Parity to Safety in Polynomial Time for Pushdown and Collapsible Pushdown Systems

Matthew Hague[1]

Royal Holloway, University of London, UK
matthew.hague@rhul.ac.uk
https://orcid.org/0000-0003-4913-3800

Roland Meyer

TU Braunschweig, Germany
roland.meyer@tu-braunschweig.de

Sebastian Muskalla

TU Braunschweig, Germany
s.muskalla@tu-braunschweig.de
https://orcid.org/0000-0001-9195-7323

Martin Zimmermann[2]

Universität des Saarlandes, Saarbrücken, Germany
zimmermann@react.uni-saarland.de

Abstract

We give a direct polynomial-time reduction from parity games played over the configuration graphs of collapsible pushdown systems to safety games played over the same class of graphs. That a polynomial-time reduction would exist was known since both problems are complete for the same complexity class. Coming up with a direct reduction, however, has been an open problem. Our solution to the puzzle brings together a number of techniques for pushdown games and adds three new ones. This work contributes to a recent trend of liveness to safety reductions which allow the advanced state-of-the-art in safety checking to be used for more expressive specifications.

---

1   Introduction

Model-checking games ask whether there is a strategy (or implementation) of a system that can satisfy required properties against an adversary (or environment). They give a natural method for reasoning about systems wrt. popular specification logics such as LTL, CTL, and the μ-calculus. The simplest specifications are reachability or safety properties, where the system either needs to reach a given good state or avoid a bad state (such as a null-pointer dereference). The most expressive logic typically studied is the μ-calculus, which subsumes LTL, CTL, and CTL* [22]. One can reduce μ-calculus model checking in polynomial time to the analysis of parity games (op cit.) via a quite natural product of system and formula.

In the finite-state setting, while reachability and safety games can be solved in linear time and space, the best known algorithms for parity games are quasi-polynomial time [8] or quasi-linear space [18, 13]. For infinite-state games described by pushdown systems, or more generally, collapsible pushdown systems, the complexities match: EXPTIME-complete for solving reachability, safety [5, 32], and parity games [32] over pushdown systems, and n-EXPTIME-complete for order-n collapsible pushdown systems [11, 7, 24, 17].

Pushdown systems are an operational model for programs with (recursive) function calls. In such systems, a configuration has a control state from a finite set and a stack of characters from a finite alphabet (modeling the call stack). Collapsible pushdown systems [17] are an operational model for higher-order recursion as found in most languages (incl. Haskell, JavaScript, Python, C++, Java, . . .). They have a nested stack-of-stacks (e.g. an order-2 stack is a stack of stacks) and *collapse links* which provide access to calling contexts.

Given that safety and parity games over collapsible pushdown systems are complete for the same complexity classes, the problems must be inter-reducible in polynomial-time. However, a direct (without a detour via Turing machines) polynomial-time reduction from parity to safety has been an open problem [14]. To see why the reduction is difficult to find, note that a safety game is lost based on a finite prefix of a play while determining the winner of a parity game requires access to the infinitely many elements of a play. Complexity theory tells us that this gap can be bridged by access to the stack, with only polynomial overhead.

Our contribution is such a polynomial-time reduction from parity to safety. From a theoretical standpoint, it explains the matching complexities despite the difference in expressible properties. From a practical standpoint, it may help building model-checking tools for μ-calculus specifications. Indeed, competitive and highly optimized tools exist for analysing reachability and safety properties of higher-order recursion schemes (HorSat [6, 31, 20] and Preface [28] being the current state-of-the-art), but implementing efficient tools for parity games remains a problem [15, 23]. Having the reduction at hand can allow the use of safety tools for checking parity conditions, suggest the transfer of techniques and optimizations from safety to parity, and inspire new algorithms for parity games. Still, a complexity-theoretic result should only be considered a first step towards practical developments.

Reductions from parity to safety have been explored for the finite-state case by Bernet et al. [1], and for pushdown systems by Fridman and Zimmermann [14]. We will refer to them as counter reductions, as they use counters to track the occurrences of odd ranks. These existing reductions are not polynomial. Berwanger and Doyen [2] showed that counter reductions can be made polynomial in the case of finite-state imperfect-information games.

Our solution to the puzzle brings together a number of techniques for pushdown games and contributes three new ones. We first show how to lift the existing counter reductions [1, 14] from first order to higher orders. For this we exploit a rank-awareness property of collapsible pushdown systems [17]. Secondly, we prove the correctness of this lifting by showing that it

commutes with a reduction from order-n to order-$(n-1)$ games [32, 17]. The polynomial-time reduction is then a compact encoding of the lifted counter reduction. It uses the ability of higher-order stacks to encode large numbers [7] and the insight that rank counters have a stack-like behavior, even in their compact encoding.

Much recent work verifies liveness properties via reductions to safety [25, 10, 26, 27, 12] or reachability [21, 4] with promising results. For finite-state generalized parity games, Sohail and Somenzi show that pre-processing via a safety property can reduce the state space that a full parity algorithm needs to explore, giving competitive synthesis results for LTL [30]. In the case of infinite-state systems (including pushdowns), reductions from liveness (but not parity games) have been explored by Biere et al. [3] and Schuppan and Biere [29].

2 Preliminaries

We define games over collapsible pushdown systems (CPDS). For a full introduction see [17]. CPDS are an operational model of functional programs that is equivalent to higher-order recursion schemes (HORS) [17]. Without collapse, they correspond to *safe* HORS [19].

In the following, let $\mathbb{N}$ be the set of natural numbers (including 0) and $[i, j]$ denote the set $\{i, i + 1, \ldots, j\}$.

2.1 Higher-Order Collapsible Stacks

Higher-order stacks are a nested stack-of-stacks structure whose stack characters are annotated by collapse links that point to a position in the stack. Intuitively, this position is the context in which the character was created. We describe the purpose of collapse links after some basic definitions.

▶ **Definition 2.1** (Order-n Collapsible Stacks). For $n \geq 1$, let Σ be a finite set of stack characters Σ together with a partition function[3] $\lambda : \Sigma \to [1, n]$. An *order-0 stack* with up-to order-n collapse links is an annotated character $a^i \in \Sigma \times \mathbb{N}$. An *order-k stack* with up-to order-n collapse links is a non-empty sequence $w = [w_1 \ldots w_\ell]_k$ (with $\ell > 0$) such that each w_i is an order-$(k-1)$ stack with up-to order-n collapse links. By $Stacks_n$ we denote the set of order-n stacks with up-to order-n links.

In the sequel, we will refer to stacks in $Stacks_n$ as order-n stacks. By *order-k* stack we will mean an order-k stack with up-to order-n links, where n will be clear from the context.

Given an order-k stack with up-to order-n links $w = [w_1 \ldots w_\ell]_k$, we define below the operation $top_{k'}$ to return the topmost element of the topmost order-k' stack. Note that this element is of order-$(k'-1)$. The top of a stack appears leftmost. The operation bot_k^i removes all but the last i elements from the topmost order-k stack. It does not change the order of the stack and requires $i \in [1, \ell]$.

$$
\begin{array}{llll}
top_k(w) & = & w_1 & bot_k^i(w) & = & [w_{\ell-i+1} \ldots w_\ell]_k \\
top_{k'}(w) & = & top_{k'}(w_1) \ (k' < k) & bot_{k'}^i(w) & = & [bot_{k'}^i(w_1) w_2 \ldots w_\ell]_k \ (k' < k).
\end{array}
$$

For technical convenience, we will also define

$$top_{n+1}(w) = w$$

which, we note, does not extend to top_{n+2} or beyond.

