) = q'$, it holds that q is returning, q' is the exit point of q, and $\operatorname{eval}(E_q)$ is the transcript of the return segment from q.

(I3) Whenever $\mathcal{H}(q) = q'$, it holds that q' is the horizontal successor of q and $\operatorname{eval}(H_q)$ is the transcript of the horizontal segment from q.

(I4) Whenever $\mathcal{W}(q) = q'$, it holds that $\delta_{+1}(q) = (q', \gamma)$ for some $\gamma \in \Gamma$.

(I5) Whenever $q \in \mathrm{NonRet}$, it holds that q is non-returning and the sequence $\operatorname{eval}(N'_q) \cdot (\operatorname{eval}(N''_q))^\omega$ is the transcript of the infinite computation starting at q.

Description of the algorithm: computing transcripts. Our algorithm has three stages: the initialization stage, the main stage, and the $\perp$-handling stage. The initialization stage of the algorithm works as follows:

— for each $q \in Q$, write $V_q \to \mu(q)\sigma(q)$, where $\mu(q)$ is f if $q \in F$ and ε otherwise, and $\sigma(q)$ is a if $q \in R$ (that is, if transitions departing from q read a symbol from the input) and ε otherwise;

— for all $q \in Q_{-1}$, set $\mathcal{E}(q) = q$ and write $E_q \to \varepsilon$;

— for all $q \in Q_0$, set $\mathcal{H}(q) = q'$ where $\delta_0(q) = q'$ and write $H_q \to V_q$;

— for all $q \in Q_{+1}$, set $\mathcal{W}(q) = q'$ where $\delta_{+1}(q) = (q', \gamma)$ for some $\gamma \in \Gamma$;

— set $\mathrm{NonRet} = \emptyset$.

It is easy to see that in this way all invariants (I1)–(I5) are initially satisfied (recall that the transcript of an empty computation is ε).

For convenience, we also introduce two auxiliary objects: a partial function $\mathcal{G} \colon Q \to Q$ and nonterminals G_q, defined as follows. The domain of $\mathcal{G}$ is $\operatorname{dom}\mathcal{G} = \operatorname{dom}\mathcal{H} \sqcup \operatorname{dom}\mathcal{W}$; note that, according to invariant (I1), this union is disjoint. We assign $\mathcal{G}(q) = q'$ iff $\mathcal{H}(q) = q'$ or $\mathcal{W}(q) = q'$. We shall assume that $\mathcal{G}$ is recomputed as $\mathcal{H}$ and $\mathcal{W}$ change. Now for every $q \in \operatorname{dom}\mathcal{G}$, we let G_q stand for H_q if $q \in \operatorname{dom}\mathcal{H}$ and for V_q if $q \in \operatorname{dom}\mathcal{W}$.

At this point we are ready to describe the main stage of the algorithm. During this stage, the algorithm applies the following rules until none of them is applicable (if at some point several rules can be applied, the choice is made arbitrarily; the rules are well-defined whenever invariants (I1)–(I5) hold):

(R1) If $\mathcal{G}(q) = q'$ where $q' \in \mathrm{NonRet}$ and $q \in Q$: remove q from either $\operatorname{dom}\mathcal{H}$ or $\operatorname{dom}\mathcal{W}$, add q to NonRet, write $N'_q \to G_q N'_{q'}$ and $N''_q \to N''_{q'}$.

(R2) If $\mathcal{H}(q) = q'$ where $q' \in \operatorname{dom} \mathcal{E}$ and $q \in Q$: remove q from $\operatorname{dom} \mathcal{H}$, define $\mathcal{E}(q) = \mathcal{E}(q')$, write $E_q \to H_q E_{q'}$.

