Electronic Colloquium on Computational Complexity, Report No. 103 (2010)

Counting paths in VPA is complete for $\#\mathsf{NC}^1$

Andreas Krebs[1], Nutan Limaye[2], and Meena Mahajan[3]

[1] University of Tübingen, Germany. mail@krebs-net.de
[2] Tata Institute of Fundamental Research, Mumbai 400 005, India. nutan@tcs.tifr.res.in
[3] The Institute of Mathematical Sciences, Chennai 600 113, India. meena@imsc.res.in

Abstract. We give a $\#\mathsf{NC}^1$ upper bound for the problem of counting accepting paths in any fixed visibly pushdown automaton. Our algorithm involves a non-trivial adaptation of the arithmetic formula evaluation algorithm of Buss, Cook, Gupta, Ramachandran ([8]). We also show that the problem is $\#\mathsf{NC}^1$ hard. Our results show that the difference between $\#\mathsf{BWBP}$ and $\#\mathsf{NC}^1$ is captured exactly by the addition of a visible stack to a nondeterministic finite-state automata.

1 Introduction

We investigate the complexity of the following problem: Fix any visibly pushdown automata V. Given a word w over the input alphabet of V, compute the number of accepting paths that V has on w. We show that this problem is complete for the counting class $\#\mathsf{NC}^1$.

The class $\#\mathsf{NC}^1$ was first singled out for systematic study in [9], and has been studied from many different perspectives; see [1, 9, 6, 5]. It consists of functions from strings to numbers that can be computed by arithmetic circuits (using the operations $+$ and $\times$ and the constants $0, 1$) of polynomial size and logarithmic depth. Equivalently, these functions compute the number of accepting proof trees in a Boolean NC^1 circuit (a polynomial size logarithmic depth circuit over $\vee$ and $\wedge$). It is known that characteristic functions of Boolean NC^1 languages can be computed in $\#\mathsf{NC}^1$ and that functions in $\#\mathsf{NC}^1$ can be computed in deterministic logspace. It is also known that functions in $\#\mathsf{NC}^1$ can be computed by Boolean circuits of polynomial size and $O(\log n \log^* n)$ depth; that is, almost in Boolean NC^1. An analogue of Barrington's celebrated thereom ([4]) stating that Boolean NC^1 equals languages accepted by families of bounded-width branching programs BWBP almost goes through here: functions computed by arithmetic BWBP, denoted $\#\mathsf{BWBP}$, are also computable in $\#\mathsf{NC}^1$, and $\#\mathsf{NC}^1$ functions are expressible as the difference of two $\#\mathsf{BWBP}$ functions. All attempts so far to remove this one subtraction and place $\#\mathsf{NC}^1$ in $\#\mathsf{BWBP}$ have failed.

ISSN 1433-8092

A nice characterization of #BWBP, extending Barrington's result for the Boolean case, is in terms of branching programs over monoids, and yields the following ([9]): there is a fixed NFA (nondeterministic finite-state automaton) N such that any function f in #BWBP can be reduced to counting accepting paths of N. In particular, $f(x)$ equals the number of accepting paths of N on a word $g(x)$ that is a projection of x (each letter in $g(x)$ is either a constant or depends on exactly one bit of x) and is of size polynomial in the length of x. There has been no similar characterization of $\#\mathsf{NC}^1$ so far (though there is a characterization of its closure under subtraction GapNC^1, using integer matrices of constant dimension). Our result does exactly this; the hardness proof shows that there is a fixed VPA (visibly pushdown automaton) V such that any function f in $\#\mathsf{NC}^1$ can be reduced via projections to counting accepting paths of V, and the algorithm shows that any #VPA function (the number of accepting paths in any VPA) can be computed in $\#\mathsf{NC}^1$. Thus, the difference (if any) between #BWBP and $\#\mathsf{NC}^1$, which is known to vanish with one subtraction, is captured exactly by the extension of NFA to VPA.

What exactly are visibly pushdown automata? These are pushdown automata (PDA) with certain restrictions on their transition functions. (They are also called input-driven automata, in some of the older literature. See [14, 15, 10, 3].) There are no ϵ moves. The input alphabet is partitioned into call, return and internal letters. On a call letter, the PDA must push a symbol onto its stack, on a return letter it must pop a symbol, and on an internal move it cannot access the stack at all. While this is a severe restriction, it still allows VPA to accept non-regular languages (the simplest example is $a^n b^n$). At the same time, VPA are less powerful than all PDA; they cannot even check if a string has an equal number of a's and b's. In fact, due to the visible nature of the stack, membership testing for VPA is significantly easier than for general PDA; it is known to be in Boolean NC^1 ([10]). In other words, as far as membership testing is concerned, VPA are no harder than NFA.

