Cost Register Automata for Nested Words

Andreas Krebs[1], Nutan Limaye[2], and Michael Ludwig[1](✉)

[1] University of Tübingen, Tübingen, Germany
{krebs,ludwigm}@informatik.uni-tuebingen.de
[2] Indian Institute of Technology, Bombay, India
nutan@cse.iitb.ac.in

Abstract. In the last two decades visibly pushdown languages (VPLs) have found many applications in diverse areas such as formal verification and processing of XML documents. Recently, there has been a significant interest in studying quantitative versions of finite-state systems as well as visibly pushdown systems. In this work, we take forward this study for visibly pushdown systems by considering a functional version of visibly pushdown automata. Our version is formally a generalization of cost register automata (CRA) defined by [Alur et al., 2013]. We observe that our model continues to have all the *good* properties of the CRAs in spite of being a generalization.

Apart from studying the functional properties of the model, we also study the complexity theoretic aspects. Recently such a study was conducted by [Allender and Mertz, 2014] with respect to CRAs. Here we show that CRAs when appended with a visible stack (i.e. in the model defined here), continue to have the same complexity theoretic upper bounds as are known for CRAs. Moreover, we observe that one of the upper bounds shown by Allender et al. which was not tight for CRAs becomes tight for our model. Hence, it answers one of the questions raised in their work.

1 Introduction

Language theory and complexity theory are two major branches of study in theoretical computer science. The study of the interplay between the two has a rich history of more than three decades. The study has not only increased our understanding of many complexity classes, but also has led to very beautiful theory. For example, a well-studied complexity class $\mathbf{NC}^1$ and the finer structure inside the class was revealed through the study of subclasses of regular languages [7,8,19]. Similarly, the class LogCFL and its equivalence with the complexity class $\mathbf{SAC}^1$ is another celebrated example of such a connection [10,15,18,21]. The study of language classes has been relevant in understanding Boolean complexity classes; and similarly, it is helpful to relate arithmetic complexity classes such as $\#\mathbf{NC}^1$, $\#\mathbf{SAC}^1$, $\#\mathbf{P}$ to formal power series (see for example [2]) for improving our understanding of these arithmetic classes.

In the literature of language theory, the study of formal power series has played a central role. One of the most well-studied models, namely weighted

T.N. Dinh and M.T. Thai (Eds.): COCOON 2016, LNCS 9797, pp. 587–598, 2016.
DOI: 10.1007/978-3-319-42634-1_47

automata, are an example of automata which compute an interesting class of formal power series.

More recently, Alur et al. [4,5] defined a generalization of weighted automata, namely *Cost Regsiter Automata*, CRA. In a subsequent work of Allender and Mertz [3] the complexity theoretic study of the formal power series defined by CRAs was performed.

Inspired by the works of [3–5], we extend the complexity theoretical study of formal power series in this work. CRAs defined and studied by [4,5] are models which compute functions from Σ^* to $\mathbb{D}$, where Σ is an input alphabet and $\mathbb{D}$ is a domain such as $\mathbb{Q}, \mathbb{Z}$ or $\mathbb{N}$. A CRA is a finite state automaton which is equipped with a set of registers and an algebra $\mathcal{A}$. The automaton reads the input from left to right; as in the case of a usual finite state automaton, but additionally various registers can get updated by using rules prescribed by the underlying algebra $\mathcal{A}$. It turns out, augmenting a finite state automaton with registers working over an algebra provides a way for performing a quantitative study of regular languages.

Here, we perform a similar quantitative analysis of visibly pushdown languages (VPLs) by augmenting the visibly pushdown automata (VPA) with registers working over an algebra. In this sense, we define a new model of computation, which is a natural extension of two well-known models of computation, namely VPAs and CRAs. We call our model *Cost Register Visibly Pushdown Automata* (CVPA) to signify this combination.

VPLs in their current incarnation were first conceptualized by Alur and Madhusudan [6]. However they have been introduced by Mehlhorn under the name input-driven pushdown languages [20] earlier. Due to their power to generalize regular languages while staying extremely structured, they have found many applications in diverse areas such as formal verification and processing of XML documents. Just like the study of regular systems through weighted automata and CRAs has helped in understanding their quantitative properties, we believe that our work is a step towards understanding quantitative properties of visibly pushdown systems.

Recently, Allender and Mertz [3] developed the complexity theoretic understanding of the arithmetic functions computed by CRAs. They proved many complexity theoretic upper bounds for functions computed by CRAs over many different algebras. In this work, we extend their results to CVPAs. We prove that CVPAs have computational complexity bounds similar to CRAs. This presents a gratifying picture, because CVPAs formally generalize CRAs, however, still stay *computationally efficient*.

Contributions. We start by proving basic language theoretic results about our model. In particular, we first show basic closure properties of CVPAs. Here we get that CVPAs are closed with respect to many natural properties (as shown by [4,5] for CRAs.) We then show that CVPAs formally generalize VPAs and CRAs. It is known that there is a version of CRAs, namely copyless CRAs over $(\mathbb{N}, +)$ which can be simulated by CRAs over $(\mathbb{N}, +c)$[1]. However, we observe

[1] Notations explained later.

that this is not true for CVPAs. That is, we give an explicit function which is computable by copyless CVPAs over $(\mathbb{N}, +)$ but cannot be computed by $(\mathbb{N}, +c)$.