(R3) If $\mathcal{W}(q) = q'$ where $q' \in \operatorname{dom} \mathcal{E}$ and $q \in Q$: remove q from $\operatorname{dom} \mathcal{W}$, define $\mathcal{H}(q) = \bar{q}$ where $\mathcal{E}(q') = q''$, $\delta_{-1}(q'', \gamma) = \bar{q}$, and $\delta_{+1}(q) = (q', \gamma)$ (that is, γ is the symbol pushed by the transition leaving q, and $\bar{q}$ is the state reached by popping γ at q'', the exit point of q'). Finally, write $H_q \to V_q E_{q'} V_{q''}$.

(R4) If $\mathcal{G}$ contains a simple cycle, that is, if $\mathcal{G}(q_i) = q_{i+1}$ for $i = 1, \ldots, k-1$ and $\mathcal{G}(q_k) = q_1$, where $q_i \neq q_j$ for $i \neq j$, then for each vertex q_i of the cycle remove it from either $\operatorname{dom} \mathcal{H}$ or $\operatorname{dom} \mathcal{W}$ and add it to NonRet; in addition, write $N'_{q_k} \to G_{q_k}$, $N''_{q_k} \to G_{q_1} \ldots G_{q_k}$, and, for each $i = 1, \ldots, k-1$, $N'_{q_i} \to G_{q_i} N'_{q_{i+1}}$ and $N''_{q_i} \to N''_{q_{i+1}}$.

We shall need two basic facts about this stage of the algorithm.

Claim 2. Application of rules (R1)–(R4) does not violate invariants (I1)–(I5).

The proof of Claim 2 is easy and left to the reader.

Claim 3. If no rule is applicable, then $\operatorname{dom} \mathcal{G} = \emptyset$.

Proof. Suppose $\operatorname{dom} \mathcal{G} \neq \emptyset$. Consider the graph associated with $\mathcal{G}$ and observe that all vertices in $\operatorname{dom} \mathcal{G}$ have outdegree 1. This implies that $\mathcal{G}$ has either a cycle within $\operatorname{dom} \mathcal{G}$ or an edge from $\operatorname{dom} \mathcal{G}$ to $Q \setminus \operatorname{dom} \mathcal{G}$. In the first case, rule (R4) is applicable. In the second case, we conclude with the help of the invariant (I1) that the edge leads from a vertex in $\operatorname{dom} \mathcal{H} \sqcup \operatorname{dom} \mathcal{W}$ to a vertex in NonRet $\sqcup$ $\operatorname{dom} \mathcal{E}$. If the destination is in NonRet, then rule (R1) is applicable; otherwise the destination is in $\operatorname{dom} \mathcal{E}$ and one can apply rule (R2) or rule (R3) according to whether the source is in $\operatorname{dom} \mathcal{H}$ or in $\operatorname{dom} \mathcal{W}$. $\qquad\square$

Now we are ready to describe the $\perp$-handling stage of the algorithm. By the beginning of this stage, the structure of deterministic computation of $\mathcal{A}$ has already been almost completely captured by the productions written earlier, and it only remains to account for moves involving $\perp$. So this last stage of the algorithm takes the initial state q_0 of $\mathcal{A}$ and proceeds as follows.

If $q_0 \in$ NonRet, then take N'_{q_0} as the axiom of $\mathcal{T}'$ and N''_{q_0} as the axiom of $\mathcal{T}''$. By invariant (I5), these nonterminals are defined and generate appropriate words, so the pair $(\mathcal{T}', \mathcal{T}'')$ indeed generates the transcript of the computation of $\mathcal{A}$.

Since at the beginning of the $\perp$-handling stage $\operatorname{dom} \mathcal{G} = \emptyset$, it remains to consider the case $q_0 \in \operatorname{dom} \mathcal{E}$. Define a partial function $\mathcal{E}^{\perp} \colon Q \to Q$ by setting, for each $q \in \operatorname{dom} \mathcal{E}$, its value according to $\mathcal{E}^{\perp}(q) = \bar{q}$ if $\mathcal{E}(q) = q'$ and $\delta_{-1}(q', \perp) = \bar{q}$. Write productions $E_q^{\perp} \to E_q V_{q'}$ accordingly. Now associate $\mathcal{E}^{\perp}$ with a graph, as earlier, and consider the longest simple path within $\operatorname{dom} \mathcal{E}^{\perp}$ starting at q_0. Suppose it ends at a vertex q_k, where $\mathcal{E}^{\perp}(q_i) = q_{i+1}$ for $i = 0, \ldots, k$. There are two subcases here according to why the path cannot go any further.