---

[3] Readers familiar with CPDS may expect links to be pairs (k, i) and the alphabet Σ not to be partitioned by link order. The partition assumption is oft-used. It is always possible to tag each character with its link order using $\Sigma \times [1, n]$. Such a partition becomes crucial in Section 4.

The destination of a collapse link i on a with $\lambda(a) = k$ in a stack w is $bot_k^i(w)$, when defined. When $i = 0$, the link is considered *null*. We often omit irrelevant collapse links from characters to improve readability.

When u is a $(k-1)$-stack and $v = [v_1 \dots v_\ell]_n$ is an n-stack with $k \in [1, n]$, we define $u :_k v$ as the stack obtained by adding u on top of the topmost k-stack of v. Formally,

$$u :_k v = [uv_1 \dots v_\ell]_n \ (k = n) \qquad \text{and} \qquad u :_k v = [(u :_k v_1)v_2 \dots v_\ell]_n \ (k < n).$$

▶ **Example 2.2.** When $\lambda(a) = 3$ and $\lambda(b) = 2$ let $w = [[[a^1 b^1]_1 [b^1]_1]_2 [[b^0]_1]_2]_3$ be an order-3 collapsible stack. The destination of the topmost link is $bot_3^1(w) = [[[b^0]_1]_2]_3$. Furthermore, $bot_2^1(w) = [[[b^1]_1]_2 [[b^0]_1]_2]_3$ and $top_2(w) = [a^1 b^1]_1$. Here, $top_2(w) :_2 bot_2^1(w) = w$.

Operations on Order-n Collapsible Stacks

CPDS are programs with a finite control acting on collapsible stacks via the operations:

$$\mathcal{O}_n = \{push_2, \dots, push_n\} \cup \{push_a, rew_a \mid a \in \Sigma\} \cup \{pop_1, \dots, pop_n\} \cup \{collapse\}.$$

Operations $push_k$ of order $k > 1$ copy the topmost element of the topmost order-k stack. Order-1 push operations $push_a$ push a onto the topmost order-1 stack and annotate it with an order-$\lambda(a)$ collapse link. When executed on a stack w, the link destination is $pop_{\lambda(a)}(w)$. A pop_k removes the topmost element from the topmost order-k stack. The rewrite rew_a modifies the topmost stack character while maintaining the link (rewrite must respect the link order). Collapse, when executed on a^i with $\lambda(a) = k$, pops the topmost order-k stack down to the last i elements, captured by bot_k^i. Formally, for an order-n stack w:

1. $push_k(w) = top_k(w) :_k w$.
2. $push_a(w) = a^{\ell-1} :_1 w$ when $top_{k+1}(w) = [w_1 \dots w_\ell]_k$, where $k = \lambda(a)$ is the link order,
3. $pop_k(w) = v$ when $w = u :_k v$,
4. $collapse(w) = bot_k^i(w)$ when $top_1(w) = a^i$ and $\lambda(a) = k$, and
5. $rew_b(w) = b^i :_1 v$ when $w = a^i :_1 v$ and $\lambda(a) = \lambda(b)$.

Note that since our definition of stacks does not permit empty stacks, pop_k is undefined if v is empty and *collapse* is undefined when $i = 0$. Thus, the empty stack cannot be reached using CPDS operations; instead, the offending operation will simply be unavailable. Likewise if a rewrite operation would change the order of the link.

▶ **Example 2.3.** Recall Example 2.2 and that $w = [[[a^1 b^1]_1 [b^1]_1]_2 [[b^0]_1]_2]_3$. Given the order-3 link 1 of the topmost stack character a, a collapse operation yields $u = [[[b^0]_1]_2]_3$. Now $push_3(u) = [[[b^0]_1]_2 [[b^0]_1]_2]_3$. A $push_a$ on this stack results in $v = [[[a^1 b^0]_1]_2 [[b^0]_1]_2]_3$. We have $pop_3(v) = u = collapse(v)$.

There is a subtlety in the interplay of collapse links and higher-order pushes. For a $push_k$, links pointing outside of $u = top_k(w)$ have the same destination in both copies of u, while links pointing within u point to different sub-stacks.

▶ Remark (*nop*). For convenience we use an operation *nop* which has no effect on the stack. We can simulate it by rew_a where a is the topmost character (by the format of rules, below, we will always know the topmost character when applying an operation). Hence, it is not a proof case.

2.2 Collapsible Pushdown Systems and Games

▶ **Definition 2.4** (CPDS). An order-n *collapsible pushdown system* is a tuple $\mathcal{C}$ given by $(\mathcal{P}, \Sigma, \lambda, \mathcal{R}, p_I, a_I, \rho)$ with $\mathcal{P}$ a finite set of control states with initial control state p_I, Σ a finite stack alphabet with initial stack character a_I and order function λ, $\rho : \mathcal{P} \to \mathbb{N}$ a function assigning ranks to $\mathcal{P}$, and $\mathcal{R} \subseteq (\mathcal{P} \times \Sigma \times \mathcal{O}_n \times \mathcal{P})$ a set of rules. The size is $|\mathcal{C}| = |\mathcal{P}| + |\Sigma|$. The remaining entries polynomially depend on $\mathcal{P}$ and Σ (note that n is fixed).

We attach ranks to CPDS instead of games as we later need the notion of *rank-aware* CPDS.

A *configuration* of a CPDS is a pair $c = \langle p, w \rangle$ with $p \in \mathcal{P}$ and a stack $w \in Stacks_n$. We have a transition $\langle p, w \rangle \longrightarrow \langle p', w' \rangle$ if there is a rule $(p, a, o, p') \in \mathcal{R}$ with $top_1(w) = a$ and $w' = o(w)$. The initial configuration is $\langle p_I, w_I \rangle$ where $w_I = [\dots [a_I^0]_1 \dots]_n$. To begin from another configuration, one can adjust the CPDS rules to build the required stack from the initial configuration. A computation is a sequence of configurations $c_0, c_1, \dots$ where $c_0 = \langle p_I, w_I \rangle$ and $c_i \longrightarrow c_{i+1}$ for all $i \in \mathbb{N}$. Recall, transitions cannot empty a stack or rewrite the order of a link.

▶ **Definition 2.5** (Games over CPDS). A *game over a CPDS* is a tuple $\mathcal{G} = (\mathcal{C}, \mathbb{O}, \mathcal{W})$, where $\mathcal{C}$ is a CPDS, $\mathbb{O} : \mathcal{P} \to \{A, E\}$ is a division of the control states of $\mathcal{C}$ by owner Elvis (E) or Agnetha (A), and $\mathcal{W} \subseteq \mathbb{N}^\omega$ is a winning condition. The size of the game is $|\mathcal{G}| = |\mathcal{C}|$.

