The Complexity of Model Checking (Collapsible) Higher-Order Pushdown Systems

Matthew Hague[1] and Anthony Widjaja To[2]

1,2 Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, OX1 3QD

Abstract

We study (collapsible) higher-order pushdown systems — theoretically robust and well-studied models of higher-order programs — along with their natural subclass called (collapsible) higher-order basic process algebras. We provide a comprehensive analysis of the model checking complexity of a range of both branching-time and linear-time temporal logics. We obtain tight bounds on data, expression, and combined-complexity for both (collapsible) higher-order pushdown systems and (collapsible) higher-order basic process algebra. At order-k, results range from polynomial to $(k+1)$-exponential time.

1998 ACM Subject Classification D.2.4

Keywords and phrases Higher-Order, Collapsible, Pushdown Systems, Temporal Logics, Complexity, Model Checking

1 Introduction

Recently, there has been a burgeoning interest in collapsible higher-order pushdown systems (CPDSs), both as generators of structures and as models of higher-order computation. Whereas an order-1 pushdown system augments a finite-state automaton with an unbounded stack memory, a higher-order pushdown system (HOPDS) provides a nested "stack-of-stacks" structure. CPDSs allow a further backtracking operation called *collapse*.

Higher-order pushdown automata (HOPDA) were introduced by Maslov [24]. *Higher-order pushdown systems* (HOPDS) are HOPDA viewed as generators of infinite trees or graphs. Recently these models have been generalised to *collapsible pushdown systems* (CPDS) [17, 19]. In terms of expressivity, order-k CPDSs generate the same class of ranked trees as deterministic order-k recursion schemes [17]. The analogous result holds for *safe* recursion schemes and HOPDSs [18]. These systems provide a natural model for higher-order programs with (unbounded) recursive function calls and are therefore useful in software verification. Further results show an intimate connection with the Caucal hierarchy [10, 11]. For verification, reachability properties — which ask whether a given set of control states can be reached from the initial configuration — are complete for $(k-1)$-ExpTime [5, see appendix], whilst μ-calculus properties are k-ExpTime-complete [7, 26, 17]. Despite these high complexities, Kobayashi has verified resource usage properties of higher-order programs [20] using a novel approach based on intersection types [21, 23].

Hitherto, there has been little work addressing the precise complexity of model checking higher-order programs with respect to the common temporal logics. In most cases, there is currently a single or double exponential gap in the best known upper and lower bounds (derived usually from μ-calculus and reachability respectively). One main contribution of this paper is a nearly complete picture of the model checking complexities against temporal

	(Collapsible) HOPDS		(Collapsible) HOBPA	
	Data	Expression & Combined	Data	Expression & Combined
μLTL / LTL	$(k-1)$-ExpTime	k-ExpTime	P-time	k-ExpTime
LTL(F, X)	$(k-1)$-ExpTime	k-ExpTime	P-time	k-ExpTime
LTL(U)	$(k-1)$-ExpTime	k-ExpTime	P-time	k-ExpTime
CTL	k-ExpTime	k-ExpTime	P-time	k-ExpTime
CTL+	k-ExpTime	$(k+1)$-ExpTime	P-time	$(k+1)$-ExpTime
CTL*	k-ExpTime	$(k+1)$-ExpTime	P-time	$(k+1)$-ExpTime
EF	$(k-1)$-ExpSpace-hard	$(k-1)$-ExpSpace-hard	P-time	$(k-1)$-ExpSpace-hard

Figure 1 The complexity of model checking order-k higher-order systems. Unless stated, all results are complete.

logics. In particular, we consider data complexity (formulas are fixed), expression complexity (systems are fixed), and combined complexity (both formulas and systems are input parameters). Table 1 (left column) summarises our results. In all cases, our lower bounds hold without the collapse operation, whilst our upper bounds allow collapse.

Basic process algebras (BPAs) are a natural and well-studied subclass of order-1 PDSs (cf. [6]), which are suitable abstractions for modelling the control-flow of sequential programs (cf. [2, 14]). We propose higher-order extensions of BPAs, called *(collapsible) higher-order basic process algebras* (HOBPAs), that form a natural subclass of (collapsible) HOPDSs. This differs from the single-state HOPDSs introduced by Bouajjani and Meyer [3]. As graph generators, (collapsible) HOBPAs are almost as powerful as (collapsible) HOPDSs in the following sense: (1) like CPDS, there exists a collapsible order-2 BPA whose graph has an undecidable monadic second-order logic (MSO) theory, and (2) the class of graphs generated by order-k BPAs coincide with those generated by order-k PDSs up to MSO interpretations. In this paper, we provide an almost complete picture for the model checking complexities of standard temporal logics over (collapsible) HOBPAs. See Table 1 (right column). We show that the restriction to HOBPA does not, in most cases, simplify the model checking problem; a notable exception is for data complexity, where the problem becomes polynomial time. Again, our lower bounds hold without collapse, whilst our upper bounds allow collapse.

Similar analyses appear across a number of papers for the special case of order-1 pushdown systems [1, 6, 30, 31, 4]. In all cases we generalise the resulting picture in a natural manner. That is, a 1-ExpTime-complete complexity becomes k-ExpTime-complete, and so on. Our upper bound results concern the data complexity of Collapsible HOBPAs and the data and combined complexities for LTL over CPDSs. Previous work studied reachability, LTL and the alternation-free μ-calculus [16, 27, 13] over HOPDS without collapse. However, we believe the LTL algorithm contains an error, and provide a new algorithm. Furthermore, the alternation-free μ-calculus algorithm in [16] is not optimal. Our remaining results concern lower bounds. We begin with two techniques from in the literature: (1) Engelfriet's characterization of complexity classes k-ExpTime by extensions of HOPDAs (e.g. with space-bounded worktape) [13], and (2) Cachat and Walukiewicz's more "direct" approach via encodings of large numbers using HOPDSs [8]. We employ Technique (1) to prove the lower bounds for LTL (and its fragments), CTL, CTL+, and CTL*. This does not mean that the proofs of the results are immediate: it was left as an open problem in [8] whether the two techniques can be used to derive these lower bounds. Since Technique (1) seems only suited to deriving k-ExpTime lower bounds (for some k), we give two variations of Technique (2) to derive $(k-1)$-ExpSpace lower bounds for EF model checking over HOPDSs and HOBPAs (the latter proof is substantially more involved). The lower bound proofs in this paper suggest that Technique (1) yields simpler proofs, while Technique (2)

offers more flexibility.

The preliminaries are given in §2. We begin in §3 with the results for fixed formulas over collapsible HOBPA. In §4 we discuss branching-time logics, and linear-time in §5. Finally, we conclude this paper with future work in §6. Due to the length and intricate nature of the proofs, we relegate the full details into the full version.

2 Preliminaries

We define (collapsible) higher-order pushdown systems and basic process algebra and give a result of Engelfriet used in some proofs. Note, after defining higher-order and collapsible stores, we only define higher-order systems. For the collapsible version, simply replace the higher-order store with a collapsible one, expanding the stack operations accordingly. Also, the definitions generalise from non-deterministic to alternating in the standard way.

Higher-Order Collapsible Pushdown Stores

We begin by defining a higher-order pushdown store. Collapse links will be introduced afterwards. Intuitively, a higher-order store is a stack of lower order stacks.

▶ **Definition 1** (k-Stores). Let C_0^Σ be a finite alphabet Σ with $[,] \notin \Sigma$. For $k \geq 1$, the set of *k-stores* C_k^Σ contains all $[\gamma_1 \dots \gamma_m]$ with $m \geq 1$ and $\gamma_i \in C_{k-1}^\Sigma$ for all $1 \leq i \leq m$.

There are two operations defined over 1-stores (for all $w \in \Sigma^*$)

$$push_w[a_1 \dots a_m] = [wa_2 \dots a_m] \quad \text{and} \quad top_1[a_1 \dots a_m] = a_1 \ .$$

We define $pop_1 = push_\varepsilon$. Let $\mathcal{O}_1 = \{\ push_w \mid w \in \Sigma^*\ \}$. When $k > 1$, a push operation creates a copy of the topmost stack, while a pop removes it. We assume w.l.o.g. that $\Sigma \cap \mathbb{N} = \emptyset$, where $\mathbb{N}$ is the set of natural numbers. Finally, let $[\gamma_1 \dots \gamma_m] \in C_k^\Sigma$ for some k.

$$\begin{array}{rcll}
push_w[\gamma_1 \dots \gamma_m] & = & [push_w(\gamma_1)\gamma_2 \dots \gamma_m] & \\
push_l[\gamma_1 \dots \gamma_m] & = & [push_l(\gamma_1)\gamma_2 \dots \gamma_m] & \text{if } 2 \leq l < k \\
push_k[\gamma_1 \dots \gamma_m] & = & [\gamma_1\gamma_1\gamma_2 \dots \gamma_m] & \\
pop_l[\gamma_1 \dots \gamma_m] & = & [pop_l(\gamma_1)\gamma_2 \dots \gamma_m] & \text{if } 1 \leq l < k \\
pop_k[\gamma_1 \dots \gamma_m] & = & [\gamma_2 \dots \gamma_m] & \text{if } m > 1 \\
top_l[\gamma_1 \dots \gamma_m] & = & top_l(\gamma_1) & \text{if } 1 \leq l < k \\
top_k[\gamma_1 \dots \gamma_m] & = & \gamma_1 &
\end{array}$$

Note, when $m = 1$, pop_k is undefined. Let $\mathcal{O}_k = \{\ push_w \mid w \in \Sigma^*\ \} \cup \{\ push_l, pop_l \mid 1 < l \leq k\ \}$. We designate $\bot$ to be a *bottom of stack* symbol that is neither pushed nor popped. Let $[w]_1 = [w]$ and $[w]_k = [[w]_{k-1}]$.

For collapse, the order-1 push operation $push_w$ is replaced with $push_{a_1^{i_1} \dots a_m^{i_m} b}$ for $1 \leq i_z \leq k$ and $a_z, b \in \Sigma$ where $1 \leq z \leq m$. A $push_{a_1^{i_1} \dots a_m^{i_m} b}$ on some stack with top_1 character a is equivalent to $push_{a_1 \dots a_m b}$ except each a_z is augmented with a pair $(i_z, 1)$. That is, the top of stack character $a^{(i,j)}$ is replaced by $a_1^{(i_1,1)} \dots a_m^{(i_m,1)} b^{(i,j)}$. The collapse operation from a character $a^{(i,j)}$ is equivalent to j applications of pop_i. The second component j is incremented at every $push_i$. Hence, (i, j) is a link to the order-$(i-1)$ stack beneath the character when it was first pushed.

Consider $[[[\bot]\,[\bot]]]$. Applying $push_{a^2\bot}$ gives $\left[\left[\left[a^{(2,1)}\ \bot\right]\,[\bot]\right]\right]$. The pair $(2, 1)$ points to $[\bot]$. A $push_2$ leads to $\left[\left[\left[a^{(2,2)}\ \bot\right]\,\left[a^{(2,1)}\ \bot\right]\,[\bot]\right]\right]$. Note the second component is incremented in the copy of a, and, thus, $(2, 2)$ also points to $[\bot]$. A subtlety occurs after a $push_3$. We

obtain the stack below on the left, where the copies of a now refer to the copies of $[\bot]$ within the order-2 stack they occupy. After a collapse, we obtain the stack on the right.

$$\left[\begin{array}{c}\left[\left[a^{(2,2)}\ \bot\right]\left[a^{(2,1)}\ \bot\right][\bot]\right]\\ \left[\left[a^{(2,2)}\ \bot\right]\left[a^{(2,1)}\ \bot\right][\bot]\right]\end{array}\right] \qquad \left[\begin{array}{c}[[\bot]]\\ \left[\left[a^{(2,2)}\ \bot\right]\left[a^{(2,1)}\ \bot\right][\bot]\right]\end{array}\right]$$

Formally, we define order-k stores with links in terms of order-k stores over the infinite alphabet $\Sigma = \{\ a^{(i,j)}\ \mid i,j \in \mathbb{N}\ \}$. The set of operations over an order-k store with links is

$$\begin{aligned}\mathcal{O}_k^c \quad = \quad & \left\{\ push_{a_1^{i_1}\ldots a_m^{i_m} b}\ \mid \forall 1 \le z \le m . 1 \le i_z \le k \wedge a_z \in \Sigma\ \right\}\\ & \cup \{\ push_l, pop_l, collapse\ \mid 1 < l \le k\ \}\ .\end{aligned}$$

Note that this set of operations is slightly different from the original definition [17]. We show, in the full version, that the definitions are equivalent. The semantics of the operations are given below, in terms of the standard order-k pushdown operators, and an order-k stack $\gamma = [\gamma_1 \ldots \gamma_m]$. Let $\gamma^{<k>}$ be the stack γ where each superscript (i,j) with $i \ge k$ is replaced with $(i, j+1)$.

$$\begin{array}{rcll} push_{a_1^{i_1}\ldots a_m^{i_m} b}(\gamma) & = & push_{a_1^{(i_1,1)}\ldots a_m^{(i_m,1)} b_m^{(i',j')}}(\gamma) & \text{where } top_1(\gamma) = b^{(i',j')}\\ collapse(\gamma) & = & pop_i^j(\gamma) & \text{where } top_1(\gamma) = b^{(i,j)}\\ push_k[\gamma_1 \ldots \gamma_m] & = & [\gamma_1^{<k>}\gamma_1 \ldots \gamma_m] & \\ push_l[\gamma_1 \ldots \gamma_m] & = & [push_l(\gamma_1)\gamma_2 \ldots \gamma_m] & \text{where } l < k \end{array}$$

Higher-Order Pushdown Systems

A HOPDS is a finite-state system with a higher-order store. The finite-state component is the control state. At each step, the applicable transitions are determined by the control state and the top_1 character of the stack. Each transition updates the control state and the stack.

▶ **Definition 2.** An *order-k PDS* is a tuple $(P, \mathcal{R}, \Sigma, p_0, \bot)$ where P is a finite set of control states, $\mathcal{R} \subseteq P \times \Sigma \times \mathcal{O}_k \times P$ is a finite set of rules, Σ is a finite stack alphabet, $p_0 \in P$ is an initial control state and $\bot \in \Sigma$ is a bottom of stack symbol.

A configuration of a higher-order PDS is a pair $\langle p, \gamma\rangle$ where $p \in P$ and γ is a k-store. We have a transition $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle$ iff we have $(p, a, o, p') \in \mathcal{R}$, $top_1(\gamma) = a$ and $\gamma' = o(\gamma)$. The initial configuration is $\langle p_0, [\bot]_k\rangle$.

Higher-Order Basic Process Algebra

An order-1 BPA is an order-1 PDS with a single control state. By applying the same restriction, Bouajjani and Meyer have obtained one definition of higher-order BPA [3]. However, consider $\langle p, [[a\ \bot]]\rangle$ and the rule (omitting the control) $(a, push_2)$. We obtain $\langle p, [[a\ \bot]\,[a\ \bot]]\rangle$ and the same rule can be applied, ad infinitum. However, at order-1 we may use $(a, push_{bc})$ to rewrite the top character before adding a new top character. Hence, at order-j, it is natural to be able to rewrite the top order-$(j-1)$ stack, before adding a new one. Consequently, we introduce $(a, push_j, b) = push_a; push_j; push_b$ for all $2 \le j \le k$. E.g., such rules can simulate $push_2; push_3; pop_2$. Let $\mathcal{O}'_k = \{\ push_w \mid w \in \Sigma^*\ \} \cup \{\ (a, push_j, b), pop_j \mid 1 < j \le k\ \}$.

▶ **Definition 3.** An *order-k BPA* is a tuple $(\mathcal{R}, \Sigma, \bot)$ where $\mathcal{R} \subseteq \Sigma \times \mathcal{O}'_k$ is a finite set of rules, Σ is a finite stack alphabet, and $\bot \in \Sigma$ is the bottom of stack symbol.

We mention two results on the expressive power of HOBPAs as graph generators. Familiarity with monadic second-order logic (MSO) is assumed (cf. [28]). As graph generators, (collapsible) HOBPAs are as powerful as (collapsible) HOPDAs up to monadic second-order logic (MSO) interpretation in the following sense. First, it is known that there exists an order-2 CPDA generating a graph with an undecidable MSO theory [17]. In contrast, over HOPDAs, MSO is decidable. This CPDA is not a collapsible HOBPA. On the other hand, using the ideas from [17], it is not difficult to come up with an order-2 collapsible HOBPA generating a graph with an undecidable MSO theory.

▶ **Proposition** 1. There exists a fixed collapsible order-2 BPA which generates a graph with an undecidable MSO theory.

We sketch the proof of this proposition in the full version. Secondly, we discuss the expressive power of HOBPAs without collapse. Carayol and Wöhrle [9, 10] gave a fixed graph Δ_2^k, for each integer $k > 0$, such that the class of graphs that are MSO-interpretable in the graphs generated by order-k PDSs coincide with the class of graphs that are MSO-interpretable in Δ_2^k. It is easy to check that Δ_2^k can be generated by a fixed order-k BPAs (e.g. see [9]), which implies the following proposition.

▶ **Proposition** 2. The class of graphs that are MSO-interpretable in the graphs generated by order-k BPAs coincide with the class of graphs MSO-interpretable in the graphs generated by order-k PDSs.

Higher-Order Pushdown Automata with an Auxiliary Work Tape

For the lower bound proofs we use HOPDA with a space-bounded work tape. That is, in addition to the control state and the stack, the machine has a bounded, two-way work tape. This tape operates identically to the tape in a Turing machine.

▶ **Definition 4.** An *order-k PDA with an $s(n)$-space work tape* is a tuple $(P, \mathcal{R}, \Sigma, \Gamma \cup \{\varepsilon\}, \Delta, p_0, \bot, \Box, \mathcal{F})$ where P is a finite set of control states, $\mathcal{R} \subseteq (P \times \Gamma \cup \{\varepsilon\} \times \Sigma \times \Delta) \times \mathcal{O}_k \times (\Delta \times \{l, r\} \times P)$ is a finite set of rules, Σ is a finite stack alphabet, Γ is a finite input alphabet, Δ is a finite tape alphabet, $p_0 \in P$ is an initial control state, $\bot \in \Sigma$ is the bottom of stack symbol, $\Box \in \Delta$ denotes a blank tape cell and $\mathcal{F} \subseteq P$ is a set of accepting control states.