The arguments we present here are combinatorial and may be useful in proving similar things for different algebras in the case of CRAs. We also show that CRAs over $(\mathbb{N}, +)$ compute the same class of functions as #NFA. Moreover, we prove that CVPAs generalize the weighted and the counting variant of VPLs as defined in [13,17].

We then perform a complexity theoretical study of CVPAs along the lines of Allender and Mertz [3]. Here our main theorem states that CVPA over the algebra $(\mathbb{Z}, +)$ are Gap$\mathbf{NC}^1$-complete. Our Gap$\mathbf{NC}^1$ hardness result also holds for a restricted version of CVPAs, namely copyless CVPAs (CCVPAs). No such result is known for the copyless variant of CRAs.

Our Gap$\mathbf{NC}^1$ upper bound for CVPA over $(\mathbb{Z}, +)$ builds on several different results. It combines ideas from Buss [11] and Dymond [16] who showed that Boolean formula evaluation (and membership in regular languages) and membership in VPLs are in $\mathbf{NC}^1$ (resp.) with some crucial concepts from [12] which showed that arithmetic formula evaluation is in $\#\mathbf{NC}^1$. This also proves the hardness. Recently, [17] showed an $\#\mathbf{NC}^1$ upper bound (and hardness) for #VPA. In our paper we take this even further and show Gap$\mathbf{NC}^1$ upper bound for CVPA over $(\mathbb{Z}, +)$ by appropriately combining all these ideas. We also prove that the class CVPA$(\mathbb{Z}, +c)$ is contained in $\mathbf{NC}^1$, i.e. for any function in CVPA$(\mathbb{Z}, +c)$, the ith bit of the function is computable by an $\mathbf{NC}^1$ circuit.

2 Preliminaries

In this section we establish notation and recall some known definitions and results from circuit complexity and language theory.

2.1 Notation

By Σ and Γ we denote finite *alphabets* and (finite) *words* are finite sequences of letters of the alphabet. The set of all words over Σ is the free monoid Σ^* with concatenation $\circ$ as operation. A *language* is a subset of Σ^*. For $w \in \Sigma^*$, we address the ith letter of w by w_i and by $|w|$ we denote the *length* of w. We use $|w|_a$ to denote the number of letters a appearing in w. The *empty word* is denoted by ϵ.

An *algebra* $\mathcal{A} = (\mathbb{D}, \oplus_1, \ldots, \oplus_k)$ is a *domain* $\mathbb{D}$ which can be infinite, together with a number of *operators* $\oplus_i \colon \mathbb{D}^{\alpha_i} \to \mathbb{D}$ where α_i is the arity of $\oplus_i$. We usually have binary operators and sometimes unary ones. For binary operator $\oplus$ and $c \in \mathbb{D}$, we use $\oplus c$ to indicate that one of the operands of $\oplus$ is a domain constant c. For $k = 1$ and depending on the properties of $\oplus_1$, we get semigroups, monoids, and groups. For $k = 2$, depending on the properties of $\oplus_1$ and $\oplus_2$ we get different kinds of rings. Examples of algebras we use in this paper include $\mathbb{B} = (\{0, 1\}, \wedge, \vee, \neg)$, $(\mathbb{N}, +)$, $(\mathbb{Z}, +)$, $(\mathbb{Z}, +, \times)$, $(\mathbb{Z}, +, \min)$, and $(\mathbb{Z}, +c)$ where $+c$ means addition with a constant.

By $E(\mathcal{A})$ we denote the set of *expressions* (or formulas) over $\mathcal{A}$: All $d \in \mathbb{D}$ are (atomic) expressions and if a and b are expressions, then $(a \oplus_i b)$ is an expression for all $1 \leq i \leq k$. Let $X = \{x_1, \ldots, x_l\}$ be a set of variable names (which we later use as *register* names). We consider these variables as atomic expressions, from which we get a set we denote by $E(\mathcal{A}, X)$. An expression $E(\mathcal{A})$ evaluates to an element in $\mathbb{D}$ and an expression in $E(\mathcal{A}, X)$ evaluates to a function $\mathbb{D}^{|X|} \to \mathbb{D}$. With this we are now ready to define the model we wish to study in this paper.

2.2 Circuit Complexity

For the basic circuit complexity we refer e.g. to [22].

A *circuit* $C_n = (V, E, \Sigma, \mathcal{A}, l, v_{\text{out}})$ is a directed acyclic graph (V, E) in which the vertices are called *gates*, the in-degree is called *fan-in*. There are n gates of fan-in 0, which are called *input gates*, which are marked with inputs to the circuit. For a circuit computing functions over an algebra $\mathcal{A} = (\mathbb{D}, \oplus_1, \ldots, \oplus_k)$, the map $l\colon V \to \{\oplus_1, \ldots, \oplus_k\}$ assigns each gate one of the functions.