We call $\mathcal{G}$ a *safety game* if $\rho(p) \in \{1, 2\}$ for all $p \in \mathcal{P}$ and $\mathcal{W} = 2^\omega$. It is a *parity game* if $\mathcal{W}$ is the set of all sequences such that the smallest infinitely occurring rank is even.

We refer to computations as plays and require them to be infinite. This means every configuration $\langle p, w \rangle$ has some successor $\langle p', w' \rangle$. This does not lose generality as we can add to the CPDS transitions to a losing (as defined next) sink state (with self-loop) for $\mathbb{O}(p)$ from any configuration $\langle p, w \rangle$. A play $\langle p_0, w_0 \rangle, \langle p_1, w_1 \rangle, \langle p_2, w_2 \rangle, \dots$ is won by Elvis, if its sequence of ranks satisfies the winning condition, i.e. $\rho(p_0)\rho(p_1)\rho(p_2)\dots \in \mathcal{W}$. Otherwise, Agnetha wins. When a play reaches (p, w), then the owner of p chooses the rule to apply. A *strategy* for player $\Upsilon \in \{E, A\}$ is a function $\sigma : (\mathcal{P} \times Stacks_n)^* \to \mathcal{R}$ that returns an appropriate rule based on the prefix of the play seen so far. A play $\langle p_0, w_0 \rangle, \langle p_1, w_1 \rangle, \langle p_2, w_2 \rangle, \dots$ is according to σ if for all i with $\mathbb{O}(p_i) = \Upsilon$ we have $\langle p_i, w_i \rangle \longrightarrow \langle p_{i+1}, w_{i+1} \rangle$ via rule $\sigma(\langle p_0, w_0 \rangle, \dots, \langle p_i, w_i \rangle)$. The strategy is winning if all plays according to σ are won by Υ. We say a player *wins* a game if they have a winning strategy from the initial configuration.

2.3 Rank-Aware Collapsible Pushdown Systems

We will often need to access the smallest rank that was seen in a play since some stack was created. *Rank-aware* CPDS record precisely this information [17]. We first define k-ancestors which, intuitively, give the position in the play where the top order-$(k-1)$ stack was pushed. Note, in the definition below, the integer j is unrelated to the collapse links.

▶ **Definition 2.6** (k-Ancestor). Let $k \in [2, n]$ (resp. $k = 1$). Given a play $c_0, c_1, \dots$ we attach an integer j to every order-$(k-1)$ stack as follows. In c_0 all order-$(k-1)$ stacks are annotated by 0. Suppose c_{i+1} was obtained from c_i using operation $push_k$ (resp. $push_a$). Then the new topmost order-$(k-1)$ stack in c_{i+1} is annotated with i. If c_{i+1} is obtained via a $push_{k''}$ with $k'' > k$, then all annotations on the order-$(k-1)$ stacks in the copied stack are also copied.

The k-*ancestor* with $k \in [1, n]$ of c_i is the configuration c_j where j is the annotation of the topmost order-$(k-1)$ stack in c_i. Let $top_1(c_i) = a^\ell$ and $\lambda(a) = k'$. The *link ancestor* of c_i is the k'-ancestor of the 1-ancestor of c_i.

Applying a pop_k operation will expose (a copy of) the topmost $(k-1)$-stack of the k-ancestor. To understand the notion of a link ancestor, remember that *collapse* executed on a stack whose topmost order-0 stack is a^ℓ with $\lambda(a) = k'$ has the effect of executing $pop_{k'}$ several times. The newly exposed topmost $(k'-1)$-stack is the same that would be exposed if $pop_{k'}$ were applied at the moment the a character was pushed. This exposed stack is the same stack as is topmost on the k'-ancestor of the 1-ancestor of a. We illustrate this with an example.

▶ **Example 2.7.** Assume some c_0. Now take some c_1 containing the stack $w_1 = [[[b^0]_1]_2]_3$. Apply a $push_3$ operation to obtain c_2 with stack $w_2 = [[[b^0]_1]_2[[b^0]_1]_2]_3$. Note, the topmost $[[b^0]_1]_2$ has 3-ancestor c_1.

Now, let $\lambda(a) = 3$ and obtain c_3 with $push_a$, which thus contains the stack $w_3 = [[[a^1b^0]_1]_2[[b^0]_1]_2]_3$ where the a^1 has the 1-ancestor c_2.

We can now apply $push_3$ again to reach c_4 with stack $w_4 = [[[a^1b^0]_1]_2[[a^1b^0]_1]_2[[b^0]_1]_2]_3$. Note that both copies of a^1 have the 1-ancestor c_2. Moreover, the link ancestor of both is c_1. That is, the 3-ancestor of the topmost stack of c_2. In particular, applying *collapse* at c_4 results in a configuration with stack $[[[b^0]_1]_2]_3$, which is the same stack contained in c_1.

In the below, intuitively, the level-k rank is the smallest rank seen since the topmost $(k-1)$ stack was created. Similarly for the link-level rank. Our rank-awareness definition is from [17] but includes level-k ranks as well.

▶ **Definition 2.8** (Level Rank). For a given play $c_0, c_1, \ldots$ the *level-k rank* with $k \in [1, n]$ (resp. *link-level rank*) at a configuration c_i is the smallest rank of a control state in the sequence $c_{j+1}, \ldots, c_i$ where j is the k-ancestor (link ancestor) of c_i.

In the following definition, ℓ is a special symbol to be read as *link*.

▶ **Definition 2.9** (Rank-Aware). A *rank-aware CPDS* is a CPDS $\mathcal{C}$ over stack characters (a, Rk), where a is taken from a finite set and function Rk has type $\mathrm{Rk} : [1, n] \cup \{\ell\} \rightarrow [0, m]$ (with m the highest rank of a state in $\mathcal{C}$). The requirement is that in all computations $c_0, c_1, \ldots$ of the CPDS all configurations $c_i = (p, w)$ with top-of-stack character (a, Rk) satisfy

$$\mathrm{Rk}(k) = \text{ the level-}k\text{ rank at } c_i, \ k \in [1, n], \quad \text{and} \quad \mathrm{Rk}(\ell) = \text{ the link-level rank at } c_i.$$

Below, we slightly generalize a lemma from [17] to include safety games and level-k ranks. Intuitively, we can obtain rank-awareness by keeping track of the required information in the stack characters and control states.

▶ **Lemma 2.10** (Rank-Aware). *Given a parity (resp. safety) game over CPDS $\mathcal{C}$ of order-n, one can construct in polynomial time a rank-aware CPDS $\mathcal{C}'$ of the same order and a parity (resp. safety) game over $\mathcal{C}'$ such that Elvis wins the game over $\mathcal{C}$ iff he wins the game over $\mathcal{C}'$.*

Note, the number of functions Rk is exponential in n. However, since n is fixed the construction is polynomial. In the sequel, we will assume that all CPDS are rank-aware.

3   Main Result and Proof Outline

In the following sections, we define a reduction Poly which takes a parity game $\mathcal{G}$ over a CPDS and returns a safety game $\mathrm{Poly}(\mathcal{G})$ of the same order. The main result follows.