Given an input word of length n, a configuration of a HOPDA with $s(n)$ bounded work tape is a tuple $\langle p, \gamma, t, j\rangle$ where $p \in P$, γ is a k-store, t (the tape contents) is a word in $\Delta^{s(n)}$ and $1 \leq j \leq s(n)$ indicates the position of the read/write head on the tape.

A rule $(p, \alpha, a, x, o, y, d, p') \in \mathcal{R}$ can be applied when the current control state is p, the input character is α, the top-of-stack character is a, and the tape contents at position j are x. The control state is then updated to p', the command o is applied to the stack, and y is written to the tape. The tape head moves accordingly for $d = l$ (left) or $d = r$ (right).

More formally, we have a transition $\langle p, \gamma, t, j\rangle \overset{\alpha}{\hookrightarrow} \langle p', \gamma', t', j'\rangle$ iff we have $(p, \alpha, a, x, o, y, l, p') \in \mathcal{R}$, $j > 1$, $top_1(\gamma) = a$, $t(j) = x$, $\gamma' = o(\gamma)$, $t'(j) = y$, $t'(h) = t(h)$ for all $h \neq j$ and $j' = j - 1$ or we have $(p, \alpha, a, x, o, y, r, p') \in \mathcal{R}$, $j < s(n)$, $top_1(\gamma) = a$, $t(j) = x$, $\gamma' = o(\gamma)$, $t'(j) = y$, $t'(h) = t(h)$ for all $h \neq j$ and $j' = j - 1$. For $\alpha \neq \varepsilon$, we write $c \overset{\alpha}{\hookrightarrow}_\varepsilon c'$ whenever there is a sequence of ε-transitions from c to some c_1, an α-transition from c_1 to c_2 and a sequence of ε-transitions to c'. A word $\alpha_1, \ldots, \alpha_n$ is accepted by the automaton iff $c_n = \langle p, \gamma\rangle$ and $p \in \mathcal{F}$ and $c_0 \overset{\alpha_1}{\hookrightarrow}_\varepsilon \cdots \overset{\alpha_n}{\hookrightarrow}_\varepsilon c_n$ where $c_0 = \langle p_0, [\bot]_k, \Box^{s(n)}, 1\rangle$.

Temporal Logics

We will assume familiarity with the temporal logics discussed, remarking only that μLTL is LTL extended with fixed point operators. Full definitions can be found in the literature [12, 29]. We assume, for all logics, the valuations of atomic propositions depend only on the control state and current top-of-stack character, referred to as a *head*. That is, $\Lambda : P \times \Sigma \to 2^{Prop}$ is an assignment of satisfied atomic propositions from the set $Prop$ to each head in $P \times \Sigma$. We say a system satisfies a formula if it holds at the initial state of the system.

Engelfriet's Results

We use the following theorem of Engelfriet [13] in some proofs. Let NSPACE($s(n)$)-P^k denote the class of languages accepted by a non-deterministic order-k PDA with an $s(n)$-space-bounded work tape, where n is the length of the input word. Similarly, ASPACE($s(n)$)-P^k denotes the class of languages accepted by an alternating order-k PDA with an $s(n)$-space-bounded work tape. Finally $\bigcup_{d>0}$ DTIME($exp_k(ds(n))$) is the class of languages accepted by a time-bounded Turing machine, where $exp_0(x) = x$ and $exp_k(x) = 2^{exp_{k-1}(x)}$.

▶ **Theorem 5** ([13], Thm. 2.5)**.** *For any $k \geq 1$ and $s(n) \geq log(n)$, we have NSPACE($s(n)$)-P^k = ASPACE($s(n)$)-P^{k-1} = $\bigcup_{d>0}$ DTIME($exp_k(ds(n))$).*

That is, a non-deterministic order-k PDA with a polynomially-bounded work tape exists for every k-ExpTime language, and an alternating order-k PDA with a polynomially-bounded work tape exists for every $(k+1)$-ExpTime language.

3 Model Checking Collapsible HOBPA Against Fixed Formulas

We begin with a P-time algorithm for model checking collapsible HOBPA against fixed formulas. Hardness follows from the P-time-hardness of context-free language emptiness [15].

▶ **Theorem 6.** *For any logic that can be translated into μ-calculus, model checking collapsible HOBPA against a fixed formula is in P-time.*

Proof. As argued in the full version, any collapsible HOBPA can be simulated by a CPDS with a fixed number of control states. Therefrom, and since the formula is fixed, we construct a CPDS parity game with a fixed number of control states. At order-k, the winner of these games can be determined in k-ExpTime in the number of control states, and polynomial in the alphabet [17]. Hence, the algorithm runs in P-time. ◀

4 Branching Time

We begin by observing, for CPDS, the upper bounds for CTL, CTL+ and CTL* can be obtained by translating into μ-calculus, which has a k-ExpTime model checking problem. For CTL, the translation is polynomial. For CTL+ and CTL* it is exponential, giving $(k+1)$-ExpTime, and k-ExpTime when the formula is fixed. For the lower bound results, we discuss EF, CTL and then CTL+.

▶ **Theorem 7.** *For a fixed formula, and a given order-k CPDS, model checking CTL, CTL+ and CTL* is in k-ExpTime. For a non-fixed system and non-fixed formula, CTL is k-ExpTime, and CTL+ and CTL* are in $(k+1)$-ExpTime.*

4.1 Lower Bounds for EF

In most cases, we are able to derive optimal lower bounds using Theorem 5. However, Theorem 5 is not immediately applicable for (e.g.) $(k-1)$-ExpSpace problems. In the case of EF-logic, the model checking problem over order-1 PDSs and BPAs is PSpace-complete [25, 31]. We now give $(k-1)$-ExpSpace lower bounds for data complexity of order-k PDSs and the expression complexity of order-k BPAs (and thus of order-k PDSs) using the technique of [8] of encoding large numbers. We conjecture that these lower bounds are tight (currently, the best upper bound is k-ExpTime, which is inherited from μ-calculus).

▶ **Theorem 8.** *Model checking EF over order-k PDS without collapse is $(k-1)$-ExpSpace-hard, even for a fixed formula.*

Proof. (sketch) We reduce membership for a given $(k-1)$-ExpSpace Turing machine M using $exp_{k-1}(p(n))$ space on an input word of length n, for some polynomial function p. Fix a number $m \in \mathbb{Z}_{>0}$, which we will later define as $p(n)$ once n is set. The proof combines the technique of [1] for proving that EF-logic over PDS is PSpace-hard and the technique of [8] for encoding and checking large numbers (i.e. k-towers of exponentials) using operations in $\mathcal{O}_k$.

We shall start by briefly recalling the encoding techniques of large numbers from [8]. For each $i \in \mathbb{Z}_{>0}$, we define $\Sigma_i := \{a_i, b_i\}$ and $\Sigma_{\leq i} := \bigcup_{j=1}^{i} \Sigma_i$. We now define the notion of *i-counters* by induction. A *1-counter* (of length m) is a word $\sigma_{m-1} \dots \sigma_0 \in (\Sigma_1)^m$. Such a word naturally represents the number $\sum_{i=0}^{m-1} \sigma_i 2^i$ where a_1 represents 0 and b_1 represents 1. Assuming that the notion of i-counter has been defined, an *$(i+1)$-counter* is simply a word $\sigma_r l_r \dots \sigma_0 l_0$ over $\Sigma_{\leq i+1}$, where $r = exp_i(m) - 1$, $\sigma_j \in \Sigma_{i+1}$, and l_j is an i-counter representing the number j. This $(i+1)$-counter represents the number $\sum_{j=0}^{r} \sigma_j 2^j$, where (as before) a_{i+1} and b_{i+1} are used to (respectively) represent 0 and 1.

Cachat and Walukiewicz [8] showed that a polynomial-size order-k pushdown game arena $\mathcal{P}$ with a reachability objective could be defined (depending only on m) with the following control states and properties: $counter_k$ — from configuration $(counter_k, \gamma)$ of $\mathcal{P}$, Player 0 wins iff γ ends with a k-counter; $first_k$ (resp. $last_k$) — from configuration $(first_k, \gamma)$ (resp. $(last_k, \gamma)$), Player 0 wins iff γ ends with a k-counter representing 0 (resp. $exp_{k-1}(m)$); $equal_k$ — from $(equal_k, \gamma)$, Player 0 wins iff γ ends with two k-counters representing equal values; $succ_k$ — from $(succ_k, \gamma)$, Player 0 wins iff γ ends with two k-counters representing successive values. We observe that the game element of $\mathcal{P}$ can easily be translated into fixed EF formulas (i.e. not depending on m) satisfying the same properties, the main reason being that the game arena $\mathcal{P}$ has a fixed number of rounds.

The rest of the proof uses the idea of [1]. Using an EF operator, we will first guess a word in $\Sigma_{\leq k+1}$ representing an accepting computation of M on the given input word $w = \alpha_1 \dots \alpha_n$. We then need to check that the guess is valid. That is, it represents a sequence of configurations, the initial configuration is the right form, the final configuration is reached, and consecutive configurations respect the transition relation. All these can be done by means of a fixed formula, thanks to the result above for encoding large numbers. ◀

▶ **Theorem 9.** *For a fixed order-k HOBPA without collapse, model checking EF is $(k-1)$-ExpSpace-hard.*

Proof. (sketch) The proof uses some general ideas from the previous proof, but, without control states to encode tests for large numbers, we need an entirely different construction. We briefly explain the order-2 case. Our HOBPA $\mathcal{P}$ will guess an accepting run of a fixed exponential space Turing machine M accepting an ExpSpace-complete language, obtaining a

stack of the form $[w]_2$. For the checking stage, our HOBPA $\mathcal{P}$ now tries to find some location inside the stack that is invalid. In doing so, we need to ensure that all of the information on top of this location is not destroyed. To this end, we will build a stair-like structure from $[w]_2$ by performing operations of the form $[push_1(a'); push_2; push_1(prime)]$ or of the form $[push_1(a''); push_2; push_1(dprime)]$ when seeing a topmost stack symbol a. Here, $prime$ and $dprime$ are simply intermediate symbols to help signify the action that was previous executed, i.e., we could simply only allow pop_1 operation when $prime$ or $dprime$ is seen as topmost symbol. The double prime marking is used to "remember" the *starting* point of (sub)configuration that we suspect is invalid. That is, we will have to make sure that it is put precisely once. At some point, $\mathcal{P}$ simply applies rules of the form $push_1(a')$ when a is seen without applying $push_2$, which marks the *end* point of a (sub)configuration that we suspect is invalid. We then only allow rules pop_2 when primed or double primed symbols are seen. To make sure that we see precisely one separator symbol (i.e. $a_3 \in \Sigma_3$), we can use an EF formula saying facts about the location of the double primed symbol a_3''. Such a stair-like structure will allow us to define EF formulas that play the roles of $counter_i$, $first_i$, $last_i$, $equal_i$, and $succ_i$ and their associated EF formulas in the previous proof. ◀

4.2 Lower Bounds for CTL

Data Complexity We know, for a fixed formula, model checking CTL, CTL+ and CTL* against HOPDSs is in k-ExpTime. Here, we show the lower bound.

▶ **Theorem 10.** *For a fixed formula, model checking CTL over a given order-k HOPDS without collapse is k-ExpTime-hard.*

Proof. (sketch) From Theorem 5 we take a language that is k-ExpTime-hard and fix an equivalent order-$(k-1)$ alternating HOPDA with a polynomially space-bounded work tape $\mathcal{P}$. The reduction is inspired by Bozzelli [4].

We use an order-k stack to navigate a computation tree of the HOPDA. To simulate the work tape, at each step, after an operation on the order-$(k-1)$ stack, a sequence of tape symbols are pushed on to the top order-1 stack. Then, the system can do a check branch to ensure the guessed tape is consistent with the previous, or continue simulating the execution. To continue, an order-k push saves the current state (for backtracking), the work tape is erased, the next rule is announced, and a $push_k$ remembers the rule. This process repeats. Consider the example order-3 stack below.

$$\left[\begin{array}{c} \left[\begin{array}{r} [tw_1] \\ [w_2] \\ \cdots \end{array}\right] \end{array}\right. \quad \left[\begin{array}{c} [r\ldots] \\ \cdots \end{array}\right] \quad \left[\begin{array}{r} [t'w_1'] \\ [w_2'] \\ \cdots \end{array}\right] \quad \left.\begin{array}{c} \cdots \end{array}\right]$$

This stack is at a configuration with the tape given by the word t and order-2 stack $[[w_1]\,[w_2]\ldots]$, which can backtrack to a configuration with tape t' and order-2 stack $[[w_1']\,[w_2']\ldots]$. The rule r connects the configurations. When an accepting configuration is seen, or the children of the current node have been fully explored, we backtrack using pop_k, and check untested universal branches. The automaton accepts when the (marked) initial stack is reached. That is, all paths have been explored, and found to be accepting.

The check branches have further branches for each of the polynomially many positions of the work tape. Each branch uses the control state to find the correct position, and then, using the control state, compares it with the corresponding positions in the previous work tape, which is recovered via pop_k operations.

The CTL formula $E((op \wedge AX(check \rightarrow AF good))\, U\, fin)$ asserts a path encoding an accepting tree exists, and checking branches all accept. The proposition *op* indicates the current path is simulating a tree, and *check* indicates a checking branch. Finally, *good* indicates that the check has been passed, and fin denotes the (successful) completion of the run. ◀

Expression Complexity The following theorem takes care of all cases.

▶ **Theorem 11.** *For a fixed order-k HOBPA without collapse, model checking CTL is k-ExpTime-hard.*

Proof. (sketch) The proof is in stages. First, we adapt Theorem 10, unfixing the formula to fix the HOPDS. A HOBPA is obtained using a more complex formula. There are two main assertions we move from the HOPDS to the formula. First, the check branch becomes a straight-line sequence of pops and the formula uses sequences of EX to compare positions. Secondly, the word position being read has to be guessed and added to the work tape information, then checked by the formula. Hence, we have the result for a fixed HOPDS.

To obtain a HOBPA the main difficulty is that control states were used to separate the check, backtrack and simulation phases of the model. Here we use the $(a, push_j, b)$ rules so that, when pushing, the automaton can read a character a, and mark it $\underline{a}$ in the next applied rule. Hence the system knows when it is moving up or down the stack, and when it has simulated a stack action. Also the automaton announces the intended phases. For example, the check branch announces "check", removes the work tape, announces "pop_k", pops, announces "check" again and removes the work tape. The formula can then use EX^j to look into the first tape, and $E(tapeU(pop_k \wedge EX(check \wedge EX^j\varphi)))$ to look j steps into the next tape, where *tape* indicates that a tape character is seen. ◀

4.3 Lower Bounds for CTL+

For data complexity, the CTL lower bound transfers to CTL+ and CTL*. For the expression complexity, the following theorem suffices.

▶ **Theorem 12.** *For a fixed order-k HOBPA without collapse, model checking CTL+ is* $(k+1)$*-ExpTime-hard.*

Proof. (sketch) First we adapt the proof of Theorem 10 to show, CTL* is $(k+1)$-ExpTime-hard. We then replace the CTL* formula with a CTL+ formula. Then we show how to fix the system, and restrict ourselves to HOBPA.

For a non-fixed formula and system, our CTL* proof adapts Bozzelli's order-1 proof [4]. Fix a language that is hard for $(k+1)$-ExpTime and an equivalent order-$(k-1)$ alternating HOPDA with an *exponentially* space-bounded work tape. The system proceeds as before, but guesses the length of the work tape and uses a word $bin_n(0)c_0bin_n(1)c_1 \cdots bin_n(2^n - 1)c_{2^n-1}$ to represent it, where $bin_n(i)$ is the n-digit binary representation of i, and c_j are cell contents. The check phase has one branch to check the cell counters are sequential, and the others, instead of just popping down the stack, mark a position in each tape. The formula asserts, when markings are sensible, the tape contents are locally consistent. This can, in fact, be encoded in CTL+ by taking advantage of straight-line parts of the execution and adding extra markings. Obtaining a fixed HOBPA is similar to the CTL case, with some extra tricks. E.g., to ensure each marker is placed once and in the correct order. ◀

5 Linear Time

We consider the linear time logics. We first deal with the upper bound for linear time μ-calculus (μLTL) — and hence LTL — before considering the lower bounds in turn.

5.1 Upper Bounds for μLTL

Since the linear time μ-calculus (μLTL) does not translate polynomially into μ-calculus, we show the k-ExpTime upper bound of model checking μLTL against CPDS separately. Note that μLTL trivially subsumes all other linear time logics considered in this paper.

▶ **Theorem 13.** *Model checking μLTL against order-k CPDSs is in k-ExpTime for a non-fixed formula, and $(k-1)$-ExpTime for a fixed formula.*

Proof. (sketch) We can translate any μLTL formula φ into a Büchi automaton $\mathcal{B}$ at an exponential cost [29]. From a given CPDS $\mathcal{P}$ we construct a product CPDS $\mathcal{P}_B = \mathcal{P} \times \mathcal{B}$ which has a Büchi acceptance condition such that $\mathcal{P}_B$ accepts iff $\mathcal{P}$ does not satisfy φ.

An order-k Büchi CPDS is a CPDS parity game with two colours and only one player. Hence, non-emptiness can be reduced to determining the winner in a parity game, which takes k-ExpTime in the size of the CPDS [17]. Since the Büchi CPDS is exponential in the size of φ, this complexity is too high. The algorithm for an order-k parity game is by a reduction to an order-$(k-1)$ game of exponential size. Because the Büchi CPDS has one player, we can avoid the exponential blow up, constructing an order-$(k-1)$ game of polynomial size. This can be solved in $(k-1)$-ExpTime in the size of the Büchi CPDS, giving an algorithm in k-ExpTime for a non-fixed formula, and $(k-1)$-ExpTime for a fixed formula. ◀

5.2 Lower Bounds for LTL

We first give a matching lower bound for data complexity (fixed formula) of LTL, which already hold for its fragments LTL(F,X) and LTL(U). Since we have previously shown that order-k HOBPA can be analysed in P-time for fixed formulas, it remains to consider HOPDS.