If a gate g has fan-in t with gates $g_1, g_2, \ldots, g_t$ feeding into it, then it computes the function $l(g)\,(g_1, g_2, \ldots, g_t)$. Fan-in greater than two of a gate g requires associativity of $(\mathbb{D}, l(g))$. The gates naturally compute functions of their fan-ins. The gate $v_{\text{out}} \in V$ is the output gate, which computes a function $\mathbb{D}^n \to \mathbb{D}$. *Formulas* are circuits, in which the underlying DAG is a tree. To treat inputs of arbitrary length, we consider *families of circuits* $C = (C_n)_{n \in \mathbb{N}}$, where there is one circuit for each input length.

For $\mathcal{A} = (\{0,1\}, \wedge, \vee, \neg)$ we get Boolean circuits, similarly, for $\mathcal{A} = (\mathbb{Z}, +, \times)$ we get arithmetic circuits.

If we want to emphasize a different algebra, we write e.g. $\mathcal{C}(\mathcal{A})$, where $\mathcal{C}$ is a circuit class. The special case of $\mathcal{C}(\mathbb{N}, +, \times)$ restricted to $\{0,1\}$-inputs is known as $\#\mathcal{C}$ and $\mathcal{C}(\mathbb{Z}, +, \times)$ restricted to $\{-1, 0, 1\}$-inputs is known as $\mathbf{Gap}\mathcal{C}$. For basics in circuit classes and arithmetic circuit complexity see e.g. [1,22].

2.3 Visibly Pushdown Automata

In our paper we define a new model which computes functions from Σ^* to some domain $\mathbb{D}$. We view our model as a combination of two well-studied models of computations, namely Visibly pushdown automata (VPA) [6] and Cost register automata (CRA) [4,5]. In order to define our model, we first start by recalling the definition of VPA. In the context of VPA we always have to specify a *visible alphabet* $\hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}})$ such that Σ is the disjoint union of Σ_{call}, Σ_{ret} and Σ_{int}, where Σ_{call} are *call* or *push* letters, Σ_{ret} *return* or *pop* letters and Σ_{int} *internal* letters. For the rest of the paper we assume a fixed visible alphabet unless stated otherwise.

Definition 1 (Visibly Pushdown Automaton (VPA) [6]). *A deterministic visibly pushdown automaton is a tuple* $M = (\hat{\Sigma}, Q, q_0, \delta, \Gamma, \bot, F)$ *where* $\hat{\Sigma}$ *is the visible alphabet,* Q *is the finite set of states,* $q_0 \in Q$ *is the initial state,* δ *is the transition function with* $\delta\colon Q \times \Sigma_{\text{int}} \to Q$, $\delta\colon Q \times \Sigma_{\text{call}} \to Q \times \Gamma$, *and*

$\delta\colon Q \times \Sigma_{\text{ret}} \times \Gamma \to Q$. *Also,* Γ *is the finite stack alphabet with bottom-of-stack symbol* $\perp \in \Gamma$*, and* $F \subseteq Q$ *is the set of final states.*

A configuration of M is an element (q, γ), where q is a state and γ is a word over Γ. A run $(q_0, \gamma_0) \dots (q_n, \gamma_n)$ of M on a word $w \in \Sigma^n$ is defined as follows:

- If $w_i \in \Sigma_{\text{int}}$, then $\gamma_i = \gamma_{i-1}$ and $q_i = \delta(q_{i-1}, w_i)$.
- If $w_i \in \Sigma_{\text{call}}$, then $\gamma_i = \gamma_{i-1}\gamma$ and $(q_i, \gamma) = \delta(q_{i-1}, w_i)$.
- If $w_i \in \Sigma_{\text{ret}}$, then $\gamma_i\gamma = \gamma_{i-1}$ and $q_i = \delta(q_{i-1}, w_i, \gamma)$. If $\gamma = \perp$, the word is rejected.

A run is said to be accepting if $q_n \in F$. The language accepted by M, denoted by $L(M)$, is defined to be the set of words $w \in \Sigma^*$ such that M has an accepting run on w starting from $(q_0, \perp)$.

As defined above, a VPA M can be thought of as computing a function $f_M : \Sigma^* \to \{0, 1\}$, where for all $w \in \Sigma^*$, $f_M(w) = 1$ if and only if $w \in L(M)$. We will append this model with *registers* and interpret its computation over an algebra $\mathcal{A}$ so as to design more general functions $\Sigma^* \to \mathbb{D}$, where $\mathbb{D}$ is a general domain such as $\mathbb{Z}, \mathbb{N}$ etc.

In what follows we will assume that all the input words are *well-matched*, i.e. all our input words have the following two properties.

- any prefix $w_i = w_1 \dots w_i$ of w the number of call letters in w_i is greater than or equal to the number of return letters in w_i.
- the number of call letters in w equals the number of return letters in w.

This assumption is for the ease of exposition and all our results hold even in the general case, unless stated otherwise. Given some word, we call two positions i, j *matching* if $w_i \dots w_j$ is well-matched, there is no prefix of $w_i \dots w_j$ which is well-matched and $w_i \in \Sigma_{\text{call}}$ and $w_j \in \Sigma_{\text{ret}}$.

