Electronic Colloquium on Computational Complexity, Revision 2 of Report No. 19 (2017)

A Unified Method for Placing Problems in Polylogarithmic Depth*

Andreas Krebs, Tübingen University, Germany, Nutan Limaye, IIT Bombay, India,
Michael Ludwig, Tübingen University, Germany.

October 27, 2017

Abstract

In this work we consider the term evaluation problem which is, given a term over some algebra and a valid input to the term, computing the value of the term on that input. In contrast to previous methods we allow the algebra to be completely general and consider the problem of obtaining an efficient upper bound for this problem. Many variants of the problems where the algebra is well behaved have been studied. For example, the problem over the Boolean semiring or over the semiring $(\mathbb{N}, +, \times)$. We extend this line of work.

Our efficient term evaluation algorithm then serves as a tool for obtaining polylogarithmic depth upper bounds for various well-studied problems. To demonstrate the utility of our result we show new bounds and reprove known results for a large spectrum of problems. In particular, the applications of the algorithm we consider include (but are not restricted to) arithmetic formula evaluation, word problems for tree and visibly pushdown automata, and various problems related to bounded tree-width and clique-width graphs.

1 Introduction

Background and motivation. Classically the notion of efficiently solvable problems is defined to be the class of problems for which there are polynomial time algorithms, namely the set of problems in the complexity class **P**. Over the last many decades a fine grained study of classically efficient computation has lead to many interesting subclasses of problems in **P**. One such class is a set of problems solvable using polynomially many processors which run in parallel for at most polylogarithmic time. This class of problems is known to be **NC**.

There are many interesting and fundamental computational problems for which the classical algorithms designed were inherently sequential in nature. Owing to a series of theoretically intriguing and practically relevant discoveries we know **NC** algorithms for some of these problems. That is, a fairly diverse set of problems from the class **P** is known to be in **NC**. This raises a natural question: is **P** the same as **NC**? That is, can any sequential algorithm be turned into an efficient parallel one? In fact, this is one of the very central open questions in computer science. The question has implications to many different practical aspects of computer science such as distributed computing and parallel algorithms for large scale data (see for instance [35, 40, 49]).

A lot of effort has gone into understanding the relative strengths and weaknesses of **P** and **NC**. The study of Boolean and arithmetic circuits and many interesting results proved therein show that some specific subclasses of **NC** are strictly less powerful than **P**. (See for instance [21, 26, 27].) This rich body of work could be thought of as attempts to separate **NC** from **P**. On the other hand, over the last many

*The work was funded by DST-DAAD project no. INT/FRG/DAAD/P-252/2015.

ISSN 1433-8092

years, surprising **NC** upper bounds have been proved for problems which were previously believed to be hard to parallelize. (See for instance [28, 38, 1, 24, 20].)

Our contributions. All these algorithmic advances raise a natural question: what makes a problem in **P** to have an **NC** algorithm? The main goal of this work is to build a theory which attempts to answer this question. Our main contributions are given below:

- We identify similarities between a large number of parallel algorithms. We observe that if a problem has a core *tree-like* structure, then it is amenable to have an **NC** algorithm. We formally define the notion of the tree-like structure and demonstrate the presence of such a structure in a large collection of problems.

- Our second important contribution is to mechanise the process of coming up with a parallel algorithm for any problem that has this tree-like structure. This can be thought of as an algorithmic contribution stemming from our work. We demonstrate the strength of this algorithmic technique by rediscovering many known **NC** upper bounds.

  This is also a technically challenging part of our work. The difficulty arises because we need an algorithm which is independent of the problem and only dependent on the underlying tree-like structure. The structure itself is dependent on the problem: for two seemingly unrelated problem these structures could be different. Therefore, the algorithm has to be general, which makes as few assumptions about the structure as possible.

  One caveat worth mentioning is that in some cases the tree-like structure is clear from the problem definition and in some cases it requires some work to notice this structure. We assume that an expert working on a specific problem may be able to notice this structure easily for the problem of her interest and then choose to use our approach mechanically to obtain an **NC** algorithm for the problem.

Significance. Over the last four decades there has been a lot of work related to design of parallel algorithms for tree-like problems. Given below is a notable and diverse (but not exhaustive) list of problems which have been considered in this literature.

- Boolean and arithmetic term evaluation [9, 11].
- Membership for language classes [36, 34, 17, 2, 31].
- Evaluation of circuits with bounded tree-width [29].
- Courcelle's Theorem and counting [14, 18].
- Computation of maximal cuts in bounded clique-width graphs [48].
- Counting Hamiltonian paths in bounded clique-width graphs [5, 48].

Using our techniques, we reprove the above results. That is, we give a unified way of proving all the above bounds. Moreover, we also consider variants of the above applications and obtain parallel ($\mathbf{NC}^1$, $\mathbf{NC}^2$, $\mathbf{SAC}^1$, $\mathbf{SAC}^2$) upper bounds.

Techniques. Our approach uses algebra as a tool for obtaining the desired abstraction. An algebra $\mathscr{A}$ consists of a set $\mathbb{D}$ and a finite set of operations $\{\oplus_1, \oplus_2, \ldots, \oplus_k\}$, where k is fixed. As mentioned earlier, we focus on coming up with an efficient parallel algorithm for problems with a tree-like structure. For this, we consider the *term evaluation problem*. A term is simply a tree in which the leaves are labelled by the elements of $\mathbb{D}$ and the internal nodes are labelled by the operations. The term evaluation problem deals with evaluating the value of the term for a given assignment of the leaves from the domain $\mathbb{D}$. The main algorithmic and technical contribution of our work is to rewrite the term T as an equivalent term

T' whose inputs are labelled by the elements of $\mathbb{D}$ and the internal nodes (gates) are labelled by the operations of an algebra $\mathscr{F}(\mathscr{A})$, which is an extended algebra derived from $\mathscr{A}$. Moreover, if T has size s then T' has size $\text{poly}(s)$ and depth $O(\log s)$[1].

This result is our primary algorithmic contribution. It may be thought of as a meta theorem for the general term evaluation problem. Using this result we obtain **NC** upper bounds as follows: say Π is a problem for which we need to design an **NC** algorithm. For a given Π, we show that solving Π is equivalent to evaluating a term T_Π over an appropriately defined algebra $\mathscr{A}_\Pi$. Now, using the above result, we get a log-depth term over the operations of $\mathscr{F}(\mathscr{A}_\Pi)$. We then observe that each operation in $\mathscr{F}(\mathscr{A}_\Pi)$ is *easy* to evaluate. Note that for a given problem Π, there may be many $\mathscr{A}_\Pi$ one could design. It is not necessary for every choice of $\mathscr{A}_\Pi$, the corresponding $\mathscr{F}(\mathscr{A}_\Pi)$ has operations which are *easy* to evaluate. This part is sensitive to the choice of $\mathscr{A}_\Pi$. However, the main result stated above is independent of Π.

The known **NC** algorithms can be thought of as algorithms which transform T_Π to T'_Π for a particular Π. What we manage to do here is to obtain a transformation from T to T', irrespective of any specific Π (and hence a specific $\mathscr{A}_\Pi$). This approach allows us to unify many known results by noticing that each had to its core a term evaluation problem over an algebra. The following are the main technical contributions in this result: (i) the algebraic notions defined and used for the abstraction, and (ii) the definition of the extended algebra $\mathscr{F}(\mathscr{A})$. The notion of an extended algebra is intricate and crucial in the algorithm design.

Related work. Our algorithm for the term evaluation problem fits in the long chain of contributions dedicated to the term evaluation problem. The origin of which can be vaguely traced back to the investigation of upper bounds for the Boolean formula value problem. In [37] Lynch studied it first and achieved a log-space upper bound. Subsequently Cook conjectured that this bound is tight [13] which, as we know today, is not (unless log space equals log depth). Earlier, a way to deal with formulas that are very deep trees was investigated by Spira [44]: by a quadratic increase in size, we can balance a Boolean formula. Brent built upon this work [8]. Going from balancing to obtaining an **NC** (in fact $\mathbf{NC}^1$, i.e. log-depth) upper bound is not tough. It is known that if the transformation can be done in $\mathbf{NC}^1$, the evaluation is in $\mathbf{NC}^1$.

Cook and Gupta [23] as well as Ramachandran [42] were the next in line who showed that $\mathscr{O}(\log n \log\log n)$ deep circuits suffice for evaluating formulas. Based on [23], Buss showed an ALOGTIME bound [9] which equals logarithmic depth [43] and is known to be tight. His proof utilized a sophisticated two-player pebbling game. From there on the research proceeded in the direction of broadening the scope of the result. This continued research is always rooted in the work of [23] and [9]. Many other interesting works have contributed to this rich line of research, each solving the term evaluation problem over a specific algebra [39], [17], [33], [31].

Subsequent work. Closely related to our work is a very recent work of Ganardi and Lohrey [22] which uses the notion of algebras, terms and extended algebras (i.e. $\mathscr{A}, T, \mathscr{F}(\mathscr{A})$) and shows optimal upper bounds on the size of the **NC** circuits obtained for some of the problems considered here. This is an interesting piece of follow up work, which improves on the size of the circuits (to $O(n/\log n)$) for some of the problems and uses some of the machinery developed here.

Organization. We give the details regarding the term evaluation problem in Section 3. We present the relevant preliminaries and some definitions in Section 2. Finally the discussion regarding the applications of the term evaluation algorithm is provided in Section 4.

[1] Please refer to [47] for definitions of circuits, size and depth notions for terms and circuits and circuits with gates coming from an algebra. Also, the size can in fact be bounded by $O(s)$, but that is not crucial here.

2 Preliminaries: notations and definitions

As mentioned before, the term evaluation problem that we deal with here is over a very general algebraic structure. In the literature, the term evaluation problem has been studied with respect to specific algebras. However, as our main goal here is to give a unified approach to solve the general term evaluation problem, we define algebraic structures which are as general as possible. We also define circuits (and terms) which *operate* over these algebraic structures and we formalize the notation of semantics for such circuits (and resp. terms).

To the best of our knowledge, the definitions appearing here have not been stated in this form in any other literature in the series of works related to the term evaluation problem. In that sense, they are new. However, some of the definitions are in fact generalizations/abstractions of well-known classical notions.

2.1 Notation

The set $\{1,\ldots,n\}$ is denoted by $[n]$ and $\{i,\ldots,j\}$ by $[i,j]$. The set $\mathbb{N}$ stands for the natural numbers containing 0, $\mathbb{Z}$ for the integers, and $\mathbb{B}$ for the Boolean values $\{\bot,\top\}$. An alphabet, denoted as Σ, is a finite sets of letters. A word $w \in \Sigma^*$ is a finite sequence of letters and hence Σ^* is the set of all words over Σ. The ith letter of w is denoted by $w(i)$. The length of w is denoted by $|w|$. The word of length 0 is denoted by ε. A language is a subset of Σ^*.

2.2 Many-sorted signatures, circuits and terms

Operations and sorts. Below we will deal with operations which get inputs from different domains. The distinct domains which any operation uses are called *sorts*. For example, consider an operation f which has four inputs, say x_1,x_2,x_3,x_4, the first two are Boolean, i.e. $x_1,x_2 \in \mathbb{B}$ and $x_3,x_4 \in \mathbb{N}$. The operation $f(x_1,x_2,x_3,x_4)$ outputs $x_3 \times x_4$ if x_1 AND x_2 is 1 and outputs $\frac{x_3}{x_4+1}$ otherwise (i.e. if x_1 AND x_2 is 0). Unlike a usual Boolean or arithmetic operation, f is more complicated. The inputs of f come from different sorts of domains, i.e. $\mathbb{B}$ and $\mathbb{N}$. The output is over yet another domain, namely $\mathbb{Q}$. Here, $\mathbb{B},\mathbb{N},\mathbb{Q}$ are the three different sorts.

We define, circuits, terms and algebras which use such generalized operations and also define the semantics for them. Towards this, we first define a many-sorted signature, which is a generalization of the notion of arity of a Boolean or arithmetic operation.

Definition 1 (Many-sorted signature)**.** *Given $S \in \mathbb{N}$ different sorts, a many-sorted signature $\boldsymbol{\sigma}$ of an operation is an element of $[S]^* \times [S]$.*

A many-sorted signature $\boldsymbol{\sigma}$ is a pair of a word and a letter (w,a). The word codes the input sorts of the operation. The length of a word $|w|$ is the arity of the operation. E.g. the operation f defined above has the signature $(1122,3)$, where the three sorts are $\mathbb{B},\mathbb{N},\mathbb{Q}$ (numbered in that order). For a operation f with signature $\boldsymbol{\sigma}$, we write $\mathsf{In}_{\boldsymbol{\sigma}}(f)$ to denote w and $\mathsf{Out}_{\boldsymbol{\sigma}}(f)$ to denote a. If we have a set of operations $f_1,\ldots,f_\ell$, with signatures $\boldsymbol{\sigma_1},\ldots,\boldsymbol{\sigma_\ell}$ respectively, we use σ to denote the tuple $(\boldsymbol{\sigma_1},\ldots,\boldsymbol{\sigma_\ell})$ and $\sigma(i)$ to address (w_i,a_i). Also, $\mathsf{In}_\sigma(i,j)$ is the jth letter of $\mathsf{In}_\sigma(f_i)$. If we have an ordering on the operations $f_1,\ldots,f_\ell$ then we simply use $\mathsf{In}_\sigma(i)$ instead of $\mathsf{In}_\sigma(f_i)$. We use $|\sigma|$ to denote the number of different operations, i.e. ℓ.

A single-sorted signature $\boldsymbol{\sigma}$ is one where $|S| = 1$. In this case $\boldsymbol{\sigma}$ corresponds to the classical notion of signature assigning just an arity to the operation. For the sake of brevity, henceforth we will simply say signature instead of many-sorted signature.

We now define a circuit which uses these operations. Suppose a gate F in the circuit is assigned the operation f defined above. Then in such a circuit, one needs to ensure that the first two inputs to F are Boolean, while the last two are from $\mathbb{N}$. That is, only valid gate types feed into one another. We ensure this using the notion of a signature defined above. Formally, we define many-sorted circuits as follows.

Definition 2 (Many-sorted circuit)**.** *For a signature σ, a many-sorted circuit over signature σ of S sorts, n inputs and m outputs is a tuple $C = (V, E, \mathsf{Order}, \mathsf{Gatetype}, \mathsf{Outputgates})$, where (V,E) is a directed acyclic graph,* $\mathsf{Order}\colon E \to \mathbb{N}$ *is an injective map giving an order on the edges,* $\mathsf{Gatetype}\colon V \to [|\sigma|] \cup \{x_1, \ldots, x_n\}$ *assigns a position of the signature or makes it an input gate,* $\mathsf{Outputgates}\colon \{y_1, \ldots, y_m\} \to V$ *makes gates output gates, such that:*

- *If some $v \in V$ has in-degree 0 then $\mathsf{Gatetype}(v) \in \{x_1, \ldots, x_n\}$ or $\mathsf{In}_\sigma(\mathsf{Gatetype}(v)) = \varepsilon$, i.e. it is 0-ary.*
- *If some $v \in V$ has in-degree $k > 0$ then $|\mathsf{In}_\sigma(\mathsf{Gatetype}(v))| = k$, hence it is k-ary.*
- *For all $i \in [n]$ there exists at most one $v \in V$ such that $\mathsf{Gatetype}(v) = x_i$*
- *All successors of an input gate can be assigned a unique sort which is fixed by the successor gates and the signature. Also, $\mathsf{In}_C \subseteq [S]^n$ denotes the sorts of the n input gates and $\mathsf{Out}_C \subseteq [S]^m$ denotes the sorts of the m output gates.*
- *For all $v \in V$, let $v_1, \ldots v_{|\mathsf{In}_\sigma(\mathsf{Gatetype}(v))|}$ be the input gates for v such that $\mathsf{Order}(v_i) \leq \mathsf{Order}(v_j)$ iff $i \leq j$. Then $\mathsf{Out}_\sigma(\mathsf{Gatetype}(v_i)) = \mathsf{In}_\sigma(\mathsf{Gatetype}(v), i)$. If v_i is an input gate then $\mathsf{In}_C(j) = \mathsf{In}_\sigma(\mathsf{Gatetype}(v), i)$ where $\mathsf{Gatetype}(v_i) = x_j$.*

By $\mathsf{Circ}_{\sigma,n,m}$ we denote the set of circuits over σ of n inputs and m outputs.

Terms and circuits are closely related. In general, a term is a circuit which is a tree. However in our setting additionally, a term has no input variables. Instead, the inputs are constants from the domain given as 0-ary operations. One can think of it as the case when all variables have already been assigned a value. Also our terms have only binary operations[2].