However, the picture changes where counting accepting paths is concerned. An obvious upper bound on #VPA functions is the upper bound for #PDA functions. It is tempting to speculate that since membership testing for VPA and NFA have the same complexity, so does counting. This was indeed claimed, erroneously, in [12]; the subsequent version in [13] retracted this claim and showed an upper bound of LogDCFL for #VPA functions. In this paper, we improve this upper bound to $\#\mathsf{NC}^1$, and show via a hardness proof that improving it to #BWBP or #NFA would imply #BWBP$= \#\mathsf{NC}^1$. Our main results are:

Theorem 1. $\#\mathsf{VPA} \subseteq \#\mathsf{NC}^1$.
For every fixed VPA V*, there is a family of polynomial-size logarithmic depth bounded fanin circuits over* $+$ *and* $\times$ *that computes, for each word* w*, the number of accepting paths of* V *on* w*.*

Theorem 2. $\#\mathsf{NC}^1 \leq \#\mathsf{VPA}$.
There is a fixed VPA V *such that for any function family* $\{f_n\}$ *in* $\#\mathsf{NC}^1$*, there is a uniform reduction (via projections)* π *such that for each word* w*,* $f(w)$ *equals the number of accepting paths of* V *on* $\pi(w)$*.*

Combining the two results, we see that branching programs over VPA characterize $\#\mathsf{NC}^1$ functions. Using notation from [13, 9]:

Corollary 1. $\#\mathsf{NC}^1 = \#\mathsf{BP}\text{-}\mathsf{VPA}$

Here is a high-level description of how we achieve the upper bound. In [7], Buss showed that Boolean formulas can be evaluated in NC^1. In [8], Buss, Cook, Gupta, and Ramachandran extended this to arithmetic formulas over semi-rings. We use the algorithm of [8], but not as a black-box. We show that counting paths on a word in a VPA can be written as a formula in a new algebra which is not a semi-ring. However, the crucial way in which semi-ring properties are used in [8] is to assert that for any specified position (called scar) in the formula, the final value is a linear function of the value computed at the scar. We show that this property holds even over our algebra, because of the behaviour of VPA. Thus the strategy of [8] for choosing scar positions can still be used to produce a logarithmic depth circuit, where each gate computes a constant number of operations over this algebra. We also note that in our algebra, basic operations are in fact computable in $\#\mathsf{NC}^0$ (constant-depth constant fanin circuits over $+$ and $\times$). Thus the circuit produced above can be implemented in $\#\mathsf{NC}^1$.

The rest of this paper is organised as follows. In Section 2 we set up the basic notations required for our main result. In Section 3 we present an overview of the arithmetic formula evaluation algorithm of [8], highlighting the points where we will make changes. Section 4 describes our adaptation of this algorithm, placing $\#\mathsf{VPA}$ in $\#\mathsf{NC}^1$. In Section 5, we show that $\#\mathsf{VPA}$ functions are hard for $\#\mathsf{NC}^1$.

2 Preliminaries

Definition 1 (Visibly pushdown automaton). *A visibly pushdown automaton on finite words over a tri-partitioned alphabet* $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$

is a tuple $V = (Q, Q_I, \Gamma, \delta, Q_F)$ where Q is a finite set of states, $Q_I \subseteq Q$ is a set of initial states, Γ is a finite stack alphabet that contains a special bottom-of-stack symbol $\perp$, δ is the transition function $\delta \subseteq (Q \times \Sigma_c \times Q \times \Gamma) \cup (Q \times \Sigma_r \times \Gamma \times Q) \cup (Q \times \Sigma_i \times Q)$, and $Q_F \subseteq Q$ is a set of final states. The letters in Σ_c, Σ_r, and Σ_i are called call letters, return letters, and internal letters, respectively.

The transitions of the VPA are of the form $p \xrightarrow{a} qX$ or $pY \xrightarrow{b} q$ or or $p \xrightarrow{c} q$, where $p, q \in Q$, $X, Y \in \Gamma$, and $\alpha \in \Sigma_c$ is a call letter, $\beta \in \Sigma_r$ is a return letter, $\nu \in \Sigma_i$ is an internal letter. The technical definition in [3] also allows pop moves on an empty stack, with a special bottom-of-stack marker $\perp$. However, we will not need such moves because of *well-matchedness*, discussed below.

Definition 2 (#VPA). *A function $f : \Sigma^* \longrightarrow \mathbb{N}$ is said to be in #VPA if there is a VPA V over the alphabet Σ such that for each $w \in \Sigma^*$, $f(w)$ is exactly the number of accepting paths of V on w.*

Without loss of generality, we can assume that there are no internal letters. (If there are, then design another VPA such that it has as many new extra call and return letters, stack letters, and states as the number of internal letters. Replace every internal letter ν by a string $\alpha\beta$ where α is a call letter and β a return letter added for ν.)

We say that a string is *well-matched* if the VPA never sees a return letter when its stack is empty, and at the end of the word its stack is empty. It can be assumed that all strings are well-matched; in [13], there is a conversion from VPA V to VPA V', and a reduction (computable in NC^1) from inputs w of V to inputs w' of V' such that the number of accepting paths is preserved.

Our Theorem 1 places #VPA functions in $\#\mathsf{NC}^1$. But $\#\mathsf{NC}^1$ is a class of functions from $\{0, 1\}^*$ to $\mathbb{N}$, while VPA may have arbitrary input alphabets. Using standard terminology (see for instance [2, 9]), we assume that the leaves of the $\#\mathsf{NC}^1$ circuits can be labeled by predicates of the form $[w_i = a]$.

For hardness, we use the notion of reductions via projections.

Definition 3 (Projections, [9]). *A function $f : \Sigma^* \longrightarrow \Delta^*$ is a projection if for each $x \in \Sigma^*$, each letter in $f(x)$ is either a constant or depends on exactly one letter of x.*

For a definition of uniformity in projections, see [9].