2.4 Cost Register Automata (CRA)

Here we recall the definition of Cost Register automata, CRA. from [4,5] with a small modification resulting in an equivalent model. In [4,5], the register updates are performed using a tree grammar, but here we consider a version wherein the rules of update are governed by the underlying algebra. It is this version which will help us to define the cost register VPA, the model which we will introduce in the next section.

Definition 2 (Cost Register Automaton (CRA) [4,5]). *A CRA is a tuple* $M = (\Sigma, Q, q_0, \delta, X, \mathcal{A}, v_0, \rho, \mu)$ *where* Q *is the finite set of states,* q_0 *is the initial state,* $\delta\colon Q \times \Sigma \to Q$ *is the transition function,* $X = \{x_1, x_2 \dots, x_k\}$ *is the finite set of registers,* $\mathcal{A} = (\mathbb{D}, \oplus_1, \dots, \oplus_m)$ *is an algebra,* $v_0\colon X \to \mathbb{D}$ *are the initial register values,* $\rho\colon Q \times \Sigma \times X \to E(\mathcal{A}, X)$ *is the partial register update function, and* $\mu\colon Q \to E(\mathcal{A}, X)$ *is the partial final cost function.*

A *configuration* of M is an element (q, v), where q is a state and $v\colon X \to E(\mathcal{A}, X)$ assigns each register an expression over the algebra $\mathcal{A}$. A *run* $(q_0, v_0) \dots (q_n, v_n)$ of M on some input $w \in \Sigma^*$ starts in (q_0, v_0), $q_i = \delta(q_{i-1}, w_i)$ for each $1 \leq i \leq n$ and for each $x \in X$, $v_i(x)$ is the expression obtained by substituting $v_{i-1}(y)$ for every occurrence of y in $\rho(q_i, w_i, x) \in E(\mathcal{A}, X)$.

The semantics of a CRA M is the function $F_M : \Sigma^* \to \mathbb{D}$ which is defined as the evaluation of the expression $\mu(q_n)$ by substituting the expression $v_n(y)$ for every occurrence of $y \in X$ in the expression $\mu(q_n)$. It is undefined if $\mu(q_n)$ is undefined.

Claim ([4, 5]). Given a DFA M, which accepts language $L(M)$, there is a CRA M' such that $F_{M'}(w) = 1$ if $w \in L(M)$ and $F_{M'}(w) = 0$ if $w \notin L(M)$, i.e. M' computes the characteristic function $\Sigma^* \to \{0, 1\}$ of the language $L(M)$.

Note that cost registers are always driven by the states while on the other hand the registers have no impact on the states.

3 Cost Register VPA

Here we define a version of a visibly pushdown automaton by augmenting it with cost registers. A *cost register visibly pushdown automaton*, which we denote as CVPA, is a VPA which along with its visible stack has a finite set of registers, say X and an underlying algebra, say $\mathcal{A}$, which influences the computations performed by the registers. The automaton reads the input from left to right and depending on whether the letter read is a push/pop/internal letter, it updates its stack. This part of the working is exactly as in a VPA.

Additionally, there is a finite number of registers $X = \{x_1, \dots, x_k\}$. While reading an internal letter the registers are updated as in the case of CRAs. While reading a call letter, the current instance of the register values is allowed to be saved on the stack as a vector $(x_1, \dots, x_k)$, and the next set of registers are obtained from these. Here, we assume that registers are stored as formal expressions and their evaluations are performed as in the case of CRAs. Finally, while reading the return letter, the registers are updated using the registers available from the previous instance as well as by using the vector of registers on the top of the stack.

Definition 3 (Cost Register VPA (CVPA)). *A cost register visibly pushdown automaton is a tuple $M = (\hat{\Sigma}, Q, q_0, \delta, \Gamma, \bot, \delta, X, \mathcal{A}, v_0, \rho, \mu)$, where, $\hat{\Sigma}$ is the visible alphabet, Q is the finite set of states, $q_0 \in Q$ is the initial state, Γ is the finite stack alphabet with bottom-of-stack symbol $\bot \in \Gamma$, $X = \{x_1, \dots, x_k\}$ is the finite set of registers, $\mathcal{A} = (\mathbb{D}, \oplus_1, \dots, \oplus_m)$ is an algebra over the domain $\mathbb{D}$, and $v_0\colon X \to \mathbb{D}$ are the initial register values. Further, δ is the (deterministic) transition function similar to the transition function of a VPA, $\delta\colon Q \times \Sigma_{\text{int}} \to Q$, $\delta\colon Q \times \Sigma_{\text{call}} \to Q \times \Gamma$, $\delta\colon Q \times \Sigma_{\text{ret}} \times \Gamma \to Q$. The partial register update function ρ is as follows:*

- $\rho\colon Q \times \Sigma_{\text{int}} \times X \to E(\mathcal{A}, X_{prev})$

- $\rho\colon Q \times \Sigma_{\text{call}} \times X \to E(\mathcal{A}, X_{prev})$
- $\rho\colon Q \times \Sigma_{\text{ret}} \times \Gamma \times X \to E(\mathcal{A}, X_{prev}, X_{match})$

Here the place holders X_{prev} and X_{match} are both copies of X. $E(\mathcal{A}, X_{prev})$ is the set of expressions over $\mathcal{A}$ using the registers X_{prev} and $E(\mathcal{A}, X_{prev}, X_{match})$ is the set of expressions using registers in X_{prev} and X_{match}. Finally, $\mu\colon Q \to E(\mathcal{A}, X)$ is the partial final cost function.