We also consider the notion of terms with an unknown. Such terms come into play when we decompose a given term in the term evaluation algorithm. Assume a term is to be evaluated over a domain $\mathbb{D}$ (e.g. $\mathbb{D}$ could be $\mathbb{B}, \mathbb{N}$ or $\mathbb{Q}$) then a term with an unknown evaluates to a map $\mathbb{D} \to \mathbb{D}$. Formally,

Definition 3 (Many-sorted term, many-sorted term with an unknown)**.** *Given a many-sorted circuit T over σ where $m = 1$ and (V,E) is a tree with a degree bounded by two. If $n = 0$ then T is a many-sorted term and if $n = 1$ then T is a many-sorted term with an unknown. By Term_σ we denote the set of terms over the signature[3] σ and by $\mathsf{Term}_\sigma[X]$ we denote the set of terms with an unknown over σ.*

We assume that the term (with or without an unknown) is encoded as a string over a fixed alphabet[4]. We call this a linearization of a term. We now define linearization of terms, which is essentially a way of writing a term as a string over a fixed alphabet[5].

Definition 4 (Linearization of many-sorted terms)**.** *Given a many-sorted term T over a signature σ, the linearization of t is a string $w(T)$ which is a word of $\{(,), \oplus_1, \ldots, \oplus_{|\sigma|}\}^*$. First we define it on the nodes inductively:*

- *If $v \in V$ has in-degree 0 then $w(v) = \oplus_{\mathsf{Gatetype}(v)}$.*
- *If $v \in V$ has in-degree 1 and v' is the predecessor, then $w(T) = (\oplus_{\mathsf{Gatetype}(v)} w(T(v')))$ where $T(v')$ is the maximal subtree of T rooted in v'.*
- *If $v \in V$ has in-degree 2 and v_1, v_2 are the predecessors, then $w(T(v)) = (w(T(v_1)) \oplus_{\mathsf{Gatetype}(v)} w(T(v_2)))$ where $T(v)$ is the maximal subtree of T rooted for some $v \in V$.*

[2] This is no restriction as an n-ary operations can be simulated by a small set of binary operations.

[3] Note that for a term to be over some signature, the signature has to admit 0-ary operations since we need them for the leafs.

[4] It is the usual way of encoding machines as strings.

[5] It is the usual way of encoding machines as strings.

Finally if v is the output gate, then $w(T) = w(v)$.

In the terms with an unknown we set $w(v) = X$ if v is the single input gate, where X is an additional letter. From now on we will not distinguish between terms and their linearizations.

From now on we will not distinguish between terms and their linearizations.

2.3 Semantics: evaluation of circuits and terms without unknown

So far we have defined circuits and terms (with and without unknowns) syntactically. Now we will give semantics for circuits and terms. Informally, the semantics of a circuit is the tuple of functions its outputs compute and that of a term is the value which its output evaluates to. We make this notion formal by introducing the notion of a many-sorted algebra, which helps us assign semantics to the operations in the circuits/terms.

Definition 5 (Many-sorted universal algebra)**.** *Given a many-sorted signature σ with S sorts, a many-sorted universal algebra is a tuple $\mathscr{A} = (\{\mathbb{D}_1, \ldots, \mathbb{D}_S\}, \circledast_1, \ldots, \circledast_{|\sigma|})$ where $\circledast_i \colon \mathbb{D}_{\mathsf{In}_\sigma(i,1)} \times \ldots \times \mathbb{D}_{\mathsf{In}_\sigma(i,\alpha_i)} \to \mathbb{D}_{\mathsf{Out}_\sigma(i)}$ where $\alpha_i = |\mathsf{In}_\sigma(i)|$. We call the sets $\mathbb{D}_i$ subdomains and the union of all subdomains $\mathbb{D}$, which is the domain.*

From now on we will simply say algebra instead of many-sorted universal algebra. Given an algebra which has the same signature as a circuit and valuations for the input gates, we can evaluate the circuit under the given algebra. Note that this in turn can be used to evaluate terms since terms are just circuits that are trees without inputs.

Definition 6 (Evaluation of many-sorted circuits)**.** *Given an algebra $\mathscr{A}$ over signature σ and a word $w \in \mathbb{D}^n$, the evaluation map $\eta_{\mathscr{A},w} \colon \mathsf{Circ}_{\sigma,n,m} \to \mathbb{D}^m$ is a map defined inductively for all $v \in V$. Here, let $T(v)$ be the maximal subcircuit of a circuit C containing all nodes from which v is reachable.*

- *If $\mathsf{Gatetype}(v) = x_i$ then $\eta_{\mathscr{A},w}(T(v)) = w_i$ provided $w_i \in \mathbb{D}_j$ implies that $\mathsf{In}_C(i) = j$.*
- *Let α be the arity $|\mathsf{In}_\sigma(\mathsf{Gatetype}(v))|$ and $v_1, \ldots v_k$ be the predecessors of v ordered by their output wire order, then $\eta_{A,w}(T(v)) = \circledast_{\mathsf{Gatetype}(v)}(\eta_{\mathscr{A},w}(T(v_1)), \ldots, \eta_{\mathscr{A},w}(T(v_\alpha)))$.*

Let $v_1, \ldots v_m$ be the output gates and C a circuit, then $\eta_{\mathscr{A},w}(C) = (\eta_{\mathscr{A},w}(T(v_1)), \ldots, \eta_{\mathscr{A},w}(T(v_m)))$ if for all $\mathsf{Outputgates}^{-1}(y_i) = v_i$ the following holds: $\mathsf{Out}_C(i) = \mathsf{Out}_\sigma(\mathsf{Gatetype}(v_i))$.

2.4 Semantics: evaluation of terms with an unknown

We have now described how to evaluate terms and circuits. However, we would like to treat terms with an unknown slightly differently. We do not give a value to the unknown but we let this term evaluate to a function. If some algebra is given, the set of functions we can get can be obtained from this algebra. In fact we now get a many-sorted algebra since it needs to contain the original algebra as well as these functions. There are operations of mixed sorts. We can e.g. combine a function $\mathbb{D} \to \mathbb{D}$ and a value $\mathbb{D}$.

Definition 7 (Functional algebra)**.** *Given an algebra $\mathscr{A} = (\mathbb{D}, \circledast_1, \ldots, \circledast_k)$ over a single-sorted signature σ which only contains operations that are at most binary, the functional algebra is defined to be $\mathscr{F}(\mathscr{A}) = (\{\mathbb{D}, \widetilde{\mathbb{D}}\}, F)$, where F is a placeholder for the operations which we will define next and $\widetilde{\mathbb{D}} \subseteq \mathbb{D}^{\mathbb{D}}$ is the smallest set containing the identity function and is closed under the operations in F which are the following:*

- *All operations of $\mathscr{A}$: $\circledast_1, \ldots, \circledast_k$.*
- *$\circ \colon \widetilde{\mathbb{D}} \to \widetilde{\mathbb{D}}$ which is the functional composition.*

- *An operation for functional evaluation* $\odot\colon \widetilde{\mathbb{D}} \times \mathbb{D} \to \mathbb{D}$, *where* $f \odot c = f(c)$.
- *For each* $\circledast_i\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ *there are two variants:* $\overleftarrow{\circledast}_i\colon \widetilde{\mathbb{D}} \times \mathbb{D} \to \widetilde{\mathbb{D}}$ *and* $\overrightarrow{\circledast}\colon \mathbb{D} \times \widetilde{\mathbb{D}} \to \widetilde{\mathbb{D}}$, *where* $(f \overleftarrow{\circledast}_i c)(x) = f(x) \circledast_i c$ *and* $(c \overrightarrow{\circledast}_i f)(x) = c \circledast_i f(x)$.
- *For each* $\circledast_i\colon \mathbb{D} \to \mathbb{D}$ *there is* $\widetilde{\circledast}_i\colon \widetilde{\mathbb{D}} \to \widetilde{\mathbb{D}}$, *where* $(\widetilde{\circledast}_i f)(x) = \circledast_i f(x)$.

The signature of $\mathscr{F}(\mathscr{A})$ *is denoted by* $\sigma(\mathscr{F}(\mathscr{A}))$.

This definition can be lifted to arbitrary arities. The evaluation of terms with an unknown can now be done using the previous definition. Again we use the linearization.

Definition 8 (Evaluation of terms with an unknown)**.** *Let* $\mathscr{A} = (\mathbb{D}, \circledast_1, \ldots, \circledast_k)$ *be an algebra over a single-sorted signature* σ *with maximal operation arity of two and let* $\mathscr{F}(\mathscr{A})$ *be the functional algebra of* $\mathscr{A}$. *The evaluation map* $\mu_{\mathscr{A}}\colon \mathsf{Term}_\sigma[X] \to \widetilde{\mathbb{D}}$ *is a map defined inductively for all* $i \in [k]$*:*

- $\mu_{\mathscr{A}}(X) = id \in \widetilde{\mathbb{D}}$,
- $\mu_{\mathscr{A}}(\oplus_i f) = \widetilde{\circledast}_i \mu_{\mathscr{A}}(f)$ *for* $f \in \mathsf{Term}_\sigma[X]$,
- $\mu_{\mathscr{A}}(f \oplus_i t) = \mu_{\mathscr{A}}(f) \overleftarrow{\circledast}_i \eta_{\mathscr{A}}(t)$ *where* $f \in \mathsf{Term}_\sigma[X]$ *and* $t \in \mathsf{Term}_\sigma$,
- $\mu_{\mathscr{A}}(t \oplus_i f) = \eta_{\mathscr{A}}(t) \overrightarrow{\circledast}_i \mu_{\mathscr{A}}(f)$ *where* $f \in \mathsf{Term}_\sigma[X]$ *and* $t \in \mathsf{Term}_\sigma$.

2.5 Interpreting classical circuit classes in this framework

In order to view the classical circuit complexity classes in this framework, we will first start with the definitions of family of algebras and family of many-sorted circuits.

Definition 9 (Family of algebras)**.** *A family of algebras* $(\mathscr{A}_n)_{n\in\mathbb{N}}$ *is a sequence of algebras, where* $\mathscr{A}_i = (\{\mathbb{D}_1^{p_1(i)}, \ldots, \mathbb{D}_S^{p_S(i)}\}, \circledast_1^i, \ldots, \circledast_k^i)$ *for* $i \in \mathbb{N}$ *and* p_j *being polynomials for* $j \in [S]$[6].

Here, let us assume that the circuits have one output gate and all input gates have the same sort[7]. Given an algebra $\mathscr{A}$ let $\mathbb{D}_I$ and $\mathbb{D}_O$ be the two subdomains that correspond to the input and output values, respectively. Then a circuit C_n of n inputs realizes a function $F_{\mathscr{A}}(C_n)\colon \mathbb{D}_I^n \to \mathbb{D}_O$. Given a family of circuits C we then get a function $F_{\mathscr{A}}(C) = \mathbb{D}_I^* \to \mathbb{D}_O$.

In general, assuming a family of circuits is very powerful, so in complexity it is natural to require that this family is computable in some complexity bound. We then speak of uniformity. Our constructions will be DLOGTIME-uniform. (See e.g. [45] or [46] for basics in circuit complexity.) We use our framework to define the classical Boolean version of $\mathbf{NC}^i$. Here $\mathscr{B}$ is the Boolean algebra $(\mathbb{B}, \wedge, \vee, \neg, \bot, \top)$.

Definition 10 ($\mathbf{NC}^i$)**.** *The set* $\mathbf{NC}^i$ *contains all functions* $F_{\mathscr{B}}(C)$, *where* C *is a family of circuits of signature same as* $\mathscr{B}$ *that contains circuits of polynomial size and* $\mathscr{O}(\log^i n)$ *depth.*

Note that the bounded fan-in of the gates is ensured through the signature of $\mathscr{B}$. For defining $\mathbf{AC}^i$ we need a different algebra to handle the unbounded fan-in gates. In fact we need a family of algebras $\mathscr{B}^* = (\mathscr{B}_n)_{n\in\mathbb{N}}$ where $\mathscr{B}_i = (\mathbb{B}, \wedge, \vee, \neg, \bot, \top, \wedge_i, \vee_i)$, $\wedge_i\colon \mathbb{B}^i \to \mathbb{B}$ and $\vee_i\colon \mathbb{B}^i \to \mathbb{B}$. Hence this is an example where the difference between the members of the families only lies in the arity of operations.

Definition 11 ($\mathbf{AC}^i$)**.** *The set* $\mathbf{AC}^i$ *contains all functions* $F_{(\mathscr{B}_n)_{n\in\mathbb{N}}}(C)$, *where* $C = (C_n)_{n\in\mathbb{N}}$ *is a family of circuits that contains circuits of polynomial size,* $\mathscr{O}(\log^i n)$ *depth and where* C_n *has the same signature as* $\mathscr{B}_n$.

The classes $\mathbf{SAC}^i$ we also get in a similar way as $\mathbf{AC}^i$; we only have to replace the unbounded AND gates $\wedge_i$ from the algebras with the bounded fan-in AND gates.

[6] Note that we are assuming a family of signatures here, rather than a single signature.

[7] This is usually the case in the classical circuit complexity models and hence the assumption.

2.6 Composition of algebras and extended circuit classes

We will now define circuit classes which are variants of the previously defined classes obtained using some algebra $\mathscr{A}$. These are circuits that have Boolean gates as well as $\mathscr{A}$-gates. In our context, such circuits arise in the algorithm design stage. Therefore, we ensure by design that in these circuits Boolean values and values of $\mathscr{A}$ interact via multiplexer gates defined below.

Definition 12 (Multiplexer operation)**.** *Given a domain $\mathbb{D}$, the ternary multiplexer operation is defined as* $\mathrm{mp}_{\mathbb{D}} \colon \mathbb{B} \times \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ *with*

$$(b, d_0, d_1) \mapsto \begin{cases} d_0 & \text{if } b = 0 \\ d_1 & \text{else} \end{cases} .$$

Now we can use multiplexer operations to compose algebras which have as subalgebras the Boolean one $\mathscr{B}$ and some other algebra $\mathscr{A}$ and they interact via multiplexer operations.

Definition 13 (Composition of algebras)**.** *Given an algebra $\mathscr{A} = (\{\mathbb{D}_1, \ldots, \mathbb{D}_S\}, \circledast_1, \ldots, \circledast_k)$, by $(\mathscr{B}, \mathscr{A})$ we denote the algebra $(\{\mathbb{B}, \mathbb{D}_1, \ldots, \mathbb{D}_S\}, \wedge, \vee, \neg, \circledast_1, \ldots, \circledast_k, (mp_{\mathbb{D}_i})_{i \in [S]})$. Given an algebra of the form $(\mathscr{B}, \mathscr{A})$ and an algebra $\mathscr{A}' = (\{\mathbb{D}'_1, \ldots, \mathbb{D}'_{|S'|}\}, \circledast'_1, \ldots, \circledast'_{k'})$, by $((\mathscr{B}, \mathscr{A}), \mathscr{A}')$ we denote the algebra*

$$(\{\mathbb{B}, \mathbb{D}_1, \ldots, \mathbb{D}_S, \mathbb{D}'_1, \ldots, \mathbb{D}'_{|S'|}\}, \wedge, \vee, \neg, \circledast_1, \ldots, \circledast_k, \circledast'_1, \ldots, \circledast'_{k'}, (mp_{\mathbb{D}_i})_{i \in [S]}, (mp_{\mathbb{D}'_i})_{i \in [S']}).$$

Hence we write $(\mathscr{B}, \mathscr{A}_1, \ldots, \mathscr{A}_k) = ((\mathscr{B}, \mathscr{A}_1, \ldots, \mathscr{A}_{k-1}), \mathscr{A}_k)$ for the composition of k algebras.

Note that the previous definition also naturally carries over to families of algebras. We can define classes similar to e.g. $\mathbf{NC}^i$ that are enriched by some algebra. Intuitively, the Boolean part is directing the non-Boolean part via multiplexer gates.

Definition 14 ($\mathscr{A}$-$\mathbf{NC}^i$, $\mathscr{A}$-$\mathbf{NC}^i_{\mathbb{D}}$)**.** *The set $\mathscr{A}$-$\mathbf{NC}^i_{\mathbb{D}}$ contains all functions $F_{(\mathscr{B},\mathscr{A})}(C)$, where C is a family of circuits having the same family of signatures as $(\mathscr{B}, \mathscr{A})$ that contains circuits of polynomial size, depth $\log^i n$, inputs of $\mathbb{D}$ and one output of a subdomain of $\mathscr{A}$. For the special case of Boolean inputs we set $\mathscr{A}$-$\mathbf{NC}^i = \mathscr{A}$-$\mathbf{NC}^i_{\mathbb{B}}$.*

The class $(\mathbb{N}, +, \times, 0, 1)$-$\mathbf{NC}^1$ is in fact $\#\mathbf{NC}^1$ and $(\mathbb{Z}, +, \times, 0, 1)$-$\mathbf{NC}^1$ is the well-studied Gap$\mathbf{NC}^1$. The $\mathscr{A}$ -$\mathbf{NC}^i$ and $\mathscr{A}$-$\mathbf{NC}^i_{\mathbb{D}}$ definitions naturally carry over to classes other than $\mathbf{NC}^i$. The idea of $\mathscr{A}$-$\mathbf{NC}^i_{\mathbb{D}}$ is that we allow Boolean and non-Boolean inputs which then help in composing such circuits, i.e. the output of one circuit is the input of another one.

3 Term evaluation algorithm

Given some term and an algebra $\mathscr{A}$ of the same signature, what does the term evaluate to over $\mathscr{A}$? This problem is the term evaluation problem. The purpose of this section is to prove our main theorem regarding the parallel algorithm for the term evaluation problem.