▶ **Theorem 14.** *Model checking HOPDS without collapse against fixed LTL(F, X) and LTL(U) formulas is $(k-1)$-ExpTime-hard.*

Proof. The non-emptiness problem for HOPDS is $(k-1)$-ExpTime-complete [13]. This problem easily reduces to checking the fixed formula $G(\neg f)$, where f holds at all accepting states. Since this formula is both in LTL(F, X) and LTL(U), we are done. ◀

Next we study the expression complexity (fixed system). This is our main result of this section: already for a fixed order-k HOBPA, both LTL(F, X) and LTL(U) are k-ExpTime-hard.

▶ **Theorem 15.** *Model checking LTL(F, X) and LTL(U) against a fixed HOBPA without collapse is k-ExpTime-hard.*

Proof. (sketch) We take a k-ExpTime-hard language $\mathcal{L}$, and, by Theorem 5, its equivalent HOPDS with $s(n)$-bounded space work tape $\mathcal{P}$, for some polynomial $s(n)$. We shall construct a fixed HOBPA $\mathcal{P}'$ such that the language $\mathcal{L}$ is polynomial-time reducible to the LTL(F,X) model checking problem over $\mathcal{P}'$. We can similarly derive the desired lower bound for LTL(U) by "weakly" simulating the next operators with the until operator in the standard way.

We shall now give an intuition of the construction of $\mathcal{P}'$. Our HOBPA $\mathcal{P}'$ is an "over-approximation" of $\mathcal{P}$ in the sense that $\mathcal{P}'$ can do whatever actions $\mathcal{P}$ can do but also more. We will then use LTL(F,X) formulas to enforce correct simulations. This is of course due to the fact that $\mathcal{P}'$ lacks control states and work tape, and its definition should not depend on the input word to $\mathcal{P}$. We shall now elaborate more on how this can be implemented. Given a word $w = \alpha_1 \dots \alpha_n \in \Gamma^*$, we would like to determine if there is an accepting run of $\mathcal{P}$ on w, i.e., a sequence of configurations of the form $\langle p, \gamma, t, j\rangle$ starting with a starting configuration and ending with a final configuration. Here, p is a control state of $\mathcal{P}$, γ is a k-store, $t \in \Delta^{s(n)}$ is a tape content, and $1 \leq j \leq s(n)$ is the position of the tape head. We will represent each such configuration as the topmost symbols of a contiguous sequence of configurations of $\mathcal{P}'$. For example, suppose that the current configuration of $\mathcal{P}$ is $\langle p, \gamma, t, j\rangle$ where $top_1(\gamma) = a$. The HOBPA $\mathcal{P}'$ will start by having (p, a) as its topmost stack symbol. It will then keep modifing its topmost symbol to reflect the tape content t and the position j. This is done by guessing each individual tape cell content from left to right. At some point, $\mathcal{P}'$ will nondetermistically choose some rule of $\mathcal{P}$ to fire. We simulate this by first executing the stack operation and then guess some new state, which we put on top of the stack (this guess is needed because pop operations will destroy control state information). It will then continue by guessing next tape content in the same manner. This process can be repeated indefinitely, unless $\mathcal{P}'$ decides to go to a final (i.e. sink) state, in which $\mathcal{P}'$ will just loop forever. Given an input word $w = \alpha_1 \dots \alpha_n \in \Gamma^*$, we may force a correct simulation of $\mathcal{P}$ on w by $\mathcal{P}'$ using an LTL(F,X) formula. That is, we give a formula φ_w such that $w \in \mathcal{L}(\mathcal{P})$ iff $\mathcal{P}', c_0 \not\models \varphi_w$, where c_0 is an appropriate initial configuration of $\mathcal{P}'$ reflecting the initial state of $\mathcal{P}$. This can be done by first ensuring that each configuration of $\mathcal{P}$ in the simulation as a contiguous sequence of configurations of $\mathcal{P}'$ is valid. In particular, the guessed tape content (reflected by the topmost symbols in this sequence of configurations of $\mathcal{P}'$) must be of length $s(n)$ and has precisely one tape head, which can be easily expressed in LTL(F,X) using a single operator G and nestings of next operators of depth $s(n)$ (approximately). Recall that $s(n)$ is a polynomial function. Using the same technique, we also express that two representations of consecutive configurations of $\mathcal{P}$ in the simulation respect the transition relation of $\mathcal{P}$. Similarly, we enforce the initial configuration in the simulation and that some final configuration of $\mathcal{P}$ is reached. ◀

6 Future Work

There are several avenues of future work. E.g., we have no matching upper bound for the complexity of EF model checking. Walukiewicz has shown the problem to be PSpace-complete at order-1 [31]. However, his techniques do not easily extend to HOPDS owing to the subtleties of higher-order stacks. We may also study simpler logics such as LTL(F).

Acknowledgments. We thank Olivier Serre for interesting discussions and the anonymous referees for their helpful remarks. This work was partly supported by EPSRC (EP/F036361 and EP/E005039), and was done while the second author was a student at the School of Informatics, University of Edinburgh.

References

1 A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, p. 135–150, 1997.

2 A. Bouajjani, P. Habermehl, and R. Mayr. Automatic verification of recursive procedures with one integer parameter. *Theor. Comput. Sci.* 295:85–106 (2003)

3 A. Bouajjani and A. Meyer. Symbolic Reachability Analysis of Higher-Order Context-Free Processes. In *FSTTCS*, p. 135–147, 2004.
4 L. Bozzelli. Complexity results on branching-time pushdown model checking. *Theor. Comput. Sci.*, 379(1-2):286–297, 2007.
5 C. Broadbent and L. Ong. On global model checking trees generated by higher-order recursion schemes. In *FOSSACS*, p. 107–121, 2009.
6 O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In *Handbook of Process Algebra*, Elsevier, 1999.
7 T. Cachat. Higher order pushdown automata, the caucal hierarchy of graphs and parity games. In *ICALP*, p. 556–569, 2003.
8 T. Cachat and I. Walukiewicz. The complexity of games on higher order pushdown automata. *CoRR*, abs/0705.0262, 2007.
9 A. Carayol. Regular sets of higher-order pushdown stacks. In *MFCS*, p. 168–179, 2005.
10 A. Carayol and S. Wöhrle. The caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *FSTTCS*, p. 112–123, 2003.
11 D. Caucal. On infinite terms having a decidable monadic theory. In *Proc. MFCS*, p. 165–176, 2002.
12 E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, p. 995–1072, Elsevier, 1990.
13 J. Engelfriet. Iterated pushdown automata and complexity classes. In *STOC*, p. 365–373, 1983.
14 J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *FoSSaCS*, p. 14–30, 1999.
15 E. M. Gurari. *An Introduction to the Theory of Computation*. W. H. Freeman & Co., New York, NY, USA, 1989.
16 M. Hague. *Global Model Checking Higher Order Pushdown Systems*. PhD thesis, Oxford University, 2009.
17 M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LICS*, p. 452–461, 2008.
18 T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS*, p. 205–222, 2002.
19 T. Knapik, D. Niwinski, P. Urzyczyn, I. Walukiewicz. Unsafe Grammars and Panic Automata. In *ICALP*, p. 1450-1461, 2005.
20 N. Kobayashi. Model-checking higher-order functions. In *PPDP*, p. 25–36, 2009.
21 N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, p. 416–428, 2009.
22 N. Kobayashi and C.-H. L. Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. In *ICALP*, p. 223–234, 2009.
23 N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, p. 179–188, 2009.
24 A. N. Maslov. Multilevel stack automata. *Probl. Inf. Transm.*, 15:1170–1174, 1976.
25 R. Mayr. Strict lower bounds for model checking BPA. *Electr. Notes Theor. Comput. Sci.*, 18, 1998.
26 C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, p. 81–90, 2006.
27 A. Seth. An Alternative Construction in Symbolic Reachability Analysis of Second Order Pushdown Systems. Int. J. Found. Comput. Sci. 19(4): 983-998, 2008.
28 W. Thomas. Constructing Infinite Graphs with a Decidable MSO-Theory. In *MFCS*, p. 113–124, 2003.
29 M. Vardi. A temporal fixpoint calculus. In *POPL*, p. 250-259, 1988.
30 I. Walukiewicz. Pushdown processes: Games and model checking. In *CAV*, p. 62–74, 1996.
31 I. Walukiewicz. Model checking CTL properties of pushdown systems. In *FSTTCS*, p. 127–138, 2000.

A Definition of Collapsible Pushdown Systems

We define order-k stores with links in terms of order-k stores over the infinite alphabet $\Sigma = \{ a^{(i,j)} \mid i,j \in \mathbb{N} \}$. We define the original set of stack operations [17]. We then define the rule $push_{a_1^{i_1} \dots a_m^{i_m} b}$ used in this paper.

The set of operations over an order-k store with links is

$$\begin{aligned} \mathcal{O}_k^c &= \{ push_a^i, rew_a, pop_1 \mid 1 \le i \le k \wedge a \in \Sigma \} \\ &\quad \cup \{ push_l, pop_l, collapse \mid 1 < l \le k \} . \end{aligned}$$

The semantics of the operations are given below, in terms of the standard order-k pushdown operators, and an order-k stack $\gamma = [\gamma_1 \dots \gamma_m]$. Let $\gamma^{<k>}$ be the stack γ where each superscript (i,j) with $i \ge k$ is replaced with $(i, j+1)$.

$$\begin{aligned} pop_1(\gamma) &= push_\varepsilon(\gamma) \\ rew_a &= push_{a^{(i,j)}} && \text{where } top_1(\gamma) = b^{(i,j)} \\ push_a^i(\gamma) &= push_{a^{(i,1)} b_m^{(i',j')}}(\gamma) && \text{where } top_1(\gamma) = b^{(i',j')} \\ collapse(\gamma) &= pop_i^j(\gamma) && \text{where } top_1(\gamma) = b^{(i,j)} \\ push_k[\gamma_1 \dots \gamma_m] &= [\gamma_1^{<k>} \gamma_1 \dots \gamma_m] \\ push_l[\gamma_1 \dots \gamma_m] &= [push_l(\gamma_1)\gamma_2 \dots \gamma_m] && \text{where } l < k \end{aligned}$$

▶ **Definition 16.** An *order-k CPDS* is a tuple $(P, \mathcal{R}, \Sigma, p_0, \bot)$ where P is a finite set of control states, $\mathcal{R} \subseteq P \times \Sigma \times \mathcal{O}_k^c \times P$ is a finite set of rules, Σ is a finite stack alphabet, $p_0 \in P$ is an initial control state and $\bot \in \Sigma$ is a bottom of stack symbol.

A configuration of a CPDS is a pair $\langle p, \gamma \rangle$ where $p \in P$ and γ is a k-store with links. We have a transition $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ iff we have $(p, a, o, p') \in \mathcal{R}$, $top_1(\gamma)$ is a or $a^{(i,j)}$ for some i, j and $\gamma' = o(\gamma)$. The initial configuration is $\langle p_0, [\bot]_k \rangle$.

In this paper we use a slight variation of CPDSs, where

$$\begin{aligned} \mathcal{O}_k^c &= \left\{ push_{a_1^{i_1} \dots a_m^{i_m} b} \mid \forall 1 \le z \le m . 1 \le i_z \le k \wedge a_z \in \Sigma \right\} \\ &\quad \cup \{ push_l, pop_l, collapse \mid 1 < l \le k \} . \end{aligned}$$

This is to emphasize the connection to HOPDS. We have

$$push_{a_1^{i_1} \dots a_m^{i_m} b} = rew_b ; push_{a_m}^{i_m} ; \dots ; push_{a_1}^{i_1}$$

This can easily be simulated by adding (a polynomial number of) intermediate states.

Finally, we observe that a collapsible HOBPA can be simulated by a CPDS of the original definition with a fixed number of states and a polynomially expanded alphabet. To do so we expand the alphabet to two kinds of tuples

$$HOPush(a, j, b) \text{ and } CPush(a, a_1^{i_1}, \dots, a_m^{i_m})$$

for all prefixes $a_1^{i_1}, \dots, a_m^{i_m}$ of some collapsible push rule. For both tuples, a is the top_1 character of the stack being simulated. We write $\langle a \rangle$ to denote any character of this form, or simply of the form a.

The first tuple $HOPush(a, j, b)$ indicates the current top_1 character is a, a $push_j$ is to be performed, then the top_1 character is to be replaced by b. We replace each rule $(a, (b, push_j, c))$ with the rules $(q_{norm}, \langle a \rangle, push_{HOPush(b,j,c)}, q_{push})$ for all $j' \in \{1, \dots, k\}$ and $a \in \Sigma$, followed by the rule $(q_{push}, HOPush(b, j, c), push_j, q_{pushed})$ and $(q_{pushed}, HOPush(b, j, c), push_c, q_{norm})$.

The second type of tuples $CPush(a, a_1^{i_1}, \ldots, a_m^{i_m})$ are used to simulate collapsible push rules. That is, we still need to perform $push_{a_1}^{i_1}; \ldots; push_{a_m}^{i_m}$. Each rule $(a, push_{a_1^{i_1} \ldots a_m^{i_m} b})$ is replaced by $(q_{norm}, \langle a \rangle, rew_{CPush(b, a_1^{i_1}, \ldots, a_m^{i_m}), q_{cpush}}$. Then we add the rule

$$(q_{cpush}, CPush(b, a_1^{i_1}, \ldots a_m^{i_m}), push^{i_1}_{CPush(a_m, a_1^{i_1}, \ldots, a_{(m-1)}^{i_{(m-1)}})}, q_{cpush})$$

whenever $m > 0$ for all prefixes of collapsible push rules. Finally we have the rule

$$(q_{cpush}, CPush(b), rew(b), q_{norm}) \ .$$

The remaining rules (a, o) are simply replaced by rules of the form $(q_{norm}, \langle a \rangle, o, q_{norm})$.

B Proofs for Branching Time

B.1 Proof of Theorem 8

Theorem 8. *Model checking EF over order-k PDS without collapse is* $(k-1)$*-ExpSpace-hard, even for a fixed formula.*

We will now elaborate our construction of the fixed EF formulas and the order-k PDSs in more details. Following [8], we could construct in polynomial-time an order-k PDS $\mathcal{P}$ over the stack alphabet $\Sigma_{\leq k+1}$ with the following pairs (s, ψ) of control states of $\mathcal{P}$ and EF formulas:

- $(counter_k, Counter_k)$ such that $(\mathcal{P}, \langle counter_k, \gamma \rangle) \models Counter_k$ iff $top_2(\gamma)$ ends with a k-counter.
- $(first_k, First_k)$ such that $(\mathcal{P}, \langle first_k, \gamma \rangle) \models First_k$ iff $top_2(\gamma)$ ends with a k-counter representing the number 0.
- $(last_k, Last_k)$ such that $(\mathcal{P}, \langle last_k, \gamma \rangle) \models Last_k$ iff $top_2(\gamma)$ ends with a k-counter representing the number $exp_{k-1}(m) - 1$.
- $(equal_k, Equal_k)$ such that $(\mathcal{P}, \langle equal_k, \gamma \rangle) \models Equal_k$ iff $top_2(\gamma)$ ends with two k-counters that have the same value (separated by one letter from Γ_{k+1}).
- $(equal_k^2, Equal_k^2)$ such that $(\mathcal{P}, \langle equal_k^2, \gamma \rangle) \models Equal_k^2$ iff the two topmost order-1 stacks in γ ends with k-counters that have the same value.
- $(succ_k, Succ_k)$ such that $(\mathcal{P}, \langle succ_k, \gamma \rangle) \models Succ_k$ iff $top_2(\gamma)$ ends with two k-counters that are of successive values (the top one is the successor of the lower one).

The EF formulas defined here depend neither on the input word w nor on the Turing machine M. In fact, [8] defines order-k pushdown game arenas instead of EF formulas. However, one can easily check that the description translates directly to pairs of EF formulas and HOPDS as they have a fixed number of rounds (more precisely, the structure of the arena is a tree with no self-loops).

We now describe how to finish the proof. Suppose that M uses $exp_{k-1}(p(n))$ space and t states. Set $m = (\log_2(t) + 1) \times p(n)$. A configuration of M can be thought of as a word from $\{0,1\}^*$ of length $(\log_2(t) + 1) \times (exp_{k-1}(p(n)) - 1)$, where each tape cell of M is now represented using $\log(t) + 1$ bits as: (1) we need one bit to tell whether head is on top of the current cell, (2) we need one bit to store whether the current tape cell has 0 or 1 (wlog we can assume that the tape alphabet of M is $\{0,1\}$), and (3) assuming that the head is on top of the current tape cell, we need $\log_2(t) - 1$ bits to describe which state M is in. In this way, we can represent a configuration of M as a k-counter. Note that the ith cell in a configuration of M corresponds to the contiguous sequence of bits in the number

represented by this k-counter starting from bit position $i \times (\log_2(t)+1)$ to bit position $(i+1)\times(\log_2(t)+1)-1$ (starting from least significant bits). Therefore, a computation sequence of M can be represented by a sequence of k-counters separated by a letter from Γ_{k+1}. The initial state of the HOPDS is $\langle q_{start}, [\bot]_k\rangle$.