A configuration of M is an element (q, γ, v), where q is a state, $\gamma \in \Gamma^*$ is the stack and $v\colon X \to E(\mathcal{A}, X)$ assigns each register as expression over $\mathcal{A}$. A run $(q_0, \gamma_0, v_0) \dots (q_n, \gamma_n, v_n)$ of M on some input $w \in \Sigma$ starts in $(q_0, \bot, v_0)$. The part of a configuration without the register (q_i, γ_i) is defined as in a VPA. The register values v_i are updated in the following way:

- If $w_i \in \Sigma_{\text{int}}$ or if $w_i \in \Sigma_{\text{call}}$ then for every $x \in X$, $v_i(x)$ is an expression obtained by substituting $v_{i-1}(y)$ in every occurrence of $y \in X$ in $\rho(q_i, w_i, x)$.
- If $w_i \in \Sigma_{\text{ret}}$ then for every $x \in X$, $v_i(x)$ is the expression obtained by substituting $v_{i-1}(y)$ (resp. $v_j(y)$) for every occurrence of $y \in X_{\text{prev}}$ (resp. $y \in X_{\text{match}}$) in $\rho(q_i, w_i, g, x)$, where $j < i$ is the matching position of the current index i and $g \in \Gamma$ is the top symbol of the stack content γ_i.

The semantics of a CVPA M is a function $F_M\colon \Sigma^* \to \mathbb{D}$ which is defined on words from $\hat{\Sigma}^*$ as the evaluation of the expression $\mu(q_n)$ by substituting the expression $v_n(y)$ for every occurrence of $y \in X$ in the expression $\mu(q_n)$. It is undefined if $\mu(q_n)$ is undefined. The set of all functions F_M computable by some CVPA M over an algebra $\mathcal{A}$ is denoted by CVPA($\mathcal{A}$). This completes the formal definition of CVPAs.

Remark 4. When specifying register updates we follow the following convention. If e.g. q is a state, a a letter and x a register, then $\rho(q, a, x) = x + x_{match}$ could be a register update rule. If on the right side a plain variable name appears it is meant to be part of X_{prev}. Variable names of the form x_{match} are part of X_{match}.

In the case of CRA interesting and natural subclasses have been defined by imposing a *copyless* restriction (CCRA). For CVPA we can also define a notion of copyless. The simple idea is that each register value may only be used once to calculate a new register value.

Definition 5 (Copyless CVPA (CCVPA)). *A CVPA is called copyless if the following conditions hold:*

- *In each register update expression each variable may only occur at most once.*
- *For any $a \in \Sigma_{\text{call}} \cup \Sigma_{\text{int}}$, variable $x \in X$ and state $q \in Q$, there exists at most one $y \in X$ such that $\rho(q, a, y)$ is an expression containing x.*
- *For any $b \in \Sigma_{\text{ret}}$, variable $x \in X$, state $q \in Q$ and $\gamma \in \Gamma$, there exists at most one $y \in X$ such that $\rho(q, b, \gamma, y)$ is an expression containing x.*
- *For any word $uavb \in \Sigma^*$ where avb is well-matched and $a \in \Sigma_{\text{call}}$ and $b \in \Sigma_{\text{ret}}$ let q_u (resp. q_{ua}, resp. q_{uav}) be the state after reading u (resp. ua, resp. uav), then if a variable x does not occur in $\rho(q_u, a, y)$ for all y there may exist one variable z such that $\rho(q_{uav}, b, \gamma, z)$ contains x_{match}, where $\delta(q_u, a) = (q_{ua}, \gamma)$.*

3.1 Examples

Example 6. Let $\hat{\Sigma} = \{a, b\}$, where $\Sigma_{\text{call}} = \{a\}, \Sigma_{\text{ret}} = \{b\}, \Sigma_{\text{int}} = \emptyset$. Consider the function $f\colon \{a, b\}^* \to \mathbb{N}$ which assigns each prefix of a well-matched word its maximal stack height of VPAs reading the word, e.g. $f(aababababaa) = 3$. This function can be implemented by a CVPA over $(\mathbb{Z}, +c, \max)$, where $+c$ is an operation only allowing adding a constant to a register. The underlying automaton is just a one-state automaton accepting everything. Besides that we need two registers, so $X = \{r, s\}$; further $v_0(r) = v_0(s) = 0$. The register update function is $\rho(q, a, r) = r + 1$, $\rho(q, b, r) = r - 1$ and $\rho(q, a, s) = \rho(q, b, s) = \max(s, r)$. The final output is $\mu(q) = s$.