3 An overview of the BCGR algorithm

The algorithm of [8] for arithmetic formula evaluation over commutative semirings $(S, +, \cdot)$ builds upon Brent's recursive evaluation method [6], but uses a pebbling game to make the construction uniform and oblivious. The recursive strategy is as follows:

For formula ϕ let A be the value of ϕ, and for a position j within it, let ϕ_j be the formula rooted at j and let A_j be the value of ϕ_j. Let $A(j, X)$ be the function corresponding to the formula ϕ with the subformula ϕ_j replaced by the indeterminate X. We say ϕ is scarred at j. Then $A(j, X) = B \cdot X + C$, and to compute the value A, we recursively determine B, C, and the correct value of X (that is, A_j). The recursive procedure ensures that there is at most one scar at each stage. Thus while considering $A(j, X)$, the next scar is always chosen to be an ancestor of ϕ_j. It is shown in [8] that there is a way of choosing scars such that the recursion terminates in $O(\log |\phi|)$ rounds. The main steps of the algorithm of [8] can thus be stated as follows:

1. Convert the given formula ϕ' to an equivalent formula represented in post-fix form, with the longer operand of each operator appearing first in the expression. Call this PLOF (Post-Fix Longer Operand First). Pad the formula with a unary identity operator, if necessary, so that its length is a power of 2. Let ϕ be the resulting formula.
2. Construct an $O(\log |\phi|)$ depth fanin 3 "circuit" $\mathcal{C}$, where each "gate" of $\mathcal{C}$ is a constant-size program, or a block, associated with a particular sub-formula. The top-most block is associated with the entire formula. A block associated with interval g of length $4m$ has as its three children the blocks associated with the prefix interval g_1, the centred interval g_2, and the suffix interval g_3, each of length $2m$. Each interval has upto 9 designated positions or sub-formulas, and the block computes the values of the subformulas rooted at these positions. The set of these positions will contain all possible scar positions that are good (that can lead to an $O(\log 4m)$ depth recursion).
3. Describe Boolean NC^1 circuitry that determines the designated positions within each block. This circuitry depends only on the position of the block within $\mathcal{C}$, and on the letters appearing in the associated interval, not on the values computed so far.
4. Using this Boolean circuitry along with the values at the designated positions in the children g_i, compute the values at designated positions for the interval g using $\#\mathsf{NC}^0$ circuits. Plug in this $\#\mathsf{NC}^0$ circuit for each block of $\mathcal{C}$ to get a $\#\mathsf{NC}^1$ circuit.

To handle the non-commutative case, add an operator $\cdot'$. In conversion to PLOF, if the operands of $\cdot$ have to be switched, then replace the operator by $\cdot'$. The actual operator is correctly applied within the $\#\mathsf{NC}^0$ block at the last step.

4 Adaption of the BCGR algorithm

The algorithm in the previous section works for any non-commutative ring. Unfortunately we were not able to find a non-commutative ring in which we can compute the value of a given VPA. So we will define an algebraic structure that uses constant size matrices as its elements and has two operations $\otimes$ and $\odot$, but which is not a ring. For example we do not have the distributivity law in our structure. Then we show that the algorithm of the previous section can be modified to work for our algebraic structure, and give the precise differences.

Let $V = (Q, Q_I, \Gamma, \delta, Q_F)$ be a fixed VPA and let $q = |Q|$.

In describing the NC^1 algorithm for membership testing in VPA, Dymond [10] constructed a formula using operators Ext and $\circ$ described below. In [13] it was shown that this formula can also evaluate the number of paths (and the number was computed using a deterministic auxiliary logspace pushdown machine which runs in polynomial time). Essentially, the formula builds up a $q \times q$ matrix $\hat{M}_w$ for the input word w by building such matrices for well-matched subwords. The (i, j)th entry of $\hat{M}_w$ gives the number of paths from state q_i to state q_j on reading w. (Due to well-matchedness, the stack contents are irrelevant for this number.) For the zero-length word $w = \epsilon$, the matrix is the identity matrix $\hat{I}$. The unary $\mathrm{Ext}_{\alpha\beta}$ operator computes, for any word w, the matrix $\hat{M}_{\alpha w \beta}$ from the matrix $\hat{M}_w$. The binary $\circ$ operator computes the matrix $\hat{M}_{ww'}$ from the matrices $\hat{M}_w$ and $\hat{M}_{w'}$. The formula over these matrices can be obtained from the input word in NC^1 (and in fact, even in TC^0, see [13, 11]). The leaves of the formula all carry the identity matrix $\hat{I} = \hat{M}_\epsilon$.

Lemma 1 ([10, 13]). *Fix a VPA V. For every well-matched word, there is a formula over $\mathrm{Ext}_{\alpha\beta}, \circ$, constructible in NC^1, which computes the $q \times q$ matrix $\hat{M}_w$. The (i, j)th entry of $\hat{M}_w$ is the number of paths from q_i to q_j while reading w.*

The unary operators $\mathrm{Ext}_{\alpha\beta}$ used above are functions mapping $\mathbb{N}^{q\times q} \longrightarrow \mathbb{N}^{q\times q}$. From the definition of the operators,

$$[\mathrm{Ext}_{\alpha\beta}(M)]_{ij} = \sum_{kl} M_{kl} \cdot |\{X \mid q_i \xrightarrow{\alpha} q_k X; \quad q_l X \xrightarrow{\beta} q_j\}|$$

it is easy to see that these functions are linear operators on $\mathbb{N}^{q \times q}$. The linear operators of $\mathbb{N}^{q \times q}$ can be written as $q \times q$ matrices with $q \times q$ matrices as entries, or simply as matrices of size $q^2 \times q^2$. So we can represent every unary $\mathrm{Ext}_{\alpha\beta}$ operator as a $q^2 \times q^2$ matrix. However, the binary $\circ$ operator works with $q \times q$ matrices. To unify these two sizes, we embed the $q \times q$ matrices from Lemma 1 into $q^2 \times q^2$ matrices in a particular way.