We shall describe the proof with games with a fixed number of rounds that can be easily translated to order-k PDS $\mathcal{P}$ and EF formulas ϕ. The players consist of Player 0 (representing EF, EX operators, and $\vee$) and Player 1 (representing AG, AX operators, and $\wedge$). At the beginning, Player 0 will arbitrarily $push_1$ symbols from $\Gamma_{\leq k+1}$ which he claims to represent an accepting computation sequence of M on w. [More precisely, this translates to one state s in $\mathcal{P}$, which performs only $push_1$ and at some point might move to another state s' (from which s is no longer accessible), and a formula ϕ of the form $EF(s' \wedge \psi)$ for some ψ that we will define shortly.] Player 1 now tries to disprove that the guessed stack content represents an accepting computation. The current stack content looks like $[\gamma]_k$, where $\gamma \in \Gamma^+_{\leq k+1}$. Player 1 has several choices. First, he could doubt and try to disprove that γ is a sequence of k-counters separated by a letter from Γ_{k+1}. This can be easily done by repeated applications of pop_1; each time a letter from Γ_{k+1} is seen, Player 1 may use $(counter_k, Counter_k)$. This can be implemented using a single AG operator using $Counter_k$ as a subformula. Second, Player 1 could show that the last configuration is not a final configuration. This could be easily done by checking whether a final state is found in the last configuration (in between the delimiters from Γ_{k+1}). Note that since each cell content is represented as a binary string of length $\log(t)+1$ (the bits are in Σ_k) we will need a finite-state automaton part that keeps a $\log(t)+1$ modulo count of the number of letters from Σ_k that have been seen thus far when performing pop_1 (this can be done in the standard way). Third, Player 1 could show that the first configuration is not an initial state. For this, we will have to make sure that the first configuration in the sequence of configurations guessed by Player 0 does not represent $q_0 w \Box^{exp_{k-1}(p(n))-n}$, where q_0 is the initial state of M. This can be done as follows. We first compute the sequence v of $(\log(t)+1)n$ bits corresponding to $q_0 w$, using a_k (resp. b_k) to represent 0 (resp. 1) and hardwire v into $\mathcal{P}$. The system $\mathcal{P}$ then performs any number of pop_1 operation (making sure that each cell content seen thus far represents $\Box$) and at some point nondeterministically guessing the end (rightmost) position of the string that must represent $q_0 w$, after which $\mathcal{P}$ will ensure that precisely v are seen at the beginning (we simply ignore all letters from Σ_{k-1} and only look at the letter from Σ_k, noting also that letters from Σ_{k+1} signify the end of the configuration). Fourth, Player 1 could show that some configurations are invalid (e.g. by showing that there are two states in them). This can be done in a similar way (by remembering the first time a state is seen in a configuration and making sure that no more is seen until we see a letter from Σ_{k+1}). Fifth, Player 1 could show that there are two consecutive configurations C_j and C_{j+1} in γ such that C_j is not a predecessor of C_{j+1}. To this end, Player 1 keeps performing pop_1, stops, and picks a position in C_{j+1}. Player 1 will then perform a $push_2$ operation and obtain $\langle[\gamma'][\gamma']\rangle$. Player 1 then keeps performing pop_1 operations and we will make sure that exactly one letter from Σ_{k+1} is seen. Once Player 1 picks a position in C_j, Player 0 can try to prove that the positions are incorrect. This can be done by an application of $(equal^2_{k-1}, Equal^2_{k-1})$. Otherwise, $\mathcal{P}$ can remember the corresponding (at most) four cells in C_j and C_{j+1} (using a total of $O(\log(t))$ bits, i.e., we need $2^{O(\log t)} = O(t)$ extra memory in the control states of $\mathcal{P}$) and check that the positions in C_{j+1} follows the corresponding positions in C_j (according to the transition function of M). This can be using pop_1 operations interleaved with exactly one application of pop_2, while remembering the content of eight cells inside the control states of $\mathcal{P}$. This technique was

first used in [1] to encode linear bounded Turing machine as an EF model checking problem (for a fixed formula) over order-1 PDSs. Finally, we can see that the resulting EF-formula is fixed and the size of the resulting HOPDS is polynomial in $|w|$ and $|M|$ (recall also that M is fixed). This completes the proof.

B.2 Proof of Theorem 9

Theorem 9. *For a fixed order-k HOBPA without collapse, model checking EF is* $(k-1)$*-ExpSpace-hard.*

We use the notion of i-counters from the previous proof. Fix positive integers n and $k > 1$. Fix a polynomial function p. For a finite set Q, a (k, Q)-*counter* is a word of the form $\sigma_r l_r \dots \sigma_0 l_0$ where

- each σ_i is a letter in $\Sigma_k \cup Q$,
- each l_i is a $(k-1)$-counter representing the number i, and
- $r = exp_k(p(n)) - 1$.

Note that the notions of k-counters and $(k, \emptyset)$-counters coincide. We shall now define an order 2-BPA $\mathcal{P} := \mathcal{P}_{2,Q} = (\mathcal{R}, \Sigma, c_0)$ that depends only on Q and show that EF model checking over $\mathcal{P}$ is 1-ExpSpace-hard. We will later set Q to be the set of states of a fixed $(k-1)$-ExpSpace-complete Turing machine. We will then argue that the same construction can be generalised to show $(k-1)$-ExpSpace-hard expression complexity of EF model checking over k-BPAs.

At the beginning, our 2-BPA must write a sequence of $(2, Q)$-counters that potentially gives an accepting computation of a fixed 1-ExpSpace-complete Turing machine. Since the definition should not depend on the input word of a fixed Turing machine, we define the languages $L_2 := (\Sigma_1^+(\Sigma_2 \cup Q))^+$ and $L_2' := (L_2\Sigma_3)^+$. Intuitively, L_2 is the smallest "overapproximation" of the notion of $(2, Q)$-counters that does not depend on n. Hence, L_2' is an overapproximation of all sequences of $(2, Q)$-counters separated by a letter from Σ_3. Initially our 2-BPA will be able to guess a word in L_2', i.e., a potential sequence of $(2, Q)$-counters separated by a letter in Σ_3. The starting configuration c_0 is $[(guess, 3)]_k$ where $(guess, 3)$ is a symbol in Σ. We add the following rules to $\mathcal{R}$:

- $((guess, 3), push((guess, 2)\sigma_3))$ for each $\sigma_3 \in \Sigma_3$,
- $((guess, 2), push((guess, 1)\sigma_2))$ for each $\sigma_2 \in \Sigma_2 \cup Q$,
- $((guess, 1), push((guess, 1)\sigma_1))$ for each $\sigma_1 \in \Sigma_1$,
- $((guess, 1), push((guess, 2')\sigma_1))$ for each $\sigma_1 \in \Sigma_1$,
- $((guess, 2'), push((guess, 3))$, and
- $((guess, 2'), pop_1)$ for each $\sigma_3 \in \Sigma_3$.

The last transition marks the end of the guessing stage as we will no longer see stack symbol of the form $(guess, i)$ afterwards. We now introduce several transitions that will allow us to use EF formulas to check the correctness of the guess from the previous stage. For $i = 1, 2, 3$, let $\Sigma_i' := \{a_i', b_i'\}$ and $\Sigma_i'' := \{a_i'', b_i''\}$. Also, let $Q' = \{q' : q \in Q\}$ and $Q'' = \{q'' : q \in Q\}$. Let $Prime := Q' \cup \bigcup_{i=1}^3 \Sigma_i'$ and $DPrime := Q'' \cup \bigcup_{i=1}^3 \Sigma_i''$. Add the following rules:

- For $i = 1, 2, 3$, when $\sigma_i \in \Sigma_i$ is seen, we can nondeterministically choose any of the following four choices:
  - $push_1(\sigma_i')$
  - $push_1(\sigma_i'')$
  - $(\sigma_i', push_2, unmarked_i)$
  - $(\sigma_i'', push_2, marked_i)$
- When $q \in Q$ is seen, we can nondeterministically choose any of the following four choices:

  - $push_1(q')$
  - $push_1(q'')$
  - $(q', push_2, unmarkedQ)$
  - $(q'', push_2, markedQ)$
- For $i = 1, 2, 3$, when we see either $\sigma_i' \in \Sigma_i' \cup Q'$ or $\sigma_i'' \in \Sigma_i'' \cup Q''$, we can only perform pop_2.
- For $i = 1, 2, 3$, when we see either $marked_i$, $unmarked_i$, $marked_Q$, or $unmarked_Q$, we can only perform pop_1.

To get an intuition, let us look at an example. Suppose that the current stack content is $[a_3 b_1 a_1 q b_1 b_1 a_3]_2$. The following is a snapshot of a run:

$$\left|\begin{array}{l} a_3 \\ q \\ a_1 \\ b_1 \\ a_3 \end{array}\right| \rightarrow \left|\begin{array}{l} a_3 \\ q \\ a_1 \\ b_1 \\ a_3 \end{array}\right| \left|\begin{array}{l} marked_3 \\ q \\ a_1 \\ b_1 \\ a_3 \end{array}\right| \rightarrow \left|\begin{array}{l} a_3 \\ q \\ a_1 \\ b_1 \\ a_3 \end{array}\right| \left|\begin{array}{l} \\ q \\ a_1 \\ b_1 \\ a_3 \end{array}\right| \rightarrow \left|\begin{array}{l} a_3 \\ q \\ a_1 \\ b_1 \\ a_3 \end{array}\right| \left|\begin{array}{l} \\ q' \\ a_1 \\ b_1 \\ a_3 \end{array}\right| \left|\begin{array}{l} \\ markedQ \\ a_1 \\ b_1 \\ a_3 \end{array}\right| \rightarrow \left|\begin{array}{l} a_3 \\ q \\ a_1 \\ b_1 \\ a_3 \end{array}\right| \left|\begin{array}{l} \\ q' \\ a_1 \\ b_1 \\ a_3 \end{array}\right| \left|\begin{array}{l} \\ \\ a_1 \\ b_1 \\ a_3 \end{array}\right|$$

In this way, we obtain a stair-like structure from the stack content $[w]_2$ from the initial guessing stage. That is, we "unroll" the content of the topmost 1-store.The following are simple properties that can be established about every reachable configuration c of $\mathcal{P}_{2,Q}$:

(P1) Every 1-store in c has at most one symbol from $(Q' \cup Q'') \cup \bigcup_{i=1}^{3}(\Sigma_i' \cup \Sigma_i'')$, i.e., on the top of the store.

(P2) If $c \models (Q' \cup Q'') \cup \bigcup_{i=1}^{3}(\Sigma_i' \cup \Sigma_i'')$ and $c \rightarrow^* c'$, then $c' \models (Q' \cup Q'') \cup \bigcup_{i=1}^{3}(\Sigma_i' \cup \Sigma_i'')$ too.

We now will define several EF formulas that play the roles of $Counter_i$, $Equal_i$, etc. in the previous proof. These formulas shall depend on the parameter n and the polynomial p. First, we can easily define $Counter_1^{p(n)}$ such that $\mathcal{P}_{2,Q}, c \models Counter_1^{p(n)}$ iff the topmost $p(n)+2$ symbols on the topmost 1-store of c is of the form $\sigma_2 v \gamma_2$ for some $\sigma_2, \gamma_2 \in \Sigma_2 \cup Q$ and some $v \in (\Sigma_1)^{p(n)}$. This can be done using nested EX operators, conjunctions, atomic propositions, and *no* EF operators, since we have to unroll the topmost 1-store up to $p(n)+2$ times. Similarly, we can define $First_1^{p(n)}$ (resp. $Last_1^{p(n)}$) such that $\mathcal{P}_{2,Q} \models First_1^{p(n)}$ (resp. $\mathcal{P}_{2,Q} \models Last_1^{p(n)}$) iff the topmost $p(n)+2$ symbols on the topmost 1-store of c is $\sigma_2 v \sigma_2'$ for $\sigma_2, \sigma_2' \in \Sigma_2 \cup Q$ and $v = a_1^{p(n)}$ (resp. $v = b_1^{p(n)}$).

We now define the formula $Equal_1^{p(n)}$ such that $\mathcal{P}_{2,Q}, c \models Equal_1^{p(n)}$ iff the topmost 1-store of c has a prefix of the form $\sigma_2 v \gamma_2 v \beta_2$such that $\sigma_2, \gamma_2, \beta_2 \in \Sigma_2 \cup Q$ and $v \in (\Sigma_1)^{p(n)}$. That is, it has two equal 1-counters on the top. To define $Equal_1^{p(n)}$, we will first define a formula ψ such that $\mathcal{P}_{2,Q}, c \models \psi$ iff the topmost 1-store of c has prefix of the form $\sigma_2 v \gamma_2 u \beta_2$ such that $\sigma_2, \gamma_2, \beta_2 \in \Sigma_2 \cup Q$ and $v, u \in (\Sigma_1)^{p(n)}$ but not necessarily $v = u$. This can be defined in a similar way as $Counter_1^{p(n)}$. Assuming now that c satisfies ψ (i.e. that the topmost 1-store of c is $\sigma_2 v \gamma_2 u \beta_2$ as above), we now define the formula ψ' such that $\mathcal{P}_{2,Q} \models \psi'$ iff $u = v$. The formula is

$$\bigwedge_{i=1}^{p(n)} (EX)^{2i} \left(\bigvee_{\sigma_1 \in \Sigma_1} \left(\sigma_1 \wedge (EX)^{2(p(n)+1)} \sigma_1 \right) \right)$$

where $(EX)^i$ is simply a nesting of i EX operators. Notice that when a configuration satisfies Σ_1, Σ_2, or Σ_3 we may still continue unrolling. Notice also that we use $2i$ instead of i because we have "intermediate" configurations (i.e. stacks whose topmost 1-store is of the form $[marked_i ...]$ or $[unmarked_i ...]$) that are needed for the purpose of checking.

Similar to the definition of $Equal_1^{p(n)}$, we can also define an EX formula $Succ_1^{p(n)}$ such that $\mathcal{P}_{2,Q}, c \models Succ_1^{p(n)}$ iff the topmost 1-store of c has prefix of the form $\sigma_2 v \gamma_2 u \gamma_2$ such that $\sigma_2, \gamma_2, \beta_2 \in \Sigma_2 \cup Q$, $v, u \in (\Sigma_1)^{p(n)}$, and the number represented by v is a successor of the number represented by v' (e.g. $v = a_1 a_1 a_1 b_1$ and $v' = a_1 a_1 b_1 a_1$).

We now define a formula $Counter_2^{p(n)}$ such that, for any *reachable* configuration c of $\mathcal{P}_{2,Q}$ (from c_0), we have $\mathcal{P}_{2,Q}, c \models Counter_2^{p(n)}$ iff the two following conditions are satisfied:

(C1) the topmost 1-store of c has a prefix of the form $\sigma_3 v \gamma_3$ where $\sigma_3, \gamma_3 \in \Sigma_3$ and v is a $(2, Q)$-counter.

(C2) each 1-store in c, except for the topmost, has some symbol from $Prime$ (but not $DPrime$) as a topmost symbol.

It is easy to check that a reachable configuration c satisfying $top_1(c) \in \Sigma_3$ must have a prefix that is in the language $top_1(c).L_2$ in its topmost 1-store. The second condition is introduced due to the fact that HOBPA lack control states, but that we will have to remember that exactly one symbol in Σ_3 is seen when we try to pinpoint parts of the hypothetical 2-counter that might be invalid. On the other hand, using EF logic, we must somehow use the EF operator here as in the worst case we may have to pop_1 exponentially many symbols to check the first condition. We start by defining a formula $Unmarked$ checking the second condition:

$$Unmarked := \left(\bigcup_{i=1}^{3} \Sigma_i \cup Q \right) \wedge EX[Prime]AGPrime.$$

where $EX[S](\phi)$ abbreviates $EX(\left(\bigvee_{a \in S} a\right) \wedge \phi)$ for any subset S of atomic propositions. It is easy to check that $Unmarked$ defines the desired formula owing to properties **(P1)** and **(P2)**. We shall now define the formula $Counter_2^{p(n)}$ checking the first condition, *assuming that the second condition is satisfied*. This formula will be the formula $\Sigma_3 \wedge EX[marked_3]EX[\Sigma_1](\theta)$, where θ is a conjunction of the following formulas:

- $Last_1$, which we already defined. This is because we expect to see a 1-counter with value $exp_1(p(n)) - 1$ on the top of the topmost 1-store.
- $ValidCounter_2$, which we define as

$$AG\left(\left(Counter_1^{p(n)} \wedge Check_2\right) \rightarrow \left(\left(First_1^{p(n)} \wedge BottomCounter_1\right) \vee Succ_1^{p(n)}\right)\right).$$

This formula tries to make sure that the structure of the hypothetical 2-counter is valid. Intuitively, the formula $ValidCounter_2$ tries to guess two consecutive 1-counters inside the hypothetical 2-counter that might not proceed in a successive fashion, unless it is the bottomost 1-counter inside the hypothetical 2-counter which has to be the 1-counter representing 0. To see this, first notice that we already forced the marking to be applied (i.e. we have operator $EX[marked_3]$ before θ). To this end, we first make sure that top_2 has a prefix that is a 1-counter. The subformula $Check_2$ ensures that the operator AG does not take it to another 2-counter (i.e. passing a symbol from Σ_3):

$$Check_2 := EX[\Sigma_2' \cup Q'](EF\Sigma_3'' \wedge \neg EF(\Sigma_3'' \wedge EXEF\Sigma_3'') \wedge \neg EF(\Sigma_3' \wedge EF\Sigma_3'')).$$

The formula simply says that precisely one 1-store below the topmost satisfies Σ_3'' and this is above any 1-store satisfying Σ_3'. Since we know that initially (i.e. in c) $Unmarked$ is satisfied, we can be sure that the 1-store satisfying Σ_3'' is the one which we applied the marking meaning that the application of AG operator has *not* passed any symbol from Σ_3, i.e., we are still in the same hypothetical 2-counter. Here, $BottomCounter_1$ is an EX formula which makes sure that the guessed 1-counter is the bottomost 1-counter in this

hypothetical 2-counter. This can be defined without EF as we only need to look forward at most $2p(n)+2$ steps using EX and so can be defined in the same way as $Counter_1^{p(n)}$.

We now define the formula $Equal_2^{p(n)}$ such that, for a reachable configuration c, we have $\mathcal{P}_{2,Q}, c \models Equal_2^{p(n)}$ iff the condition **(C2)** above is satisfied and that

(C1') top_2 has a prefix of the form $\sigma_3 v \gamma_3 v \beta_3$, where $\sigma_3, \gamma_3, \beta_3 \in \Sigma_3$ and v a 2-counter.