Example 7. Arithmetic formulas over $(\mathbb{Z}, +, \times)$ can be evaluated by a CCVPA$(\mathbb{Z}, +, \times)$ machine. An arithmetic formula is e.g. $(-1 \times ((1 + 1) \times (1 + 1 + 1)))$ and it evaluates to -6. For the rest of the example we assume for convenience that the formulas are maximally parenthesized, i.e. we have $(1 + (1 + 1))$ instead of $(1 + 1 + 1)$. We can understand strings that are formulas as visibly words over the alphabet $\{-1, 1, (,), +, \times\}$, where (is a push letter,) a pop letter and $-1, 1, +$, and $\times$ are internal letters. However for the rest we write [and] instead of (and) for better readability.

We now give a formal description of the CVPA $M = (\hat{\Sigma}, Q, q_0, \Gamma, \delta, X, \mathcal{A}, v_0, \rho, \mu)$ evaluating formulas. There are states $Q = \{q_0, q_+, q_\times\}$ and as stack alphabet we introduce the symbols P, T and I (used to indicate that the push happened on the $q_+, q_\times, q_0$, resp.). The transition function is as follows:

- transitions from q_0: $\delta(q_0, [) = (q_0, I)$, $\delta(q_0, 1) = \delta(q_0, -1) = \delta(q_0,], I) = \delta(q_0,], P) = \delta(q_0,], T) = q_0$, $\delta(q_0, +) = q_+$, $\delta(q_0, \times) = q_\times$,
- transitions from q_+ on call and internal letters: $\delta(q_+, [) = (q_0, P)$, $\delta(q_+, 1) = \delta(q_+, -1) = q_0$,
- transitions from $q_\times$ on call and internal letters: $\delta(q_\times, [) = (q_0, T)$, $\delta(q_\times, 1) = \delta(q_\times, -1) = q_0$.

The initial value for x is 0, so $v_0(x) = 0$. The register update function can be described as follows: $\rho(q_0, [, x) = 0$, $\rho(q_0, 1, x) = 1$, $\rho(q_0, -1, x) = -1$, $\rho(q_0, +, x) = \rho(q_0, \times, x) = x$ and $\rho(q_0,], I, x) = x$, $\rho(q_0,], P, x) = x + x_{\text{match}}$ and $\rho(q_0,], T, x) = x \times x_{\text{match}}$. The other states: $\rho(q_+, [, x) = \rho(q_\times, [, x) = 0$, $\rho(q_+, 1, x) = x + 1$, $\rho(q_+, -1, x) = x + (-1)$, $\rho(q_\times, 1, x) = x \times 1$, $\rho(q_\times, -1, x) = x \times (-1)$. Finally we let $\mu(q_0) = x$.

Remark 8. Note that the automaton is copyless. In particular, while pushing we do not use x on the right side for the immediate update; it is only used when the matching return letter is read. Note also that this construction can be easily generalized from $\mathcal{A} = (\mathbb{Z}, +, \times)$ to arbitrary $\mathcal{A}$.

3.2 Language Theoretic Properties of CVPAs

In this section we study some language theoretic properties of CVPAs. The first simple observation is that by definition CVPAs (copyless CVPAs) generalize CRAs (copyless CRAs, resp.), just by not using the stack. We note this as an observation explicitly.

Observation 9. *Let $f : \Sigma^* \to \mathbb{D}$ be a function computed by a CRA over an algebra $\mathcal{A}$, then it can also be computed by a CVPA over the same algebra.*

To see that CVPAs generalize VPAs, we first prove a closure property of CVPAs. Given f and g in CVPA($\mathcal{A}$) and a visibly pushdown language L, then we define the function `if` L `then` f `else` g as mapping w to $f(w)$ if $w \in L$ and to $g(w)$ otherwise.

Lemma 10. *If $f, g \in$ CVPA($\mathcal{A}$) then* `if` L `then` f `else` g *is in* CVPA($\mathcal{A}$).

A similar result was proved for CRAs by [4]. Our result can be thought of a as a generalization to CVPAs. We can use the previous closure property to attain the characteristic function: `if` L `then` 1 `else` 0.

This gives that the characteristic function of L can be computed by a CVPA over the algebra $\mathbb{B}$. This therefore gives us the following corollary.

Corollary 11. *CVPA generalize VPA, i.e. for a given visibly pushdown language L, its characteristic function is in* CVPA($\mathbb{B}$).

Lemma 12. *Let f_1, f_2 be two functions over the same visible alphabet computable in* CVPA($\mathcal{A}$) *and $\oplus$ be a binary operator from $\mathcal{A}$, then $f_1 \oplus f_2 \in$* CVPA($\mathcal{A}$).

3.3 Relationship Between CVPA($\mathbb{Z}, +c$) and CCVPA($\mathbb{Z}, +$)

In [4] the relationship between the copyless restriction and restrictions on the multiplication was investigated. In the case of CRA it holds that CRA($\mathbb{Z}, +c$) = CCRA($\mathbb{Z}, +$). Unfortunately this is not true for CVPA which can be seen by the following separating example. Let Σ consist of the letters $(,),], a$, where (is a call letter, $),]$ are return letters and a is an internal letter. Consider all well-matched expressions over Σ. Let f be the function counting the number of a's which are immediately enclosed by (and). E.g. on $((aa(a)a)aa]$ the function f evaluates to 4 and on $(a(aa(aaa))]$ it evaluates to 5.