▶ **Theorem 3.1** (From Parity to Safety, Efficient). *Given a parity game $\mathcal{G}$, Elvis wins $\mathcal{G}$ iff he wins $\mathrm{Poly}(\mathcal{G})$. $\mathrm{Poly}(\mathcal{G})$ is polynomially large and computable in time polynomial in the size of $\mathcal{G}$.*

We outline how to define Poly and prove it correct. First, we give a function Counter_B reducing an order-n parity game to an equivalent order-n safety game. It extends Fridman and Zimmermann's reduction [14] from first order to higher orders. In the finite-state setting, a related reduction appeared already in [1]. The idea is to count in the stack characters the occurrences of odd ranks. Elvis has to keep the counter values below B, a threshold that is a parameter of the reduction. For completeness, this threshold has to be n-fold exponential in the size of $\mathcal{G}$. Let $\text{Exp}_0(f) = f$ and $\text{Exp}_n(f) = 2^{\text{Exp}_{n-1}(f)}$. We have the following lemma.

▶ **Lemma 3.2** (From Parity to Safety, Inefficient). *Given a parity game $\mathcal{G}$ played over an order-n CPDS, there is a bound $B(\mathcal{G}) = Exp_n(f(|\mathcal{G}|))$ for some polynomial f so that for all $B \geq B(\mathcal{G})$ Elvis wins $\mathcal{G}$ iff he wins the safety game $\text{Counter}_B(\mathcal{G})$.*

The size of $\text{Counter}_B(\mathcal{G})$ is not polynomial, even for constant B. The next step is to give an efficient reduction Poly_B producing a safety game equivalent to $\text{Counter}_B(\mathcal{G})$. In particular $\text{Poly}_{B(\mathcal{G})}(\mathcal{G})$ can be computed in time polynomial only in the size of $\mathcal{G}$, not in $B(\mathcal{G})$. Thus, we can define Poly from the main theorem to be $\text{Poly}(\mathcal{G}) = \text{Poly}_{B(\mathcal{G})}(\mathcal{G})$.

Technically, Poly relies on the insight that counter increments as performed by Counter_B follow a stack discipline. Incrementing the rth counter resets all counters for $r' > r$ to zero. The upper bound combines this with the fact that collapsible pushdown systems can encode large counters [7]. The second step is summarized as follows.

▶ **Lemma 3.3** (From Inefficient to Efficient). *Elvis wins $\text{Counter}_B(\mathcal{G})$ iff he wins $\text{Poly}_B(\mathcal{G})$. Moreover, $\text{Poly}(\mathcal{G}) = \text{Poly}_{B(\mathcal{G})}(\mathcal{G})$ is polynomial-time computable.*

It should be clear that the above lemmas, once proven, yield the main theorem. For the equivalence stated there, note that $\text{Poly}(\mathcal{G}) = \text{Poly}_{B(\mathcal{G})}(\mathcal{G})$ is equivalent to the game $\text{Counter}_{B(\mathcal{G})}(\mathcal{G})$ by Lemma 3.3. This game, in turn, is equivalent to $\mathcal{G}$ by Lemma 3.2.

The proof of Lemma 3.3 will be direct and is given in Section 7. We explain the proof of Lemma 3.2 here, which relies on a third reduction. We define a function called Order that takes an order-n parity or safety game and produces an equivalent order-$(n-1)$ parity or safety game. The reduction already appears in [17], and generalizes the one from [32]. Let

$$
\begin{array}{llll}
\mathcal{G}_O & = & \text{Order}(\mathcal{G}), & \mathcal{G}_{C_B} & = & \text{Counter}_B(\mathcal{G}), \\
\mathcal{G}_{O,C_B} & = & \text{Counter}_B(\text{Order}(\mathcal{G})), & \mathcal{G}_{C_B,O} & = & \text{Order}(\text{Counter}_B(\mathcal{G})) .
\end{array}
$$

The proof of Lemma 3.2 chases the diagram below. We rely on the observation that the games $\text{Counter}_B(\text{Order}(\mathcal{G}))$ and $\text{Order}(\text{Counter}_B(\mathcal{G}))$ are equivalent, as stated in Lemma 3.4. The proof of Lemma 3.4 needs the reductions and can be found in Section 6. The commutativity argument yields the following proof, almost in category-theoretic style.

$$
\begin{array}{ccc}
\mathcal{G} & \dashrightarrow \text{Counter}_B \dashrightarrow & \mathcal{G}_{C_B} \\
\downarrow & & \downarrow \\
\text{Order} & & \text{Order} \\
\downarrow & & \downarrow \\
\mathcal{G}_O & \xrightarrow{\quad \text{Counter}_B \quad} & \mathcal{G}_{O,C_B} \Longleftrightarrow \mathcal{G}_{C_B,O}
\end{array}
$$

▶ **Lemma 3.4** ($\mathcal{G}_{O,C_B}$ vs. $\mathcal{G}_{C_B,O}$). *Given $B \in \mathbb{N}$ and a parity game $\mathcal{G}$ over an order-n CPDS, Elvis wins $\text{Counter}_B(\text{Order}(\mathcal{G}))$ iff Elvis wins $\text{Order}(\text{Counter}_B(\mathcal{G}))$.*

Proof of Lemma 3.2. We induct on the order. At order-1, the result is due to Fridman and Zimmerman [14]. For the induction, without the bound, at order-n, take a winning strategy for Elvis in $\mathcal{G}$. By [17], he has a winning strategy in $\mathcal{G}_O$. By induction, Elvis has a winning

strategy in $\mathcal{G}_{O,C_B}$ and by Lemma 3.4 also in $\mathcal{G}_{C_B,O}$ when B is suitably large. Finally, again by [17], Elvis can win $\mathcal{G}_{C_B}$. I.e., we chase the diagram above from $\mathcal{G}$ to $\mathcal{G}_O$ to $\mathcal{G}_{O,C_B}$ to $\mathcal{G}_{C_B,O}$ and then up to $\mathcal{G}_{C_B}$. To prove the opposite direction, simply follow the path in reverse.

To obtain the required bound, we argue as follows: Intuitively, we have an exponential bound at order-1 by Fridman and Zimmerman. Thus, assume by induction we have a $(n-1)$-fold exponential bound for order-$(n-1)$. From an order-n system we obtain an exponentially large order-$(n-1)$ system for which an n-fold exponential bound is therefore needed. ◀

In Sections 4 and 5, we define Order and Counter$_B$, and show Lemma 3.4 in Section 6. The reduction Poly is defined in Section 7, which also sketches the proof of Lemma 3.3.

4 Order Reduction

We recall the reduction of [17] from order-n to order-$(n-1)$ parity games. This reduction also works for safety games. It is a natural extension of Carayol et al. [9] for higher-order pushdown systems without collapse, which extended Walukiewicz's reduction of pushdown parity games to finite-state parity games [32]. Due to space constraints, we only give the intuition here. It is useful when explaining the motivation behind the constructions in our parity to safety reduction.

Given an order-n CPDS $\mathcal{C}$ and a game $\mathcal{G} = (\mathcal{C}, \mathbb{O}, \mathcal{W})$ we define an order-$(n-1)$ game Order$(\mathcal{G})$ over a CPDS $\mathcal{C}'$. The order-$(n-1)$ CPDS $\mathcal{C}'$ simulates $\mathcal{C}$. The key operations are $push_n$, pop_n, $push_a$ with $\lambda(a) = n$, and *collapse* when the link is order-n. We say these operations are order-n. The remaining operations are simulated directly on the stack of $\mathcal{C}'$.