Theorem 15 (Main Theorem)**.** *Given an algebra $\mathscr{A}$ of single-sorted signature σ and domain $\mathbb{D}$, the evaluation function $\eta_{\mathscr{A}} \colon \mathsf{Term}_\sigma \to \mathbb{D}$ can be computed in $\mathscr{F}(\mathscr{A})$-$\mathbf{NC}^1$. Moreover, the construction ensures that the we get a DLOGTIME-uniform $\mathscr{F}(\mathscr{A})$-$\mathbf{NC}^1$ circuit family.*

Note that the theorem and its proof are independent of the actual algebra $\mathscr{A}$. The algebra can be arbitrary, with no restrictions such as commutativity or associativity on the operations.

The rest of the section is devoted to proving the above theorem.

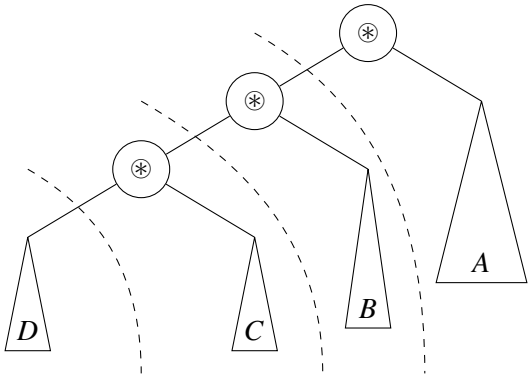

Figure 1: The figure shows a PNF term T with the first three most-left operation symbols from the top pointed out. The term T is of the form $DC \circledast B \circledast A \circledast$, where A, B, C, and D are again terms. Note that $|A| \geq |DC \circledast B \circledast|$, $|B| \geq |DC \circledast|$, and $|C| \geq |D|$. The dashed lines indicate where we can split the term such that the left part corresponds to a closed term. E.g. the middle line gives us the prefix $DC \circledast$ which is again a valid term. What is left on the right is an open term.

3.1 Representing terms in a normal form

Recall that we assume without loss of generality that all the operations used in the terms are binary. A term $(\phi \circledast \psi)$, where ϕ and ψ are also terms, is in infix notation. If ϕ' and ψ' are the equivalent postfix notations for terms ϕ and ψ, then $\phi'\psi' \circledast$ is the postfix equivalent of $(\phi \circledast \psi)$. Note that parentheses are not needed in the postfix notation.

A postfix term is in *postfix normal form (PNF)* if for subterms ϕ and ψ of $\phi\psi \circledast$, $|\phi| \geq |\psi|$. This normal-form was defined by Buss to design an **NC** algorithm [9] for evaluating Boolean formulas. Our algorithm has its roots in this and other related works [11, 23]. We are aware of a simplified version [10] of [9] which directly operates on the infix notation, however we use PNF as that is more convenient here. We now present the details of how the given term is converted into the PNF form.

In order to convert any term into PNF we need to take care of possible non-commutative operations. For that, we assume that all algebras have symmetric variants of operations. That is, for an operation $\circledast$ there exists an operation $\circledast'$ in the algebra such that $x \circledast y = y \circledast' x$.

For the algorithm we always assume that the terms are trees given as strings in the PNF notation. We refer to the terms seen so far as the closed terms. We define a variant called an open term which is informally a suffix of a closed term. Formally, we call a string T *an open term* if there exists a closed non-empty term T' such that $T'T$ is a closed term. A closed term corresponds to a tree, while an open term, which is a sufix of a closed term, corresponds to the subtree obtained by chopping off the left-most subtree. (See Figure 1.) The concatenation of two open terms gives an open term and an open term concatenated with a closed term gives a closed term.

Given a term T (coded as a string in PNF form) of length n, for $1 \leq i \leq j \leq n$, we use $i <_T j$ to denote that in the tree T, j is an ancestor of i. We use $[i, j]$ to denote the interval $\{i, \ldots, j\}$ and the subword $T_i \ldots T_j$ interchangeably (will be clear from the context). If $i <_T j$ and $[i, j]$ is a term, then we write $i \lhd_T j$.

A crucial property of a term in PNF is that it always has suffixes that are *open* terms, which correspond to terms with an unknown.

3.2 Dividing terms: some structural lemmas regarding terms

Let T be a term given as a string of length n in PNF. Let $[l, r]$ be a subinterval. The size of the interval is $s = r - l + 1$. Typically, an interval is specified by l and r and additionally a position m which is approximately the middle position. It is either can be $\lfloor l + \frac{s}{2} \rfloor$ or $\lceil l + \frac{s}{2} \rceil$. Its exact position is not decided

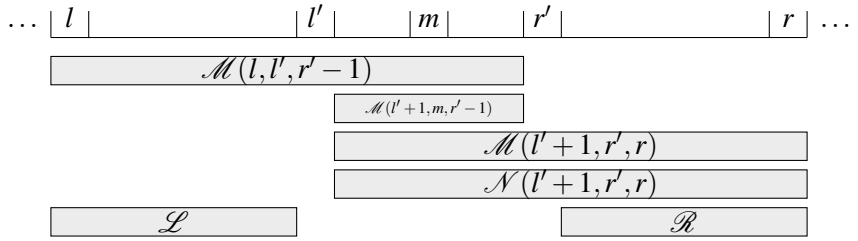

Figure 2: The figure shows how a recursion interval is subdivided into smaller recursion intervals. In this case the subdivision for computing $\mathscr{M}(l,m,r)$ is shown. The six intervals yield recursively six values which may be used to be combined in order to get the evaluation of the $\mathscr{M}(l,m,r)$ interval.

a priori, but comes through a recursive evaluation of subintervals of $[l,r]$. We also use *interval borders* $l' = \lfloor l + \frac{s}{3} \rfloor$ and $r' = \lceil l + \frac{2s}{3} \rceil$. This divides the interval $[l,r]$ in approximately thirds. We not only consider the three intervals but also the intervals obtained by concatenating first two thirds and the second two thirds. These five intervals will be our recursion intervals. Based on those static intervals we define some dynamic intervals, i.e. intervals depending on the specific input:

- The largest closed or open subterm in $[l,r]$ that contains m. This interval is denoted as $\mathscr{M}(l,m,r) = [\mathscr{M}^1(l,m,r), \mathscr{M}^2(l,m,r)]$.
- The open subterm in $[l,r]$ that begins with $\max\{p \mid l \leq p \leq m \wedge l-1 <_T p-1 \wedge l-1 \not<_T p\}$ and ends with the largest position $q \in [m,r]$ such that $[p,q]$ is an open subterm. This interval is denoted as $\mathscr{N}(l,m,r) = [\mathscr{N}^1(l,m,r), \mathscr{N}^2(l,m,r)]$.
- The largest open subterm in $[l,r]$ that precedes $\mathscr{M}(l,m,r)$. This interval is denoted as $\mathscr{L}(l,m,r) = [\mathscr{L}^1(l,m,r), \mathscr{L}^2(l,m,r)]$. It is $\mathscr{L}^2(l,m,r)+1 = \mathscr{M}^1(l,m,r)$.
- The largest open subterm in $[l,r]$ that follows $[\mathscr{M}^1(l,m,r), \mathscr{M}^2(l,m,r)+1]$. This interval is denoted as $\mathscr{R}(l,m,r) = [\mathscr{R}^1(l,m,r), \mathscr{R}^2(l,m,r)]$. It is $\mathscr{R}^1(l,m,r)-2 = \mathscr{M}^2(l,m,r)$ and $\mathscr{M}^2(l,m,r)+1$ is a binary operation symbol. If this operation exists, we denote the set containing its position by $O(l,m,r) = \{\mathscr{M}^2(l,m,r)+1\}$.

If values of (l,m,r) are clear from the context, we drop them. Note that an $\mathscr{M}$ interval could correspond to an open or a closed term. The $\mathscr{N}, R$ intervals always correspond, by definition, to open terms. (That is the only way we use them in the algorithm.) The $\mathscr{L}$ interval is also defined to be open however even if we allowed it to be closed, it can only be an open term. This is because it is strictly shorter than $\mathscr{M} \cup \mathscr{N}$ and does not contain the complete second operand of O. Figure 2 shows the considered intervals. The intervals might be empty. Note that the intervals are well-defined. We now state some properties regarding these intervals. The proofs of these properties are presented in Section A of the appendix.

Lemma 16. *Given intervals $[p_1,q_1] \subseteq [l,r]$ and $[p_2,q_2] \subseteq [l,r]$ which address closed or open terms with $[p_1,q_1] \cap [p_2,q_2] \neq \emptyset$ then $[p_1,q_1] \cup [p_2,q_2]$ is also a closed or open term.*

Lemma 17. *It holds $\mathscr{M}^2 = \mathscr{N}^2$ and $\mathscr{M}^2(l,l',r'-1)+1 = \mathscr{N}^1(l'+1,r',r)$.*

The key lemmas which later constitute the recursive evaluation algorithm are given below. They show how the algorithm actually composes a term by subterms coming from static subintervals. For instance, Figure 2 shows the subintervals relevant for the next lemma.

Lemma 18. *Given a term T and subinterval $[l,r]$ with middle m, $\mathscr{M}$ equals one of the following intervals:*
1. $\mathscr{M}(l,l',r'-1)$*, 2.* $\mathscr{M}(l'+1,r',r)$*, 3.* $\mathscr{M}(l'+1,m,r'-1)$*,*
4. $\mathscr{M}(l,l',r'-1)\cup\mathscr{N}(l'+1,r',r)$*,*
5. $\mathscr{L}(l,m,r)\cup\mathscr{M}(l,l',r'-1)\cup\mathscr{N}(l'+1,r',r)\cup O(l,m,r)\cup\mathscr{R}(l,m,r)$*. Further the sets involved in the unions of case 4 and 5 are disjoint unions.*

In a very similar way we can treat $\mathscr{N}$ as well.

Lemma 19. *Given a term T and subinterval $[l,r]$ with middle m, $\mathscr{N}$ equals one of the following intervals:*
1. $\mathscr{N}(l,l',r'-1)$*, 2.* $\mathscr{N}(l'+1,r',r)$*, 3.* $\mathscr{N}(l'+1,m,r'-1)$*,*
4. $\mathscr{N}(l,l',r'-1)\cup\mathscr{N}(l'+1,r',r)$*,*
5. $\mathscr{N}(l,l',r'-1)\cup\mathscr{N}(l'+1,r',r)\cup O(l,m,r)\cup\mathscr{R}(l,m,r)$*. Further the sets involved in the unions of case 4 and 5 are disjoint.*

The intervals $\mathscr{M}$ and $\mathscr{N}$ are built around the property of containing a middle position m. The intervals $\mathscr{L}$ and $\mathscr{R}$ are different. They can lie arbitrarily in $[l,l'-1]$ resp. $[r'+1,r]$ and we initially do not know anything about the location of the middle points. Our goal is to reduce $\mathscr{L}$ and $\mathscr{R}$ to some $\mathscr{M}(\bar{l},\bar{m},\bar{r})$ where find $\bar{l}$, $\bar{m}$, and $\bar{r}$ using a binary search.

Lemma 20. *Given a term T and subinterval $[l,r]$ with middle m, there is an interval $[\bar{l},\bar{r}]\subseteq[l,l'-1]$ with middle $\bar{m}$ such that $\mathscr{L}=\mathscr{M}(\bar{l},\bar{m},\bar{r})$ and this $\bar{m}$ can be computed by a binary search inside an appropriate range defined by l,m,r.*

Lemma 21. *Given a term T and subinterval $[l,r]$ with middle m there is an interval $[\bar{l},\bar{r}]\subseteq[r'+1,r]$ with middle $\bar{m}$ such that $\mathscr{R}=\mathscr{M}(\bar{l},\bar{m},\bar{r})$ and $\bar{m}$ can be computed by a binary search inside an appropriate range defined by l,m,r.*

3.3 The evaluation algorithm

The algorithm we present is a recursive one which is given in terms of circuits. Lemmas 18, 19, 20, and 21 directly suggest how the recursive evaluation will work: to evaluate an interval compute smaller fixed subintervals and use their evaluations to obtain the overall evaluation.

In particular, we proceed as follows: we wish to compute $\mathscr{M}(1,\lfloor n/2\rfloor,n)$. In order to do that, we design a procedure called EVAL, which at any given stage of recursion receives as input an $\mathscr{M},\mathscr{N},\mathscr{L}$ or an $\mathscr{R}$ interval along with the current values of l,m,r.

If it is an $\mathscr{M}$ or an $\mathscr{N}$ interval then the procedure first determines which among the five cases applies for the subsequent recursion call. By cases, we mean the five possibilities listed in Lemmas 18, 19. Moreover, if it is an $\mathscr{M}$ interval then it also determines whether the term defined by it is an open term or a closed term and keeps that information in a local flag. (Say the flag is by default set to 0 and then it is toggled to 1 if the term is closed.)

If it is an $\mathscr{L}$ or an $\mathscr{R}$ interval then it first computes the appropriate $\bar{l}$, $\bar{r}$ and $\bar{m}$ values and makes recursive calls for the appropriate $\mathscr{M}$ interval defined using these $\bar{l},\bar{r},\bar{m}$ values.

Finally, once the recurive calls return the values, the EVAL procedure combines these values. If the flag is set to 1 then we know that the current call is dealing with a closed term and therefore, the procedure outputs an evaluated value. On the other hand, if the flag is false, the procedure outputs a function from $\mathbb{D}\to\mathbb{D}$.

For the evaluation we need to implement circuits which on a given interval $[l,r]$ evaluate the intervals $\mathscr{M}$, $\mathscr{N}$, $\mathscr{L}$, and $\mathscr{R}$. In the case of $\mathscr{M}$ we need to distinguish whether the evaluation is a value of $\mathbb{D}$ or a function of $\widetilde{\mathbb{D}}$. In the other cases the result is always a function. Correspondingly, the following circuits may arise: EVAL$(\mathscr{M}(l,m,r))$, EVAL$(\mathscr{N}(l,m,r))$, EVAL$(\mathscr{L}(l,m,r),\bar{l},\bar{m},\bar{r})$, EVAL$(\mathscr{R}(l,m,r),\bar{l},\bar{m},\bar{r})$.

The variables $\bar{l},\bar{m},\bar{r}$ exist to serve the binary search as mentioned in Lemma 20 and 21.

Those circuits all work in a similar way: Depending on the structure of the term one of a number of cases holds which determines how the output value is composed of the recursion results. So the recursion results are combined according to the cases and then fed into a multiplexer-gate which chooses the correct output. The circuits determining the cases are called $\textsc{Case}(\mathscr{M}(l,m,r))$, $\textsc{Case}(\mathscr{N}(l,m,r))$, $\textsc{Case}(\mathscr{L}(l,m,r),\bar{l},\bar{m},\bar{r})$, $\textsc{Case}(\mathscr{R}(l,m,r),\bar{l},\bar{m},\bar{r})$.

Here we have already assumed that the term is given in the PNF form. To ensure this, as the first step we perform this conversion to the PNF form as done in [9]. The conversion is of complexity $\mathbf{TC}^0$. The resulting term is T as above. We now present the details regarding recursive calls made in our algorithm.

3.4 The details of term evaluation algorithm

The details of the working of various recursive calls are presented below. In the description below we use two variants of the EVAL procedure. One to deal with closed terms, called $\textsc{Eval}_{\text{closed}}$, which occur only for $\mathscr{M}$ intervals and the other used for all the open terms, denoted as EVAL.