We require that for a matrix $\hat{M}$ at some position in Dymond's formula, our corresponding matrix M satisifes the following: $\hat{M}_{ij} = M_{(ij)(oo)}$ for all indices o. (The values at $M_{(ij)(kl)}$ when $k \neq l$ are not important.)

The operators to capture $\mathrm{Ext}_{\alpha\beta}$ and $\circ$ are defined as follows.

Definition 4. *Let $\mathbb{M}$ be the family of $q^2 \times q^2$ matrices over $\mathbb{N}$.*

1. *The matrix I is defined as the "pointwise" identity matrix, i.e. $I_{(ij)(kl)} = 1$ if $i = j \wedge k = l$ and $I_{(ij)(kl)} = 0$ otherwise.*
2. *For each well-matched string $\alpha\beta$ of length 2, the matrix $EXT^{\alpha\beta} \in \mathbb{M}$ is the matrix corresponding to $\mathrm{Ext}_{\alpha\beta}$ and is defined as follows:*

$$EXT^{\alpha\beta}_{(ij)(kl)} = [\mathrm{Ext}_{\alpha\beta}(E_{kl})]_{ij} \,,$$

   where E_{kl} is a $q \times q$ matrix with a 1 at position (k, l) and zeroes everywhere else.
3. *The operator $\otimes : \mathbb{M} \longrightarrow \mathbb{M}$ is matrix multiplication, i.e.*

$$M = S \otimes T \iff M_{(ij)(kl)} = \sum_{u,v} S_{(ij)(uv)} T_{(uv)(kl)}$$

4. *The operator $\odot : \mathbb{M} \longrightarrow \mathbb{M}$ is defined as a "point-wise" matrix multiplication:*

$$M = S \odot T \iff M_{(ij)(kl)} = \sum_{u} S_{(iu)(kl)} T_{(uj)(kl)}$$

5. *The algebraic structure $\mathbb{V}'$ is defined as follows.*

$$\mathbb{V}' = (\mathbb{M}, \otimes, \odot, I, \{EXT^{\alpha\beta}\}_{\alpha \in \Sigma_c, \beta \in \Sigma_r})$$

Now we change Dymond's formula into a formula that works over the algebra $\mathbb{V}'$ and produces a matrix in $\mathbb{M}$. The $\circ$ operator that represented concatenation is now replaced by the new $\odot$ operator defined above. Each unary $\mathrm{Ext}_{\alpha\beta}$ operator with argument ψ is replaced by the sub-formula $EXT^{\alpha\beta} \otimes \psi$. Each leaf matrix $\hat{I}$ is replaced by I.

Let $w = \alpha\alpha\beta\alpha\alpha\beta\beta\beta$ be word over $\{\alpha, \beta\}$, where α is a call letter and β is a return letter. Figure 1 shows Dymond's formula and our translated formula.

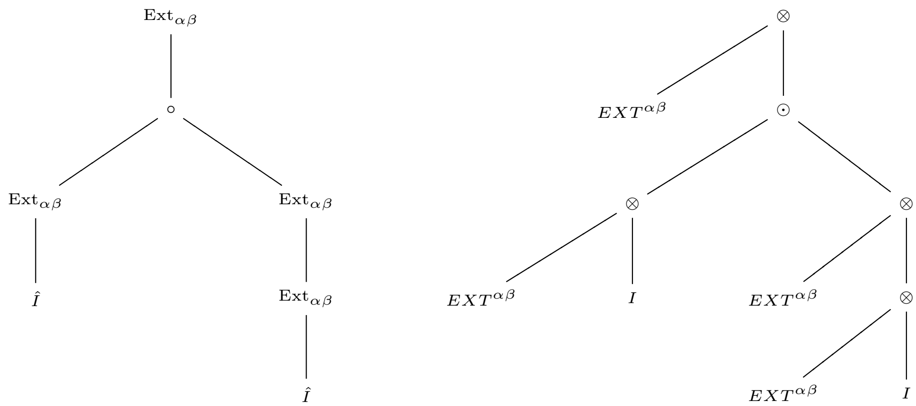

Fig. 1. Dymond's formula (left) and our translated formula (right) for $w = \alpha\alpha\beta\alpha\alpha\beta\beta\beta$

Lemma 2. *Let w be a well-matched word, and let $\hat{\phi}$ be the corresponding formula from Lemma 1. Let ϕ be the formula over $\mathbb{V}'$ obtained by changing $\hat{\phi}$ as described above. Then, for each sub-formula $\hat{\psi}$ of $\hat{\phi}$ and corresponding ψ of ϕ, if $\hat{\psi}$ computes $\hat{P} \in \mathbb{N}^{q\times q}$ and ψ computes $P \in \mathbb{M}$, the following holds:*

$$\forall i, j, o \in [q] \quad \hat{P}_{(ij)} = P_{(ij)(oo)}$$

Proof. We will prove this by induction over the structure of the formula.

Base Case: For the matrix $\hat{I}$ translated to I this is clear.

Case 1: $\hat{\psi} = \mathrm{Ext}_{\alpha\beta}(\hat{\tau})$. Then $\phi = EXT^{\alpha\beta} \otimes \tau$, where τ is the formula corresponding to $\hat{\tau}$. By induction we already know that:

$$\forall i, j, o \in [q] \quad \hat{T}_{(ij)} = T_{(ij)(oo)}$$

where $\hat{\tau}$ computes $\hat{T}$ and τ computes T. We let $\hat{P}$ be the matrix computed by $\hat{\psi}$ and P be the matrix computed by ψ. Then

$$P_{(ij)(kl)} = \sum_{u,v} EXT^{\alpha\beta}_{(ij)(uv)} T_{(uv)(kl)}$$

Hence for all $o \in [q]$ we have $P_{(ij)(oo)} = \sum_{u,v} EXT^{\alpha\beta}_{(ij)(uv)} T_{(uv)(oo)} = \sum_{u,v} EXT^{\alpha\beta}_{(ij)(uv)} \hat{T}_{(uv)}$.