Lemma 13. *The function f defined above is in* CCVPA($\mathbb{Z}, +$) *but is not in* CVPA($\mathbb{Z} + c$).

It is easy to see that f has a CCVPA($\mathbb{Z}, +$). To prove that it does not have a CVPA($\mathbb{Z}, +c$), we make use of the fact that if the values can only be incremented or decremented by a constant, then the different configurations that all the registers can reach is limited, thereby proving the lower bound.

4 Comparing the Power of #VPA and CVPA

There are counting variants of nondeterministic VPA and NFA, called #VPA and #NFA. For a fixed VPA (resp. an NFA), the semantics then is a function $\Sigma^* \to \mathbb{N}$ which assigns each input the number of accepting runs. In [17] it

was shown that #VPA is #$\mathbf{NC}^1$-complete, whereas #NFA is only complete for counting paths in bounded-width branching programs, which may be a weaker class [14]. This is interesting since their Boolean counterparts are known to have equal power. We first prove that the class of functions $\mathrm{CRA}(\mathbb{N}, +)$ and #NFA are equal. One direction of this is similar to [4]. We also prove the other direction. To the best of our knowledge this result has not been formally proved in any other work.

Lemma 14. *The set of functions* #NFA *equals* $\mathrm{CRA}(\mathbb{N}, +)$.

In the case of #VPA we need a CVPA with multiplication. Formally,

Lemma 15. *The functions in #VPA can be expressed as CVPA over* $(\mathbb{N}, +, \times)$ *which only use multiplication between a just computed register value and a register value from the stack, i.e. between* X_{prev} *and* X_{match} *variables.*

It remains open whether #VPA equals $\mathrm{CVPA}(\mathbb{N}, +)$.

A major motivation for considering cost register models are weighted automata. Weighted automata are NFA where each transition is assigned a weight. Weights are part of an algebra $(\mathbb{D}, \otimes, \odot)$ and in a run all weights are multiplied via $\otimes$. Then the result of the computation is the $\odot$-product of all runs. We consider now the example $(\mathbb{Z}, +, \min)$, which serves a good intuition for the usage of weighted automata as devices quantitatively searching for a "cheapest" run.

Consider a natural weighted variant of VPA over $(\mathbb{Z}, +, \min)$, in which each transition of δ is labeled with a weight and at the end of a computation the weight of the path with the smallest weight is the output.

Lemma 16. *Each weighted VPA function can be expressed as a CVPA over* $(\mathbb{Z}, +c, \min)$. *The resulting automaton does not store register values on the stack.*

5 Complexity Theoretic Results

In this section we present complexity theoretic results about CVPAs and CCVPAs over different algebras. It is known that the membership problem of regular and the visibly pushdown languages is $\mathbf{NC}^1$-complete [9,16]. It is also known that computing a certain bit of the image of a function in $\mathrm{CCRA}(\mathbb{Z}, +)$ or $\mathrm{CCRA}(\Gamma^*, \circ)$ is $\mathbf{NC}^1$-complete [3]. $\mathrm{CRA}(\mathbb{Z}, +)$ is Gap$\mathbf{NC}^1$-complete [3]. $\mathrm{CCRA}(\mathbb{Z}, +, \times)$ is contained in Gap$\mathbf{NC}^1$ [3]. #NFA is #BWBP-complete [14] and #VPA is #$\mathbf{NC}^1$-complete [17]. We add to this landscape a few more complexity results by studying CVPAs using the circuit complexity lense.

Theorem 17. *The set* $\mathrm{CVPA}(\mathbb{Z}, +)$ *is Gap*$\mathbf{NC}^1$*-complete.*

We are given a CVPA M over $(\mathbb{Z}, +)$ and will show that F_M is in Gap$\mathbf{NC}^1$. The completeness then follows because of the fact that CRA over $(\mathbb{Z}, +)$ are already Gap$\mathbf{NC}^1$-complete.

Theorem 18. *The set* CCVPA$(\mathbb{Z}, +, \times)$ *is hard for Gap*$\mathbf{NC}^1$.

Proof. To prove the above statement, we need to show that any maximally bracketed arithmetic formula over $(\mathbb{Z}, +, \times)$ with inputs from the set $\{-1, 0, 1\}$ can be evaluated by CCVPA$(\mathbb{Z}, +, \times)$, as this is a Gap$\mathbf{NC}^1$ hard problem. Recall that the automaton described in Example 7 exactly performed this. □

Note that a similar Gap$\mathbf{NC}^1$ hardness is not known for CCRA$(\mathbb{Z}, +, \times)$.

Theorem 19. *Let f be a function computed by a* CVPA$(\mathbb{Z}, +c)$ *then the ith bit of the output of f can be computed in* $\mathbf{NC}^1$.