3.4.1 The evaluation algorithm - $\textsc{Case}(\mathscr{M}(l,m,r))$, $\textsc{Eval}(\mathscr{M}(l,m,r))$, and $\textsc{Eval}_{\text{closed}}(\mathscr{M}(l,m,r))$

This part is based on Lemma 18. Consider its five cases:

1. $\mathscr{M} = \mathscr{M}(l,l',r'-1)$
2. $\mathscr{M} = \mathscr{M}(l'+1,r',r)$
3. $\mathscr{M} = \mathscr{M}(l'+1,m,r'-1)$
4. $\mathscr{M} = \mathscr{M}(l,l',r'-1) \cup \mathscr{N}(l'+1,r',r)$
5. $\mathscr{M} = \mathscr{L}(l,m,r) \cup \mathscr{M}(l,l',r'-1) \cup \mathscr{N}(l'+1,r',r) \cup O(l,m,r) \cup \mathscr{R}(l,m,r)$

The circuit $\textsc{Case}(\mathscr{M}(l,m,r))$ determines which case holds for given l, m and r. It actually has five output bits - one for each case. The circuit for the i'th output bit is $\textsc{Case}_i(\mathscr{M}(l,m,r))$. Instead of actually stating a circuit we specify MAJ[$<$] formulas for each output. This is sufficient since MAJ[$<$] equals $\mathbf{TC}^0$ which is a subset of $\mathbf{NC}^1$. Also note that $\lhd_T$ is also expressible in MAJ[$<$] logic [9].

$$\begin{aligned}\textsc{Case}_1(\mathscr{M}(l,m,r)) = {} & \exists x \;\; m \le x < r' \wedge (\exists y \;\; l \le y < l' \wedge y \lhd_T x) \\ & \wedge \forall x \;\; r' \le x \le r \Rightarrow \neg(\exists y \;\; l \le y < m \wedge y \lhd_T x)\end{aligned}$$

$$\begin{aligned}\textsc{Case}_2(\mathscr{M}(l,m,r)) = {} & \exists x \;\; r' \le x \le r \wedge (\exists y \;\; l' < y \le m \wedge y \lhd_T x) \\ & \wedge \forall x \;\; r' \le x \le r \Rightarrow \neg(\exists y \;\; l \le y \le l' \wedge y \lhd_T x)\end{aligned}$$

$$\begin{aligned}\textsc{Case}_3(\mathscr{M}(l,m,r)) = {} & \exists x \;\; m \le x < r' \wedge (\exists y \;\; l' < y \le m \wedge y \lhd_T x) \\ & \wedge \forall x \;\; r' \le x \le r \Rightarrow \neg(\exists y \;\; l \le y \le m \wedge y \lhd_T x) \\ & \wedge \forall x \;\; m \le x < r' \Rightarrow \neg(\exists y \;\; l \le y \le l' \wedge y \lhd_T x)\end{aligned}$$

$$\begin{aligned}\textsc{Case}_4(\mathscr{M}(l,m,r)) = {} & \exists x\ \ r' < x \le r \wedge \exists y\ \ l \le y < l' \wedge y \lhd_T x \\ & \wedge \forall u \forall v\ \ x \le u \wedge (x < u \vee v < y) \Rightarrow \neg(v \lhd_T u) \\ & \wedge \exists z\ \ l' \le z < r' \wedge y \lhd_T z \wedge z+1 \lhd_T x\end{aligned}$$

$$\begin{aligned}\textsc{Case}_5(\mathscr{M}(l,m,r)) = {} & \exists x\ \ r' < x \le r \wedge \exists y\ \ l \le y < l' \wedge y \lhd_T x \\ & \wedge \forall u \forall v\ \ x \le u \wedge (x < u \vee v < y) \Rightarrow \neg(v \lhd_T u) \\ & \wedge \exists z \exists u \exists v\ \ y \lhd_T v \wedge v+1 \lhd_T z \\ & \wedge z+1 \lhd_T u \wedge u+2 \lhd_T x \\ & \wedge v+1 \lhd_T u+1\end{aligned}$$

Now that we have the means of deciding the case of Lemma 18 for a given interval, we can actually evaluate the interval. Recursively we get the results for the intervals $\mathscr{M}(l,l',r'-1)$, $\mathscr{M}(l'+1,r',r)$, $\mathscr{M}(l'+1,m,r'-1)$, $\mathscr{N}(l'+1,r',r)$, $\mathscr{L}(l,m,r)$, and $\mathscr{R}(l,m,r)$. By combining those we can obtain the output value.

- In cases one to three the combination is trivial as we only pass a recursively computed value.
- In case four the output of $\textsc{Eval}_{\text{closed}}(\mathscr{M}(l,m,r))$ we use a functional application gate ($\odot$) which gets the results from $\textsc{Eval}_{\text{closed}}(\mathscr{M}(l,l',r'-1))$ and $\textsc{Eval}(\mathscr{N}(l'+1,r',r))$. For the output of $\textsc{Eval}(\mathscr{M}(l,m,r))$ we use a composition gate ($\circ$) which gets the outputs of $\textsc{Eval}(\mathscr{M}(l,l',r'-1))$ and $\textsc{Eval}(\mathscr{N}(l'+1,r',r))$.
- Case five is composed as $\mathscr{L}(l,m,r) \cup \mathscr{M}(l,l',r'-1) \cup \mathscr{N}(l'+1,r',r) \cup O(l,m,r) \cup \mathscr{R}(l,m,r)$. The subinterval $\mathscr{M}(l,l',r'-1) \cup \mathscr{N}(l'+1,r',r)$ is a term an can be obtained like in case four. For interval $\mathscr{M}(l,l',r'-1) \cup \mathscr{N}(l'+1,r',r) \cup O(l,m,r)$ we use that result and feed it together with an identity function into a $\overleftarrow{\circledast}$-gate if $O(l,m,r)$ points to a symbol $\circledast$. Then we take this value and the result of $\textsc{Eval}(\mathscr{R}(l,m,r))$ and feed it into a composition gate which then yields the value for $\mathscr{M}(l,l',r'-1) \cup \mathscr{N}(l'+1,r',r) \cup O(l,m,r) \cup \mathscr{R}(l,m,r)$. Finally we take this value and compose it with the result of $\textsc{Eval}(\mathscr{L}(l,m,r))$ to get the value for the whole interval; see figure 3.

In case five we need a multiplexer construction to select the right operation $\circledast$, i.e. we do the construction for all possible operations and then select the right one by the multiplexer which is directed by the following:

$$\begin{aligned}\textsc{Operation}_{\circledast}(\mathscr{M}(l,m,r)) = {} & \exists x\ \ r' < x \le r \wedge \exists y\ \ l \le y < l' \wedge y \lhd_T x \\ & \wedge \forall u \forall v\ \ x \le u \wedge (x < u \vee v < y) \Rightarrow \neg(v \lhd_T u) \\ & \wedge \exists z \exists u \exists v\ \ y \lhd_T v \wedge v+1 \lhd_T z \\ & \wedge z+1 \lhd_T u \wedge u+2 \lhd_T x \\ & \wedge v+1 \lhd_T u+1 \\ & \wedge Q_{\circledast}(u+1)\end{aligned}$$

This is the same as the formula for $\textsc{Case}_5(\mathscr{M}(l,m,r))$ but it also checks whether $\circledast$ is in the place of $O(l,m,r)$; see figure 4.

Finally we have these five possible combinations, we use a multiplexer gate and the results of $\textsc{Case}_i(\mathscr{M}(l,m,r))$ to select the right one as output; see figure 5.

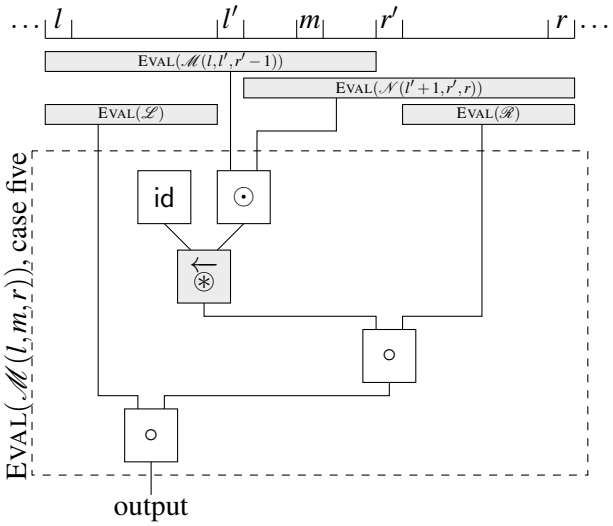

Figure 3: The dashed box represents the subcircuit of EVAL$(\mathscr{M}(l,m,r))$ which performs the combination in case five. Note that the box $\circledast$ corresponds to the operation symbol in position B in figures 6 and 7. This box actually is not a single gate but also a construction which is shown in figure 4.

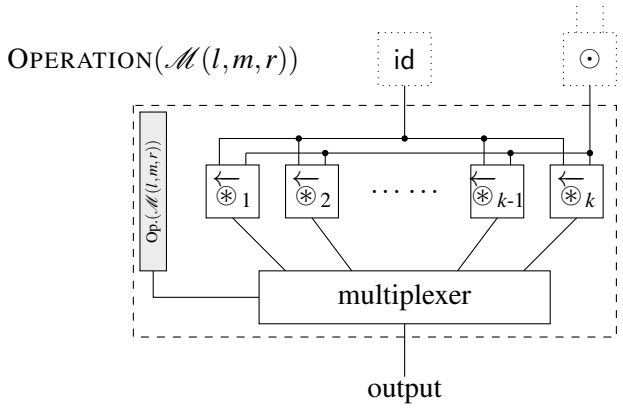

Figure 4: In case five of the computation of EVAL$(\mathscr{M}(l,m,r)$, the operation has to be computed and used. Figure 3 shows where the operation circuit shown here has to be inserted.

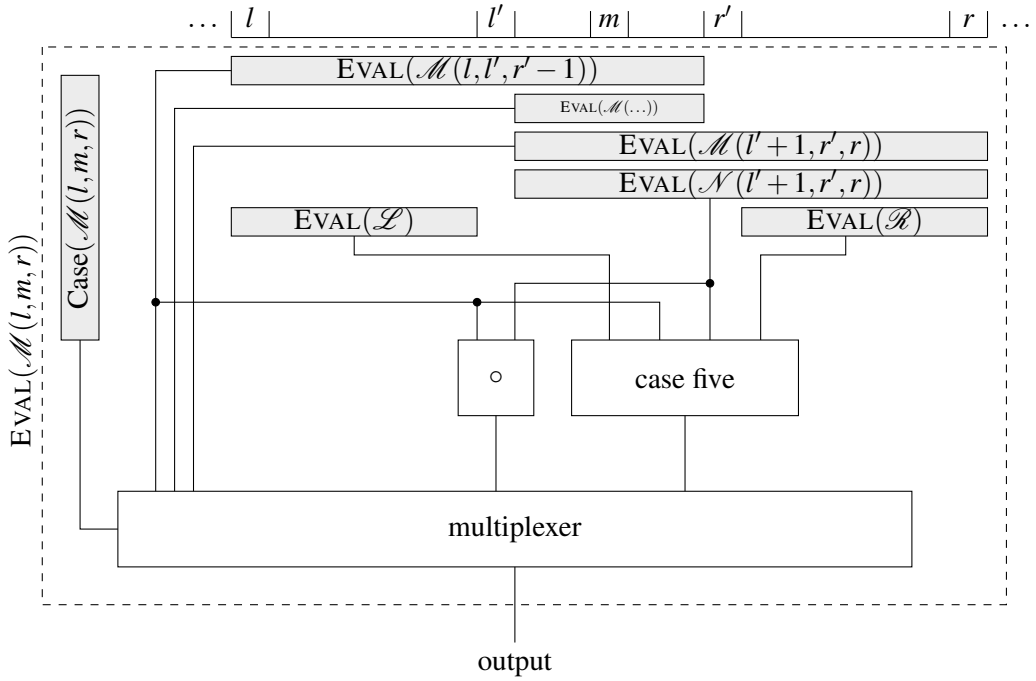

Figure 5: Construction for the $\textsc{Eval}(\mathscr{M}(l,m,r))$ circuit. It consists of 5 recursive calls, a circuit for determining the case and a subcircuit performing the combination for case five as shown in figure 3.

3.4.2 The evaluation algorithm - $\textsc{Case}(\mathscr{N}(l,m,r))$ and $\textsc{Eval}(\mathscr{N}(l,m,r))$

The evaluation of $\mathscr{N}$ intervals is very similar to the one previously described for $\mathscr{M}$. First, we only evaluate open terms in this case. Then the difference is for one that we need to have adjusted circuits $\textsc{Case}(\mathscr{N}(l,m,r))$ computing the case:

$$\begin{aligned}\textsc{Case}_1(\mathscr{N}(l,m,r)) = \exists y\ & l \le y \le l' \wedge l-1 <_T y-1 \wedge \neg(l-1 \le_T y)\\ \wedge\ \forall u\ & (l \le u \le m \wedge l-1 <_T u-1 \wedge \neg(l-1 \le_T u)) \Rightarrow u \le y\\ \wedge\ \exists x\ & m \le x < r' \wedge y \lhd_T x\\ \wedge\ \forall v\ & x < v \le r \Rightarrow \neg(y \lhd_T v)\end{aligned}$$

$$\begin{aligned}\textsc{Case}_2(\mathscr{N}(l,m,r)) = \exists y\ & l' < y \le m \wedge l-1 <_T y-1 \wedge \neg(l-1 \le_T y)\\ \wedge\ \forall u\ & (l \le u \le m \wedge l-1 <_T u-1 \wedge \neg(l-1 \le_T u)) \Rightarrow u \le y\\ \wedge\ \exists x\ & r' \le x < r \wedge y \lhd_T x\\ \wedge\ \forall v\ & x < v \le r \Rightarrow \neg(y \lhd_T v)\end{aligned}$$

$$\begin{aligned}\textsc{Case}_3(\mathscr{N}(l,m,r)) = \exists y\ & l' < y \le m \wedge l-1 <_T y-1 \wedge \neg(l-1 \le_T y)\\ \wedge\ \forall u\ & (l \le u \le m \wedge l-1 <_T u-1 \wedge \neg(l-1 \le_T u)) \Rightarrow u \le y\\ \wedge\ \exists x\ & m \le x < r' \wedge y \lhd_T x\\ \wedge\ \forall v\ & x < v \le r \Rightarrow \neg(y \lhd_T v)\end{aligned}$$

$$\begin{aligned}\textsc{Case}_4(\mathscr{N}(l,m,r)) = \exists y\ \ & l \le y \le l' \wedge l-1 <_T y-1 \wedge \neg(l-1 \le_T y)\\ \wedge\ \forall u\ \ & (l \le u \le m \wedge l-1 <_T u-1 \wedge \neg(l-1 \le_T u)) \Rightarrow u \le y\\ \wedge\ \exists x\ \ & r' \le x < r \wedge y \lhd_T x\\ \wedge\ \forall v\ \ & x < v \le r \Rightarrow \neg(y \lhd_T v)\\ \wedge\ \exists w\ \ & l' \le w < r' \wedge y \lhd_T w \wedge w+1 \lhd_T x\end{aligned}$$

$$\begin{aligned}\textsc{Case}_5(\mathscr{N}(l,m,r)) = \exists y\ \ & l \le y \le l' \wedge l-1 <_T y-1 \wedge \neg(l-1 \le_T y)\\ \wedge\ \forall u\ \ & (l \le u \le m \wedge l-1 <_T u-1 \wedge \neg(l-1 \le_T u)) \Rightarrow u \le y\\ \wedge\ \exists x\ \ & r' \le x < r \wedge y \lhd_T x\\ \wedge\ \forall v\ \ & x < v \le r \Rightarrow \neg(y \lhd_T v)\\ \wedge\ \exists w \exists z\ \ & l' \le w < r' \wedge y \lhd_T w \wedge w+1 \lhd_T z \wedge z+2 \lhd_T x\end{aligned}$$

Now by applying Lemma 19 we can build $\textsc{Eval}(\mathscr{N}(l,m,r))$. Consider the cases:

1. $\mathscr{N} = \mathscr{N}(l,l',r'-1)$
2. $\mathscr{N} = \mathscr{N}(l'+1,r',r)$
3. $\mathscr{N} = \mathscr{N}(l'+1,m,r'-1)$
4. $\mathscr{N} = \mathscr{N}(l,l',r'-1) \cup \mathscr{N}(l'+1,r',r)$
5. $\mathscr{N} = \mathscr{N}(l,l',r'-1) \cup \mathscr{N}(l'+1,r',r) \cup O(l,m,r) \cup \mathscr{R}(l,m,r)$

The construction for $\textsc{Eval}(\mathscr{N}(l,m,r))$ is similar to the one for $\textsc{Eval}(\mathscr{M}(l,m,r))$ with the exception that we use the appropriate recursive calls and do not use the $\mathscr{R}$ interval. Also we of course use $\textsc{Case}(\mathscr{N}(l,m,r))$ instead of $\textsc{Case}(\mathscr{M}(l,m,r))$.

3.4.3 The evaluation algorithm - $\textsc{Case}(\mathscr{L}(l,m,r),\bar{l},\bar{m},\bar{r})$ and $\textsc{Eval}(\mathscr{L}(l,m,r),\bar{l},\bar{m},\bar{r})$

The key idea of evaluating an interval in our algorithm is that we evaluate e.g. the largest subterm in the interval that contains the middle. If we want to evaluate a $\mathscr{L}$ interval we face the problem that it can lie arbitrarily in the considered interval. So the idea is that we do a binary search in order to find an interval whose middle is part of $\mathscr{L}$; see Lemma 20.