Now we compute $\hat{P}$. We can write the matrix $\hat{T}$ as a sum of its entries and since $\mathrm{Ext}_{\alpha\beta}$ is a linear operator pull the sum and the factors out:

$$\hat{P} = \mathrm{Ext}_{\alpha\beta}(\hat{T}) = \mathrm{Ext}_{\alpha\beta}\left(\sum_{u,v} E_{uv} \cdot \hat{T}_{uv}\right) = \sum_{u,v} \mathrm{Ext}_{\alpha\beta}\left(E_{uv}\right)\hat{T}_{uv}$$

By definition of $EXT^{\alpha\beta}$ we get $P_{(ij)(oo)} = P_{ij}$.

Case 2: $\hat{\psi} = \hat{\sigma} \odot \hat{\tau}$. Then $\psi = \sigma \odot \tau$, where $\hat{\sigma}$ and $\hat{\tau}$ are the formulas corresponding to σ and τ. Let $\hat{S}$, $\hat{T}$, S, T be the values of these formulas. By induction we already know that:

$$\forall i, j, o \in [q] \quad \hat{S}_{ij} = S_{(ij)(oo)} \text{ and } \hat{T}_{ij} = T_{(ij)(oo)}$$

For a fixed $o \in [q]$, the definition of $\odot$ becomes simple pointwise matrix multiplication: $P_{(ij)(oo)} = \sum_u S_{(iu)(oo)} T_{(uj)(oo)} = \sum_u \hat{S}_{iu} \hat{T}_{uj} = \hat{P}_{ij}$.

Hence by induction the result follows. □

From Lemma 2, and since we only did a syntactic local replacement at each node of Dymond's formula, we conclude:

Lemma 3. *For every well-matched word w, there is a formula over $\mathbb{V}'$, constructible in NC^1, which computes a $q^2 \times q^2$ matrix M satisfying the following: For each i, j, the number of paths from q_i to q_j while reading w is $M_{(ij)(oo)}$ for all o.*

For the purposes of using the template of [8], we need to convert our formula to PLOF format. But our operators $\otimes$ and $\odot$ are not commutative. We handle this exactly as in [8], extending the algebra to include the antisymmetric operators. We also need that the length of the formula is a power of 2. In order to handle this, we introduce a unary identity operator $\ominus(S)$.

Definition 5. *The operators $\otimes' : \mathbb{M} \times \mathbb{M} \longrightarrow \mathbb{M}$, $\odot' : \mathbb{M} \times \mathbb{M} \longrightarrow \mathbb{M}$, $\ominus : \mathbb{M} \longrightarrow \mathbb{M}$ are defined as follows: $S \otimes' T = T \otimes S$; $S \odot' T = T \odot S$, $\ominus(S) = S$.*

The algebraic structure $\mathbb{V}$ is the extension of the structure $\mathbb{V}'$ to include the operators $\otimes'$, $\odot'$, and $\ominus$.

Since the conversion to PLOF as outlined in [8] does not depend on the semantics of the structure, we can do the same here. Further, since our formula has a special structure (at each $\otimes$ node, the left operand is a leaf of the form $EXT^{\alpha\beta}$), we can rule out using some operators.

Lemma 4. *Given a formula over $\mathbb{V}$ in infix notation as constructed in Lemma 3, there is a formula over $\mathbb{V}$ in postfix longest operator first PLOF form, constructible in NC^1, which computes the same matrix. The formula over $\mathbb{V}$ in PLOF form will not make use of the operator $\otimes$.*

Proof. We only need to make sure that the resulting formula does not contain the $\otimes$ operator. But since the formula constructed in Lemma 3 uses $\otimes$ only when the left operand is a single matrix $EXT^{\alpha\beta}$, for every subformula $(EXT^{\alpha\beta} \otimes \psi)$ we have $|\psi| \geq 1$, and hence we can assume that the arguments are switched and only $\otimes'$ is used. $\square$

In the following we do not want to allow arbitrary formulas but only formulas that describe the run of the VPA V. We say that a formula over $\mathbb{V}$ is valid if it is constructed as in Lemma 3 and then converted to PLOF as in Lemma 4.

Definition 6. *Let ϕ be a valid formula in $\mathbb{V}$, and let ϕ_j be a sub-formula of ϕ appearing as a prefix in the PLOF representation of ϕ. If we replace ϕ_j by an indeterminate X and obtain a formula ψ over $\mathbb{V}[X]$, we call ψ a formula with a left-most scar. In a natural way a formula with a left-most scar represents a function $f : \mathbb{M} \to \mathbb{M}$ which we call the value of ψ.*

We say a formula/formula with a left-most scar over $\mathbb{V}$ is valid if it is obtained by Lemma 3 and converted to PLOF and then scarred at a prefix subformula. In the following we will only consider valid formulas. Following the algorithm we need to show that we can write every formula with a left-most scar as a fixed expression.

From Lemma 4 it follows that the valid formulas with a left-most scar have the following form.