There is no $push_n$ on an order-$(n-1)$ stack. Instead, observe that if the stack is w before the $push_n$ operation, it will return to w after the corresponding pop_n (should it occur). Thus, we simulate $push_n$ by splitting the play into two branches. The first simulates the play between the $push_n$ and corresponding pop_n. The second simulates the play after the pop_n.

Instead of applying a $push_n$ operation, Elvis makes a claim about the control states the play may pop to. Also necessary is information about the smallest rank seen in the play to the pop. This claim is recorded as a vector of sets of control states $\vec{P} = (P_0, \ldots, P_m)$ which is held in the current control state. Each $p \in P_r$ is a potential future of the play, meaning that the pushed stack may be popped to p and the minimum rank seen since the push could be r. Because Elvis does not have full control of the game, he cannot give a single control state and rank: Agnetha may force him to any of a number of situations.

Once this guess has been made, Agnetha chooses whether to simulate the first play (between the push and the pop) or the second (after the pop). In the first case, $\vec{P}$ is stored in the control state. Then, when the pop occurs, Elvis wins if the destination control state is in P_r where r is the minimum rank seen (his claim was correct). In the second case, Agnetha picks a rank r and moves the play directly to some control state in P_r. This move has rank r (as the minimum rank seen needs to contribute to the parity/safety condition). In both cases, the topmost order-$(n-1)$ stack does not change (as it would be the same in both plays).

To simulate a $push_a$ with $\lambda(a) = n$ and a corresponding *collapse* we observe that the stack reached after the collapse is the same as that after a pop_n applied directly. Thus, the simulation is similar. To simulate the play up to the collapse, the current target set $\vec{P}$ is stored with the new stack character a. Then Elvis wins if a move performs a *collapse* to a control state $p \in P_r$, where r is the smallest rank seen since the order-$(n-1)$ stack, that was topmost at the moment of the original $push_a$, was pushed. To simulate the play after the collapse, we can simulate a pop_n as above.

5 Counter Reduction

We reduce parity to safety games, generalizing Fridman and Zimmermann [14] which extended Bernet et al. [1]. This reduction is not polynomial and we show in Section 7 how to achieve the desired complexity. Correctness is Lemma 3.2 (From Parity to Safety, Inefficient) .

We give the intuition here. The reduction maintains a counter for each odd rank, which can take any value between 0 and B. We also detail the counters below as they are needed in Section 7.

The insight of Bernet et al. is that, in a finite-state parity game of ℓ states, if Agnetha can force the play to pass through some odd rank r for $\ell + 1$ times without visiting a state of lower rank in between, then some state p of rank r is visited twice. Since parity games permit positional winning strategies, Agnetha can repeat the play from p ad infinitum. Thus, the smallest infinitely occurring rank must be r, and Agnetha wins the game.

Thus, Elvis plays a safety game: he must avoid visiting an odd rank too many times without a smaller rank being seen. In the safety game, counters

$$\vec{\alpha} = (\alpha_1, \alpha_3, \ldots, \alpha_m)$$

are added to the states, one for each odd rank. When a rank r is seen, then, if it is odd, α_r is incremented. Moreover, whether r is odd or even, all counters $\alpha_{r'}$ for $r' > r$ are reset to 0.

As the number of configurations is infinite, Bernet's insight does not immediately generalize to pushdown games. However, Fridman and Zimmermann observed that, from Walukiewicz [32], a pushdown parity game can be reduced to a finite-state parity game (of exponential size) as described in the previous section. This finite-state parity game can be further reduced to a safety game with the addition of counters. Their contribution is then to transfer back the counters to the pushdown game, with the following reasoning.

Recall, a push move at $(p, [aw]_1)$ is translated into a branch from a corresponding state $(p, a, \vec{P})$ in the finite-state game. There are several moves from $(p, a, \vec{P})$, some of them simulate the push, the remaining moves simulate the play after the corresponding pop. When augmented with counters the states take the form $(p, a, \vec{P}, \vec{\alpha})$. We see that, when simulating the pop in the finite-state game, the counter values are the same as in the moment when the push is simulated. That is, if we lift the counter construction to the pushdown game, after each pop move we need to reset the counters to their values at the corresponding push. Thus we store the counter values on the stack. For example, for a configuration $(p, [(a, \vec{\alpha})(b, \vec{\alpha}')]_1)$ where the current top of stack is a and the current counter values are $\vec{\alpha}$, the counter values at the moment when a was first pushed are stored on the stack as $\vec{\alpha}'$.

This reasoning generalizes to any order n. We store the counter values on the stack so that, when a pop_k operation occurs, we can retrieve the counter values at the corresponding $push_k$, and similarly for *collapse*. Note also that, when reducing from order-n to order-$(n-1)$, any branch corresponding to a play after a pop passes through a rank r which is the smallest rank seen between the push and pop. Thus, in the safety game, after each pop or collapse we need to update the counter values using r. Hence we require a rank-aware CPDS.

Let m be the maximum rank, and, for convenience, assume it is odd. We maintain a vector of counters $\vec{\alpha} = (\alpha_1, \alpha_3, \ldots, \alpha_m)$, one for each odd rank, stored in the stack alphabet as described above. We update these counters with operations $\oplus_r$ that exist for all $r \in [0, m]$ (including the even ranks). Operation $\oplus_r$ resets the counters $\alpha_{r'}$ with $r' > r$ to zero. If r is odd, it moreover increments α_r. If the bound is exceeded, an overflow occurs. Formally, $\oplus_r(\vec{\alpha}) = \text{NaN}$ if r is odd and $\alpha_r + 1 > B$. Otherwise, $\oplus_r(\vec{\alpha}) = \vec{\alpha}'$ where for each $\tilde{r}$

$$\alpha'_{\tilde{r}} = \alpha_{\tilde{r}} \text{ (if } \tilde{r} < r), \quad \alpha'_{\tilde{r}} = \alpha_r + 1 \text{ (if } \tilde{r} = r), \text{ and } \quad \alpha'_{\tilde{r}} = 0 \text{ (if } \tilde{r} > r).$$

6   Equivalence Result

We need equivalence of $\mathcal{G}_{O,C_B} = \mathrm{Counter}_B(\mathrm{Order}(\mathcal{G}))$ and $\mathcal{G}_{C_B,O} = \mathrm{Order}(\mathrm{Counter}_B(\mathcal{G}))$ for Lemma 3.4. The argument is that the two CPDS only differ in order of the components of their control states and stack characters. A subtlety is that when $\mathrm{Counter}_B$ is applied first, the contents of $\vec{P}$ are not control states of $\mathcal{G}$, but control states of $\mathcal{G}_{C_B}$. However, the additional information in the control states after $\mathrm{Counter}_B$ has to be consistent with $\vec{P}$, which means we can directly translate between guesses over states in the original CPDS, and those over states of the CPDS after the counter reduction.