Proof (Sketch). Let $M = (\hat{\Sigma}, Q, q_0, \Gamma, \delta, X, \mathcal{A}, v_0, \rho, \mu)$ be a CVPA over $(\mathbb{Z}, +c)$. First thing to note is that computing the height of the stack at any step can be done in TC^0. (See for instance [17].) Let us denote the circuit that computes the height of the stack (more precisely, it decides whether the jth bit of the height reached after reading i bits of the input is 1 or not) as Height_n. Also, as membership testing for VPLs is in $\mathbf{NC}^1$ checking whether after having read $w_1 \dots w_i$ the state reached is q or not can be done in $\mathbf{NC}^1$. Let us call the circuit which does this for length n inputs the circuit State_n. Therefore, an entire run of the underlying VPA of M can be computed in $\mathbf{NC}^1$ (by creating $|w|$ many copies of $\mathrm{State}_{|w|}$, one for each $1 \leq i \leq |w|$).

On $w = w_1 \dots w_n$ suppose the run of the machine has the following states $q_0, q_1, \dots, q_n$ and say $\mu(q_n) = x$ then we say that x is the *relevant register* at the nth step. In general, for $i < n$ if x was relevant at $i + 1$th step and at step i $\rho(q_{i-1}, w_i, x) = y + c$ was the register update function, then we say that y is the relevant register at step $i - 1$ and c is the relevant constant at step $i - 1$. Our proof proceeds in the following two steps: We first observe that for a given input the relevant registers and relevant constants can be computed in $\mathbf{NC}^1$. We then show that once we have this information, computing the final value of the register can be done in $\mathbf{NC}^1$. □

References

1. Allender, E.: Arithmetic circuits and counting complexity classes. In: Krajek, J. (ed.) Complexity of Computations and Proofs, Quaderni di Matematica (2004)
2. Allender, E., Arvind, V., Mahajan, M.: Arithmetic complexity, kleene closure, and formal power series. Technical report (1997)
3. Allender, E., Mertz, I.: Complexity of regular functions. In: Dediu, A.-H., Formenti, E., Martín-Vide, C., Truthe, B. (eds.) LATA 2015. LNCS, vol. 8977, pp. 449–460. Springer, Heidelberg (2015)
4. Alur, R., D'Antoni, L., Deshmukh, J.V., Raghothaman, M., Yuan,Y.: Regular functions, cost register automata, and generalized min-cost problems. CoRR, abs/1111.0670 (2011)
5. Alur, R., D'Antoni, L., Deshmukh, J.V., Raghothaman, M., Yuan, Y.: Regular functions and cost register automata. In: 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, 25–28 June 2013, pp. 13–22 (2013)

6. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, 13–16 June 2004, pp. 202–211 (2004)
7. Barrington, D., Therien, D.: Finite monoids and the fine structure of NC1. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, STOC 1987, pp. 101–109, New York, NY, USA. ACM (1987)
8. Barrington, D.A.: Bounded-width polynomial-size branching programs recognize exactly those languages in NC1. In: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing, STOC 1986, pp. 1–5, New York, NY, USA. ACM (1986)
9. Mix Barrington, D.A., Compton, K.J., Straubing, H., Thérien, D.: Regular Languages in NC^1. J. Comput. Syst. Sci. **44**(3), 478–499 (1992)
10. Bédard, F., Lemieux, F., McKenzie, P.: Extensions to Barrington's M-program model. Theor. Comput. Sci. **107**(1), 31–61 (1993)
11. Buss, S.R.: The boolean formula value problem is in ALOGTIME. In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA, pp. 123–131 (1987)
12. Buss, S.R., Cook, S.A., Gupta, A., Ramachandran, V.: An optimal parallel algorithm for formula evaluation. SIAM J. Comput. **21**(4), 755–780 (1992)
13. Caralp, M., Reynier, P.-A., Talbot, J.-M.: Visibly pushdown automata with multiplicities: finiteness and k-boundedness. In: Yen, H.-C., Ibarra, O.H. (eds.) DLT 2012. LNCS, vol. 7410, pp. 226–238. Springer, Heidelberg (2012)
14. Caussinus, H., McKenzie, P., Thérien, D., Vollmer, H.: Nondeterministic NC^1 computation. J. Comput. Syst. Sci. **57**(2), 200–212 (1998)
15. Cook, S.A.: A taxonomy of problems with fast parallel algorithms. Inf. Control **64**(13), 2–22 (1985). International Conference on Foundations of Computation Theory
16. Dymond, P.W.: Input-driven languages are in log n depth. Inf. Process. Lett. **26**(5), 247–250 (1988)
17. Krebs, A., Limaye, N., Mahajan, M.: Counting paths in VPA is complete for $\#NC^1$. Algorithmica **64**(2), 279–294 (2012)
18. Lange, K.-J.: Complexity and structure in formal language theory. Fundam. Inf. **25**(3,4), 327–352 (1996)
19. McKenzie, P., Péladeau, P., Therien, D.: NC1: the automata-theoretic viewpoint. Comput. Complex. **1**(4), 330–359 (1991)
20. Mehlhorn, K.: Pebbling moutain ranges and its application of dcfl-recognition. In: Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, 14–18 July 1980, Proceedings, pp. 422–435 (1980)
21. Venkateswaran, H.: Properties that characterize LOGCFL. J. Comput. Syst. Sci. **43**(2), 380–404 (1991)
22. Vollmer, H.: Introduction to Circuit Complexity - A Uniform Approach. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (1999)