Lemma 5. *Let ψ be a valid formula in PLOF over $\mathbb{V}$ with a left-most scar X. Then ψ is of the form:*

1. *X*
2. *$\sigma\ominus$, where σ is a valid formula $\mathbb{V}$ with a left-most scar X.*
3. *$\sigma EXT^{\alpha\beta}\otimes'$, where σ is a valid formula $\mathbb{V}$ with a left-most scar X, and $\alpha \in \Sigma_c, \beta \in \Sigma_r$.*
4. *$\sigma\tau\odot$, where σ is a valid formula with a left-most scar X, and τ is a valid formula.*
5. *$\sigma\tau\odot'$, where σ is a valid formula over $\mathbb{V}$ with a left-most scar X, and τ is a valid formula.*

Let ψ be a formula over an algebraic structure $\mathbb{V}$ with a left-most scar X. Then the value of this formula with a scar in general can be represented by a function $f : \mathbb{M} \to \mathbb{M}$. In our case the situation is much simpler; we show that all the functions that occur in our construction can be represented by functions of the form $f(X) = B \cdot X$, where $B \in \mathbb{M}$ is an element of our structure, *i.e.* a $q^2 \times q^2$ matrix of the natural numbers.

(By the definition of $\otimes$ we know that $B \otimes X$ is also given by matrix multiplication $B \cdot X$, but still we use the $\cdot$ operator here to distinguish between the different uses of the semantical equivalent expressions.) Actually the situation is a bit more technical. Since we are only interested in computing the values in the "diagonal", we only need to ensure that these values are computed correctly. In the following lemma we show that the algebraic structure does allow us to represent the computation of these values succinctly. In Lemma 7 we will show that these succinct representations can be computed as required.

Lemma 6. *Let ψ be a valid formula in PLOF over $\mathbb{V}$ with a left-most scar X, and let $f : \mathbb{V} \to \mathbb{V}$ be the value of ψ. Then there is an element $B \in \mathbb{M}$ such that $f(X)_{(ij)(oo)} = (B \cdot X)_{(ij)(oo)}$ for all $i, j, o \in [q]$.*

Proof. We will prove this by induction over the structure of ψ.

1. Let $\psi = X$, then we let $B = id$, and hence $f(X) = X = id \cdot X$.
2. Let $\psi = \sigma \ominus$, where σ is a valid formula with a left-most scar that evaluates to $f'(X) = B' \cdot X$. Then $f(X) = \ominus(B' \cdot X)$. By the definition of $\ominus$, $f(X) = B' \cdot X$.
3. Let $\psi = \sigma EXT^{\alpha\beta} \otimes'$, where σ is a valid formula with a left-most scar that evaluates to $f'(X) = B' \cdot X$. Then $f(X) = EXT^{\alpha\beta} \otimes (B' \cdot X)$, which by associativity of the matrix multiplication can be rewritten as $f(X) = (EXT^{\alpha\beta} \otimes B') \cdot X$. Hence f is of the correct form with $B = (EXT^{\alpha\beta} \otimes B')$.
4. Let $\psi = \sigma\tau\odot$, where σ is a valid formula with a left-most scar that evalutes to $f'(X) = B' \cdot X$, and τ is a valid formula that evaluates to B''. Then $f(X) = (B' \cdot X) \odot B''$. We need to show that we can find a matrix B such that $f(X)$ agrees with $B \cdot X$ at all the "diagonal" positions $(ij)(oo)$. Since B'' is the evalutation of a valid formula we know that $B''_{(ij)(oo)}$ is the same value for all o, and these are the only "important" values of B'' for our computation.

We define B as $B_{(ij)(kl)} = \sum_m B'_{(im)(kl)} B''_{(mj)(11)}$. Then

$$\begin{aligned}
f(X)_{(ij)(oo)} &= \big((B' \cdot X) \odot B''\big)_{(ij)(oo)} \\
&= \sum_m \left(\sum_{u,v} B'_{(im)(uv)} X_{(uv)(oo)} \right) \cdot B''_{(mj)(oo)} \\
&= \sum_{u,v} \left(\sum_m B'_{(im)(uv)} B''_{(mj)(oo)} \right) \cdot X_{(uv)(oo)} \\
&= \sum_{u,v} \big(B_{(ij)(uv)}\big) \cdot X_{(uv)(oo)} \\
&\qquad \text{(since by induction, } B''_{(mj)(oo)} = B''_{(mj)(11)}\text{)} \\
&= (B \cdot X)_{(ij)(oo)} \quad \text{(by definition of } B\text{)}
\end{aligned}$$

5. Let $\psi = \sigma\tau\odot'$, where σ is a valid formula with a left-most scar that evalutes to $f(X) = B' \cdot X$, and τ is a valid formula that evaluates to B''. This case is similar to the previous case: $f(X) = B'' \odot (B' \cdot X)$ and we define B as $B_{(ij)(kl)} = \sum_m B''_{(im)(11)} B'_{(mj)(kl)}$. Then

$$\begin{aligned}
f(X)_{(ij)(oo)} &= \sum_m B''_{(im)(oo)} \cdot \left(\sum_{u,v} B'_{(mj)(uv)} X_{(uv)(oo)} \right) \\
&= \sum_{u,v} \left(\sum_m B''_{(im)(11)} B'_{(mj)(uv)} \right) \cdot X_{(uv)(oo)} \\
&= (B \cdot X)_{(ij)(oo)}
\end{aligned}$$

□

Since we are able to represent every scarred formula by a constant size matrix, we can apply a conversion similar to the BCGR algorithm and end up with a circuit of logarithmic depth. The only thing that remains is to show that computations within the blocks can be computed by $\#\mathsf{NC}^0$ circuits, so that the total circuit will be in $\#\mathsf{NC}^1$. Note that each block is expected to compute for its associated formula the element $B \in \mathbb{V}$ guaranteed by Lemma 6, assuming the corresponding elements at designated subformulas have been computed.