The first possible reason is that it reaches $Q \setminus \operatorname{dom} \mathcal{E}^{\perp} =$ NonRet, that is, that q_{k+1} belongs to NonRet. In this subcase write $N'_{q_0} \to E_{q_0}^{\perp} \ldots E_{q_k}^{\perp} N'_{q_{k+1}}$ and

$N_{q_0}'' \to N_{q_{k+1}}''$. The second possible reason is that $q_{k+1} = q_i$ where $0 \leq i \leq k$. In this subcase write $N_{q_0}' \to E_{q_0}^\perp \ldots E_{q_{i-1}}^\perp$ and $N_{q_0}'' \to E_{q_i}^\perp \ldots E_{q_k}^\perp$.

In any of the two subcases above, take N_{q_0}' and N_{q_0}'' as axioms of $\mathcal{T}'$ and $\mathcal{T}''$, respectively. The correctness of this step follows easily from the invariants (I2) and (I5). This gives a polynomial algorithm that converts a udpda $\mathcal{A}$ into a pair of SLPs $(\mathcal{T}', \mathcal{T}'')$ that generates the transcript of the infinite computation of $\mathcal{A}$, and the only remaining bit is bounding the size of $(\mathcal{T}', \mathcal{T}'')$.

Claim 4. The size of $(\mathcal{T}', \mathcal{T}'')$ is $O(|Q|)$.

Proof. There are three types of nonterminals whose productions may have growing size: N_{q_k}'' in rule (R4), and N_{q_0}' and N_{q_0}'' in the $\perp$-handling stage. For all three types, the size is bounded by the cardinality of the set of states involved in a cycle or a path. Since such sets never intersect, all such nonterminals together contribute at most $|Q|$ productions to the Chomsky normal form. The contribution of other nonterminals is also $O(|Q|)$, because they all have fixed size and each state q is associated with a bounded number of them. □

Combined with Claim 1 in Subsection 5.2, this completes the proof of Lemma 4 and Theorem 1.

6 Universality of unpda

In this section we settle the complexity status of the universality problem for unary, possibly nondeterministic pushdown automata. While $\Pi_2\mathbf{P}$-completeness of equivalence and inclusion is shown by Huynh [14], it has been unknown whether the universality problem is also $\Pi_2\mathbf{P}$-hard.

For convenience of notation, we use an auxiliary descriptional system. Define *integer expressions* over the set of operations $\{+, \cup, \times 2, \times \mathbb{N}\}$ inductively: the base case is a non-negative integer n, written in binary, and the inductive step is associated with binary operations $+$, $\cup$, and unary operations $\times 2$, $\times \mathbb{N}$. To each expression E we associate a set of non-negative integers $S(E)$: $S(n) = \{n\}$, $S(E_1 + E_2) = \{s_1 + s_2 \colon s_1 \in S(E_1), s_2 \in S(E_2)\}$, $S(E_1 \cup E_2) = S(E_1) \cup S(E_2)$, $S(E \times 2) = S(E + E)$, $S(E \times \mathbb{N}) = \{sk \colon s \in S(E), k = 0, 1, 2, \ldots\}$.

Expressions E_1 and E_2 are called *equivalent* iff $S(E_1) = S(E_2)$; an expression E is *universal* iff it is equivalent to $1 \times \mathbb{N}$. The problem of deciding universality is denoted by INTEGER-$\{+, \cup, \times 2, \times \mathbb{N}\}$-EXPRESSION-UNIVERSALITY.