7   Polynomial Reduction

For a game $\mathcal{G}$ over an order-n CPDS, the counters in the game $\mathrm{Counter}_{B(\mathcal{G})}(\mathcal{G})$ blow up $\mathcal{G}$ by an n-fold exponential factor. To avoid this we use the stack-like behaviour of the counters and a result due to Cachat and Walukiewicz [7], showing how to encode large counter values into the stack of a CPDS with only polynomial overhead (in fact, collapse is not needed).

7.1   Counter Encoding

Cachat and Walukiewicz propose a binary encoding that is nested in the sense that a bit is augmented by its position, and the position is (recursively) encoded in the same way. For example, number 5 stored with 16 bits is represented by $(0,1).(1,0).(2,1).(3,0).(4,0)\dots(15,0)$. Since four bits are required to index 16 bits, we encode position 4 as $(0,0').(1,0').(2,1').(3,0')$. Finally, position 2 of this encoding stored as $(0,0'').(1,1'')$. The players compete to (dis)prove that the indexing is done properly.

Formally we introduce distinct alphabets to encode counters for all odd ranks r:

$$\Gamma_r \;=\; \hat{\Gamma}_r \cup \{0_r, 1_r\} \;.$$

Here, $\hat{\Gamma}_r$ is a polynomially-large set of characters for the indexing. The set $\{0_r, 1_r\}$ are the bits to encode numbers. Let Γ be the union of all Γ_r.

The values of the counters are stored on the order-1 stack, with the least-significant bit topmost. The indices appear before each bit character. E.g., value 16 for counter r stored with five bits yields a sequence from $\hat{\Gamma}_r^* . 0_r . \hat{\Gamma}_r^* . 0_r . \hat{\Gamma}_r^* . 0_r . \hat{\Gamma}_r^* . 0_r . \hat{\Gamma}_r^* . 1_r$. Actually, the encoding will always use all bits, which means its length will be $(n-1)$-fold exponential.

Cachat and Walukiewicz provide game constructions to assert properties of the counter encodings. For this, play moves to a dedicated control state, from which Elvis wins iff the counters have the specified property. In [7], Elvis plays a reachability game from the dedicated state. We need the dual, with inverted state ownership and a safety winning condition, where the target state of the (former) reachability game has rank 1. Elvis's goal will be to prove the encoding wrong (it violates a property) by means of safety, Agnetha tries to build up the counters correctly and, if asked, demonstrate correctness using reachability.

For all properties, the counter to be checked must appear directly on the top of the stack (topmost on the topmost order-1 stack). If any character outside Γ_r is found, Agnetha loses. When two counters are compared, the first counter must appear directly at the top of the stack, while the second may be separated from the first by any sequence of characters from outside Γ_r (these can be popped away). The first character found from Γ_r begins the next encoding. Agnetha loses the game if none is found. The required properties are listed below.

- Encoding Check ($encoding_r$): For each rank r, we have a control state $encoding_r$. Agnetha can win the safety game from $\langle encoding_r, w \rangle$ only if the topmost sequence of characters from Γ_r is a correct encoding of a counter, in that all indices are present and correct.
- Equals Check ($equal_r$): For each r, we have a control state $equal_r$, from which Agnetha can win only if the topmost sequence of characters from Γ_r is identical to the next topmost sequence of Γ_r-characters. I.e., the two topmost rth counter encodings are equal.
- Counter Increment: Cachat and Walukiewicz do not define increment but it can be done via the basic rules of binary addition. We force Agnetha to increment the counter by first using pop_1 to remove characters from $\hat{\Gamma}_r \cup \{1_r\}$ until 0_r is found. Then, Agnetha must rewrite the 0_r to 1_r. Agnetha then performs as many $push_a$ operations as she wishes, where $a \in \hat{\Gamma}_r \cup \{0_r\}$. Next, Elvis can accept this rewriting by continuing with the game, or challenge it by moving to $encoding_r$. This ensures that Agnetha has put enough 0_r characters on the stack (with correct indexing) to restore the number to its full length.

In this encoding one can only increment the topmost counter on the stack. That is, to increment a counter, all counters above it must be erased. Fortunately, $\oplus_r$ resets to zero all counters for ranks $r' > r$, meaning the counter updates follow a stack-like discipline. This enables the encoding to work. To store a character with counter values from the counter reduction $(a, \vec{\alpha})$ with $\vec{\alpha} = \alpha_1, \ldots, \alpha_m$ we store the character a on top and beneath we encode α_m, then α_{m-2} and so on down to α_1.

7.2   The Simulation

The following definition is completed in the following sections. Correctness is stated in Lemma 3.3 (From Inefficient to Efficient) .

▶ **Definition 7.1** (Poly_B)**.** Given a parity game $\mathcal{G} = (\mathcal{C}, \mathbb{O}, \mathcal{W})$ over the order-n CPDS $\mathcal{C} = (\mathcal{P}, \Sigma, \lambda, \mathcal{R}, p_I, a_I, \rho)$ and a bound B n-fold exponential in the size of the game, we define the safety game $\text{Poly}_B(\mathcal{G}) = (\mathcal{C}', \mathbb{O}', 2^\omega)$ where $\mathcal{C}' = (\mathcal{Q}, \Sigma', \lambda', \mathcal{R}', p_I', a_I', \rho')$. The missing components are defined below.

We aim to simulate $\text{Counter}_B(\mathcal{G})$ compactly. This simulation is move-by-move, as follows.

A $push_{(a, \vec{\alpha})}$ of a character with counter values $(a, \vec{\alpha})$ with $\vec{\alpha} = \alpha_1, \ldots, \alpha_m$ (where the max-rank is m) is simulated by first pushing a special character ℓ_k to save the link (with $push_{\ell_k}$). Then, since the counter values are a copy of the preceding counter values on the stack, Agnetha pushes an encoding for α_1 using Γ_1 after which Elvis can accept the encoding, check that it is a proper encoding using $encoding_1$, or check that it is a faithful copy of the preceding value of α_1 using $equal_1$. We do this for all odd ranks through to m. Then the only move is to push a with $push_a$.

Each $push_k$ and pop_k, with $k \in [2, n]$, is simulated directly by the same operation. For a pop_1 we (deterministically) remove all topmost characters (using pop_1) up to and including the first $\ell_{k'}$ (for some k'). We simulate $collapse$ like pop_1, but we apply $collapse$ to $\ell_{k'}$.

A $rew_{(a, \vec{\alpha})}$ that does not change the counters can be simulated by rewriting the topmost character. If $\oplus_r$ is applied, we force Agnetha to play as follows. If r is even, Agnetha removes the counters for $r' > r$. She replaces them with zero values by pushing characters from $\hat{\Gamma}_{r'} \cup \{0_{r'}\}$. After each counter is rewritten, Elvis can accept the encoding, or challenge it with $encoding_{r'}$. Finally, a is pushed onto the stack. If r is odd, the counters for r' are removed as before. Then we do an increment as described above, with Elvis losing if the increment fails. Note, it fails only if there is no 0_r in the encoding, which means the counter is at its maximum value and there is an overflow (indicating Elvis loses the parity game). If it succeeds, zero values for the counters $r' > r$ and a are pushed to the stack as before.