The construction of the blocks is very restricted by the BCGR algorithm. We know that there are only two major cases that happen:

- Either we need to compute a formula ψ with a scar X from the value of the formula ψ with a scar Y and the value of the formula Y with the scar X (here X can be the empty scar),

– or we need to compute a formula ψ with operand one of $\otimes'$ $\odot$, $\odot'$, $\ominus$, and a scar X, where the left argument contains the scar X and we are already given the value of the left argument with the scar X and the value of the right argument.

Hence we need to show that we can compute the operators in our algebraic structure in constant depth, as well as the computations for the matrix representations of our functions as in the proof of Lemma 6.

Lemma 7. *Let $S, T \in \mathbb{M}$. There are $\#\mathsf{NC}^0$ circuits that compute the matrix P, where P is defined in any one of the following ways:*

1. $P = S \cdot T$
2. $P = S \otimes' T$
3. $P = S \odot T$
4. $P = S \odot' T$
5. $P_{(ij)(kl)} = \sum_m S_{(im)(kl)} T_{(mj)(11)}$.
6. $P_{(ij)(kl)} = \sum_m S_{(im)(11)} T_{(mj)(kl)}$.

Proof. This is trivial since all operations use only 2 constant size matrices as inputs and hence are defined by a constant expression over $+, \times$. $\square$

Hence we can replace the blocks by $\#\mathsf{NC}^0$ circuits and obtain a $\#\mathsf{NC}^1$ circuit. But this implies Theorem 1.

5 Hardness

In this section we will show that there is a hardest function in $\#\mathsf{NC}^1$ and that it can be computed by a VPA. Let $\Sigma = \{[,], \bar{]}, +, \times, 0, 1\}$. We will define a function $f : \Sigma^* \to \mathbb{N}$ that is $\#\mathsf{NC}^1$ hard under projections and is computable by a VPA.

Informally speaking, the following language will represent equations with $+, \times$ over the natural numbers, where the operation $+$ is always bracketed by $[$ and $\bar{]}]$. The 2 closing brackets are necessary for the technical reasons.

Definition 7. *Define L to be the smallest language such that:*

1. $0, 1 \in L$,
2. $[u + v\bar{]}] \in L$ *for* $u, v \in L$,
3. $uv \in L$ *for* $u, v \in L$.

Note that L is uniquely determined as the closure of the set of expressions $\{0, 1\}$ under appropriately bracketed $+$ and $\times$.

Also, define the function $f : \Sigma^* \to \mathbb{N}$ by:

1. $f(0) = 0, f(1) = 1$,
2. $f([u + v\bar{]}]) = f(u) + f(v)$ *for* $u, v \in L$,
3. $f(uv) = f(u) \cdot f(v)$ *for* $u, v \in L$,
4. $f(u) = 0$ *for* $u \notin L$.

Please note that the value of $f(w)$ does not depend on the decomposition of w since multiplication on natural numbers is associative.

Lemma 8. *Computing the value of $f(w)$ on words $w \in L$ is $\#\mathsf{NC}^1$ hard under Projections.*

Proof. Given a $\#\mathsf{NC}^1$ circuit we can expand the circuit to a tree and do a traversal of the tree to obtain a formula over the inputs $x_1, \ldots, x_n$ with $+, \times$. It is easy to see that for a fixed circuit, we can get a program that outputs the expression w where $+$ is enclosed by $[, \bar{]}]$, removes all $\times$, and $x_1, \ldots, x_n$ are replaced by 0 or 1 depending on the input x. By the definition $f(w)$ evalutates to the same value as the $\#\mathsf{NC}^1$ circuit for the input x. $\square$

We will design a fixed VPA V. The idea for this VPA is to compute inductively the value of f as paths from one state q_C to q_C, while always keeping a single path from another state q_I to q_I. This allows us to temporarily store a value in the computation, and with the stack of the VPA we use this as a stack for storing values.

We will first give the definition of the VPA and then prove that it computes f on all words of L. It will have a tripartitioned input alphabet $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$ where $\Sigma_c = \{[, +\}$, $\Sigma_r = \{\bar{]},]\}$, and $\Sigma_i = \{0, 1\}$.

We let $V = \{\{q_C, q_I\}, \{q_C\}, \{T_1, T_2\}, \delta, \{q_C\}\}$, where the transition function is defined as (ordered by input letters):

Σ_i	q_C	q_I
1	q_C	q_I
0	$-$	q_I

Σ_c	q_C	q_I
$[$	q_CT_1, q_IT_1	q_IT_2
$+$	q_IT_1	q_CT_1, q_IT_2

Σ_r	q_CT_1	q_IT_1	q_CT_2	q_IT_2
$\bar{]}$	q_C	q_C	$-$	q_I
$]$	q_C	$-$	$-$	q_I

For a better understanding of δ we provide a graphical version of δ (see Figure 2).

We now show that this VPA actually computes f.