Our search interval will be $[\bar{l},\bar{r}]$ with middle $\bar{m}$. We then distinguish three cases:

1. $\bar{m} \in \mathscr{L}$
2. $\mathscr{L} \subseteq [\bar{l},\bar{m}-1]$
3. $\mathscr{L} \subseteq [\bar{m}+1,\bar{r}]$

In the first case we can fall back to $\textsc{Eval}(\mathscr{M}(\bar{l},\bar{m},\bar{r}))$. In the second we recurse using $\textsc{Eval}(\mathscr{L}(l,m,r),(\bar{l},\bar{l}+\bar{m}-1)/2,\bar{m}-1)$ and in the third case we use $\textsc{Eval}(\mathscr{L}(l,m,r),(\bar{m}+1,(\bar{r}+\bar{m}+1)/2,\bar{r})$. So we have three recursive calls which we feed into a multiplexer gate. The multiplexer gate is directed by $\textsc{Case}(\mathscr{L}(l,m,r),\bar{l},\bar{m},\bar{r})$ which decides which of the tree cases hold:

$$\begin{aligned}\textsc{Case}_1(\mathscr{L}(l,m,r),\bar{l},\bar{m},\bar{r}) = \textsc{Case}_5(\mathscr{M}(l,m,r)) \wedge y \leq \bar{m} \leq v \\ =\exists x \;\; r' < x \leq r \wedge \exists y \;\; l \leq y < l' \wedge y \lhd_T x \\ \wedge \forall u \forall v \;\; x \leq u \wedge (x < u \vee v < y) \Rightarrow \neg(v \lhd_T u) \\ \wedge \exists z \exists u \exists v \;\; y \lhd_T v \wedge v+1 \lhd_T z \\ \wedge z+1 \lhd_T u \wedge u+2 \lhd_T x \\ \wedge v+1 \lhd_T u+1 \\ \wedge y \leq \bar{m} \leq v\end{aligned}$$

$$\textsc{Case}_2(\mathscr{L}(l,m,r),\bar{l},\bar{m},\bar{r}) = \textsc{Case}_5(\mathscr{M}(l,m,r)) \wedge y \leq v < \bar{m}$$

$$\textsc{Case}_3(\mathscr{L}(l,m,r),\bar{l},\bar{m},\bar{r}) = \textsc{Case}_5(\mathscr{M}(l,m,r)) \wedge \bar{m} < y \leq v$$

3.4.4 The evaluation algorithm - $\textsc{Case}(\mathscr{R}(l,m,r),\bar{l},\bar{m},\bar{r})$ and $\textsc{Eval}(\mathscr{R}(l,m,r),\bar{l},\bar{m},\bar{r})$

Evaluating an $\mathscr{R}$ interval is again very similar to $\mathscr{L}$. For $\textsc{Eval}(\mathscr{R}(l,m,r),\bar{l},\bar{m},\bar{r})$ we use the same multiplexer construction for the binary search as in $\textsc{Eval}(\mathscr{L}(l,m,r),\bar{l},\bar{m},\bar{r})$ and only have to adjust the case computation:

$$\textsc{Case}_1(\mathscr{R}(l,m,r),\bar{l},\bar{m},\bar{r}) = \textsc{Case}_5(\mathscr{M}(l,m,r)) \wedge u+2 \leq \bar{m} \leq x$$

$$\textsc{Case}_2(\mathscr{R}(l,m,r),\bar{l},\bar{m},\bar{r}) = \textsc{Case}_5(\mathscr{M}(l,m,r)) \wedge u+2 \leq x \leq \bar{m}$$

$$\textsc{Case}_3(\mathscr{R}(l,m,r),\bar{l},\bar{m},\bar{r}) = \textsc{Case}_5(\mathscr{M}(l,m,r)) \wedge \bar{m} \leq u+2 \leq x$$

3.5 Complexity and correctness

In this section we present the claimed complexity bounds.

The correctness of our construction follows from the lemmas of Section 3.2 as is only directly implements those lemmas.

Our circuit construction uses the kind of gates which we may use for $\mathscr{F}(\mathscr{A})$-$\mathbf{NC}^1$ circuits. We used multiplexer gates of three resp. five instead of two inputs which can be easily implemented.

We now observe that the construction can be done in logarithmic depth with regard to the input length. The PNF conversion is doable in $\mathbf{TC}^0$. The same is true for the case computations. Finally the evaluation circuits entail the case circuits as well as recursive calls. As in every call the range becomes smaller by at least a factor of $2/3$, the depth is logarithmic.

Analyzing the size of our construction, we see that we use a polynomial number of circuits which originate in MAJ$[<]$ formulas which result in polynomial size circuits. In particular that are circuits computing the PNF term and those computing the cases. Further each recursive evaluation circuit covers a certain subinterval and since there is only a quadratic number of subintervals, we get the polynomial bound for the whole construction.

Lastly we give the idea for DLOGTIME-uniformity. To that end we have to show how to address states and then state FO$[<,+,\times]$ formulas which take such addresses and tell what function some gate is assigned as well as how the gates are wired. Consider a circuit $\textsc{Eval}(\mathscr{M}(l,m,r))$. It consists of several recursively defined subcircuits and a fixed number of extra gates to combine the results of the subcircuits which we call combination gates of $\textsc{Eval}(\mathscr{M}(l,m,r))$. More concretely, an addressing scheme can look like this: we assign each $\textsc{Eval}(\mathscr{M}(l,m,r))$ circuit a string w for the six subcircuits we assign strings $w000$, $w001$, $w010$, $w011$, $w100$, and $w101$. The finitely many combination gates which are left we address by $w\$x$ where x is unique string for each occurring gate. It can be easily seen that this scheme can be applied for all kinds of circuits we defined.

Now it is easy to come up with a FO$[<,+,\times]$ formula which assigns each gate its type. On an input $w\$x$ it is only a look-up to which kind of gate x corresponds. The wiring between gates can also expressed as follows: for a pair of combination gates of some $\textsc{Eval}(\mathscr{M}(l,m,r))$, where the task is again just a look-up. If we have a pair such that one is the output of a recursion, we can also model that by looking at the last letter of w in the address $w\$x$. In the case of small intervals $r-l$, the computation $\textsc{Eval}(\mathscr{M}(l,m,r))$ becomes a look-up table which accesses input gates which we can also model. The output gate is a gate with an address of the form $\$x$ for an appropriate x.

Note that we also have circuits like $\textsc{Case}(\mathscr{M}(l,m,r))$ which are given in terms of MAJ$[<]$ formulas. By [6] we know that these are also in DLOGTIME-uniform $\mathbf{NC}^1$.

4 Application

Our main theorem can be used for many different applications. We first present a template or recipe used for deriving these applications. It consists of the following steps:

1. **Find an algebra** $\mathscr{A}$. Given a problem Π, which could be a language or a function, find an algebra $\mathscr{A}_\Pi = (\mathbb{D}, \circledast_1, \ldots, \circledast_k)$, such that Π reduces to term evaluation over $\mathscr{A}_\Pi$.

2. **Find a coding for** $\mathscr{F}(\mathscr{A}_\Pi)$. Now we know by our main theorem that Π is in $\mathscr{F}(\mathscr{A}_\Pi)$-$\mathbf{NC}^1$. However what we want is a "real" class like $\mathbf{NC}^1$ or $\#\mathbf{SAC}^7$. Hence we encode $\mathscr{F}(\mathscr{A}_\Pi)$ in a way that we end up with a Boolean or an arithmetic class. So find a code c mapping (details in Section B.1) into a family of algebras, that have domains based on $\mathbb{B}$, $\mathbb{N}$, or $\mathbb{Z}$, depending on whether we wish to prove Boolean or arithmetic circuit upper bounds.

3. **Analyze the complexity of the operations used in** $c(\mathscr{F}(\mathscr{A}_\Pi))$**-**$\mathbf{NC}^1$. Now we know that Π is in $c(\mathscr{F}(\mathscr{A}_\Pi))$-$\mathbf{NC}^1$ since the coding admits a reduction. If we have chosen c well, we can implement the operations of $c(\mathscr{F}(\mathscr{A}_\Pi))$ efficiently. Note that $c(\mathscr{F}(\mathscr{A}_\Pi))$ could be a family. The members of the family all contain the following coded operations:

   - The operations of $\mathscr{A}_\Pi$: $\circledast_1^c \ldots \circledast_k^c$.
   - The functional versions for each binary operation $\circledast_i$: $\overleftarrow{\circledast}_i^c$ and $\overrightarrow{\circledast}_i^c$. Recall that $\overleftarrow{\circledast}_i \colon \mathbb{D}^\mathbb{D} \times \mathbb{D} \to \mathbb{D}^\mathbb{D}$ and $\overrightarrow{\circledast}_i \colon \mathbb{D} \times \mathbb{D}^\mathbb{D} \to \mathbb{D}^\mathbb{D}$.
   - The functional versions for each unary operation $\circledast_i$ which is $\widetilde{\circledast}_i^c \colon \mathbb{D}^\mathbb{D} \to \mathbb{D}^\mathbb{D}$.
   - The functional composition of $\mathscr{F}(\mathscr{A}_\Pi)$: $\circ^c \colon \mathbb{D}^\mathbb{D} \times \mathbb{D}^\mathbb{D} \to \mathbb{D}^\mathbb{D}$.
   - The function application operation of $\mathscr{F}(\mathscr{A}_\Pi)$: $\odot^c \colon \mathbb{D}^\mathbb{D} \times \mathbb{D} \to \mathbb{D}$.

   An algebra usually also has 0-ary operations, but here there is no complexity to analyze.

   The following is not a part of the algebra but comes into play in the construction of the $c(\mathscr{F}(\mathscr{A}_\Pi))$-$\mathbf{NC}^1$ circuits: Multiplexer operations for all subdomains $\mathbb{D}$ of $c(\mathscr{F}(\mathscr{A}_\Pi))$: $\mathrm{mp}_\mathbb{D}$. These operations are used in the $c(\mathscr{F}(\mathscr{A}_\Pi))$-$\mathbf{NC}^1$ circuit as black boxes. In this third step we have to come

up with a efficient implementation of all these operations in order to derive a good upper bound. In general, the depth increases by a logarithmic factor when comparing the complexity of the functions and the overall circuit. So, if, say, all the functions are in $\#\mathbf{NC}^1_{\mathbb{D}}$, then $c(\mathscr{F}(\mathscr{A}_\Pi))$-$\mathbf{NC}^1 \subseteq \#\mathbf{NC}^2$.

All the applications we show follow this template.

4.1 Evaluating arithmetic terms and distributive algebras

We consider evaluating terms over $\mathscr{N} = (\mathbb{N}, +, \times, 0, 1)$ and $\mathscr{Z} = (\mathbb{Z}, +, \times, 0, 1)$.

Theorem 22 ([11]). *Evaluating terms over $\mathscr{N}$ ($\mathscr{Z}$) is in $\#\mathbf{NC}^1$ (Gap$\mathbf{NC}^1$ resp.).*

Proof. We give the proof for $\mathscr{N}$. The case of $\mathscr{Z}$ is handled similarly.
step 1. The problem is directly a term evaluation problem, hence no reduction is needed.
step 2. Consider the algebra $\mathscr{F}(\mathscr{N}) = (\{\mathbb{N}, \widetilde{\mathbb{N}}\}, +, \times, 0, 1, \overleftarrow{+}, \overleftarrow{\times}, \overrightarrow{+}, \overrightarrow{\times}, \circ, \odot)$, where $\widetilde{\mathbb{N}} \subseteq \mathbb{N}^{\mathbb{N}}$. We choose a coding c such that $c(\mathbb{N}) = \mathbb{N}$ and $c(\widetilde{\mathbb{N}}) = \mathbb{N}^2$. The functions in $\widetilde{\mathbb{N}}$ are of the form $x \mapsto ax + b$ for some $a, b \in \mathbb{N}$: We begin with the identity function $x \mapsto 1x + 0$ which is clearly of this form. Now we show that the operations of $\mathscr{F}(\mathscr{N})$ keep functions in this form.

- $\circ^c$: Given some functions $f(x) = a_f x + b_f$ and $g(x) = a_g x + b_g$, then $f \circ g$ is of this form: $x \mapsto a_f a_g x + a_f b_g + b_f$. So $c(f \circ g) = c(f) \circ^c c(g) = (a_f, b_f) \circ^c (a_g, b_g) = (a_f a_g, a_f b_g + b_f)$.
- $\overleftarrow{+}^c$: Consider $c(f + e)$ for $f \in \mathbb{N}^{\mathbb{N}}$ and $e \in \mathbb{N}$. Now $c(f) +^c c(e) = (a, b) +^c e = (a, b + e)$ where $f(x) = ax + b$. The operation $\overrightarrow{+}^c$ follows similarly.
- $\overleftarrow{\times}^c$: Consider $c(f \times e)$ for $f \in \mathbb{N}^{\mathbb{N}}$ and $e \in \mathbb{N}$. Now $c(f) +^c c(e) = (a, b) +^c e = (a \times e, b \times e)$ where $f(x) = ax + b$. The operation $\overrightarrow{\times}^c$ follows similarly.

This shows that c is indeed a valid coding.
step 3. We now have an upper bound of $c(\mathscr{F}(\mathscr{B}))$-$\mathbf{NC}^1$. As all operations use constantly many inputs of natural numbers, there exist arithmetic circuit implementations for all operations. Further, all Boolean gates and multiplexer gates can be simulated by arithmetic circuit constructions, so all operations are in $\#\mathbf{NC}^0_{\mathbb{N}}$. Hence we get $c(\mathscr{F}(\mathscr{B}))$-$\mathbf{NC}^1 \subseteq \#\mathbf{NC}^1$. □

In the previous proof we used distributivity of $+$ and $\times$ which allows us to represent functions by two values. This we can do in general, so we get the following:

Theorem 23. *Given a distributive algebra $\mathscr{A} = (\mathbb{D}, \circledast_1, \circledast_2)$, then evaluating terms over $\mathscr{A}$ is in $\mathscr{A}$-$\mathbf{NC}^1$.*

4.2 The Boolean formula value problem and finite algebras

Problem description.

The BFVP is the problem of evaluating Boolean formulas. That is evaluating terms over the algebra $\mathscr{B} = (\mathbb{B}, \wedge\vee, \neg, \bot, \top)$.

Theorem 24 ([9]). *The Boolean formula value problem is in $\mathbf{NC}^1$.*

Proof. **step 1.** We do not need a reduction, since the problem directly is a term evaluation problem over the algebra $\mathscr{B}$

step 2. Consider the algebra $\mathscr{F}(\mathscr{B}) = (\{\mathbb{B}, \mathbb{B}^{\mathbb{B}}\}, \wedge, \vee, \neg, \bot, \overleftarrow{\wedge}, \overleftarrow{\vee}, \overrightarrow{\wedge}, \overrightarrow{\vee}, \widetilde{\neg}, \circ, \odot)$. Here $\mathbb{B}^{\mathbb{B}}$ has four elements. We choose some coding c with $c(\mathbb{B}) = \mathbb{B}$ and $c(\mathbb{B}^{\mathbb{B}}) = \mathbb{B}^2$.

step 3. Consider the algebra $c(\mathscr{F}(\mathscr{B}))$. The operations of $c(\mathscr{B}) = \mathscr{B}$ can be implemented directly by single gates. The other operations need constant size circuits, i.e. $\mathbf{NC}^0$. The same is true for multiplexer gates. Hence $c(\mathscr{F}(\mathscr{B}))$-$\mathbf{NC}^1 \subseteq \mathbf{NC}^1$. □ □

In the previous proof we used that the algebra is finite. If it is finite we only need constant size circuits to implement the operations. Hence we can state a general theorem:

Theorem 25. *If $\mathscr{A}$ is a finite algebra then evaluating terms over $\mathscr{A}$ is in* $\mathbf{NC}^1$.

4.3 Tree automata

Tree automata are finite state machines operating on tree; a counter-part of finite automata over strings. We are interested in proving upper bounds for the membership problems known from [36].

For the sake of a clear presentation, in this section we focus on binary trees but all results can be generalized for arbitrary trees. The nodes of the trees are labeled with a letter of the alphabet Σ. We use a linearization similar to the one we used for terms: $(t_1 a t_2)$ is a tree whose root is labeled a and has t_1 as a left and t_2 as a right descendant.

Definition 26. *A nondeterministic bottom-up tree automaton (BUTA) is a tuple $\mathscr{T} = (Q, \Sigma, Q_0, F, \delta)$ where Q is a finite set of states, Σ a finite alphabet, $Q_0 \subseteq Q$ a set of initial states, $F \subseteq Q$ the set of final states and $\delta \colon Q \times Q \times \Sigma \to 2^Q$ the transition function.*

The deterministic version has only one initial state q_0 and the transition function is deterministic: $\delta \colon Q \times Q \times \Sigma \to Q$.

Given a BUTA and a tree t, a run is an assignment of states to the edges of the tree in the following way: for each leaf we add two new children which then become leaves and label them an initial state. Also we add a new root whose only child is the old root. If the children of a node are labeled q_1 and q_2 and the node is labeled a, then it must be labeled with a state of $\delta(q_1, q_2, a)$. If the label of the root is labeled with a state of F, then we call the run accepting. If a tree has an accepting run, the tree is accepted by the automaton.

Nondeterministic BUTA can be determinized by the classical power-set construction. There is also the notion of top-down tree automata. In the nondeterministic version they have the same power as BUTA but the deterministic version is strictly weaker. Hence we will focus on BUTA.

Further, in reference to [36] we consider the uniform membership problem where the automaton is not fixed but part of the input. In fact, we consider a slightly general counting problem, which can be stated as follows: given a BUTA M as an input and a tree t, what is the number of accepting runs in the BUTA on the tree t?

Theorem 27. *Given a nondeterministic BUTA M and a tree t as input, then computing the number of accepting runs of M on t is in* $\#\mathbf{SAC}^1$

Proof. **step 1.** Let Σ be a fixed alphabet. The input tree we interpret as a term over a family of algebras $(\mathscr{A}_n)_{n\in\mathbb{N}}$ with

$$\mathscr{A}_n = (\mathbb{N}^{[n]} \times \mathbb{M}_n, (\circledast_a)_{a\in\Sigma}, \dagger),$$

where n is the length of the term and $\mathbb{M}_n$ the set of all BUTA having n states; we assume it to have a state set $[n]$. Note that BUTA with less than n states can be simulated by inserting unused states. The operation family $(\circledast_a)_{a\in\Sigma}$ consists of binary operations and $\dagger$ is a constant. Now an input tree w yields a term as follows: If $(t_1 a t_2)$ is a tree and $T(t_1)$, $T(t_2)$ are terms for t_1 and t_2, then $(T(t_1) \circledast_a T(t_2))$ is the term for $(t_1 a t_2)$. If $t = a$ is a leaf, then $T(t) = \dagger \circledast_a \dagger$.