Decision problems for integer expressions have been studied for more than 40 years: Stockmeyer and Meyer [31] showed that for expressions over $\{+, \cup\}$ compressed membership is $\mathbf{NP}$-complete and equivalence is $\Pi_2\mathbf{P}$-complete (universality is, of course, trivial). For recent results on such problems with operations from $\{+, \cup, \cap, \times, \bar{}\}$, see McKenzie and Wagner [23] and Glaßer et al. [8].

Lemma 5. INTEGER-$\{+, \cup, \times 2, \times \mathbb{N}\}$-EXPRESSION-UNIVERSALITY *is* $\Pi_2\mathbf{P}$-*hard.*

Proof. The reduction is from the GENERALIZED-SUBSET-SUM problem, which is defined as follows. The input consists of two vectors of naturals, $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_m)$, and a natural t, and the problem is to decide whether for all $y \in \{0,1\}^m$ there exists an $x \in \{0,1\}^n$ such that $x \cdot u + y \cdot v = t$, where the middle dot $\cdot$ once again denotes the inner product. This problem was shown to be hard by Berman et al. [1, Lemma 6.2].

Start with an instance of GENERALIZED-SUBSET-SUM and let M be a big number, $M > \sum_{i=1}^n u_i + \sum_{j=1}^m v_j$. Assume without loss of generality that $M > t$. Consider the integer expression E defined by the following equations:

$$E = E' \cup E'',$$
$$E' = (2^m M + 1 \times \mathbb{N}) \cup (M \times \mathbb{N} + ([0, t-1] \cup [t+1, M-1])),$$
$$E'' = \sum_{j=1}^m (0 \cup (2^{j-1} M + v_j)) + \sum_{i=1}^n (0 \cup u_i),$$
$$[a, b] = a + [0, b-a],$$
$$[0, t] = [0, \lfloor t/2 \rfloor] \times 2 + (0 \cup (t \bmod 2)),$$
$$[0, 1] = 0 \cup 1,$$
$$[0, 0] = 0.$$

Note that the size of E is polynomial in the size of the input, and E can be constructed in logarithmic space. We show that E is universal iff the input is a yes-instance of GENERALIZED-SUBSET-SUM.

It is immediate that E is universal if and only if $S(E)$ contains 2^m numbers of the form $kM + t$, $0 \le k < 2^m$. We show that every such number is in $S(E)$ if and only if for the binary vector $y = (y_1, \ldots, y_m) \in \{0,1\}^m$, defined by $k = \sum_{j=1}^m y_j \, 2^{j-1}$, there exists a vector $x \in \{0,1\}^n$ such that $x \cdot u + y \cdot v = t$.

First consider an arbitrary $y \in \{0,1\}^m$ and choose k as above. Suppose that for this y there exists an $x \in \{0,1\}^n$ such that $x \cdot u + y \cdot v = t$. One can easily see that appropriate choices in E'' give the number $kM + y \cdot v + x \cdot u = kM + t$. Conversely, suppose that $kM + t \in S(E)$ for some k, $0 \le k < 2^m$; then $kM + t \in S(E'')$. Since $(1, \ldots, 1) \cdot u + (1, \ldots, 1) \cdot v < M$, it holds that $t = y \cdot v + x \cdot u$ for binary vectors $y \in \{0,1\}^m$ and $x \in \{0,1\}^n$ that correspond to the choices in the addends. Moreover, the same inequality also shows that kM is equal to the sum of some powers of two in the first sum in E'', and so $k = \sum_{j=1}^m y_j \, 2^{j-1}$. This completes the proof. □

Remark. With *circuits* instead of formulae (see also [23] and [8]) we would not need doubling. Furthermore, we only use $\times \mathbb{N}$ on fixed numbers, so instead we could use any feature for expressing an arithmetic progression with fixed common difference.