To define this formula, we first define the formula $2Counter_2$ which expresses that top_2 has a prefix of the form $\sigma_3 v \gamma_3 u \beta_3$, where v and u are (not necessarily the same) 2-counters. This can be done in the same way as we defined $Counter_2^{p(n)}$, i.e., we need to place another marking $\sigma_3'' \in \Sigma_3''$ immediately after the previous 2-counter. Assuming $2Counter_2$ is satisfied, to violate $Equal_2^{p(n)}$, we will locate a position (i.e. 1-counter) inside v and show that the corresponding position in u contains a different symbol. In order to pick a position, we will use the technique of "marking" used for defining $Counter_2^{p(n)}$. Therefore, $Equal_2^{p(n)}$ can be defined as

$$EX[marked_3]EX[\Sigma_1]EF(\theta' \wedge Rightposn \wedge Counter_1^{p(n)} \wedge \neg Last_1^{p(n)}).$$

Here, $Rightposn$ makes sure that we are in the right position:

$$Check_2 \wedge EX[\Sigma_2' \cup Q']AG(\neg(\Sigma_2'' \vee Q'')).$$

This makes sure that we are still inside the current 2-counter v and that we did not mark symbols in $\Sigma_2 \cup Q$ in the process (i.e. not replace them by symbols in $\Sigma_2'' \cup Q''$). We shall now mark the current top_1 symbol and mark the corresponding position in u. Therefore, θ' is the following formula

$$EX[\{marked_2, markedQ\}]EF(\theta'' \wedge Rightposn' \wedge Counter_1^{p(n)} \wedge \neg Last_1^{p(n)}).$$

Here, $Rightposn'$ makes sure that we are in the right position, i.e., it's a conjunction of the following formulas:

- That we are inside u (it is a modification of $Check_2$):

$$EX[\Sigma_2' \cup Q'] \begin{pmatrix} EF\Sigma_3'' \wedge \neg EF(\Sigma_3'' \wedge EXEF\Sigma_3'') \wedge \\ EF(\Sigma_3' \wedge EF\Sigma_3'') \wedge \\ \neg EF(\Sigma_3' \wedge EXEF(\Sigma_3' \wedge EF\Sigma_3'')) \end{pmatrix}.$$

  The first three conjuncts ensure that we have passed the symbol $\gamma_3 \in \Sigma_3$. The last conjunct ensures that we have not passed the conjunct $\beta \in \Sigma_3$. The reasoning is similar to how we define the formula $Check_2$.
- That there is only one 1-store below the topmost 1-store satisfying $\Sigma_2'' \cup Q''$, i.e., the position that we marked in θ':

$$EX[\Sigma_2' \cup Q'](EF(\Sigma_2'' \vee Q'') \wedge \neg EF((\Sigma_2'' \vee Q'') \wedge EXEF(\Sigma_2'' \vee Q''))).$$

Now, the formula θ'' will mark the current position and check that the two marked positions are the same and the corresponding content of the position (i.e. two symbols from $\Sigma_2'' \cup Q''$) are also the same:

$$EX[\Sigma_2'' \cup Q''] \left(Equal_1' \wedge \bigvee_{\alpha'' \in \Sigma_2'' \cup Q''} (\alpha'' \wedge EF\alpha'') \right).$$

Here, the second conjunct tests that the contents are indeed the same. Here, $Equal'_1$ is an easy modification of $Equal_1^{p(n)}$ which tests that the two positions marked (i.e. they are 1-counters) are the same.

Likewise, we can define a formula $Succ_2^{p(n)}$ such that, for a reachable configuration c, we have $\mathcal{P}_{2,Q}, c \models Succ_2^{p(n)}$ iff the condition **(C2)** above is satisfied and that

(C3') top_2 has a prefix of the form $\sigma_3 v \gamma_3 u \beta_3$, where $\sigma_3, \gamma_3, \beta_3 \in \Sigma_3$ and v, u are two successive 2-counters.

Now, we are ready to show that the expression complexity of EF is 1-ExpSpace-hard. Let us take a fixed exponential space bounded, say $exp_1(p(n))$, Turing machine T with states Q. Our fixed 2-BPA is then $\mathcal{P}_{2,Q}$. Combining the technique that we use for establishing lower bounds for data complexity of EF over HOPDS in the previous subsection and the formulas $Counter_i^{p(n)}$, $Last_i^{p(n)}$, $First_i^{p(n)}$, $Equal_i^{p(n)}$ that we just defined, it is not hard to reduce membership of T to model checking EF over $\mathcal{P}_{2,Q}$. There are three extra things that we will have to ensure, which can be done easily:

- The bottomost configuration guessed is an initial configuration. That is, it contains an encoding of $q_0 w$ followed by $exp_1(p(n)) - n$ blank symbols.
- The topmost configuration is an accepting configuration, i.e., it has a final state.
- In between two consecutive symbols from Σ_3 in the initial guess, there is a state symbol, i.e., symbol from Q.

All these can be defined easily in EF using the marking technique that we used to define $Counter_2^{p(n)}$, i.e., we have to mark symbols from Σ_3.

Extensions to order-k BPA

We shall briefly sketch this extension to order-3 (this can be easily extended to the general case in the same manner). The guessing stage can be adapted easily to instead write a sequence of potential $(3, Q)$-counters on the stack by performing only $push_1$ operations as before. The checking stage is also similar. The idea is to create a "stair-of-stairs" structure for the purpose of checking. We shall elaborate more details on this. We will now also use symbols from Σ_4, $\Sigma'_4 := \{a'_4, b'_4\}$, and $\Sigma''_4 := \{a''_4, b''_4\}$. When we see symbols from $\Sigma_1 \cup \Sigma_2$, we have the same choices as in our previous construction. On the other hand, when we see a symbol $\sigma \Sigma_3 \cup \Sigma_4 \cup Q$, we can no longer perform $(\sigma', push_2, unmarked_i)$ and $(\sigma'', push_2, marked_i)$ operations, but instead we can use the operations $(\sigma', push_3, unmarked_i)$ and $(\sigma'', push_3, marked_i)$ (where i is one of $3, 4, Q$ as before). In this way, we are building a 3-store whose 2-stores (except the topmost) displays the projection of a prefix of the guessed configuration $[w]_3$ onto $\Sigma_3 \cup \Sigma_4 \cup Q$ as top_1 symbols. Note that we use Σ_3 for displaying tape contents of the Turing tape, Q as the states of the Turing machine, and Σ_4 as separators of two consecutive configurations. Also, note that each 2-counter encoding the address of a cell position can be doubly exponentially large and are displayed inside each 2-store (in the same way as the previous construction).

B.3 Proof of Theorem 10

Theorem 10. *For a fixed formula, model checking CTL over a given order-k HOPDS without collapse is k-ExpTime-hard.*

Given an alternating order-$(k-1)$ pushdown automaton $\mathcal{P}$ augmented with an $s(n)$-space bounded two-way work tape, we show that the membership problem can be polynomially

reduced to the satisfaction of a fixed CTL formula φ by an order-k pushdown system $\mathcal{P}^w_{\mathcal{P}}$. Here, we take $s(n)$ to be a polynomial.

The reduction is similar to that used by Bozzelli for CTL model checking of pushdown systems [4] and borrows further ideas from Engelfriet [13].

The idea of the reduction is as follows: an order-k stack is used to navigate a computation tree of the order-$(k-1)$ alternating HOPDA in a standard way. To simulate the work tape, at each step, after simulating an operation on the order-$(k-1)$ stack, a sequence of $s(n)$ work tape symbols (with the head position marked) are pushed on to the top order-1 stack. After this is done, the system can either check that the guessed work tape is consistent with the previous tape, or continue the execution. To continue the execution, an order-k push operation saves the current state (for backtracking), then the work tape is erased. This process repeats. When an accepting configuration is seen, or the children of the current node have been fully explored, the automaton backtracks, and checks untested universal branches. The automaton accepts when the empty stack is reached. That is, all paths have been explored, and found to be accepting.

The CTL formula asserts that a path encoding an accepting tree exists, and that after guessing the work tape, the branches checking consistency all accept. The formula is

$$\varphi = E\left((op \wedge AX(check \rightarrow AF good))\, U\, fin\right)$$

where *op* indicates the current path is simulating a tree, and *check* indicates a consistency checking branch. The proposition *good* indicates that the check has been passed, and fin denotes the (successful) completion of the run.

▶ **Definition 17.** Given a word w of length n and an alternating order-$(k-1)$ pushdown automaton $\mathcal{P}$ augmented with an $s(n)$-space bounded two-way work tape, we define the order-k pushdown system $\mathcal{P}^w_{\mathcal{P}}$. We assume without loss of generality that two rules can be applied from each configuration, and which can be referred to as the first and second rules respectively. We also assume the initial state is existential. Finally, let $\square$ denote an empty work tape cell, and $\bot_{\mathcal{P}}$ denote the bottom of stack character for $\mathcal{P}$.

We define $\mathcal{P}^w_{\mathcal{P}}$ to have the following transitions. The set of states is defined implicitly. The initial state is *init*. To initialise the automaton we have

- $(init, \bot, (push_{fin}; push_k; push_w), (continue, 0))$ where w describes the initial work tape. That is, $w = E(\square, p_0)\square^{s(n)-1}\# \bot_{\mathcal{P}}$.

Note that the $push_w$ erases fin from the copy of the stack created by $push_k$.

The main loop of the simulation is given by the *continue* states, which have the following rules. We store the current word position i on the stack (to aid backtracking) before performing a $push_k$ to continue the execution. The clear phase then removes the current work tape from the stack, remembering the important details. We then simulate a move which may update the stack. The branch phase is where the automaton can either check the consistency of the guessed tape, or continue the execution.

- $((continue, i), done, push_{done}, (back, i))$
- $((continue, i), x, (push_{ix}; push_k; pop_1), (clear, i, x))$ for $x \in \{E, A_1, A_2\}$.
- $((clear, i, x), a, pop_1, (clear, i, x))$ for all $a \in \Delta$ and $x \in \{E, A_1, A_2\}$.
- $((clear, i, x)), (a, p), pop_1, (clear, i, x, a, p))$ for all $a, b \in \Delta$ and $p \in P$ and $x \in \{E, A_1, A_2\}$.
- $((clear, i, x, a, p), b, pop_1, (clear, i, x, a, p))$ for all $a, b \in \Delta$ and $p \in P$ and $x \in \{E, A_1, A_2\}$
- $((clear, i, x, a, p), \#, pop_1, (move, i, x, a, p))$ for all $a \in \Delta$ and $p \in P$ and $x \in \{E, A_1, A_2\}$ and p is not accepting or $i \neq |w| + 1$.
- $((clear, i, x, a, p), \#, pop_1, (back, i))$ for all $a \in \Delta$ and $p \in P$ and $x \in \{E, A_1, A_2\}$ and p is accepting and $i = |w| + 1$.

- $((move, i, x, a, p), \gamma, (o; push_{yw}), (branch, r, i))$ for all $a \in \Delta$, $p \in P$, $x \in \{E, A_1, A_2\}$, $\gamma \in \Sigma$. Furthermore when $x = E$, r is any rule applicable at $p, a, \gamma, w(i)$; when $x = A_1$, r is the first transition for $p, a, \gamma, w(i)$; and when $x = A_2$, r is the second transition for $p, a, \gamma, w(i)$. The operation o is the pushdown operation of r and y is E when r moves to an existential state, and A_1 when moving to a universal state. Finally, w is any work tape configuration of length $s(n)$ and with the read head containing state q reached by r. Note that there are an exponential number of w, but we can easily gain all w with a linear number of intermediate states and rules, each guessing the next character of w.
- $((branch, r, i), a, push_a, (continue, j))$ where $j = i$ if r is an ε-transition in the input, or $j = i + 1$ otherwise; and $((branch, r, i), a, push_a, (check, r))$ for all r and a.

The back phase implements the backtracking, and requires the following rules.

- $((back, i), a, pop_k, (back_1, i))$.
- $((back_1, i), fin, push_{fin}, fin)$.
- $((back_1, i), j, pop_1, (back_2, j))$.
- $((back_2, i), E, push_{done}, (continue, i))$.
- $((back_2, i), A_1, push_{A_2}, (continue, i))$.
- $((back_2, i), A_2, push_{done}, (continue, i))$.

Finally, the check phase has a branch for all $0 \leq j \leq s(n)$ which ensures the jth position in the work tape is consistent with the previous work tape, given the rule r has been executed. If there is no previous configuration, the work tape is automatically good.

- $((check, r), x, pop_1, (check, r, j))$ for all $x \in \{E, A1, A2\}$ and $0 \leq j \leq s(n)$.
- $((check, r, j), a, (pop_1)^j, (check_1, r, j))$.
- $((check_1, r, j), a, pop_2, ((check_2, r, j, a))$.
- $((check_2, r, j, a), fin, push_{fin}, good)$.
- $((check_2, r, j, a), x, (pop_1)^{j+1}, (check_3, r, j, a))$ if a is not a pair (b, q), and also the rules $((check_3, r, j, a), b, push_b, good)$ where $b = a$ or $b = (b', q)$ where r reads b' at state q and writes a to the tape.
- $((check_2, r, j, (a, q)), x, (pop_1)^j, (check_L, r, j, (a, q)))$ if r moves left, q is moved to by r, and $((check_L, r, j, (a, q)), (b, q'), pop_1, (check'_L, r, j, (a, q)))$ where q is moved from by r and b is read by r. Finally, we have the following rule $((check'_L, r, j, (a, q)), a, push_a, good)$.
- $((check_2, r, j, (a, q)), x, (pop_1)^{j+1}, (check_R, r, j, (a, q)))$ if r moves right, q is moved to by r, and $((check_R, r, j, (a, q)), a, pop_1, (check'_R, r, j, (a, q)))$. Finally, we have additional rules $((check'_R, r, j, (a, q)), (b, q'), push_{(b,q')}, good)$ where b is read by r and q' is moved from by r.

The atomic proposition *op* is true during the init, continue, clear, move, branch and back phases. The proposition *check* is true during the check phase, *good* true at state *good* and fin at state fin.

▶ **Property** 1. A word w is accepted by $\mathcal{P}$ iff $\mathcal{P}^w_{\mathcal{P}}$ satisfies the fixed formula φ. Furthermore, $\mathcal{P}^w_{\mathcal{P}}$ is polynomial in the size of w and $\mathcal{P}$.

▶ **Corollary 18.** *CTL model checking of order-k pushdown systems is k-ExpTime-hard, even for a fixed formula φ.*

Proof. Take any language $\mathcal{L}$ in $\bigcup_{d>0}$ DTIME$(exp_k(ds(n)))$ for some polynomial $s(n)$. There exists a Turing machine M such that M accepts a word w iff $w \in \mathcal{L}$. From Theorem 5 we know that there is an order-$(k-1)$ alternating pushdown automaton $\mathcal{P}_M$ with an $s(n)$-space auxiliary work tape such that $w \in \mathcal{L}$ iff w is accepted by $\mathcal{P}$. Then, by Property 1, we have, by a polynomial reduction, that $w \in \mathcal{L}$ iff $\mathcal{P}^w_{\mathcal{P}}$ satisfies φ. ◀

B.4 Proof of Theorem 11

Theorem 11. *For a fixed order-k HOBPA without collapse, model checking CTL is k-ExpTime-hard.*

We adapt the reduction for CTL to give a k-ExpTime expression complexity for CTL over HOBPA.

▶ **Definition 19.** Given an alternating order-$(k-1)$ pushdown automaton $\mathcal{P}$ augmented with an linear-space bounded two-way work tape, we define the order-k pushdown system $\mathcal{P}_\mathcal{P}$. We assume without loss of generality that two rules can be applied from each configuration, and which can be referred to as the first and second rules respectively. We also assume the initial state is existential. Since there is only one control state, we write (a, o) instead of (q, a, o, q) where o is a pushdown operation. Finally, let $\Box$ denote an empty work tape cell, and $\bot_\mathcal{P}$ denote the bottom of stack character for $\mathcal{P}$.

Note, the format of the tape simulation information on the stack is a string, for example, $E0101ab(c, p)d\#$ to indicate an existential position, and word position 0101 with cell contents $ab(c, p)d$. The structure of the whole stack is an sequence of $(k-1)$ stacks. The top $(k-1)$ stack represents the current configuration being simulated. The next $(k-1)$ stack simply contains, on its top character, a representation of the rule r used to reach the configuration. The $(k-1)$ stack underneath is the previous configuration, and so on.

We define $\mathcal{P}_\mathcal{P}$ to have the following transitions. The set of states is defined implicitly. To initialise the automaton we have the following rules. Since we cannot use a control state, these rules take care of adding work tape and simulation information to the stack in general. We use the formula to assert a correct flow. For every character x, we also use $\underline{x}$ to indicate that the character has been reached via a pop operation.

- $(\bot, (fin; push_k; \bot'))$ and $(\bot', push_{\#\bot_\mathcal{P}})$.
- $(\#, push_{c\underline{\#}})$, $(c, push_{c'\underline{c}})$ where $c, c' \in \Delta \cup (\Delta \times P)$.
- $(c, push_{i\underline{c}})$, $(i, push_{i'\underline{i}})$ where $i, i' \in \{0, 1\}$ and $c \in \Delta \cup (\Delta \times P)$.
- $(i, push_{x\underline{i}})$ where $i \in \{0, 1\}$ and $x \in \{E, A_1, \checkmark\}$.

After writing the work tape we have E, A_1 or $\checkmark$ on the stack, we continue the execution with a $push_k$ or check the contents of the work tape. Alternatively, if we reach $\underline{x}$ for $x \in \{E, A_1, A_2\}$ then we are either checking the tape or backtracking, hence we announce which before continuing appropriately.

- $(x, (\underline{x}; push_k; continue))$ where $x \in \{E, A_1\}$ and $(\underline{A_1}, (\underline{A_2}; push_k; continue))$.
- $(x, push_{(x,check)})$ where $x \in \{E, A_1, A_2, \checkmark\}$.
- $(\underline{x}, push_{check_2})$ where $x \in \{E, A_1, A_2, \checkmark\}$.
- $(x, push_{back})$ where $x \in \{\underline{E}, \underline{A_2}, \checkmark\}$.
- $(back, pop_k)$ and (y, pop_1) for $y \in \{continue, (x, check), check_2\}$.
- $(\underline{x}, pop_1)$ for all $x \neq \#$.
- $(\underline{\#}, push_{move})$, $(\underline{\#}, push_{pop})$.
- (pop, pop_k), $(\underline{r}, pop_k)$ for all rules r.
- $(move, pop_1)$.

To simulate a move we first announce on the stack which move will be executed. Then we copy the stack to remember the move joining the configurations. Then we fire the rule and continue the execution. Let $op(r)$ be the stack operation of the rule r and $top(r)$ be the top of stack character after the rule has fired (which can be obtained from the rule alone), if the rule is a push rule.

- $(\underline{c}, push_{(c,r)})$, $((c,r),(\underline{r}; push_k; (c,r)'))$.
- $((c,r)', (c; op(r); (top(r), done))$, $((c, done), push_{\#)\underline{c}})$ when $op(r)$ is a push rule, and
- $((c,r)', op(r))$, $(a, push_{\#\underline{a}})$ when $op(r)$ is a pop rule and a is a stack character from the simulated HOPDA.

We define propositions to be true when forming part of the top of stack character. For example x will be true when x, $\underline{x}$ or (x,y) are on the top of the stack. We also use op to indicate all states that are not check declarations.