The idea of the algebra is that each element stores for each state q, how many runs there are to end up in q in a certain subtree. Also the automaton has to be part of the algebra. It is stored in the second component of the domain and is constant for the whole term evaluation. Now $\dagger$ should correspond to the initial states, hence $\dagger = (f, M)$, where M is the automaton from the input and $f(q) = 1$ if q is an initial state and $f(q) = 0$ else. Given $a \in \Sigma$, the operation $\circledast_a$ is defined as $(f, M) \circledast_a (g, M) = (h, M)$ where $h(q) = \sum_{q\in\delta(q_1,q_2,a)} f(q_1) \cdot g(q_2)$ and δ is the transition function of M.

Clearly if we evaluate the term we get from the input over the algebra, we get the desired value. The evaluation is a pair where the first component holds for each state how many runs there are to end up in it. Now add all values which correspond to a final state; this is the output.

step 2. We design a coding to later be able to show the upper bound. We need to code

$$\mathscr{F}(\mathscr{A}) = (\{\mathbb{D}, \widetilde{\mathbb{D}}\}, (\circledast_a)_{a\in\Sigma}, \dagger, (\overleftarrow{\circledast}_a)_{a\in\Sigma}, (\overrightarrow{\circledast}_a)_{a\in\Sigma}, \circ, \odot).$$

Note that for a given term, the automaton always stays the same, so we just assume $\mathbb{D}$ to be $\mathbb{N}^{[n]}$, so we omit explicitly coding the automaton. We focus on coding the functions $\mathbb{N}^{[n]} \to \mathbb{N}^{[n]}$ and constants $\mathbb{N}^{[n]}$. The constants we can code as $c(\mathbb{N}^{[n]}) = \mathbb{N}^n$, so the functions of $\mathbb{N}^{[n]}$ become an n-tuple. Further we code

$$c\left(\left(\mathbb{N}^{[n]}\right)^{\mathbb{N}^{[n]}}\right) = \mathbb{N}^{n,n} \times \mathbb{N}^n.$$

So each function $f\colon \mathbb{N}^{[n]} \to \mathbb{N}^{[n]}$ can be represented by a matrix M and a vector b. Note that a matrix can be seen as a sequence of numbers. A function then is a map of the form $x \mapsto xM + b$ for $x \in \mathbb{N}^n$. We now show that all functions of the algebra conform to this:

- The identity function is represented by $x \mapsto xI + 0$ where I is the identity matrix.
- Consider $f \circ g$ and let $c(f) = (M,b)$ and $c(g) = (M',b')$, then $c(f \circ g) = c(f) \circ^c c(g)$ is a map of the form $x \mapsto xMM' + bM' + b'$, so $c(f \circ g) = (MM', bM' + b')$.
- For $f \in \widetilde{\mathbb{N}^{[n]}}$, $d \in \mathbb{N}^{[n]}$, and $c(f) = (M,b)$ we get $c(f) \odot c(d) = c(f(d)) = c(f) \odot^c c(d) = c(d)M + b$.
- For $a \in \Sigma$ consider $f \overleftarrow{\circledast}_a d$ where $c(f) = (M,b)$, then $c(f \overleftarrow{\circledast}_a d) = c(f) \overleftarrow{\circledast}^c_a c(d) = (MM^a_d, bM^a_d)$ where the matrix M^a_d is defined as follows: For $i, j \in [n]$, let $S \subseteq [n]$ be the set of all states such that $i \in \delta(j,S,a)$. Then in M^a_d the position (i,j) is $\sum_{s\in S} d_s$. The operation $\overrightarrow{\circledast}_a$ follows similarly.

step 3. Multiplying matrices requires multiplication gates of fan-in 2 and addition gates of fan-in n. The depth is constant. Hence the gates used in the $c(\mathscr{F}(\mathscr{A}))$-$\mathbf{NC}^1$ circuits can be replaced by $\#\mathbf{SAC}^0_{\mathbb{N}}$ constructions which yields an overall complexity of $\#\mathbf{SAC}^1$. □

If we consider the previous proof, then it is immediate that we end up with a Boolean circuit if we are not interested in counting but only the existence of accepting runs. Hence we get the following result.

Theorem 28 ([36])**.** *The uniform membership problem for nondeterministic BUTA is in* $\mathbf{SAC}^1$.

Also we can look at the proof and consider the situation for a fixed automaton. The complexity we determined for the operations is $\#\mathbf{SAC}^0$, where the unbounded addition gates have a fan-in equal to the number of states of the automaton. If now the automaton is fixed, a constant depth construction with bounded fan-in suffices, hence we get:

Theorem 29. *For a fixed BUTA, counting the number of accepting runs is in* $\#\mathbf{NC}^1$.

And again, if we are only interested in acceptance and not in counting, the proof directly yields a Boolean upper bound:

Theorem 30 ([36])**.** *For a fixed BUTA, the membership problem is in* $\mathbf{NC}^1$.

4.4 Visibly pushdown languages and quantitative automata models

Visibly pushdown languages (VPL) are a proper subclass of context-free languages which contains regular languages. They were first defined and studied by [39]. They were referred to as the input-driven pushdown languages in this early reference. Subsequently they were rediscovered by Alur and Madhusudan in [4]. VPL is the set of languages for which there is a visibly pushdown automaton (VPA). A VPA is a pushdown automaton M which has the following restrictions: If Σ is the input alphabet then there is a partition of Σ into subsets Σ_{call}, Σ_{ret}, and Σ_{int} such that M pushes one symbol onto the stack if a letter of Σ_{call} is read. If a letter of Σ_{ret} is read then one symbol is popped of the stack. If a letter of Σ_{int} is read, the stack is not accessed at all. This model yields many desirable properties with regard to decidability and closure properties. Also determinism equals nondeterminism in this model. VPA have their applications in fields like XML or verification. The intuition is that a word accepted by a VPA basically represents an unranked tree.

Definition 31 (Visibly pushdown automaton)**.** *Given a partitioned alphabet $\Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{ret}} \cup \Sigma_{\text{int}}$, a VPA is a tuple $M = (Q, Q_0, F, \Gamma, \bot, \delta_{\text{call}}, \delta_{\text{ret}}, \delta_{\text{int}})$ where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $F \subseteq Q$ a set of final states, Γ the stack alphabet, $\bot \in \Gamma$ the bottom-of-stack symbol, $\delta_{\text{call}} \subseteq Q \times \Sigma_{\text{call}} \times Q \times \Gamma$ is the transition relation for call letters, $\delta_{\text{ret}} \subseteq Q \times \Sigma_{\text{ret}} \times \Gamma \times Q$ is the transition relations for return letters and $\delta_{\text{int}} \subseteq Q \times \Sigma_{\text{int}} \times Q$ is the transition relation for internal letters.*

We omit details for the semantic here. However note that we impose that VPLs only contain well-matched words. A word is well-matched if all positions of call or return letters have a matching position. Two positions $i < j$ in a word w match if $w_i \in \Sigma_{\text{call}}$, $w_j \in \Sigma_{\text{ret}}$, the number of call letters equals the number of return letters in $w_i \dots w_j$, and in all prefixes of $w_i \dots w_j$ there are at least as many call letters as return letters. That way well-matched words can be seen as well-parenthesized expressions or as valid representations of trees. If there is only one initial state and the transition relations are functions, the automaton is called deterministic.

Using our evaluation algorithm we can derive upper bounds for membership and counting problems.

Theorem 32. *Given a nondeterministic VPA M and a well-matched word $w \in \Sigma^*$ as input, computing the number of accepting runs of M on w can be done in* $\#\mathbf{SAC}^1$

Proof. **step 1.** A well-matched word can be considered to be a linearization of a tree or a term. So what we will do is to interpret the input word as a term and the input automaton can be found again in the family of algebras $(\mathscr{A}_n)_{n\in\mathbb{N}}$ we will evaluate the term over. We choose

$$\mathscr{A}_n = (\mathbb{N}^{[n]\times[n]} \times \mathbb{M}_n, \circledast, (\otimes_{a,b})_{a\in\Sigma_{\text{call}}, b\in\Sigma_{\text{ret}}}, (\dagger_e)_{e\in\Sigma_{\text{int}}\cup\{\varepsilon\}}).$$

Now given as input a well-matched input word, we construct a term. If the input w is either the empty word or an internal letter, then the corresponding term is $t(w) = \dagger_w$. If $w = w_1 w_2$ where w_1, w_2 are well matched then $t(w) = t(w_1) \circledast t(w_2)$. If $w = aw'b$ where $a \in \Sigma_{\text{call}}$, $b \in \Sigma_{\text{ret}}$, and w' is well-matched, then $t(w) = \otimes_{a,b} t(w)$. The intuition of the algebra is that the second part $\mathbb{M}_n$ of the domain stores the automaton which is then constant for the whole term. We assume it to have state set $[n]$, where n is the input length which is no restriction. Then the first component $\mathbb{N}^{[n]\times[n]}$ then assigns each pair of states q_1, q_2 the number of runs from q_1 to q_2 there are by passing through the corresponding well-matched word. Also note that in the construction we did not explicitly convert the automaton into a term. It is easy to do and we leave out the details for the sake of readability. The only point worth noting in building a term is that the same automaton is accessible in every step of the evaluation. The definition of the algebra operations is as follows:

- $\dagger_\varepsilon$ is a 0-ary operation, hence an element of the domain, which is a function $[n] \times [n] \to \mathbb{N}$. We define it as $(q, q') \mapsto 1$ iff $q = q'$ and $(q, q') \mapsto 0$ otherwise.

- $\dagger_e$ for $e \in \Sigma_{\text{int}}$ is defined as $(q,q') \mapsto 1$ if $q' \in \delta_{\text{int}}(q,e)$ and $(q,q') \mapsto 0$ otherwise.
- $\circledast$ is a binary operation and $\alpha \circledast \beta$ is defined as $(\alpha \circledast \beta)(q,q') = \sum_{r \in Q} \alpha(x,r)\beta(r,y)$.
- $\otimes_{a,b}$ is unary and $(\otimes_{a,b}\alpha)(q,q')$ is defined as the sum of all $\alpha(p,p')$ such that there exists $\gamma \in \Gamma$ and $(p,\gamma) \in \delta_{\text{call}}(q,a)$ and $q' \in \delta_{\text{ret}}(p',b,\gamma)$.

If we evaluate the term over this algebra we get the number of runs.

step 2. The algebra $\mathscr{F}(\mathscr{A}_n)$ has a subdomain which consists of maps of the form $\mathbb{N}^{[n]\times[n]} \to \mathbb{N}^{[n]\times[n]}$. Potentially the set of such maps is too large, but actually they are made up in a regular manner. The idea for a function $[n] \times [n] \to \mathbb{N}$ was to store how many paths there are between a pair of states for a given well-matched word. For functions $\mathbb{N}^{[n]\times[n]} \to \mathbb{N}^{[n]\times[n]}$ there is a similar picture. Given a well-matched word w_1w_2 where w_1 and w_2 not necessarily have to be well-matched, then a function f can be considered to be storing for given states q_1,q_2,q_3,q_4 how many ways there are to from q_1 to q_2 via w_1 and from q_3 to q_4 via w_2. We can now consider $f(d)$, where d is a function $d\colon [n] \times [n] \to \mathbb{N}$ which fills in the transitions from q_2 to q_3. If d resulted from evaluating a well matched word w, $f(d)$ is the evaluation corresponding to w_1ww_2.

The idea for our coding c is that we have to store natural numbers for these four-tuples of states. We set $c(\mathbb{N}^{[n]\times[n]}) = \mathbb{N}^{n,n}$ and

$$c((\mathbb{N}^{[n]\times[n]})^{\mathbb{N}^{[n]\times[n]}}) = (\mathbb{N}^{n,n})^{n,n}.$$

To assign a semantic to these matrices we define $\odot^c$ first:

- $\odot^c$: Given $c(f) \in (\mathbb{N}^{n,n})^{n,n}$ and $d \in \mathbb{N}^{n,n}$ we define the matrix $c(f(d)) = c(f \odot d) = c(f) \odot^c c(d) = A$ as follows. For a matrix like A we write $A(q_1,q_2)$ to address the entry which corresponds to the pair q_1,q_2. If we are given a matrix like $c(f)$ we write $c(f)(q_1,q_2)$ to address the matrix corresponding to q_1,q_2 and we set $c(f)(q_1,q_2)(q_3,q_4) = c(f)(q_1,q_2,q_3,q_4)$. Now $A(q_1,q_2)$ is defined as $\sum_{q_3,q_4 \in [n]} c(f)(q_1,q_2,q_3,q_4)c(d)(q_3,q_4)$. This is the sum of the entries of the point-wise matrix multiplication of $c(f)(q_1,q_2)$ and $c(d)$. Note that the coding of the identity map is $c(\text{id}) = I^{n,n}$, where I is the identity map of size n times n.
- $\circ^c$: Given $c(f)$ and $c(g)$ of $(\mathbb{N}^{n,n})^{n,n}$, then $c(f) \circ^c c(g)(q_1,q_2,q_3,q_4) = \sum_{q_5,q_6 \in [n]} c(f)(q_1,q_2,q_5,q_6)c(g)(q_5,q_6,q_3,q_4)$.
- $\circledast^c$: This is just the normal matrix multiplication.
- $\otimes^c_{a,b}$: Consider the matrix $M_{a,b} \in (\mathbb{N}^{n,n,})^{n,n}$, where $M_{a,b}(q_1,q_2,q_3,q_4) = 1$ if there exists $\gamma \in \Gamma$ such that $(q_2,\gamma) \in \delta_{\text{call}}(q_1,a)$ and $q_4 \in \delta_{\text{ret}}(q_3,b,\gamma)$ and $M_{a,b}(q_1,q_2,q_3,q_4) = 0$ else. Now we set $\otimes^c_{a,b}c(d) = M_{a,b} \odot^c c(d)$.
- $\widetilde{\otimes}^c_{a,b}$: This is similar to the previous case and we set $\widetilde{\otimes}^c_{a,b}c(f) = M_{a,b} \circ^c c(f)$.
- $\overleftarrow{\circledast}^c$: We set $c(f \overleftarrow{\circledast} d) = c(f) \overleftarrow{\circledast}^c c(d)$ as $c(f \overleftarrow{\circledast} d)(q_1,q_2) = \sum_{q_3 \in [n]} c(f)(q_1,q_3)c(d)(q_3,q_2)$ where the summation is a point-wise matrix summation and the multiplication is a scalar multiplication. The operation $\overrightarrow{\circledast}^c$ is defined similarly.

step 3. So far we have reduced the problem such that we know it is in $c(\mathscr{F}(\mathscr{A}))$-$\mathbf{NC}^1$. By considering the definition of the algebra operations above, one can see that in all cases arithmetic circuits of constant depth suffice. In particular we only use multiplication between two elements. The fan-in of addition gates is n. Hence we have a $\#\mathbf{SAC}^0_{\mathbb{N}}$ bound for the the operations. This again yields the $\#\mathbf{SAC}^1$ bound for the actual problem. □ □

Here the setting is very similar to the one for tree automata. By fixing a automaton or restriction to the Boolean case, we get the following theorems. Recall that the uniform membership problem is the membership problem, where the automaton is part of the input.

Theorem 33. *The uniform membership problem for nondeterministic VPA is in* $\mathbf{SAC}^1$.

Theorem 34 ([34])**.** *For a fixed nondeterministic VPA, counting the number of accepting runs is in* $\#\mathbf{NC}^1$.

Theorem 35 ([17])**.** *For a fixed VPA, the membership problem is in* $\mathbf{NC}^1$.

Weighted automata theory is a branch of theory which received ample attention since the 70s and has been decades of research. The original concept is based on finite automata however also generalizations to VPA have been investigated [12]. We define a weighted VPA (WVPA) based on a nondeterministic VPA and a semiring $(\mathbb{D},\oplus,\otimes)$. Then each transition of the VPA is assigned an element of $\mathbb{D}$. In a run, all weights are added by $\oplus$. Then the results for all the runs are multiplied by $\otimes$ which then is the output. That way a WVPA implements a function $\Sigma^* \to \mathbb{D}$. A typical example for the semiring is $(\mathbb{N},+,\min)$.

Theorem 36. *Functions of WVPA over a semiring* $\mathscr{A} = (\mathbb{D},\oplus,\otimes)$ *are in* $\mathscr{A}$-$\mathbf{NC}^1$.

Proof. **step 1.** In a WVPA M, the value of the word is computed by taking the sum (i.e. $\oplus$) of weights along all the computational runs regenarted by that word, where the weight of a run is nothing but the product of the weights of the steps taken by the WVPA along that path. An approach of computing the value of a given word by this approach is clearly inefficient since there can exist exponentially many runs.