Theorem 6. UNARY-PDA-UNIVERSALITY *is* $\Pi_2\mathbf{P}$-*complete.*

Proof. A reduction from INTEGER-$\{+, \cup, \times 2, \times \mathbb{N}\}$-EXPRESSION-UNIVERSALITY, which is $\Pi_2\mathbf{P}$-hard by Lemma 5, shows hardness. Indeed, an integer expression

over $\{+, \cup, \times 2, \times \mathbb{N}\}$ can be transformed into a unary CFG in a straightforward way. Binary numbers are encoded by poly-size SLPs, summation is modeled by concatenation, and union by alternatives. Doubling is a special case of summation, and $\times \mathbb{N}$ gives rise to productions of the form $N' \to \varepsilon$ and $N' \to NN'$. The obtained CFG is then transformed into a unary PDA $\mathcal{A}$ by a standard algorithm (see, e. g., Savage [26, Theorem 4.12.1]). The result is that $L(\mathcal{A}) = \{1^s \colon s \in S(E)\}$, and $\mathcal{A}$ is computed from E in logarithmic space. This concludes the proof. $\square$

Remark. We give a simple proof of the $\Pi_2\mathbf{P}$ upper bound. Let $\varphi_{\mathcal{A}}(x)$ be an existential Presburger formula of size polynomial in the size of $\mathcal{A}$ that characterizes the Parikh image of $L(\mathcal{A})$ (see Verma, Seidl, and Schwentick [32, Theorem 4]). To show that an udpda $\mathcal{A}$ is non-universal, we find an $n \geq 0$ such that $\neg\varphi_{\mathcal{A}}(n)$ holds. Now we note that for any udpda $\mathcal{A}$ of size m, there is a deterministic finite automaton of size $2^{O(m)}$ accepting $L(\mathcal{A})$ (see discussion in Section 7 and Pighizzini [24]). Thus, n is bounded by $2^{O(m)}$. Therefore, checking non-universality can be expressed as a predicate: $\exists n \leq 2^{O(m)}.\neg\varphi_{\mathcal{A}}(n)$. This is a $\Sigma_2\mathbf{P}$-predicate, because the $\exists^*$-fragment of Presburger arithmetic is **NP**-complete [33].

Corollary 1. *Universality, equivalence, and inclusion are $\Pi_2\mathbf{P}$-complete for (possibly nondeterministic) unary pushdown automata, unary context-free grammars, and integer expressions over $\{+, \cup, \times 2, \times \mathbb{N}\}$.*

Another consequence of Theorem 6 is that deciding equality of a (not necessarily unary) context-free language, given as a context-free grammar, to any fixed context-free language L_0 that contains an infinite regular subset, is $\Pi_2\mathbf{P}$-hard and, if $L_0 \subseteq \{a\}^*$, $\Pi_2\mathbf{P}$-complete. The lower bound is by reduction due to Hunt III, Rosenkrantz, and Szymanski [12, Theorem 3.8], who show that deciding equivalence to $\{a\}^*$ reduces to deciding equivalence to any such L_0. The reduction is shown to be polynomial-time, but is easily seen to be logarithmic-space as well. The upper bound for the unary case is by Huynh [14]; in the general case, the problem can be undecidable.

7 Corollaries and discussion

Descriptional complexity aspects of udpda. Theorem 1 can be used to obtain several results on descriptional complexity aspects of udpda proved earlier by Pighizzini [24]. He shows how to transform a udpda of size m into an equivalent deterministic finite automaton (DFA) with at most 2^m states [24, Theorem 8] and into an equivalent context-free grammar in Chomsky normal form (CNF) with at most $2m + 1$ nonterminals [24, Theorem 12]. In our construction m gets multiplied by a small constant, but the advantage is that we now see (the slightly weaker variants of) these results as easy corollaries of a single underlying theorem. Indeed, using an indicator pair $(\mathcal{P}', \mathcal{P}'')$ for L, it is straightforward to construct a DFA of size $|\mathrm{eval}(\mathcal{P}')| + |\mathrm{eval}(\mathcal{P}'')|$ accepting L,

as well as to transform the pair into a CFG in CNF that generates L and has at most thrice the size of $(\mathcal{P}', \mathcal{P}'')$.