Control States and Alphabet

We define the control states $\mathcal{Q}$ with $\mathbb{O}'$ and ρ' as well as the alphabet. First,

$$\mathcal{Q} \;=\; \mathcal{P} \cup (\mathcal{P} \times [0, m]) \cup \{\#, \$\} \cup \mathcal{P}_{CW} \;\cup \mathcal{P}_{OP} \;.$$

where m is the maximum rank. The set $\mathcal{P}_{CW}$ is the control states of the Cachat-Walukiewicz games implementing $encoding_r$ and $equal_r$. The size is polynomial in $\mathcal{G}$. We have $\mathcal{P}_{OP} =$

$$\left\{ \begin{array}{l} (\mathrm{inc}, r, p), (\mathrm{copy}, r, a, p), (\mathrm{zero}, r, a, p), (\mathrm{pop}_1, \Upsilon, r, p), \\ (\mathrm{inc}, r, a, p), (\mathrm{cchk}, r, a, p), (\mathrm{zchk}, r, a, p), (\mathrm{collapse}, \Upsilon, r, p) \end{array} \;\middle|\; \begin{array}{l} r \in [0, m] \wedge a \in \Sigma \wedge \\ p \in \mathcal{P} \wedge \Upsilon \in \{\mathrm{A}, \mathrm{E}\} \end{array} \right\}$$

to control the simulation of the operations as sketched above. We describe the states below.

The states in $\mathcal{P}_{CW}$ have the same rank and owner as in the Cachat-Walukiewicz games (more precisely the dual, see above). All other states have rank 2 except $\#$ which has rank 1. It (resp. $\$$) is the losing sink for Elvis (resp. Agnetha). The states in $\mathcal{P} \cup (\mathcal{P} \times [0, m]) \cup \{\#\}$ are used as in $\mathrm{Counter}_B(\mathcal{G})$ to directly simulate $\mathcal{G}$. The owners are as in $\mathcal{G}$.

A state (inc, r, p) begins an application of $\oplus_r$. The top-of-stack character is saved by moving to (inc, r, a, p). The owner of these states does not matter, we give them to Agnetha. In (inc, r, a, p), the stack is popped down to the counter for r. If r is odd, the least significant zero is set to one. Then, control moves to (zero, r, a, p). In (zero, r, a, p), zero counters for ranks r and above are pushed to the stack, followed by a push of a and a return to control state p. The state is owned by Agnetha. The state (zchk, r, a, p) is used by Elvis to accept or challenge that the encoding has been re-established completely. It is owned by Elvis.

The controls (copy, r, a, p) copy the counters for ranks r and above (the current values) and push the copies to the stack, followed by a push of a and a return to control state p. The state is owned by Agnetha. After this phase, the play moves to (cchk, r, a, p) where Elvis can accept or test whether the copy has been done correctly. This state is owned by Elvis.

The controls $(\mathrm{pop}_1, \Upsilon, r, p)$ and $(\mathrm{collapse}, \Upsilon, r, p)$ where $\Upsilon \in \{\mathrm{A}, \mathrm{E}\}$ are used to execute a pop_1 or $collapse$. For the latter, we pop to the next ℓ_k character, perform the collapse and record that the rth counter needs to be incremented. In case the collapse is not possible (because it would empty the stack) play may also move to a sink state that is losing for the player Υ who instigated the collapse. The case of pop_1 pops ℓ_k. The owner in each case is Υ as they will avoid moving to their (losing) sink state if the pop_1 or $collapse$ is possible.

The alphabet and initial control state and stack character are

$$\Sigma' = \Sigma \cup \Gamma \cup \Delta \quad \text{and} \quad p'_I = (\mathrm{zero}, 1, a_I, p_I) \quad \text{and} \quad a'_I = \ell_k \text{ where } \lambda(a_I) = k \;.$$

The alphabet is extended by the characters required for the counter and link encodings. Recall that Γ is the union of the counter alphabets, which are of polynomial size. We use $\Delta = \{\ell_1, \ldots, \ell_m\}$ for the link characters. We assign $\lambda'(\ell_k) = k$ and $\lambda'(a) = 1$ for all other a.

The task of the initial state and initial stack character is to establish the encoding of $(a_I, (0, \ldots, 0))$ in $\mathrm{Counter}_B(\mathcal{G})$ and then move to the initial state of $\mathcal{G}$. With the above description, $(\mathrm{zero}, 1, a_I, p_I)$ will establish zeros in all counters (from 1 to m), push the initial character a_I of the given game, and move to state p_I. The initial character ℓ_k is the correct bottom element for the encoding of $(a_I, (0, \ldots, 0))$.

Rules

The rules of $\mathcal{C}'$ follow $\mathcal{C}$ and maintain the counters. $\mathcal{R}'$ contains (only) the following rules. First, we have $\mathcal{R}_{CW}$ which are the (dual of the) rules of Cachat and Walukiewicz implementing $encoding_r$ and $equal_r$. The rules simulating the operations appear below. Note, pop and collapse use rank-awareness. We give the increment and copy rules after the basic operations.

- Order-k push: $(p, a, push_k, (\text{inc}, \rho(p'), p'))$ when $(p, a, push_k, p') \in \mathcal{R}$.
- Character push: $(p, a, push_{\ell_k}, (\text{copy}, 1, b, p'))$ when $(p, a, push_b, p') \in \mathcal{R}$ and $\lambda(b) = k$.
- Rewrite: $(p, a, rew_b, (\text{inc}, \rho(p'), p'))$ when $(p, a, rew_b, p') \in \mathcal{R}$.
- Pop (> 1): $(p, a, pop_k, (\text{inc}, r, p'))$ when $(p, a, pop_k, p') \in \mathcal{R}$ and $r = \min(\rho(p'), \text{Rk}(k))$.
- Pop $(= 1)$ and Collapse: $(p, a, pop_1, (o, \Upsilon, r, p'))$ when $(p, a, o, p') \in \mathcal{R}$ with $\mathbb{O}(p) = \Upsilon$, operation o being pop_1 or *collapse*, and $r = \min(\rho(p'), r')$. Here, if $o = pop_1$ then $r' = \text{Rk}(1)$. Otherwise, if $o = collapse$ then $r' = \text{Rk}(\ell)$.
  Then, we have all rules $((o, \Upsilon, r, p'), a', pop_1, (o, \Upsilon, r, p'))$ for $a' \in \Gamma$. We perform the operation with $((o, \Upsilon, r, p'), \ell_{k'}, o, (\text{inc}, r, p'))$. To allow for the case where the pop or collapse cannot be performed (because the stack would empty), we also have the rules $((o, \text{A}, r, p'), \ell_{k'}, nop, \$)$ and $((o, \text{E}, r, p'), \ell_{k'}, nop, \#)$.
- Sink states: $(\$, a, nop, \$)$ and $(\#, a, nop, \#)$.