Here, we again think of the input word as a term over an appropriate algebra. Then we can assign each well-matched subword w a value which is a map $Q \times Q \to \mathbb{D}$ where $(q_1,q_2) \to d \in \mathbb{D}$ gives a placeholder for the weight accumulated when going from q_1 to q_2 by reading w. Hence let

$$\mathscr{A}' = \left(\mathbb{D}^{Q\times Q}, \circledcirc, (\circledast_{a,b})_{a\in\Sigma_{\text{call}},b\in\Sigma_{\text{ret}}}, (\dagger_e)_{e\in\Sigma_{\text{int}}\cup\{\varepsilon\}}\right)$$

where $(f \circledcirc f)(q_1,q_2) = \bigotimes_{q\in Q} f(q_1,q) \oplus g(q,q_2)$ and $\dagger_e$ is 0-ary. Further $\circledast_{a,b}(f)(q_1,q_2) = \bigotimes_{q_1',q_2'\in Q,\gamma\in\Gamma} \mathsf{weight}(q_1,q_1',a,\gamma) \oplus f(q_1',q_2') \oplus \mathsf{weight}(q_2',q_2,b,\gamma)$. Here, $\mathsf{weight}\colon Q\times Q\times\Sigma_{\text{call}}\cup\Sigma_{\text{ret}}\times\Gamma\to\mathbb{D}$ maps to each transition its weight; if $a\in\Sigma_{\text{call}}$ then $\gamma\in\Gamma$ is the letter which is pushed on stack and if $b\in\Sigma_{\text{ret}}$ it is the one popped of stack. Now we build a term: The empty word has the identity map as corresponding value: $\dagger_\varepsilon$. An internal letter e corresponds to a map f where $f(q_1,q_2)$ is the weight associated with the transition $\delta(q_1,q_2,e)$: $\dagger_e$. If w_1 and w_2 are well matched words and f_1,f_2 are the corresponding terms, then $(f_1 \circledcirc f_2)$ is the term for w_1w_2. If w is well-matched and f is the term belonging to w then the term for the word awb is $\circledast_{a,b,}(f)$. It can be seen by induction that the constructed term evaluates to the function which tells us the weight for each pair of states. By assuming that there exists one initial and one final state, looking up at the pair of initial and final state we get the final output.

Using the construction and regarding the pair of initial state and final state one can obtain an $\mathscr{A}$-term which evaluates to the final output because the maps of $\mathscr{A}'$ can be considered matrices and the matrix operations can be made explicit.

step 2. and step 3. By Theorem 23 it follows that those terms over $\mathscr{A}$ can be evaluated in the bounds of $\mathscr{A}$-$\mathbf{NC}^1$. □

Applied, we directly obtain:

Theorem 37. *Functions of WVPA over* $(\mathbb{N},+,\times)$ *resp.* $(\mathbb{Z},+,\times)$ *are in* $\#\mathbf{NC}^1$ *resp. Gap*$\mathbf{NC}^1$.

A prime example for $\mathscr{A}$ in the context of weighted automata is $(\mathbb{N},+,\min)$, the tropical semiring. For this we get the following:

Theorem 38. *Functions of WVPA over* $(\mathbb{N},+,\min)$ *or* $(\mathbb{Z},+,\min)$ *where the output is coded binary are in* $\mathbf{SAC}^1$.

Proof. By Theorem 36 we know that this problem is in $(\mathbb{N},+,\min)$-$\mathbf{NC}^1$. The class $\mathbf{SAC}^1$ is an upper bound because addition and minimum can be computed in Boolean constant-depth circuits. □ □

Cost register automata(CRA) [3] are a different generalization of automata to capture quantitative properties. They are more powerful than weighted automata, however work somewhat differently. The idea is to let the states direct act on registers. A register holds a value of some algebra. Note that, here registers do not have influence on the states, i.e. the states which one reaches is only a function of the better being read and not the registered being updated. In [2] the complexity of CRA has been analyzed and in [31] a generalization to visibly pushdown automata (CVPA) has been made where complexity aspects also were considered. CVPA are based on deterministic VPA. Say it has a state set Q then we have in addition an algebra $\mathscr{A}$, a finite set of registers X, an initial valuation $v_0 \colon X \to \mathbb{D}$, and a register update function ρ:

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

Here $E(\mathscr{A}, X_{\text{prev}})$ is the set of expressions over the algebra $\mathscr{A}$, which may use variable names of X_{prev} which is a copy of X. In $E(\mathscr{A}, X_{\text{prev}}, X_{\text{prev}})$ there are two copies of the variable set X involved. The final cost function $\mu \colon Q \to E(\mathscr{A}, X)$ completes the definition. Now a CVPA implements a function $\Sigma^* \to \mathbb{D}$. On some input word we get the output value by updating the registers in each step according to the state and the letter read beginning with the initial valuation before the first letter is read. If a push letter is read, the register values are pushed onto the stack and when the corresponding return letter is read these values are made available again X_{match}-variables. After the word is read, all values can be combined, depending on the state, by μ. An equivalent and sometime beneficial interpretation is that a CVPA generates an $\mathscr{A}$-term which then is evaluated. However there are algebras which may lead to exponentially large terms, like $(\mathbb{N},+)$, so splitting the computation in term generation and evaluation is not feasible. Also note that there are algebras which lead to output values which need exponentially many bits. The algebra $(\mathbb{N},+\times)$ is an example for that. For details see [31].

Theorem 39 ([31])**.** *Functions realized by CVPA over* $(\mathbb{Z},+)$ *are in Gap*$\mathbf{NC}^1$.

Proof. **step 1.** We proceed as in the previous proofs by interpreting the well-matched input word as a term over an algebra such that the evaluation yields the desired value but here it is desirable to have the state information precomputed. Given a well-matched word w we can compute a word $r(w) \in (\Sigma \cup \{\varepsilon\} \times Q \times (\Gamma \cup \{\varepsilon\}))^{|w|+1}$ where $r(w)(i) = (w(i), q, \gamma)$ means that that automaton is in state q when $w(1)\dots w(i-1)$ is already read. Also if $w(i) \in \Sigma_{\text{ret}}$, then $\gamma \neq \varepsilon$ is the symbol which can be seen on the stack. For $r(|w|+1)$ we set $(\varepsilon, q, \varepsilon)$. Using Theorem 35, the word $r(w)$ can be computed in $\mathbf{NC}^1$.

Let the algebra be

$$\mathscr{A}_n = ((\mathbb{Z}^X)^{\mathbb{Z}^X}, \\ \circledcirc, (\otimes_{a,q_a,b,q_b,\gamma})_{a\in\Sigma_{\text{call}},b\in\Sigma_{\text{ret}},q_a,q_b\in Q,\gamma\in\Gamma}, (\dagger_{e,q})_{q\in Q,e\in\Sigma_{\text{int}}}, \dagger\varepsilon),$$

where X is the set of registers of the automaton and Q the set of states. The domain can be understood as a function which takes a valuation $\mathbb{Z}^X$ and transforms is into another valuation. If we evaluate the term which will correspond to a well-matched word, we get the transformation of the register values. So we define $\circledcirc$ as $\alpha \circledcirc \beta$ as the usual functional composition. The operation $\otimes_{a,q_a,b,q_b,\gamma}$ takes a valuation α

and then $\otimes_{a,q_a,b,q_b,\gamma}\alpha$ is defined as $\rho(q_a,a)\circledcirc\alpha\circledcirc\rho_1(q_b,b,\gamma)+\rho_2(q_b,b,\gamma)$, where $\rho(q_a,a)\colon \mathbb{Z}^X\to\mathbb{Z}^X$ is the register transformation map we naturally can derive of ρ. For return letters we distinguish between $\rho_1(q_b,b,\gamma)$ and $\rho_2(q_b,b,\gamma)$. An assignment of $\rho(q_b,b,\gamma,x)$ has the form $v_1x_1+\ldots+v_mx'_m+v_1x_1+\ldots+v_mx'_m$ where variables x_i correspond to values computed in the previous step and variables x'_i correspond to values which have been stored onto the stack in the matching position. Now $\rho_1(q_b,b,\gamma)$ is the map we get by omitting all variables x'_i and $\rho_2(q_b,b,\gamma)$ is the map we get by omitting all variables x_i. Finally $\dagger_{e,q}=\rho(q,e)$ and $\dagger_\varepsilon$ is the identity map.

From $r(w)$ we can then build the term. Note that $r(w)$ can also be considered a well-matched word. Inductively if there is a well-matched factor r in $r(w)$ with $r=r_1r_2$ such that r_1 and r_2 are also well-matched then let $T(r_1)$ and $T(r_2)$ be the terms of t_1 and t_2. Now the term for r is $T(r_1)\circledcirc T(r_2)$. All factors of length one which correspond to an internal letter e are assigned a term $\dagger_{e,q}$ for q. If there is a factor which corresponds to a word awb where $w\neq\varepsilon$ is well-matched, then the term for awb is $\otimes_{a,q_a,b,q_b,\gamma}T(w)$ for appropriate q_a,q_b,γ. A factor ab becomes $\otimes_{a,q_a,b,q_b,\gamma}\dagger_\varepsilon$.

Now the term evaluation yields the mapping f the automaton implements. If we then insert the initial valuation v_0 and apply the final update function, we have computed the output value $\mu(f(v_0))$.

step 2. The algebra $\mathscr{F}(\mathscr{A})$ has the domains $\mathbb{D}=(\mathbb{Z}^X)^{\mathbb{Z}^X}$ and

$$\widetilde{\mathbb{D}}\subseteq\left((\mathbb{Z}^X)^{\mathbb{Z}^X}\right)^{(\mathbb{Z}^X)^{\mathbb{Z}^X}}.$$

An element of $\mathbb{D}$ can be understood as an m-dimensional matrix of integers, where $m=|X|$. Hence the other domain consists of matrix-manipulating functions. As it turns out, these functions can be captured by functions of the form $x\mapsto AxB+C$ where A and B are matrices. So we choose $c(\mathbb{D})=\mathbb{Z}^{m,m}$ and $c(\widetilde{\mathbb{D}})=\mathbb{Z}^{m,m}\times\mathbb{Z}^{m,m}\times\mathbb{Z}^{m,m}$. By checking all operations of $\mathscr{F}(\mathscr{A})$, we show that this is actually a coding.

- $\odot^c$: Given $d\in\mathbb{D}$ and $f\in\widetilde{\mathbb{D}}$, then $f\odot d=f(d)$. Now $c(f)$ is a map $x\mapsto AxB+C$ and $c(d)$ is a matrix. So $c(f)\odot^c c(d)=c(f(d))=Ac(d)B+C$.
- $\circ^c$: Given $f,g\in\widetilde{\mathbb{D}}$, then f is of the form $x\mapsto A_fxB_f+C_f$ and g is of the form $x\mapsto A_gxB_g+C_g$. Now $c(f\circ g)=c(f)\circ^c c(g)$ is the map $x\mapsto A_f(A_gxB_g+C_g)B_f+C_f=A_fA_gxB_gB_f+A_fC_gB_f+C_f$, so $c(f\circ g)=(A_fA_g,B_gB_f,A_fC_gB_f+C_f)$.
- $\circledcirc^c$: When coded, $\circledcirc^c$ takes two matrices and multiplies them.
- $\otimes^c_{a,q_a,b,q_b,\gamma}$: This operation translates also into matrix multiplication. As by definition we have that $\otimes_{a,q_a,b,q_b,\gamma}\alpha$ translates to $\rho(q_a,a)\circledcirc\alpha\circledcirc\rho_1(q_b,b,\gamma)+\rho_2(q_b,b,\gamma)$. So we define a matrix $M_{q_a,a}$ from $\rho(q_a,a)$ and matrices $M^1_{q_b,b,\gamma}$ and $M^2_{q_b,b,\gamma}$ from $\rho_1(q_b,b,\gamma)$ and $\rho_2(q_b,b,\gamma)$. Now for $d\in\mathbb{D}$ we have $c(\otimes_{a,q_a,b,q_b,\gamma}d)=\otimes^c_{a,q_a,b,q_b,\gamma}c(d)=M_{q_a,a}c(d)M^1_{q_b,b,\gamma}+M^2_{q_b,b,\gamma}$.
- $\dagger_{e,q})^c_{q\in Q,e\in\Sigma_{\text{int}}}$: The coded version $\dagger_{e,q})^c_{q\in Q,e\in\Sigma_{\text{int}}}$ is a matrix $M_{e,q}$ corresponding to $\rho(q,e)$ and $\dagger^c_\varepsilon$ is the identity matrix.
- $\overleftarrow{\circledcirc}^c$: Given a function $f\in\widetilde{\mathbb{D}}$ and some $d\in\mathbb{D}$, we have $c(f\overleftarrow{\circledcirc}d)=c(f)\overleftarrow{\circledcirc}^c c(d)$ where $\overleftarrow{\circledcirc}^c$ is again a multiplication: If $c(f)$ is of the form $x\mapsto AxB+C$ then $c(f)\overleftarrow{\circledcirc}^c c(d)$ is of the form $x\mapsto(AxB+C)c(D)=AxBc(D)+Cc(D)$. The operation $\overrightarrow{\circledcirc}^c$ is defined analogously.
- $\widetilde{\otimes}^c_{a,q_a,b,q_b,\gamma}$: Given $f\in\widetilde{\mathbb{D}}$, we have $c(\otimes_{a,q_a,b,q_b,\gamma}f)=\otimes^c_{a,q_a,b,q_b,\gamma}c(f)$. If $c(f)$ is of the form $x\mapsto AxB+C$ then $\otimes^c_{a,q_a,b,q_b,\gamma}c(f)$ is of the form $x\mapsto M_{q_a,a}(AxB+C)M^1_{q_b,b,\gamma}+M^2_{q_b,b,\gamma}=M_{q_a,a}AxBM^1_{q_b,b,\gamma}+M_{q_a,a}CM^1_{q_b,b,\gamma}+M^2_{q_b,b,\gamma}$.

step 3. All operations of the algebra $c(\mathscr{F}(\mathscr{A}))$ are based on matrix operations and the domains are based on matrices of fixed dimensions. Because of that and since the matrices are of integer values, all operations of $c(\mathscr{F}(\mathscr{A}))$ are in Gap$\mathbf{NC}^0_{\mathbb{Z}}$. This leads to the upper bound of Gap$\mathbf{NC}^1$ for the problem in question. □

4.5 Definitions: Tree and clique width of graphs

The rest of the applications rely on width notions of graphs. In this subsection we present the basic graph theoretic definitions and also the different width concepts.

A graph is a tuple (V,E) where V is a set of vertices (or nodes) and $E \subseteq \binom{V}{2}$ is the set of edges. Here, $\binom{S}{n} \subseteq 2^S$ denotes the set of all subsets of S of size n. A directed graph is a tuple (V,E) where $E \subset V \times V$. A path in a graph is a sequence of connected edges and a cycle is a non-trivial path starting and ending in the same node. Directed acyclic graphs are abbreviated DAG. For basics in graph theory we refer e.g. to [16].

The tree width [25] is a parameter which has been successfully utilized to bound complexity, where Courcelle's Theorem is a prime example [14].

Definition 40. *Given a graph $G = (V,E)$ then (T,τ) is a tree decomposition, where $T = (V(T),E(T))$ is a tree and $\tau\colon V(T) \to 2^V$ is a map for which the following conditions hold:*

- *For each $v \in V$ there exists $b \in V(T)$ such that $v \in \tau(b)$.*
- *For each $(u,v) \in E$ there exists $b \in V(T)$ such that $\{u,v\} \subseteq t(b)$.*
- *If there is a path from $r \in V(T)$ to $s \in V(T)$ then for all nodes $t \in V(T)$ on the path holds that $\tau^{-1}(r) \cap \tau^{-1}(s) \subseteq \tau^{-1}(t)$*

The elements of $V(T)$ are called bags. The size of the largest bag minus one is the width of the decomposition $\mathsf{width}(T,\tau)$. *The minimal width of all decompositions of G is called the tree width of G which we denote as* $\mathsf{width}(G)$.

Besides tree decompositions we also consider a generalized notion of decomposition: NLC decompositions resp. clique decompositions [15, 48]. Both notions are closely related. A set of graphs has bounded clique width iff it has bounded NLC-width [15]. We will be only interested in the case of bounded width. Clique width has emerged as the more popular notion, so we speak mostly of clique width, but when it comes to decompositions we stick to the NLC variant as this has been used in the work our proofs are based on. For the rest we will only speak about clique width and decompositions even though it technically is NLC.

Given a graph $G = (V,E)$ and $k \in \mathbb{N}$, we can assign a coloring $l\colon V \to [k]$. A graph together with coloration using k colors is called a k-colored graph: (V,E,l).