Another result which follows, even more directly, from ours is a lower bound on the size of udpda accepting a specific language L_1 [24, Theorem 15]. To obtain this lower bound, Pighizzini employs a known lower bound on the SLP-size of the word $W = W[0] \dots W[K-1] \in \{0,1\}^K$ such that $a^n \in L_1$ iff $W[n \bmod K] = 1$. To this end, a udpda $\mathcal{A}$ accepting L_1 is intersected (we are glossing over some technicalities here) with a small deterministic finite automaton that "captures" the end of the word W. The obtained udpda, which only accepts a^K, is transformed into an equivalent context-free grammar. It is then possible to use the structure of the grammar to transform it into an SLP that produces W (note that such a transformation in general is **NP**-hard). While the proof produces from a udpda for L_1 a related SLP with a polynomial blowup, this construction depends crucially on the structure of the language L_1, so it is difficult to generalize the argument to *all* udpda and thus obtain Theorem 1. Our proof of Theorem 1 therefore follows a very different path.

Relationship to Presburger arithmetic. An alternative way to prove the upper bound in Theorem 5 is via Presburger arithmetic, using the observation that there is a poly-time computable existential Presburger formula that expresses the membership of a word a^n in $L(\neg \mathcal{A}_1)$ and $L(\mathcal{A}_2)$. This technique distills the arguments used by Huynh [13,14] to show that the compressed membership problem for unary pushdown automata is in **NP**. It is used in a purified form by Plandowski and Rytter [25, Theorems 4 and 8], who developed a much shorter proof of the same fact (apparently unaware of the previous proof). The same idea was later rediscovered and used in a combination with Presburger arithmetic by Verma, Seidl, and Schwentick [32, Theorem 4].

Another application of this technique provides an alternative proof of the $\Pi_{\mathbf{2}}\mathbf{P}$ upper bound for unpda inclusion (Theorem 6): to show that $L(\mathcal{A})$ is universal, we check that $L(\mathcal{A})$ accepts all words up to length $2^{O(m)}$ (this bound is sufficient because there is a deterministic finite automaton for the language with this size—see the discussion above). The proof known to date, due to Huynh [14], involves reproving Parikh's theorem and is more than 10 pages long. Reduction to Presburger formulae produces a much simpler proof.

Also, our $\Pi_{\mathbf{2}}\mathbf{P}$-hardness result for unpda shows that the $\forall_{\text{bounded}} \exists^*$-fragment of Presburger arithmetic is $\Pi_{\mathbf{2}}\mathbf{P}$-complete, where the variable bound by the universal quantifier is at most exponential in the size of the formula. The upper bound holds because the $\exists^*$-fragment is **NP**-complete [33]. In comparison, the $\forall \exists^*$-fragment, without any restrictions on the domain of the universally quantified variable, requires co-nondeterministic $2^{n^{\Omega(1)}}$ time, see Grädel [10]. Previously known fragments that are complete for the second level of the polynomial hierarchy involve alternation depth 3 and a fixed number of quantifiers, as in Grädel [11] and Schöning [28]. Also note that the $\forall^s \exists^t$-fragment is **coNP**-complete for all fixed $s \geq 1$ and $t \geq 2$, see Grädel [11].

Problems involving compressed words. Recall Theorem 4: given two SLPs, it is **coNP**-complete to compare the generated words componentwise with re-

spect to any partial order different from equality. As a corollary, we get precise complexity bounds for SLP equivalence in the presence of wildcards or, equivalently, compressed matching in the model of partial words (see, e. g., Fischer and Paterson [6] and Berstel and Boasson [2]). Consider the problem SLP-PARTIAL-WORD-MATCHING: the input is a pair of SLPs $\mathcal{P}_1$, $\mathcal{P}_2$ over the alphabet $\{a, b, ?\}$, generating words of equal length, and the output is "yes" iff for every i, $0 \leq i < |\mathcal{P}_1|$, either $\mathcal{P}_1[i] = \mathcal{P}_2[i]$ or at least one of $\mathcal{P}_1[i]$ and $\mathcal{P}_2[i]$ is ? (a *hole*, or a single-character *wildcard*).