Lemma 9. *For all $w \in L$, the number of accepting paths of the* VPA V *is exactly computes $f(w)$.*

Proof. We will prove this by induction on the structure of L. Please note that all words $w \in L$ are well-matched inputs to the VPA, since the

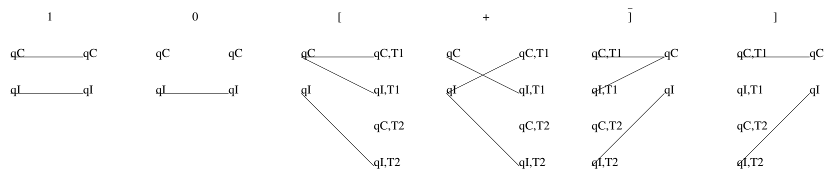

Fig. 2. Graphical version of δ

number of call and return letters is always equal, and the number of return letters never exceeds the number of return letters in any prefix.

We will show that for every word $w \in L$, the number of paths from q_I to q_I is 1, and from q_C to q_C is $f(w)$, and there are no paths from q_C to q_I or from q_I to q_C.

For $w = 0$ and $w = 1$ this is clear by the definition of δ.

Also for $w = uv$ with $u, v \in L$ it is clear that the VPA has $f(w) = f(u)f(v)$ paths for w. And also the other properties of the induction hypothesis are clear.

For $w = [u + v\bar{]}]$ with $u, v \in L$ this requires a small computation. We give a picture below (see Figure 3) with all the paths generated by δ, which should help to check the computation.

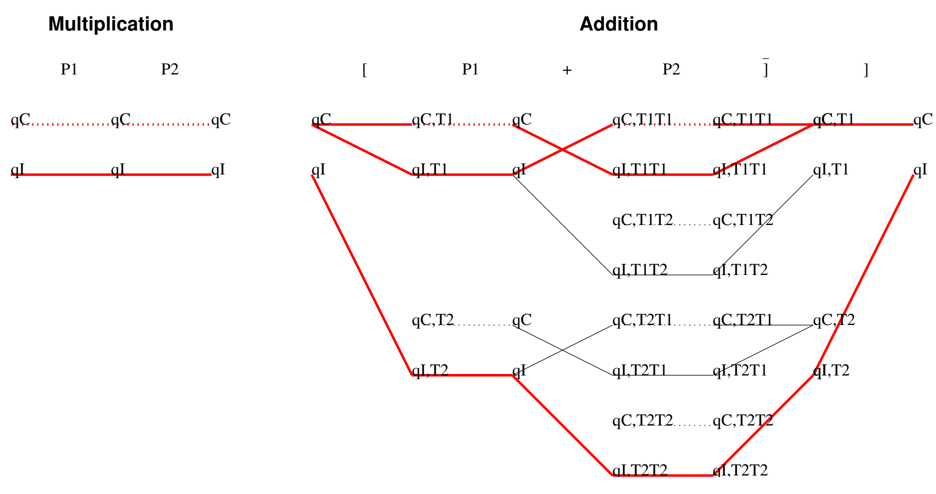

Fig. 3. Paths for uv and $[u + v\bar{]}]$

$\square$

It is also easy to see that L itself can be recognized by an VPA (with the same partition into call and return letters). Since f is 0 outside L, hence there is another VPA that computes f on all of Σ^*.

References

1. E. Allender. Arithmetic circuits and counting complexity classes. In J. Krajicek, editor, *Complexity of Computations and Proofs*, Quaderni di Matematica Vol. 13, pages 33–72. Seconda Universita di Napoli, 2004. An earlier version appeared in the Complexity Theory Column, SIGACT News 28, 4 (Dec. 1997) pp. 2-15.
2. E. Allender, J. Jiao, M. Mahajan, and V. Vinay. Non-commutative arithmetic circuits: depth reduction and size lower bounds. *Theoretical Computer Science*, 209:47–86, 1998.
3. R. Alur and P. Madhusudan. Visibly pushdown languages. In *ACM Symposium on Theoretical Aspects of Programming STOC*, pages 202–211, 2004.
4. D. Barrington. Bounded-width polynomial size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38:150–164, 1989.
5. M. Ben-Or and R. Cleve. Computing algebraic formulas using a constant number of registers. *SIAM Journal on Computing*, 21:54–58, 1992.
6. R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–206, 1974.
7. S. Buss. The Boolean formula value problem is in ALOGTIME. In *ACM Symposium on Theoretical Aspects of Programming STOC*, pages 123–131, 1987.
8. S. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM Journal on Computing*, 21(4):755–780, 1992.
9. H. Caussinus, P. McKenzie, D. Thérien, and H. Vollmer. Nondeterministic NC^1 computation. *Journal of Computer and System Sciences*, 57:200–212, 1998.
10. P. W. Dymond. Input-driven languages are in $\log n$ depth. In *Information processing letters*, pages 26, 247–250, 1988.
11. N. Limaye, M. Mahajan, and A. Meyer. On the complexity of membership and counting in height-deterministic pushdown automata. In *3nd International Computer Science Symposium in Russia CSR, LNCS vol. 5010*, pages 240–251, 2008.
12. N. Limaye, M. Mahajan, and B. V. R. Rao. Arithmetizing classes around NC^1 and L. In *Symposium on Theoretical Aspects of Computer Science STACS, LNCS vol. 4393*, pages 477–488, 2007.
13. N. Limaye, M. Mahajan, and B. V. R. Rao. Arithmetizing classes around NC^1 and L. *Theory of Computing Systems*, 46(3):499–522, 2010.
14. K. Mehlhorn. Pebbling mountain ranges and its application to DCFL recognition. In *7th International Colloquium on Automata, Languages and Programming ICALP*, pages 422–432, 1980.
15. B. von Braunmuhl and R. Verbeek. Input-driven languages are recognized in $\log n$ space. In *Conference on Fundamentals of Computation Theory FCT*, pages 40–51, 1983.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de