The corresponding CTL formula is defined below.

▶ **Definition 20.** For any w, let $n = |w|$. We define

$$\varphi_w = E\left((op \wedge AX(check \rightarrow \varphi_{good}))\, U fin\right)$$

where φ_{good} is of the form

$$\varphi_{good} = \varphi_{first} \vee (\varphi_{AE} \wedge \varphi_{fmt} \wedge \varphi_i \wedge \varphi_r \wedge \varphi_{tape})$$

where each component formula is defined below. Let $Bin(i, b_m)(j)$ be the jth bit of the binary encoding of i. To ease readability we define $EAtNext(\varphi, \varphi') = E((\neg\varphi)U(\varphi \wedge \varphi'))$. That is, at the next time φ is true, φ' is also true. We also define $ChkNext(\varphi)$ that looks into the next work tape on a check branch and $Rule(r)$ that is used to match the rule r connecting the configurations. That is,

$$\begin{aligned} ChkNext(\varphi) &= EAtNext\left(\#, EX\left(pop \wedge EX^3\left(check_2 \wedge \varphi\right)\right)\right) \\ Rule(r) &= EAtNext\left(\#, EX\left(pop \wedge EXr\right)\right) . \end{aligned}$$

When there is just a single configuration on the stack, we use φ_{first} to ensure it is correct. Note that, given a correct initialisation, φ_{tape} (defined below) ensures the tape format is correct in following configurations.

$$\varphi_{first} = \begin{cases} E \wedge \bigwedge_{j=1}^{b_m} EX^j 0 \wedge EX^{b_m+1}(\square, p_0) \\ \bigwedge_{j=1}^{s(n)} EX^{b_m+j}\square \wedge \\ EX^{b_m+s(n)+1}\# \wedge EX^{b_m+s(n)+4} fin \end{cases} .$$

For the remaining positions φ_{AE} ensures the placement of E, A_1, A_2 and $\checkmark$ is correct. Let φ_x^{to} for $x \in \{E, A_1, A_2, \checkmark\}$ denote the disjunction of all rules of leading to a state of type x.

$$\varphi_{AE} = \bigvee_{x \in \{E, A_1, A_2, \checkmark\}} x \wedge EAtNext\left(\#, EX\left(pop \wedge EX\varphi_x^{to}\right)\right)$$

The formula φ_{fmt} ensures the top_k stack has $yBin(i, b_m)w\#$ as the beginning of the first two top_1 stacks, where $y \in \{E, A_1, A_2\}$ and w is a work tape of length $s(n)$.

$$\begin{aligned} \varphi'_{fmt} &= \left(\bigwedge_{j=1}^{b_m} EX^j(0 \vee 1)\right) \wedge \left(\bigwedge_{j=1}^{s(n)} EX^{b_m+j} \bigvee_{a,(p,a)} (a \vee (p,a))\right) \\ \varphi_{fmt} &= \varphi'_{fmt} \wedge ChkNext(\varphi'_{fmt}) . \end{aligned}$$

The next formula φ_i ensures that the input position counter is consistent with the previous counter. Note that this will ensure, when backtracking to a universal position, the second branch will have the same counter as the first: both must be consistent with the ancestor's counter.

$$\varphi_i = \bigwedge_r Rule(r) \Rightarrow EX\varphi_i^r$$

Furthermore, if r is an ε-transition, we define

$$\begin{array}{rcl} \varphi_i^r & = & \bigvee_{j=0}^{n} Num(j) \wedge ChkNext(EXNum(j)) \\ Num(j) & = & \bigwedge_{j'=0}^{b_m-1} EX^{j'} Bin(j, b_m)(j') \end{array}$$

otherwise

$$\varphi_i^r = \bigvee_{j=1}^{n} \left(\begin{array}{l} Num(j) \wedge \\ ChkNext(EXNum(j-1)) \end{array} \right) .$$

The formula φ_r asserts that the applied rule r matches the claimed input positions. Let $inp(r)$ denote the input character associated with a rule r.

$$\varphi_r \quad = \quad \bigvee_r Rule(r) \Rightarrow \bigvee_{inp(r)=w(j)} ChkNext(EXNum(j)) .$$

Finally, φ_{tape} ensures that the tape contents have been updated accurately. We use, for shorthand, functions $Next_r(\sigma_1, \sigma_2, \sigma_3) = d$ that specifies the contents of the current (jth) cell d with respect to the rule fired r and the contents of the previous configuration's $(j-1)th$, jth and $(j+1)th$ cells σ_1, σ_2 and σ_3 respectively. Let $\#_l$ and $\#_r$ denote, for the benefit of $Next_r(\sigma_1, \sigma_2, \sigma_3)$, the left and right boundaries of the tape respectively.

$$\begin{array}{rcl} \varphi_{tape} & = & \bigvee_r Rule(r) \wedge EX^{b_m} \varphi_{tape}^r \\ \varphi_{tape}^r & = & \varphi_{left}^r \wedge \varphi_{middle}^r \wedge \varphi_{right}^r \\ \varphi_{left}^r & = & \bigvee_{\sigma_1,\sigma_2} \left(\begin{array}{l} Next_r(\#_l, \sigma_1, \sigma_2) \wedge \\ ChkNext(EXb_m(\sigma_2 \wedge EX\sigma_2)) \end{array} \right) \\ \varphi_{middle}^r & = & \bigwedge_{j=1}^{s(n)-1} \bigvee_{\sigma_1,\sigma_2,\sigma_3} \varphi_{\sigma_1,\sigma_2,\sigma_3}^r \\ \varphi_{\sigma_1,\sigma_2,\sigma_3}^r & = & \left(\begin{array}{l} EX^j Next_r(\sigma_1, \sigma_2, \sigma_3) \wedge \\ ChkNext(EX^{b_m+j-1}(\sigma_1 \wedge EX\sigma_2 \wedge EX^2\sigma_3)) \end{array} \right) \\ \varphi_{right}^r & = & \bigvee_{\sigma_1,\sigma_2} \left(\begin{array}{l} EX^{s(n)} Next_r(\sigma_1, \sigma_2, \#_r) \wedge \\ ChkNext(EX^{s(n)-1+b_m}(\sigma_1 \wedge EX\sigma_2)) \end{array} \right) . \end{array}$$

▶ **Property** 2. A word w is accepted by $\mathcal{P}$ iff $\mathcal{P}_\mathcal{P}$ satisfies the formula φ_w. Furthermore, $\mathcal{P}_\mathcal{P}$ and φ_w are polynomial in the size of w and $\mathcal{P}$.

▶ **Corollary 21.** *CTL model checking of fixed order-k HOBPA is k-ExpTime-hard.*

Proof. Take any language $\mathcal{L}$ in $\bigcup_{d>0}$ DTIME$(exp_k(ds(n)))$ for some polynomial $s(n)$. There exists a Turing machine M such that M accepts a word w iff $w \in \mathcal{L}$. From Theorem 5 we know that there is an order-$(k-1)$ alternating pushdown automaton $\mathcal{P}_M$ with an $s(n)$-space auxiliary work tape such that $w \in \mathcal{L}$ iff w is accepted by $\mathcal{P}$. Then, by Property 2, we have, by a polynomial reduction, that $w \in \mathcal{L}$ iff $\mathcal{P}_\mathcal{P}$ satisfies φ_w. ◀

C Proof of Theorem 12

Theorem 12. *For a fixed order-k HOBPA without collapse, model checking CTL+ is* $(k+1)$*-ExpTime-hard.*

We adapt the reduction for CTL to give an $(n+1)$-EXPTIME expression complexity for CTL+ over HOBPA. Will begin with an informal description of the proof for CTL* and explain how to adapt it for CTL+.

The reduction we use to show CTL* is $(k+1)$-ExpTime-hard over order-n pushdown systems is similar to Bozzelli's proof for the order-1 case. In fact, only a simple adjustment

to the HOPDS construction for the k-ExpTime-hard result of CTL has to be made. The main technical steps of the proof are in the definition of the required CTL* formula, which was already given by Bozzelli.

We reduce from an order-$(k-1)$ alternating HOPDA with an exponential space work tape. The order-k PDS we construct manages the alternation in the same manner as the reduction for CTL. The only difference is in the generation of work tapes. For the CTL reduction we could simply output a polynomial number of tape symbols, and check for consistency using only a polynomial number of states. However, the same strategy for an exponential space bounded tape would require an exponential number of states. The trick is to output an arbitrary length tape, and use the power of CTL* to reject any tape configurations that are not exponential.

In more detail, the HOPDS represents tape configurations as words of the form

$$bin_n(0)c_0 bin_n(1)c_1 \cdots bin_n(2^n-1)c_{2^n-1}$$

where $bin_n(i)$ is the n-digit binary representation of i with the least significant bit first. It is beyond the power, however, of a polynomially sized HOPDS to output only sequences that count correctly from 0 to 2^n-1. Hence, the HOPDS generates, nondeterministically, a word of the form

$$\left(\{0,1\}^n (\Delta \cup (\Delta \times \mathcal{Q}))\right)^* \ .$$

One can check whether a configuration of this form is of the correct length as follows: first assert that the first n-digit binary number is 0^n. Then check that every pair $b_i c_i b_{i+1} c_{i+1}$ has $b_i = b_{i+1} + 1$. Finally, the last number must be all 1s.

The checking phase of the HOPDS now proceeds as follows. One branch simply pops the configuration from the stack. This branch allows the CTL formula to check the length of the configuration.

The remaining branches remove the tape configuration from the stack, but nondeterministically mark one position with the proposition $check_1$ (hence there are a number of branches for this step). Next pop_ks are performed to retrieve the previous work tape. Then the HOPDS does the same as before: pops the tape from the stack, nondeterministically marking a position with $check_2$. The CTL* formula can test consistency by checking whether the two positions correspond to the same position in the work tape. If so, the markers $check_1$ and $check_2$ can be used to test the contents of the cell (and it's neighbours) to ensure a valid update has taken place.

To refine the construction to show CTL+ is $(k+1)$-ExpTime-hard over order-k pushdown systems, the key restriction of CTL+ that has to be overcome is that the CTL* formulas may contain nested path operators within a single path quantification. We overcome this adding extra marking information to the HOPDS. For example, we avoid the need to count $(n+2)$ steps ahead when comparing one cell position with the next by adding a $check2pre$ and a $check2post$ marker into the HOPDS. That is, before marking the $check_2$ position, the automaton will place a $check2pre$ on the preceding tape character. Then, after placing $check_2$, $check2post$ will be placed on the next tape character.

In the case of HOBPA, it is not possible for the system to enforce the expected placing of the propositions $check2pre$, $check_2$ and $check2post$. That is, we can use the until operator to ensure that the propositions appear in order $((\neg check_2 U check2pre))$ but cannot enforce that they appear in adjacent cells.

The trick is to encode cell numberings as sequences of tuples (b_j^1, b_j^2) such that the encoded numbers differ by one. That is $b^2 + 1 = b^1$. We can then ensure that the three propositions

are on adjacent cells in the same way we ensured $check_1$ and $check_2$ are on corresponding cells. Recall that all binary numbers are encoded with the least significant digit first.

The cost of this encoding is that our work tape will contain $2^n - 2$ cells, but this is clearly not a problem for the complexity result.

▶ **Definition 22.** Given an alternating order-$(k-1)$ pushdown automaton $\mathcal{P}$ augmented with an 2^n-space bounded two-way work tape, we define the order-k pushdown system $\mathcal{P}_\mathcal{P}$. We assume without loss of generality that two rules can be applied from each configuration, and which can be referred to as the first and second rules respectively. We also assume the initial state is existential. Since there is only one control state, we write (a, o) instead of (q, a, o, q) where o is a pushdown operation. Finally, let $\Box$ denote an empty work tape cell, and $\bot_\mathcal{P}$ denote the bottom of stack character for $\mathcal{P}$.

Note, the format of the simulation information on the stack is, for example,

$$E\vec{0}\vec{1}\vec{0}\vec{1}(0,0)(0,1)(0,0)a(0,1)(0,1)(0,0)b\ldots\#$$

to indicate an existential position, and word position $\vec{0}\vec{1}\vec{0}\vec{1}$ with cell contents $ab\ldots$.

We define $\mathcal{P}_\mathcal{P}$ to have the following transitions. The set of states is defined implicitly. The initial state is $init$. To initialise the automaton we have the following rules. Since we cannot use a control state, these rules take care of adding work tape and simulation information to the stack in general. We use the formula to assert a correct flow.

- $(\bot, (fin; push_k; \bot'))$ and $(\bot', push_{\#)\bot_\mathcal{P}})$.
- $(\#, push_{c\underline{\#}})$, $(c, push_{i\underline{c}})$, where $i \in \{0,1\}^2$ and $c \in \Delta \cup (\Delta \times P)$.
- $(i, push_{i'\underline{i}})$ where $i \in \{0,1\}^2$ and $i' \in \{0,1\}^2 \cup \{\hat{0}, \hat{1}\}^2$.
- $(i, push_{c\underline{i}})$ where $i \in \{\hat{0}, \hat{1}\}^2$ and $c \in \Delta \cup (\Delta \times P)$.
- $(c, push_{i\underline{c}})$, $(i, push_{i'\underline{i}})$ where $i, i' \in \{\vec{0}, \vec{1}\}$ and $c \in \Delta \cup (\Delta \times P)$.
- $(i, push_{x\underline{i}})$ where $i \in \{\vec{0}, \vec{1}\}$ and $x \in \{E, A_1, \checkmark\}$.

After writing the work tape we have E, A_1 or $\checkmark$ on the stack, we continue the execution with a $push_k$ or check the contents of the work tape. Alternatively, if we reach $\underline{x}$ for $x \in \{E, A_1, A_2\}$ then we are either checking the tape or backtracking, hence announce which before continuing appropriately.

- $(x, (\underline{x}; push_k; continue))$ where $x \in \{E, A_1, \checkmark\}$ and $(\underline{A_1}, (\underline{A_2}; push_k; continue))$.
- $(x, push_{(x,check)})$ for $x \in \{E, A_1, A_2, \checkmark, \underline{E}, \underline{A_1}, \underline{A_2}\}$.
- $(x, push_{back})$ for $x \in \{E, A_2, \checkmark\}$.
- $((x, check), push_{(x,check_{fmt}})$, $((x, check), push_{check_{cells}})$ for $x \in \{E, A_1, A_2, \checkmark\}$.
- (y, pop_1) for $y \in \{continue, (x, check_{fmt}), check_{cells}\}$.
- $(back, pop_k)$.
- $(\underline{x}, pop_1)$ for all $x \neq \#$.
- $(\underline{\#}, push_{move})$, $(\underline{\#}, push_{pop})$.
- (pop, pop_k) and $(\underline{r}, pop_k)$ for all rules r.
- $(move, pop_1)$.

To simulate a move we first announce on the stack which move will be executed. Then we copy the stack to remember the move joining the configurations. Then we fire the rule and continue the execution. Let $op(r)$ be the stack operation of the rule r and $top(r)$ be the top of stack character after the rule has fired (which can be obtained from the rule alone), if the rule is a push rule.

- $(\underline{c}, push_{(c,r)})$, $((c, r), (\underline{r}; push_k; (c, r)'))$.
- $((c, r)', (c; op(r); (top(r), done))$, $((c, done), push_{\#)\underline{c}})$ when $op(r)$ is a push rule, and

- $((c,r)', op(r))$, $(a, push_{\#\underline{a}})$ when $op(r)$ is a pop rule and a is a stack character from the simulated HOPDA.

If we announced that we intend the perform a check of the tape contents, then we must mark positions in the tape with $check_1$, $check_2$, $check2pre$ and $check2post$. Hence, these options must be available when popping down the tape.

- $(\underline{x}, push_{(x,c)})$ for $x \in \{\hat{0}, \hat{1}\}$ and $c \in \{check_1, check_2, check2pre, check2post\}$.
- $((x,c), pop_1)$ for $i \in \{1,2\}$, $c \in \{check_1, check_2, check2pre, check2post\}$ and $x \in \{\hat{0}, \hat{1}\}$.

We define propositions to be true when forming part of the top of stack character. For example x will be true when x, $\underline{x}$ or (x,y) are on the top of the stack. We also define i_1 and i_2 for $i \in \{0, 1, \hat{0}, \hat{1}\}$ to allow us to identify the components of a tuple (i_1, i_2).

The corresponding CTL+ formula is defined below. Recall b_m is the number of digits required to encode the length of w in binary.

▶ **Definition 23.** For any w, let $n = |w|$. We define

$$\varphi_w = E\left((op \wedge AX(check \rightarrow \varphi_{good}))\, U\, fin\right)$$

where

$$\varphi_{good} = \varphi_{first} \vee (\varphi_{fmt} \wedge \varphi_{cells}) \ .$$

When there is just a single configuration on the stack, we use φ_{first} to ensure it is correct. Let $EAtNext(\varphi, \varphi') = E(\neg\varphi U \varphi \wedge \varphi')$. Note that the formula is written as if there is a single path. This is not the case because the automaton may choose to mark a cell position. However, due to the strict formatting required by the formula, these paths will fail. The path that does not mark any cells should accept. Let $\varphi_{\square}^{cell}$ accept either $\square$ or characters that are not part of the tape (i.e. the cell numberings).

$$\varphi_{first} = \left\{ \begin{array}{l} AX\left(check_{len} \Rightarrow \left(\begin{array}{l} E \wedge \bigwedge_{j=1}^{b_m} EX^j \vec{0} \wedge \\ EX^{b_m+n+1}(\square \wedge EXE(\varphi_{\neg\square}^{cells} U \#)) \wedge \\ \varphi'_{fmt} \end{array} \right)\right) \wedge \\ EAtNext(\#, EX^2 fin) \end{array} \right.$$

where φ'_{fmt} is defined below, and asserts the tape is of the correct format.