Schmidt-Schauß [27] defines a problem equivalent to SLP-PARTIAL-WORD-MATCHING, along with another related problem, where one needs to find occurrences of eval($\mathcal{P}_1$) in eval($\mathcal{P}_2$) (as in pattern matching), $\mathcal{P}_2$ is known to contain no holes, and two symbols match iff they are equal or at least one of them is a hole. For this related problem, he develops a polynomial-time algorithm that finds (a representation of) all matching occurrences and operates under the assumption that the number of holes in eval($\mathcal{P}_1$) is polynomial in the size of the input. He also points out that no solution for (the general case of) SLP-PARTIAL-WORD-MATCHING is known—unless a polynomial upper bound on the number of ?s in eval($\mathcal{P}_1$) and eval($\mathcal{P}_2$) is given. Our next proposition shows that such a solution is not possible unless $\mathbf{P} = \mathbf{NP}$. It is an easy consequence of Theorem 4.

Proposition 2. SLP-PARTIAL-WORD-MATCHING *is* **coNP**-*complete.*

Proof. Membership in **coNP** is obvious, and the hardness is by a reduction from SLP-COMPONENTWISE-$(0 \leq 1)$. Given a pair of SLPs $\mathcal{P}_1$, $\mathcal{P}_2$ over $\{0, 1\}$, substitute ? for 0 and a for 1 in $\mathcal{P}_1$, and b for 0 and ? for 1 in $\mathcal{P}_2$. The resulting pair of SLPs over $\{a, b, ?\}$ is a yes-instance of SLP-PARTIAL-WORD-MATCHING iff the original pair is a yes-instance of SLP-COMPONENTWISE-$(0 \leq 1)$. □

The wide class of *compressed membership* problems (deciding eval($\mathcal{P}$) $\in L$) is studied and discussed in Jeż [16] and Lohrey [20]. In the case of words over the unary alphabet, $w \in \{a\}^*$, expressing w with an SLP is poly-time equivalent to representing it with its length $|w|$ written in binary. An easy corollary of Theorem 2 is that deciding $w \in L(\mathcal{A})$, where $\mathcal{A}$ is a (not necessarily unary) deterministic pushdown automaton and $w = a^n$ with n given in binary, is **P**-complete.

Finally, we note that the precise complexity of SLP equivalence remains open [20]. We cannot immediately apply lower bounds for udpda equivalence, since we do not know if the translation from udpda to indicator pairs in Theorem 1 can be implemented in logarithmic (or even polylogarithmic) space.

References

1. Berman, P., Karpinski, M., Larmore, L.L., Plandowski, W., Rytter, W.: On the complexity of pattern matching for highly compressed two-dimensional texts. JCSS 65(2), 332–350 (2002)
2. Berstel, J., Boasson, L.: Partial words and a theorem of Fine and Wilf. TCS 218(1), 135–141 (1999)

3. Bertoni, A., Choffrut, C., Radicioni, R.: Literal shuffle of compressed words. In: Ausiello, G., Karhumki, J., Mauri, G., Ong, L. (eds.) IFIP TCS 2008. IFIP, vol. 273, pp. 87–100. Springer, Boston (2008)

4. Böhm, S., Göller, S., Jančar, P.: Equivalence of deterministic one-counter automata is **NL**-complete. In: STOC'13, pp. 131–140. ACM (2013)

5. Caussinus, H., McKenzie, P., Thérien, D., Vollmer, H.: Nondeterministic $\mathbf{NC}^1$ computation. JCSS 57(2), 200–212 (1998)

6. Fischer, M.J., Paterson, M.S.: String-matching and other products. In: Karp, R. (ed.) SIAM-AMS proceedings, vol. 7. AMS (1974)

7. Ginsburg, S., Rice, H.G.: Two families of languages related to ALGOL. Journal of the ACM 9(3), 350–371 (1962)

8. Glaßer, C., Herr, K., Reitwießner, C., Travers, S., Waldherr, M.: Equivalence problems for circuits over sets of natural numbers. Theory of Computing Systems 46(1), 80–103 (2010)