To copy counters, for each odd r and $b \in \Gamma_r$ we have $((\text{copy}, r, a, p), *, push_b, (\text{copy}, r, a, p))$. We use $*$ to indicate that the transition exists for all stack symbols. When a counter has been pushed, like in the case of pushing zeros, Agnetha hands over the control to Elvis to check the result: $((\text{copy}, r, a, p), *, nop, (\text{cchk}, r, a, p))$. Elvis can challenge the copied counter or accept it was copied correctly. To challenge, we use $((\text{cchk}, r, a, p), *, nop, equal_r)$. To accept, the behavior depends on r. If $r < m$, we move to copying the next counter $((\text{cchk}, r, a, p), *, nop, (\text{copy}, r+2, a, p))$. When $r = m$, we finish copying and move to incrementing with rules of the form $((\text{cchk}, r, a, p), *, nop, (\text{inc}, \rho(p), a, p))$.

To increment a counter, we first pop and store the topmost stack character with the rule $((\text{inc}, r, p), a, pop_1, (\text{inc}, r, a, p))$. Agnetha then removes all counters for ranks higher than the given r with the following rules, where $b \in \Gamma_{r'}$ with $r' > r$: $((\text{inc}, r, a, p), b, pop_1, (\text{inc}, r, a, p))$.

When r is even we add back 0 counters once enough have been removed using (with $b \in \Gamma_{r-1}$ if $r > 1$ else $b \in \Delta$) the rules $((\text{inc}, r, a, p), b, nop, (\text{zero}, r+1, a, p))$. If r is odd, we start incrementing the rth counter with $((\text{inc}, r, a, p), b, pop_1, (\text{inc}, r, a, p))$ for all $b \in \hat{\Gamma}_r \cup \{1_r\}$.

When 0_r is found, we use $((\text{inc}, r, a, p), 0_r, rew_{1_r}, (\text{zero}, r, a, p))$. If no zero bit is found, we have an overflow and move to the sink state with $((\text{inc}, r, a, p), b, nop, \#)$ for $b \in \Gamma_{r-2}$ if $r > 2$ and $b \in \Delta$ otherwise. With $((\text{zero}, r, a, p), *, push_b, (\text{zero}, r, a, p))$ for $b \in \hat{\Gamma}_r \cup \{0_r\}$ we add back zeros to the incremented counter and reset all erased counters. To finish the phase that adds zeros for the rth counter, Agnetha hands over the control to Elvis, $((\text{zero}, r, a, p), *, nop, (\text{zchk}, r, a, p))$.

Elvis can now check if all bits of the counter are present or accept the result. To challenge the encoding, he uses $((\text{zchk}, r, a, p), *, nop, encoding_r)$. When accepting it, if $r < m$, more counters need to be reset. We move to the next using $((\text{zchk}, r, a, p), *, nop, (\text{zero}, r+2, a, p))$. If $r = m$, there are no more counters to handle and with the rule $((\text{zchk}, r, a, p), *, push_a, p)$ Elvis re-establishes the control state and stack character.

8 Conclusion

We gave a polynomial-time reduction from parity games played over order-n CPDS to safety games over order-n CPDS. Such a reduction has been an open problem [14] (related are also [1, 2]). It builds counters into the stack to count occurrences of odd ranks at the current stack level (without seeing a smaller rank). If this number grows large then Elvis would lose the parity game (if play continued). To obtain a polynomial reduction we use the insight that the counters follow a stack discipline. For correctness, we use a commutativity argument for the rank counter and order reductions. As a theoretical interest, the result explains the matching complexities of parity and safety games over CPDS. From a practical standpoint, the reduction may inspire the use of advanced safety checking tools for, and the transfer of technology from safety to, the empirically harder problem of parity game analysis.

───── **References** ─────

1   J. Bernet, D. Janin, and I. Walukiewicz. Permissive strategies: from parity games to safety games. *ITA*, 2002.

2   D. Berwanger and L. Doyen. On the Power of Imperfect Information. In *FSTTCS*, 2008.

3   A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electr. Notes Theor. Comput. Sci.*, 2002.

4   R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. Decidability in parameterized verification. *SIGACT News*, 47(2), 2016.

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

6   C. H. Broadbent and N. Kobayashi. Saturation-based model checking of higher-order recursion schemes. In *CSL*, 2013.

7   T. Cachat and I. Walukiewicz. The complexity of games on higher order pushdown automata. *CoRR*, abs/0705.0262, 2007.

8   C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. Deciding parity games in quasipolynomial time. In *STOC*, 2017.

9   A. Carayol, M. Hague, A. Meyer, C.-H. L. Ong, and O. Serre. Winning Regions of Higher-Order Pushdown Games. In *LICS*, 2008.

10 J. Daniel, A. Cimatti, A. Griggio, S. Tonetta, and S. Mover. Infinite-state liveness-to-safety via implicit abstraction and well-founded relations. In *CAV*, 2016.

11 J. Engelfriet. Iterated stack automata and complexity classes. *Inf. Comput.*, 95(1):21–75, 1991.

12 A. Farzan, Z. Kincaid, and A. Podelski. Proving liveness of parameterized programs. In *LICS*, 2016.

13 J. Fearnley, S. Jain, S. Schewe, F. Stephan, and D. Wojtczak. An ordered approach to solving parity games in quasi polynomial time and quasi linear space. In *SPIN*, 2017.

14 W. Fridman and M. Zimmermann. Playing pushdown parity games in a hurry. In *GandALF*, 2012.

15 K. Fujima, S. Ito, and N. Kobayashi. Practical alternating parity tree automata model checking of higher-order recursion schemes. In *APLAS*, 2013.

16 M. Hague, R. Meyer, S. Muskalla, and M. Zimmermann. Parity to safety in polynomial time for pushdown and collapsible pushdown systems. *CoRR*, abs/1805.02963, 2018. `arXiv:1805.02963`.

17 M. Hague, A. S. Murawski, C.-H. Luke Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LICS*, 2008.

18 M. Jurdzinski and R. Lazic. Succinct progress measures for solving parity games. In *LICS*, pages 1–9, 2017.

19 T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS '02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, pages 205–222, London, UK, 2002. Springer-Verlag.

20 N. Kobayashi. HorSat2: A model checker for HORS based on SATuration. A tool available at `http://www-kb.is.s.u-tokyo.ac.jp/~koba/horsat2/`.

21 I. V. Konnov, M. Lazic, H. Veith, and J. Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *POPL*, 2017.

22 G. Lenzi. The modal μ-calculus: a survey. *Task quarterly*, 9(3):293–316, 2005.

23 R. P. Neatherway and C.-H. L. Ong. Travmc2: higher-order model checking for alternating parity tree automata. In *SPIN*, 2014.

24 C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, 2006.

25 O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham. Reducing liveness to safety in first-order logic. *PACMPL*, 2017.

26 A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, 2004.

27 A. Podelski and A. Rybalchenko. Transition invariants and transition predicate abstraction for program termination. In *TACAS*, 2011.

28 S. J. Ramsay, R. P. Neatherway, and C.-H. L. Ong. A type-directed abstraction refinement approach to higher-order model checking. In *POPL*, 2014.

29 V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.*, 2005.

30 S. Sohail and F. Somenzi. Safety first: A two-stage algorithm for ltl games. In *FMCAD*, 2009.

31 T. Terao and N. Kobayashi. A zdd-based efficient higher-order model checking algorithm. In *APLAS*, 2014.

32 I. Walukiewicz. Pushdown processes: Games and model checking. In *CAV*, 1996.