We now define φ_{fmt} which checks that the format of the tape is correct, that the correct rule has been fired, &c., but does not check the cell contents. Let $EX^j_{\varphi}\varphi'$ abbreviate $EX(\varphi \wedge EX(\varphi \wedge \ldots EX(\varphi \wedge \varphi')))$ and $\neg c$ abbreviate the negation of any check propositions. Finally, we define for convenience,

$$ChkNextFmt(\varphi) = EAtNext(\#, EX(pop \wedge EX^4(check_{fmt} \wedge \varphi)))$$

and the required formula is

$$\begin{array}{rcl}\varphi_{fmt} & = & AX\left(check_{fmt} \Rightarrow \varphi'_{fmt} \wedge ChkNextFmt(\varphi'_{fmt}) \wedge \varphi_i \wedge \varphi_r \wedge \varphi_{AE}\right)\\ \varphi'_{fmt} & = & \varphi^i_{fmt} \wedge \varphi^{tape}_{fmt} \wedge \varphi^{len}_{fmt}\end{array}$$

$$\varphi^i_{fmt} = \bigwedge_{j=1}^{b_m} EX^j_{\neg c}(\vec{0} \vee \vec{1})$$

$$\begin{array}{rcl}\varphi^{tape}_{fmt} & = & EX^{b_m}_{\neg c} E\left(\left(b \Rightarrow \varphi^{num}_{fmt}\right) U \#\right)\\ \varphi^{num}_{fmt} & = & \bigwedge_{j=1}^{n} EX^j_{\neg c}(0 \vee 1)\end{array}$$

$$\varphi^{len}_{fmt} = EX^{b_m} EG \left(\begin{array}{l} \left((b \wedge f) \rightarrow EX^1_{\neg c}(0,1) \wedge \bigwedge_{j=2}^{n-1} (EX)^j_{\neg c}(0,0)\right) \wedge \\ \left((b \wedge l) \rightarrow EX^1_{\neg c}(1,0) \wedge \bigwedge_{j=3}^{n-1} EX^j_{\neg c}(1,1)\right) \wedge \\ \left[\begin{array}{l} (b \wedge \neg f) \rightarrow \\ \bigwedge_{l=1}^{2} \bigvee_{j}^{n-1} \left[\begin{array}{l} EX^j_{\neg c}(0_l \wedge EX^{n+2}_{\neg c} 1_l) \wedge \\ \bigwedge_{i>j} EX^i_{\neg c}(1_l \wedge EX^{n+2}_{\neg c} 0_l) \wedge \\ \bigwedge_{i<j} EX^i_{\neg c}(1_c \iff EX^{n+2}_{\neg c} 1_l) \end{array} \right] \end{array} \right] \end{array} \right)$$

and φ_{AE} ensures the placement of E, A_1, A_2 and ✓ is correct. Let φ^{to}_x for $x \in \{E, A_1, A_2, ✓\}$ denote the disjunction of all rules of leading to a state of type x.

$$\varphi_{AE} = \bigvee_{x \in \{E, A_1, A_2, ✓\}} x \wedge EAtNext\left(\#, EX\left(pop \wedge EX\varphi^{to}_x\right)\right)$$

and φ_i checks that the word positions are updated correctly.

$$\begin{array}{rcl}\varphi_i & = & \bigwedge_r Rule(r) \Rightarrow EX\varphi^r_i\end{array}$$

$$\begin{array}{rcl}Num(j) & = & \bigwedge_{j'=0}^{b_m} EX^{j'} Bin(j, b_m)(j')\end{array}$$

and, if r is an ε-transition, then

$$\varphi^r_i = \bigvee_{j=0}^{n} Num(j) \wedge ChkNextFmt(EXNum(j))$$

and otherwise

$$\bigvee_{j=1}^{n} Num(j) \wedge ChkNextFmt(EXNum(j-1))$$

The next formula φ_r ensures that the fired rule is applicable at the current word position. Let $inp(r)$ denote the input character associated with a rule r.

$$\begin{array}{rcl}\varphi_r & = & \bigvee_r Rule(r) \Rightarrow \bigvee_{inp(r)=w(j)} ChkNextFmt(EXNum(j)) \ .\end{array}$$

Finally φ_{cells} takes advantage of the fact that there is no branching while popping binary blocks to avoid nesting temporal operators.

$$\begin{array}{rcl}\varphi_{cells} & = & AX(check_{cells} \rightarrow \varphi'_{cells})\\ \varphi'_{cells} & = & EAtNext(\#, EX^3 fin) \vee AG((check_1 \wedge b) \rightarrow E(\theta_1 \wedge \theta_2))\end{array}$$

where θ_1 asserts that the checks are placed correctly, and θ_2 tests the consistency of the tape. As well as asserting that the propositions are in the right position, θ_1 must also assert

that there is only one copy of each check proposition. We require *check2pre* and *check2post* to hold at the beginning of the binary blocks.

$$\theta_1 = \theta^{pos}_{pre} \wedge \theta^{pos}_{2} \wedge \theta^{pos}_{post} \wedge \theta^{one}_{pre} \wedge \theta^{one}_{2} \wedge \theta^{one}_{post}$$

$$\theta^{pos}_{2} = \bigwedge_{j=0}^{n-1} \begin{pmatrix} EX^j_{\neg_c} 1_2 \Rightarrow F(check_2 \wedge EX^j_{\neg_c} 1_2) \wedge \\ EX^j_{\neg_c} 0_2 \Rightarrow F(check_2 \wedge EX^j_{\neg_c} 0_2) \end{pmatrix}$$

$$\theta^{pos}_{pre} = \bigwedge_{j=0}^{n-1} \begin{pmatrix} \big(F(check2pre \wedge EX^j_{\neg_c} 1_2) \Rightarrow F(check_2 \wedge EX^j_{\neg_c} 1_1)\big) \wedge \\ \big(F(check2pre \wedge EX^j_{\neg_c} 0_2) \Rightarrow F(check_2 \wedge EX^j_{\neg_c} 0_1)\big) \end{pmatrix}$$

$$\theta^{pos}_{post} = \bigwedge_{j=0}^{n-1} \begin{pmatrix} \big(F(check_2 \wedge EX^j_{\neg_c} 1_2) \Rightarrow F(check2post \wedge EX^j_{\neg_c} 1_1)\big) \wedge \\ \big(F(check_2 \wedge EX^j_{\neg_c} 0_2) \Rightarrow F(check2post \wedge EX^j_{\neg_c} 0_1)\big) \end{pmatrix}$$

$$\theta^{one}_{pre} = \bigwedge_{n-1}^{j=0} \neg \big(F(check2pre \wedge EX^j_{\neg_c} 0_1) \wedge F(check2pre \wedge EX^j_{\neg_c} 1_1)\big)$$

$$\theta^{one}_{2} = \bigwedge_{n-1}^{j=0} \neg \big(F(check_2 \wedge EX^j_{\neg_c} 0_1) \wedge F(check_2 \wedge EX^j_{\neg_c} 1_1)\big)$$

$$\theta^{one}_{post} = \bigwedge_{n-1}^{j=0} \neg \big(F(check2post \wedge EX^j_{\neg_c} 0_1) \wedge F(check2post \wedge EX^j_{\neg_c} 1_1)\big)$$

To define θ_2, we use, for shorthand, functions $Next_r(\sigma_1, \sigma_2, \sigma_3) = d$ that specifies the contents of the current (jth) cell d with respect to the rule fired r and the contents of the previous configuration's $(j-1)th$, jth and $(j+1)th$ cells σ_1, σ_2 and σ_3 respectively. Let $\#_l$ and $\#_r$ denote, for the benefit of $Next_r(\sigma_1, \sigma_2, \sigma_3)$, the left and right boundaries of the tape respectively.

$$\theta_2 = \bigvee_r Rule(\#, r) \wedge \big((f \rightarrow \theta^r_f) \wedge ((\neg f \wedge \neg l) \rightarrow \theta^r_i) \wedge (l \rightarrow \theta^r_l)\big)$$

where

$$\theta^r_f = \bigvee_{\sigma_1,\sigma_2} \begin{pmatrix} F\left(check_2 \wedge EX^n \sigma_1\right) \wedge \\ F\left(check2post \wedge EX^n \sigma_2\right) \end{pmatrix} \rightarrow Next_r(\#_l, \sigma_1, \sigma_2)$$

$$\theta^r_i = \bigvee_{\sigma_1,\sigma_2,\sigma_3} \begin{pmatrix} F\left(check2pre \wedge EX^n \sigma_1\right) \wedge \\ F\left(check_2 \wedge EX^n \sigma_2\right) \wedge \\ F\left(check2post \wedge EX^n \sigma_3\right) \end{pmatrix} \rightarrow Next_r(\sigma_1, \sigma_2, \sigma_3)$$

$$\theta^r_l = \bigvee_{\sigma_1,\sigma_2} \begin{pmatrix} F\left(check2pre \wedge EX^n \sigma_1\right) \wedge \\ F\left(check_2 \wedge EX^n \sigma_2\right) \end{pmatrix} \rightarrow Next_r(\sigma_1, \sigma_2, \#_r)\ .$$

▶ **Property** 3. A word w is accepted by $\mathcal{P}$ iff $\mathcal{P}_\mathcal{P}$ satisfies the formula φ_w. Furthermore, $\mathcal{P}_\mathcal{P}$ and φ_w are polynomial in the size of w and $\mathcal{P}$.

▶ **Corollary 24.** *CTL model checking of fixed order-k HOBPA is $(k+1)$-ExpTime-hard.*

Proof. Take any language $\mathcal{L}$ in $\bigcup_{d>0}$ DTIME$(exp_{k+1}(ds(n)))$ for some polynomial $s(n)$. There exists a Turing machine M such that M accepts a word w iff $w \in \mathcal{L}$. From Theorem 5 we know that there is an order-$(k-1)$ alternating pushdown automaton $\mathcal{P}_M$ with an 2^n-space auxiliary work tape such that $w \in \mathcal{L}$ iff w is accepted by $\mathcal{P}$. Then, by Property 3, we have, by a polynomial reduction, that $w \in \mathcal{L}$ iff $\mathcal{P}_\mathcal{P}$ satisfies φ_w. ◀

D Proofs For Linear Time

D.1 Proof of Theorem 13

Theorem 13. *Model checking μLTL against order-k CPDSs is in k-ExpTime for a non-fixed formula, and $(k-1)$-ExpTime for a fixed formula.*

It is well known that any formula of μLTL has a Büchi automaton representation. We form the product of the CPDS and the Büchi automaton corresponding to the negation of the μLTL formula φ. This gives us a Büchi CPDS; that is, a CPDS with a set $\mathcal{F}$ of accepting control states. Thus, model checking reduces to the non-emptiness problem for Büchi CPDSs.

The non-emptiness problem for Büchi CPDSs is a special case of determining the winner in a parity game played over a CPDS. In particular, it is a single-player game with a fixed number (two) of colours. It is known that an order-k game can be reduced to an order-$(k-1)$ game with an exponential blow-up [17]. This leads to a recursive algorithm resulting in a finite-state parity game of k-exponential size, and since known finite-state algorithms are polynomial in the number of states (and exponential in the number of colours), the algorithm runs in k-ExpTime.

We demonstrate, in this simple case, the single-player order-k game can be reduced to a two-player order-$(k-1)$ game with only a polynomial blow-up. This order-$(k-1)$ game is then solved as above, giving, an overall algorithm in $(k-1)$-ExpTime in the size of the Büchi CPDS. That is, $(k-1)$-ExpTime for a fixed formula, and k-ExpTime for a non-fixed formula.

We begin by recalling some definitions.

▶ **Definition 25.** Let $\mathcal{P} = (P, \mathcal{R}, \Sigma, p_0, \bot)$ be an order-k CPDS. Then $\mathcal{G} = (\mathcal{P}, P_E, P_A, \Omega)$ is an order-k CPDS parity game, where $P_E \uplus P_A = P$ is a partitioning of the control states and $\Omega : P \to Col$ is a function assigning to each control state a colour in the set $Col \subset \mathbb{N}$ of colours.

A play Λ of a parity game between two players, Éloïse and Abelard, proceeds as follows. Play begins with the initial configuration $\langle p_0, [\bot]_k \rangle$. If the control state of the current configuration is is P_E, then Éloïse chooses an applicable rule in $\mathcal{R}$. Otherwise Abelard chooses a rule. If a player cannot move, they lose the game. The cycle repeats until a player loses, or, in the case of an infinite play, the winner is determined by the colouring function Ω. If the minimal colour occurring infinitely often is even, Éloïse wins the game. Otherwise Abelard wins.

An important part of the reduction is the notion of *collapse rank-aware*.

▶ **Definition 26** ([17])**.** Consider a partial play Λ in $\mathcal{G}$ ending in a configuration $\langle p, \gamma \rangle$ such that $top_1(\gamma)$ has an k-link. Hence there is in Λ at least one configuration of the form $\langle p', collapse(\gamma) \rangle$ for some $p' \in P$. Then the closest to $\langle p, \gamma \rangle$ is called the *collapse ancestor* of $\langle p, \gamma \rangle$. The *collapse rank* of $\langle p, \gamma \rangle$ is the minimal colour of a state occurring in Λ between the collapse ancestor of $\langle p, \gamma \rangle$ and $\langle p, \gamma \rangle$. Note that these notions are not defined if $top_1(\gamma)$ has a j-link for some $j < k$: indeed it may happen that no configuration of the form $\langle p, collapse(\gamma) \rangle$ was visited in Λ, and therefore the collapse ancestor notion can not be adapted.

▶ **Definition 27** ([17])**.** An k-CPDS equipped with a colouring function is *collapse rank-aware* iff there exists a function $ColRnk : \Sigma \to \mathbb{N}$ such that, when defined, the collapse rank of every configuration $\langle p, \gamma \rangle$ is equal to $ColRnk(top_1(\gamma))$. On other words, the collapse rank is stored in the top_1-element of the stack.

Beginning with a Büchi CPDS, we must first make the CPDS collapse rank-aware. This can be done using the following known results. Note that, since both the size of Col and k are fixed, the transformation to a collapse rank-aware CPDS costs only a constant blow up.

▶ **Lemma 28** ([17], Lemma 14, Remark 6.6)**.** *For any order-k CPDS $\mathcal{P}$ and parity game $\mathcal{G}$ played over it, one can construct a collapse rank-aware order-k CPDS $\mathcal{P}'$ and order-k parity*

game $\mathcal{G}'$ such that Éloïse wins in $\mathcal{G}$ iff she wins in $\mathcal{G}'$. Moreover, $\mathcal{P}'$ has $\mathcal{O}(|P||Col|^{\mathcal{O}(k)})$ control states and a stack alphabet of size $\mathcal{O}(|\Sigma||Col|^{\mathcal{O}(k)})$.

We are now ready to give the main reduction.

▶ **Lemma 29.** *Given an order-k Büchi CPDS $\mathcal{P}$, one can construct an order-$(k-1)$ CPDS parity game $\mathcal{G}$ of size polynomial in the size of $\mathcal{P}$ such that $\mathcal{P}$ has an accepting run iff Éloïse wins the game $\mathcal{G}$.*

Proof. We will describe $\mathcal{G}$ somewhat informally. The formal definition should be clear from the description. Control states of the game are of the form (p, p_r, f) for control states p and p_r and the boolean flag f. We also allow $p_r = null$. Intuitively, p is the current control state of the configuration being simulated, p_r is a promise to reach the control state p_r when a pop_k, and f is an obligation to visit a final state before the pop_k. Similarly, the stack characters are either simply characters a, or, when simulating a character with a k-link, of the form (a, p_r, f), which play the same role as (p, p_r, f) does, but in the case of a *collapse* over a k-link.

We describe how each rule application is simulated. Except when stated, Éloïse makes all moves. Consider the rule (p, a, o, p'). In cases:

- When o is a pushdown command that is not $push_k$, pop_k, $push_a^k$, or collapse when the top_1 character contains a k-link, the simulation is direct: o is simply applied and we reach the control state (p', p_r, f') where f' is true iff f is true and p was not final.
- When o is pop_k, play proceeds to a winning (sink) state for Éloïse if $p' = p_r$ and f is false. Otherwise we reach a winning state for Abelard. The stack is not changed.
- When o is *collapse* and the top_1 character is (p, p_r, f), play proceeds to a winning (sink) state for Éloïse if $p' = p_r$ and $ColRnk(p, p_r, f)$ indicates a final state has been seen if f is true. Otherwise we reach a winning state for Abelard. The stack is not changed.
- When o is $push_k$, Éloïse declares a pair p'_r and f', indicating that, when the order-$(k-1)$ stack to be created is finally popped (if it is popped), then play will reach the state p'_r, and f' indicates whether a final state will have been seen on the way. Abelard can accept the declaration and move play to (p'_r, p_r, f'') without changing the stack, where f'' is true only if f is true and f' is not. Alternatively, Abelard may question the declaration and move play to (p', p'_r, f') (without changing the stack).
- When o is $push_a^k$ we proceed as follows. We apply $push^1_{(a,p_r,f)}$ and move to (p', p_r, f). The idea is that Éloïse's commitments are saved to the stack in place of the k-link. If following the collapse link is simulated, it must be in line with her previous declarations.

The final states are all control states where the first component p is final in the original game. It is also the case that, when Abelard accepts a declaration by Éloïse that she can pass through a final state, we ensure that, in the simulation, an intermediate final state is passed through. The initial state is $(p, null, false)$ and stack bottom is $\perp$.

Observe that, since there are no order-k stack operations in the simulation, the resulting game is order-$(k-1)$.

To see why the simulation is correct, first consider an accepting run of the Büchi CPDS. At each $push_k$, then either the current stack contents appear again on the run, in which case Éloïse simply declares the control state at this return, and whether a final state was encountered en-route. Otherwise, she declares *null* and play proceeds to simulate the $push_k$. Since the stack contents below the new top stack are never encountered again, they can be safely forgotten.

In the other direction, take a winning strategy for Éloïse. This strategy can be linearised, constructing a run as follows: at each $push_k$, play has two branches. On the first, Abelard

questions Éloïse's declaration, on the second, he accepts it. We recursively compute the run from the first branch and append to it the run from the second branch. This leads to an infinite run with a infinite number of final states seen. ◀

D.2 Proof of Theorem 15

Theorem 15. *Model checking LTL(F, X) and LTL(U) against a fixed HOBPA without collapse is k-ExpTime-hard.*

We give proofs for the two logics in turn.