9. Goldschlager, L.M.: ε-productions in context-free grammars. Acta Informatica, 16(3), 303–308 (1981)

10. Grädel, E.: Dominoes and the complexity of subclasses of logical theories. Annals of Pure and Applied Logic 43(1), 1–30 (1989)

11. Grädel, E.: Subclasses of Presburger arithmetic and the polynomial-time hierarchy. TCS 56(3), 289–301 (1988)

12. Hunt III, H.B., Rosenkrantz, D.J., Szymanski, T.G.: On the equivalence, containment, and covering problems for the regular and context-free languages. JCSS 12(2), 222–268 (1976)

13. Huynh, D.T.: Commutative grammars: the complexity of uniform word problems. Information and Control 57, 21–39 (1983)

14. Huynh, D.T.: Deciding the inequivalence of context-free grammars with 1-letter terminal alphabet is Σ_2^p-complete. TCS 33(2–3), 305–326 (1984)

15. Jančar, P.: Decidability of DPDA language equivalence via first-order grammars. In: LICS 2012, pp. 415–424. IEEE (2012)

16. Jeż, A.: The complexity of compressed membership problems for finite automata. Theory of Computing Systems, 1–34 (2013)

17. Jones, N.D., Laaser, W.T.: Complete problems for deterministic polynomial time. TCS 3(2), 105–117 (1976)

18. Kopczyński, E., To, A.W.: Parikh images of grammars: complexity and applications. In: LICS 2010, pp. 80–89. IEEE Computer Society (2010)

19. Lifshits, Y., Lohrey, M.: Querying and embedding compressed texts. In: MFCS 2006. LNCS, vol. 4162, pp. 681–692. Springer (2006)

20. Lohrey, M.: Algorithmics on SLP-compressed strings: a survey. Groups Complexity Cryptology 4(2), 241–299 (2012)

21. Lohrey, M.: Leaf languages and string compression. Information and Computation 209(6), 951–965 (2011)

22. Lohrey, M.: Word problems and membership problems on compressed words. SIAM Journal on Computing 35(5), 1210–1240 (2006)

23. McKenzie, P., Wagner, K.W.: The complexity of membership problems for circuits over sets of natural numbers. Computational Complexity 16(3), 211–244 (2007)

24. Pighizzini, G.: Deterministic pushdown automata and unary languages. International Journal of Foundations of Computer Science 20(4), 629–645 (2009)

25. Plandowski, W., Rytter, W.: Complexity of language recognition problems for compressed words. In: Karhumäki, J., Maurer, H., Păun, G., Rozenberg, G. (eds.) Jewels are Forever, pp. 262–272. Springer (1999)

26. Savage, J.E.: Models of computation: Exploring the power of computing. Addison-Wesley (1998)
27. Schmidt-Schauß, M.: Matching of compressed patterns with character variables. In: RTA 2012. LIPIcs, vol. 15, pp. 272–287. Dagstuhl (2012)
28. Schöning, U.: Complexity of Presburger arithmetic with fixed quantifier dimension. Theory of Computing Systems 30(4), 423–428 (1997)
29. Sénizergues, G.: $L(A) = L(B)$? A simplified decidability proof. TCS 281(1–2), 555-608 (2002)
30. Stirling, C.: Deciding DPDA equivalence is primitive recursive. In: ICALP 2002. LNCS, vol. 2380, pp. 821–832. Springer (2002)
31. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time: Preliminary report. In: STOC 1973, pp. 1–9. ACM, New York (1973)
32. Verma, K.N., Seidl, H., Schwentick, T.: On the complexity of equational Horn clauses. In: CADE 2005. LNAI, vol. 3632, pp. 337–352. Springer (2002)
33. Von zur Gathen, J., Sieveking, M.: A bound on solutions of linear integer equalities and inequalities. Proceedings of the AMS 72(1), 155–158 (1978)