D.2.1 Lower Bound for LTL(F, X)

Given a fixed k-EXPTIME complete language $\mathcal{L}$, Theorem 5 implies that there exists a fixed order-k pushdown automaton $\mathcal{P} = (P, \mathcal{R}, \Sigma, \Gamma \cup \{\varepsilon\}, \Delta, p_0, \bot, \mathcal{F})$ augmented with an $s(n)$-space bounded two-way work tape that accepts precisely the language $\mathcal{L}$ from $\langle p_0, [\bot]_k \rangle$. We now construct an order-k BPA $\mathcal{P}' = (\mathcal{R}', \Sigma, c_0')$ and later show that there is a reduction of the membership problem for $\mathcal{L}(\mathcal{P})$ to LTL model checking over $\mathcal{P}'$, which shall imply that expression complexity of order-k BPA is k-ExpTime-hard.

Let $\Delta' = \Delta \times \{h, \bar{h}\}$ (h stands whether head is in this position or not). For an alphabet Ω, we let $\Omega_? = \Omega \cup \{?\}$ assuming that $? \notin \Omega$. Similarly, we let $\Omega_\#$ be $\Omega \cup \{\#\}$ assuming that $\# \notin \Omega$. Let $\mathcal{W} = \mathcal{R} \cup \{sink\}$. Let $\Sigma' = (P_? \times \mathcal{W}_? \times \Sigma) \cup \Sigma \cup (\Sigma \times \Delta')$. The initial configuration c_0' is $\langle [(p_0, ?, \bot)]_k \rangle$. We now add the following rules to $\mathcal{R}'$:

1. for each $p \in P$, $a \in \Sigma$, and $r \in \mathcal{R}$ where r is of the form $((p, *, a, *), o, *)$[1], add

   $$((p, ?, a), push(p, r, a)).$$

   and add

   $$((p, r, a), f_a(o)).$$

   where $f_a : \mathcal{O}_k \to \mathcal{O}_k$ is defined as $f_a(o') = o'$, if $o' \in \mathcal{O}_1$ or of the form pop_j, and $f_a(o') = (push_a; push_j; push_a)$, if $o' = push_j$ for some $j > 1$.
2. for each $a \in \Sigma$ and $x \in \Delta'$, add $(a, push((a, x)))$.
3. for each $x, y \in \Delta'$ and $a \in \Sigma$, add $((a, x), push((a, y)))$.
4. for each $x \in \Delta'$ and $p \in P$, add $((a, x), push((p, ?, a)))$.
5. we add "sink" transitions: $((p, ?, a), push(p, sink, a))$ and $((p, sink, a), push(p, sink, a))$ for any $p \in \mathcal{F}$ and $a \in \Sigma$.

Intuitively, each run of $\mathcal{P}'$ will have valuations of the form

$$[(P \times \{?\} \times \Sigma) \cdot (P \times \mathcal{R} \times \Sigma) \cdot \Sigma \cdot (\Sigma \times \Delta')^+]^* \cdot (\mathcal{F} \times \{sink\} \times \Sigma)^* \qquad (*)$$

or of the form

$$[(P \times \{?\} \times \Sigma) \cdot (P \times \mathcal{R} \times \Sigma) \cdot \Sigma \cdot (\Sigma \times \Delta')^+]^\omega. \qquad (**)$$

We will now enforce the correct form via LTL formulas.

[1] Here, we use the wildcard $*$ to match a symbol or a tuple of symbols.

Given a word $w = \alpha_1 \dots \alpha_n \in \Gamma^*$, we now construct an LTL formula φ^w using only F and X temporal operators such that $w \in \mathcal{L}(\mathcal{P})$ iff $\mathcal{P}', c'_0 \not\models \neg\phi^w$. We first define a formula ψ that makes sure that precisely the word w is consumed during a guessed run:

$$\psi = \psi_1 \wedge \psi_2 \wedge \psi_3.$$

Let $\mathcal{V}$ be precisely the subset of $P \times \mathcal{R} \times \Sigma$, where the rule component is of the following form $((*, \alpha, *, *), *, *)$, for some $\alpha \in \Gamma$; in other words, these are rules which "consume" an input letter. The formula ψ_1 states that *at least* n letters are consumed in the run; let

$$\begin{aligned} \psi_1^0 &:= True \\ \psi_1^{i+1} &:= XF(\mathcal{V} \wedge \psi_1^i) \\ \psi_1 &:= \psi_1^n. \end{aligned}$$

The formula ψ_2 states that *at most* n letters are consumed in the run:

$$\psi_2 := \neg\psi_1^{n+1}.$$

To define ψ_3, let $\mathcal{V}_\alpha$ be subset of $\mathcal{V}$ where the rule component is the form $((*, \alpha, *, *), *, *)$. The formula ψ_3 states that the ith letter consumed in the run is w_i:

$$\begin{aligned} \psi_3^0 &:= True \\ \psi_3^{i+1} &:= XF(\mathcal{V}_{\alpha_{n-i+1}} \wedge \psi_3^i) \\ \psi_3 &:= \psi_3^n. \end{aligned}$$

We now ensure that after each occurrence of $(P \times \mathcal{R} \times \Sigma)$ in the run (as in (*) and (**)), we see precisely a stack symbol in Σ, followed by precisely $s(n)$ symbols from $\Sigma \times \Delta'$, which are in turn followed by a symbol $(p, ?, a)$ of the "right" form. Notice that this would mean that we may partition each run as a contiguous sequence of blocks of size $m := 3 + s(n)$ according to (*) and (**). More formally, define θ as follows:

$$\theta := \bigvee_{(p,r,a) \in P \times \mathcal{R} \times \Sigma} G((p, r, a) \rightarrow P \times \{?\} \times \Sigma.$$

If r is *not* of the form $((p, *, a, *), *, *)$, then we define $\theta_{(p,r,a)} = False$. Otherwise, letting $r = ((p, *, a, *), o, (*, *, p'))$, we define $\theta_{(p,r,a)}$ as follows:

$$\theta_{(p,r,a)} := (p, r, a) \wedge \left(\bigwedge_{i=1}^{s(n)} X^{i+1} \Sigma \times \Delta' \right) \wedge X^{s(n)+2}(p', ?, a).$$

Each block of size $s(n)$ (note: Σ is always followed by such a block) encoding the tape content needs to have exactly one head:

$$Onehead := G\left(\Sigma \rightarrow \bigvee_{i=1}^{s(n)} \left(Head_i \wedge \bigwedge_{1 \le j \le s(n), j \neq i} \neg Head_j \right) \right),$$

where $Head_j := X^j \Sigma \times (\Delta \times \{h\})$. Furthermore, two consecutive blocks encoding tape contents, each of size $s(n)$, must follow in the correct way:

$$Tapefollow := \bigwedge_{(p,r,a) \in P \times \mathcal{R} \times \Sigma} G((p, r, a) \rightarrow XX(\bigwedge_{i=1}^{s(n)} Follow_{i,r})).$$

Here, $Follow_{i,r}$ means that the ith cell of the subsequent tape content follows from the ith cell of the current tape content using rule r. This formula can be easily defined by noticing that the ith cell of the subsequent tape content can be uniquely defined from cells $i-1, i, i+1$ of the current tape content and rule r. More formally, suppose that $r = ((*,*,*,x),*,(y,d,*))$. We define the formula for the case when $d = l$ and $1 < i < s(n)$ (the remaining cases being similar). In this case, the formula $Follow_{i,r}$ can be defined as a conjunction of four formulas:

- First formula assumes that head is not in cells $i-1, i, i+1$:

$$\bigwedge_{x',y',z' \in \Delta} \left(X^{i-1}(x',\bar{h}) \wedge X^{i}(y',\bar{h}) \wedge X^{i+1}(z',\bar{h})\right) \to X^{i+m}(y',\bar{h}).$$

  in which case the ith cell of the subsequent tape content is the same as the ith content of the current tape content.
- Second formula assumes that head is in cell $i-1$. This case can be defined in the same way as the previous case (ith cell remains the same).
- Third formula assumes that head is in cell i:

$$\left(X^{i}(x,h) \to X^{i+m}(y,\bar{h})\right) \wedge \bigwedge_{z \in \Delta \setminus \{x\}} \left(X^{i}(z,h) \to \mathit{False}\right).$$

  This formula forces that the tape symbol read by the head is x, in which case the ith symbol of the subsequent block becomes $(y,\bar{h})$, i.e., x is rewritten to y and the head is no longer in the ith cell.
- Fourth formula assumes that the head is in cell $i+1$:

$$\bigwedge_{z \in \Delta} \left(X^{i}(z,\bar{h}) \wedge X^{i+1}(x,h)\right) \to X^{i+m}(z,h).$$

  That is, the head moves from $i+1$st cell to ith cell.

The first tape content needs to be initialized to $(\bot,h)(\bot,\bar{h})^{s(n)-1}$, i.e., all blank symbols and the head on the leftmost cell:

$$\mathit{Tapeinit} := X^{3}(\bot,h) \wedge \bigwedge_{i=2}^{s(n)-1} X^{2+i}(\bot,\bar{h}).$$

Finally, we need to define the formula Sink stating that the run needs to eventually end up in a "sink" state;

$$\mathit{Sink} := F(P \times \{\mathit{sink}\} \times \Sigma).$$

The formula φ^w can now be defined as

$$\varphi^w := \psi \wedge \theta \wedge \mathit{Onehead} \wedge \mathit{Tapefollow} \wedge \mathit{Tapeinit} \wedge \mathit{Sink}.$$

It is easy to see that the formula φ^w ensures the existence of a correct run of $\mathcal{P}$ on input w. That is, we have $\mathcal{P}', c_0' \not\models \varphi^w$ iff $w \in \mathcal{L}(\mathcal{P})$. This gives us the following theorem.

▶ **Theorem 30.** *The expression complexity of LTL(F,X) over order-k BPA is k-ExpTime-hard*

D.2.2 Lower Bound for LTL(U)

As in the previous subsection, we shall give a sketch on how to show the same expression complexity for the fragment LTL(U). The basic idea is quite simple: weakly simulate X operators with U operators, while making sure (in the system) that each position in each run alternately satisfy/unsatisfy some new atomic proposition p'. We may start with the order-k BPA $\mathcal{P}'$ that we defined in the previous subsection and replace rule (3) above by the following rule (3'): for $x, y \in \Delta'$ and $a \in \Sigma$, add the rules $((a, x), push(a, !))$ and $((a, !), push(a, y))$ to $\mathcal{R}'$.

We shall now show how to modify some of these subformulas that we previously defined (the idea works in general). We can redefine ψ_1^{i+1} as $(\neg \mathcal{V} U(\mathcal{V} U \psi_1^i))$; the reason we do this being that $\mathcal{V}$ cannot hold true in two consecutive points in each run of $\mathcal{P}'$. Let us now consider how to modify $\theta_{(p,r,a)}$. Notice that due to new rule (3'), $\Sigma \times \Delta'$ and $\Sigma \times \{!\}$ must alternate. This allows us to replace nestings of X operators by nestings of U operators, e.g., $X^2 \Sigma \times \Delta'$ becomes

$$((p, r, a) U (\Sigma \times \Delta' U (\Sigma \times \{!\} U \Sigma \times \Delta'))).$$

In fact, it is easy to check that all uses of X operators can be replaced by U in this way. We therefore have the following theorem.

▶ **Theorem 31.** *The expression complexity of LTL(U) over order-k BPA is k-ExpTime-hard.*

E MSO over Collapsible HOBPA

A order-k collapsible HOBPA defines an MSO structure $(C_k^\Sigma, R, (a)_{a \in \Sigma})$, where C_k^Σ is the set of all order-k stores with links, $R(c_1, c_2)$ holds if there is a transition from configuration c_1 to c_2 and $a(c)$ holds for $a \in \Sigma$ if $top_1(c) = a$.

We show that MSO over this structure is undecidable. We give an order-2 collapsible HOBPA $\mathcal{P}$ such that the infinite half-grid is MSO-interpretable in the configuration graph of $\mathcal{P}$.

Theorem 1. *There exists a fixed collapsible order-2 BPA which generates a graph with an undecidable MSO theory.*

Proof. Let $\mathcal{P}$ be the order-2 collapsible HOBPA with the following commands

$(\bot, push_{x\bot})$ $(x, (x, push_2, x'))$ $(x', push_{a^2x'})$
$(a, (a, push_2, b))$ $(a, push_{col})$ $(a, push_{pop})$ $(a, push_c)$
$(b, push_{a^2b})$
$(c, push_{col})$ $(c, push_{pop})$
$(pop, push_\varepsilon)$
$(col, collapse)$

The configuration graph is shown in Figure 3. Note that transitions to $\cdots$ may represent more than one transition.

An MSO interpretation consists of a formula defining which nodes appear in the interpretation, and formulas defining the predicates of the interpretation. We will define an MSO interpretation that restricts the graph to only the configurations shown in Figure 3 (except the *pop* and x configurations at the bottom of each hanging branch). We will then define two MSO relations $A(x, y)$ and $B(x, y)$ denoting an A- or B-edge from node x to node y.

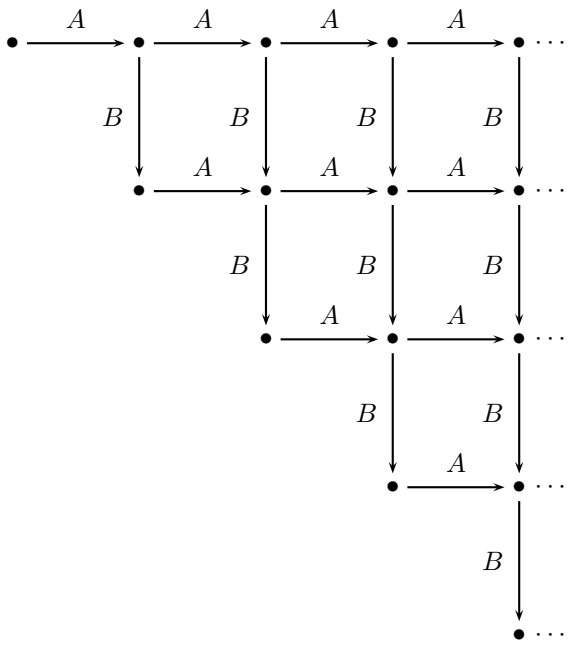

Figure 2 The infinite half-grid.

With the given domain restriction and interpretation of A and B, we obtain the infinite half-grid in Figure 2. Since decidability is preserved by MSO interpretation, we obtain the undecidability of MSO over collapsible HOBPA.

The domain restriction allows any node appearing on a path satisfying the following regular expression r_{dom}. The paths are paths of stack symbols. A path in the graph satisfies this expression if the projection to its top_1 elements does. This expression can be translated to MSO using standard techniques. We define

$$r_{dom} = \bot\ x\ x'\ (a\ b)^*\ c\ (col\ \cup ((pop\ a)^*\ col))\ .$$

Similarly, we define A and B in terms of regular expressions. For $X \in \{A, B\}$, $X(x, y)$ holds if there is a path (which may take some edges backwards) from x to y satisfying the regular expression r_X. We write $\overline{a}b$ to mean that the transition from the node satisfying a to the node satisfying b is a backwards edge. We define

$$\begin{array}{lcl} r_A & = & r_A^1 \cap r_A^2 \\ r_A^1 & = & (\overline{a}\ \overline{pop})^*\ \overline{c}\ a\ b\ a\ c\ (pop\ a)^* \\ r_A^2 & = & (c\ \cup\ a)\ col\ ((\bot\ \ x\ x') \cup (a\ b))\ \overline{a}\ \overline{col}\ (c\ \cup\ a)\ . \end{array}$$

In this definition r_A^1 takes configurations with top_1 character a or c in a hanging branch to any configuration with top_1 character a or c in the next branch along. The expression r_A^2 uses the collapse operations to make sure that the only available node in the next hanging branch is at the same depth. Finally,

$$r_B = (c\ \cup\ a)\ pop\ a$$

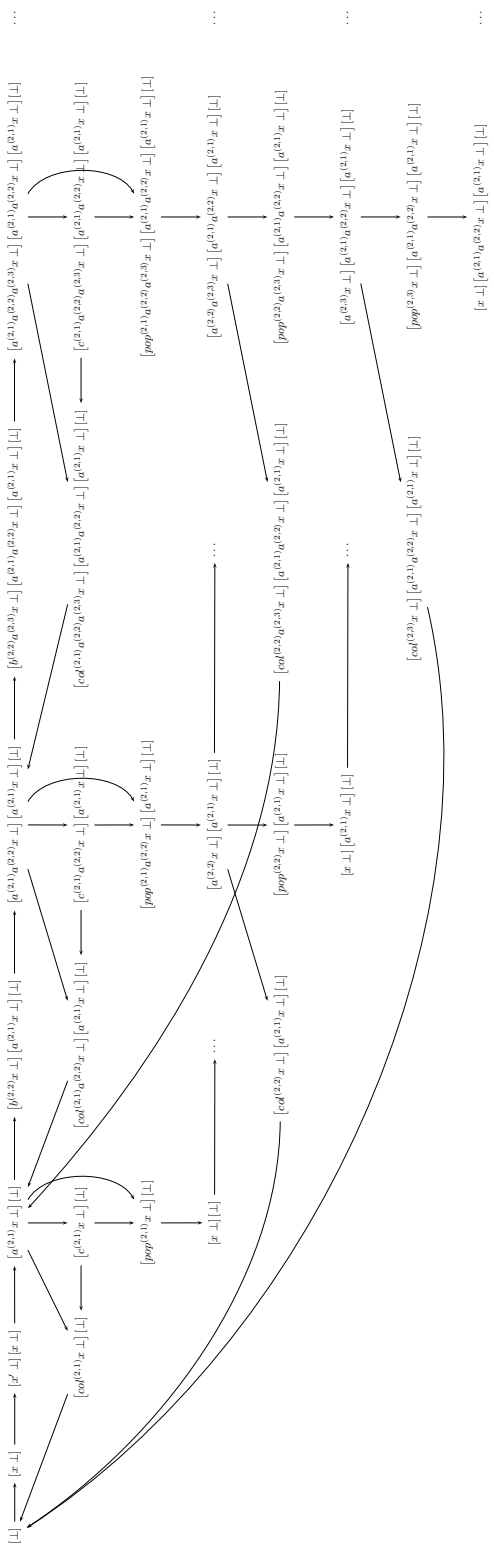

Figure 3 The configuration graph of $\mathcal{P}$.

and, to finish the definition of $B(x, y)$, we add the further constraint that the node x is reachable from the initial configuration via a path of the form $\bot\ \ x\ x'\ a\ (b\ a)^*\ c\ (pop\ a)^*$. The above interpretation gives the infinite half-grid, shown in Figure 2. ◀