Definition 41 (Clique decomposition of width k)**.** *A clique decomposition of width k of a graph G is a expression defined as follows:*

- *All k-colored graphs of the form $(\{v\},\emptyset,l)$ have clique width k.*
- *Given a colored graph (V,E,l) of width k and a map $l'\colon [k] \to [k]$ then $(V,E,l' \circ l)$ is also a k-colored graph.*
- *Given k-colored disjoint graphs $G_1 = (V_1,E_1,l_1)$ and $G_2 = (V_2,E_2,l_2)$ of width k and $S \subseteq [k] \times [k]$ then $G_1 \times_S G_2$ has also width k, where $G_1 \times_S G_2$ is defined as $(V_1 \cup V_2, E_1 \cup E_2 \cup E', l')$ and $E' = \{\{v_1,v_2\} \mid \exists (i,j) \in S \in [k]\colon v_1 \in l_1^{-1}(i) \wedge v_2 \in l_2^{-1}(j)\}$ and $l'(v) = l_1(v)$ if $v \in V_1$ and $l'(v) = l_2(v)$ if $v \in V_2$.*

If for a graph G a clique decomposition of width k exists, then G has clique width k.

If we want to compute our bounded width tree decompositions, we know from [18] that this is possible in log-space. For clique width the complexity is poly-time [41].

4.6 Circuits of bounded tree width

We apply the term evaluation algorithm to a recent result concerning circuits of bounded tree width [29]. It states that Boolean circuit families of polynomial size and bounded tree width can be balanced to obtain logarithmically deep circuit families. We show a short and generalized proof using term evaluation.

Whenever we speak of tree decompositions and tree width of a circuit we mean the corresponding underlying DAG of the circuit. The graph of a circuit satisfies some desirable properties, e.g. it is a DAG which has input and output gates. We want to decompose the graphs of circuits in a way to preserve these properties which leads to the following lemma.

Lemma 42. *For all $w \in \mathbb{N}$ there exists $c \in \mathbb{N}$ such that given a graph G of a circuit C and its minimal decomposition (T,τ) of width w, there exists a decomposition (T',τ') of C with $c \cdot \mathsf{width}(T,\tau) \geq \mathsf{width}(T',\tau')$ which satisfies:*

- *The tree T' is binary.*
- *If $u \in V(G)$ is a parent of v then let $p,q \in V(T')$ be the bags closest to the root satisfying $u \in t'(p)$ and $v \in t'(q)$. Then p is not closer to the root than q.*
- *For each input node $v \in V(G)$ there is a leaf $l \in V(T')$ such that $v \in \tau^{-1}(l)$.*
- *The output node of the circuit can be found in $\tau^{-1}(r)$, where r is the root of the tree.*

Proof. We can assume the tree T' to be binary without increasing the width, because for minimal decompositions the maximal rank of nodes is dependent on the width, hence bounded. Nodes with a rank greater than 2 can be resolved by a constant size construction.

The second requirement can be achieved by labeling those nodes by u which are labeled v and are closer to the root than all nodes labeled u. Since there is some $v \in V'$ such that $u,v \in l^{-1}(v)$, the result is again a valid tree decomposition.

The third requirement can be met by picking a node u labeled v and label the shortest path from u to some leaf with v. The last requirement can be implemented by labeling a path from a node labeled r to the root.

All operations at most need a constant factor in the width. □

By the lemma we get that assuming the stated properties preserves boundedness of tree width.

The proof idea for the following theorem is to interpret the tree decomposition as a term and evaluate it.

Consider a circuit C_n over an algebra $\mathscr{A} = (\mathbb{D}, \circledast_1, \ldots, \circledast_k)$ and let $G = (V,E)$ be the graph of C_n. Let the smallest tree decomposition following the previous lemma have width $w-1$. We define the algebra

$$\mathscr{A}(C_n,w) = \left(\mathbb{D}', (\circledast_{A,B,C})_{A,B,C \in \binom{V}{w}}, (\dagger_s)_{s \in S^{2w}}\right)$$

where $\mathbb{D}' = (\mathbb{D} \cup \{\bot\})^{2w}$ and $\circledast_{A,B,C}$ is an operation $\mathbb{D}' \times \mathbb{D}' \to \mathbb{D}'$ and S consists of all 0-ary operation values of $\mathscr{A}$ and $\bot$.. To define the operations assume V to be of the form $\{1,2,\ldots m\}$. Then A,B and C are sets of numbers. Also let $A = \{a_{g_1},\ldots,a_{g_{|A|}}\}$, $B = \{b_{h_1},\ldots,b_{h_{|A|}}\}$ and $C = \{c_{i_1},\ldots,c_{i_{|C|}}\}$. Consider $\alpha \circledast_{A,B,C} \beta$ where $\alpha,\beta \in (\mathbb{D} \cup \{\bot\})^{2w}$. For a node $a_{g_j} \in A$ the elements α_j and α_{w+j} correspond to the left and right parent of a_{g_j}. The situation for B and β resp. C and γ is similar. The following rules define the operation:

- If $a_{g_j} = c_{i_l}$ then $\alpha_j = \gamma_l$.
- If $b_{h_j} = c_{i_l}$ then $\beta_j = \gamma_l$.

- If c_{i_l} has parents which appear in α or β with values $v_1 \neq \perp$ and $v_2 \neq \perp$ and $\circledast$ is the operation of the gate c_{i_l}, then $\gamma_l = v_1 \circledast v_2$.
- In all other cases the result is $\perp$.

As the sets A, B and C are finite and there are only finitely many possibilities of ways how the gates can be wired we get that there is only a finite number of operations - independently of the actual circuit. Hence we write $\mathscr{A}(w)$ while dropping the circuit in the notation.

Theorem 43. *Given a family of circuits C of bounded tree width and polynomial size over an algebra $\mathscr{A}$, we can find an equivalent $\mathscr{F}(\mathscr{A}(w))$-$\mathbf{NC}^1$ circuit family in the sense that inputs and outputs are constant vectors which contain the input or output values respectively in some position, where $w-1$ is the width of the decomposition satisfying the conditions of Lemma 42.*

Proof. We take the decomposition of width $w-1$ satisfying the conditions of Lemma 42. We interpret it as a term over the algebra $\mathscr{A}(w)$ where each node v is assigned the operation $\circledast_{A,B,C}$ where $C = \tau(v)$ and B and C are the bags of the parents of v. To the left and right of the leafs must be constants. Such a constant s is a vector which is $\perp$ in all positions but those corresponding to an input gate; here the right input value is present. This then ends up being the operation $\dagger_s$.

This term can be evaluated by a $\mathscr{F}(\mathscr{A}(w))$-$\mathbf{NC}^1$ circuit. The output will be a vector which has positions p and $w+p$ which corresponds to the inputs of the output gate. Applying the function of the output gate to those two values yields the overall output. □

In summary, in the previous proof the word problem is solved in $\mathscr{F}(\mathscr{A}(w))$-$\mathbf{NC}^1$ by constructing a term and then evaluating it. For each circuit of the family we get one fixed term; only the constants are input-dependent. That means that actually we could fall back to a static version of our algorithm. In the algorithm in every step decisions have to be made which determine how to split subterms. Now since the structure of the term is fixed we could also fix those decisions (recall the circuits computing the cases) and end up directly with a logarithmic depth circuit without multiplexing.

Theorem 44 ([29])**.** *Languages accepted by families of Boolean circuits of polynomial size and bounded tree width are in* $\mathbf{NC}^1$.

Proof. Since $\mathscr{F}(\mathscr{A}(w))$ is finite, $\mathscr{F}(\mathscr{A}(w))$-$\mathbf{NC}^1 \subseteq \mathbf{NC}^1$ follows from Theorem 25 and 43. □

4.7 Courcelle's Theorem

Courcelle's Theorem [14] is a famous example of a meta theorem. It makes a claim concerning the complexity of the word problem if a restriction in the input set is imposed. In particular, given an MSO formula over graphs then Courcelle's Theorem states that it is decidable in linear time whether a graph is a model for the formula if we only consider graphs of some bounded tree width. The generality of the theorem stems from the fact that many relevant problems are expressible in MSO.

The algorithm has two steps. First a tree decomposition has to be computed and secondly the formula has to be fitted to tree decompositions. Checking a MSO formula on trees is then $\mathbf{NC}^1$. Elberfeld et al. [18] improved the overall complexity to log-space. In a follow-up paper they looked at the second step more closely and analyzed the complexity under the assumption that the tree decomposition is already given [19]. Besides confirming the $\mathbf{NC}^1$ bound in the Boolean case they considered an arithmetic version: Given an MSO formula and a free second-order variable X, how many valuations are there for X which satisfy the formula. The upper bound they achieved is $\#\mathbf{NC}^1$. We will re-prove this, however note that [19] has a bit more general setting of finite model theory. For simplicity of presentation we restrict ourselves to ordinary graphs and trees.

Here we consider finite graphs (V,E) with a labeling $V \to \Sigma$. A MSO formula is made of Boolean combinations, first and second order vertex quantification and predicates which are $Q_a(x)$, where $a \in \Sigma$ and x is a first order variable and tells whether position x is labeled a. Also $X(x)$ and $X \subseteq Y$ are predicates, where x is a first order variable and X and Y are second order variables. Lastly there is a predicate $E(x,y)$ for two first order variables which codes the edge relation E of the input graph.

Our proof of the theorem requires some preliminaries on forest algebras. Regular tree languages are accepted by finite forest algebras [7]. A forest algebra (H,V) consists of two (finite) monoids, the horizontal and the vertical monoid. The setting is very similar to the word case. There is also the concept of a syntactic forest algebra and recognition. Each tree corresponds to an element of H and depending whether this element is in the accepting set or not, the tree is accepted or rejected. We can turn the input tree into a term. If there is a node v labeled with $a_1 \in \Sigma$ and children v_1 and v_2 labeled with a_2 and a_3 and f_1 and f_2 are terms for v_1 and v_2 which are inductively given then the formula for v is $\odot_a^V(f_1 \odot^H f_2)$. The algebra then is $(H, \odot^H, (\odot_a^V)_{a\in\Sigma})$, where $\odot^H$ is the monoid operation of H and $\odot_a^V : H \to H$ is a unary operation which maps $t \mapsto c(a) \odot^V t$ where $\odot^V$ is the monoid operation of V and $c(a)$ is the context consisting of an node labeled a and one child which is a hole. Note that this is isomorphic to the algebra we used to show that visibly pushdown languages are in $\mathbf{NC}^1$.

Now we consider a counting problem. If we are given a formula with a free second-order variable, how many valuations for this variable exist satisfying the formula. This can be used to formulate counting versions of MSO-expressible problems.

Theorem 45 ([19])**.** *Given $w \in \mathbb{N}$, a graph G of tree width w, and its tree decomposition T as well as a MSO formula $\phi(X)$ with one free second-order variable X then the problem of counting how many valuations for X there are such that G satisfies $\phi(X)$ is in $\#\mathbf{NC}^1$.*

Proof. **step 1.**

Consider the proof for Courcelle's Theorem. Proving it takes the following steps:

- Compute the tree decomposition of the input graph.
- Compile the MSO formula into a new one which fits to tree decomposition.
- Check if the tree decomposition is a model for the new MSO formula.

The first one we do not care about since in our case the input already is a decomposition. So at this point we are interested in the second step. The standard construction [14] gives us the following: if $\psi(X)$ is an MSO formula over G with free second order variable X then the corresponding new formula $\psi'(X_1,\ldots,X_{w+1})$ over the tree decomposition T has $w+1$ free second order variables. For each $S \subseteq V(G)$ there exists exactly one corresponding $S' \subseteq V(T)^{w+1}$, i.e. $G \models \psi(S)$ iff $T \models \psi'(S')$. Note that subsets of $V(T)^{w+1}$ must have a certain form which is imposed by the construction of ψ'. Valuations that are not well-formed are dismissed. By the reasoning above it follows that the number of valuations for X which satisfy $G \models \psi(X)$ is equal to the number of valuations for $X_1,\ldots,X_{w+1}$ which satisfy $T \models \psi'(X)$. Hence we only have to show that we can count the number of fulfilling valuations in the formula over the tree decomposition.

In the following we assign formulas with free variables the semantics of accepting $\mathscr{V}$-structures [45]. In this case a $\mathscr{V}$-structure is a tree which is not only labeled with Σ but also with a bit which tells whether a position is in X or not; hence the alphabet then is $\Sigma \times \{0,1\}$ (or $\Sigma \times \{0,1\}^{w+1}$ if we have several free variables, respectively).

The idea then is that a formula with a free variable models a set of $\mathscr{V}$-structures. And each $\mathscr{V}$-structure belongs to a tree which we get by stripping it of the variable information. In the following we consider the language of $\mathscr{V}$-structures. Given a formula with a free variable and an input tree, we count how many $\mathscr{V}$-structures based on this tree fulfill the formula. This we will do using forest algebras to build an algebra.

Let $\phi'(X_1,\ldots,X_{w+1})$ be the MSO formula we get from $\phi(X)$ by the standard construction of [14]. Let (H,V) be the syntactic forest algebra of the tree language defined by $\phi'(X_1,\ldots,X_{w+1})$ interpreted over V-structures and consider the algebra

$$\mathscr{A} = (\mathbb{N}^H, \oplus^H, (\oplus_a^V)_{a\in\Sigma}).$$

The idea is that an element $f\colon H \to \mathbb{N}$ of this algebra keeps track of how many possibilities there are to end up with some element of H. The different possibilities are generated by the ways we can choose $X_1,\ldots,X_{w+1}$. So $\mathscr{A}$ can be used to count the number of assignments for $X_1,\ldots,X_{w+1}$. The operation $\oplus^H$ is defined as $f_1 \oplus^H f_2 = f$ where $f(h) = \sum_{h_1 \odot^H h_2 = h} f_1(h_1) f_1(h_2)$. The operation $\oplus_a^V$ is defined as $\oplus_a^V(f)(h) = \sum_{\odot_a^V(h')=h} f(h')$. From T we can construct a term inductively ψ over the algebra $\mathscr{A}$. For a node t labeled a and its descendants $t_1,\ldots,t_d$ the formula is $\oplus_a^V(f_1 \oplus^H \ldots \oplus^H f_d)$, where f_i is the formula for t_i.

If we evaluate ψ we get a map which tells us for each element of H how many ways there are to obtain it. If we sum all values which correspond to elements of the accepting subset of H we have the final output.

step 2. The algebra $\mathscr{F}(\mathscr{A})$ has the domains $\mathbb{N}^H$ and $\widetilde{\mathbb{D}} \subseteq (\mathbb{N}^H)^{\mathbb{N}^H}$. We code $c(\mathbb{N}^H) = \mathbb{N}^n$ where $n = |H|$ which is straight forward. As we only use addition and multiplication the result is that we can represent the elements of $\widetilde{\mathbb{D}}$ as functions of the form $x \mapsto xA + b$ where A is a matrix and b is a vector. Hence $c(\widetilde{\mathbb{D}}) = \mathbb{N}^{n,n} \times \mathbb{N}^n$. This conforms with the operations of the algebra:

- $\oplus^{H,c}$ and $\oplus_a^{V,c}$: Those two operations stay basically the same as $\oplus^H$ and $\oplus_a^V$.
- $\circ^c$: Given $f,g \in \widetilde{\mathbb{D}}$ with $c(f)\colon x \mapsto xA_1 + b_1$ and $c(g)\colon x \mapsto xA_2 + b_2$ we have that $c(f \circ g) = c(f) \circ^c c(g)$ is a map $x \mapsto (xA_2 + b_2)A_1 + b_1 = xA_2A_1 + b_2A_1 + b_1$, so $c(f) \circ^c c(g) = (A_2A_1, b_2A_1 + b_1)$.
- $\odot^c$: Given a function $f \in \widetilde{\mathbb{D}}$ with $c(f)\colon xA+b$ and a vector $c(d) \in \mathbb{N}^n$ we have $c(f \odot d) = c(f(d)) = c(f) \odot^c c(d) = x \mapsto dA + b$.
- $\overleftarrow{\oplus}^{H,c}$: Given a function $f \in \widetilde{\mathbb{D}}$ with $c(f)\colon xA + b$ and a vector $d \in \mathbb{N}^n$ we have that $c(f \overleftarrow{\oplus}^H d) = c(f) \overleftarrow{\oplus}^{H,c} c(d)$ is of the form $x \mapsto xAM_d + bM_d$ where M_d is a matrix where position (i,j) has value $\sum_{h_i = h_j h} d_h$ where $h_i, h_j \in H$ are the elements corresponding to vector positions i and j and d_h is the value of d representing h. The operation $\overrightarrow{\oplus}^{H,c}$ is done by a similar construction.
- $\widetilde{\oplus}_a^{V,c}$: Given a function $f \in \widetilde{\mathbb{D}}$ with $c(f)\colon xA + b$ we have that $c(\oplus_a^V f) = \oplus_a^{V,c} c(f)$ is the map $x \mapsto xAM_a + bM_a$ where M_a is a matrix where position (i,j) is 1 iff $\odot_a^V(h_i) = h_j$ $h_i, h_j \in H$ are the elements corresponding to vector positions i and j. In all other positions M_a is 0.

step 3. All operations operate on matrices and vectors of a fixed size with natural values. Hence we can implement them in $\#\mathbf{NC}^0_{\mathbb{N}}$ which yields the overall complexity of $\#\mathbf{NC}^1$. □

Since $\#\mathbf{NC}^1$ is a subset of log-space, we get that counting MSO problems on bounded tree-width graphs are also log-space.

4.8 Maximal cuts in graphs of bounded clique width

We consider the problems of finding maximal cuts in graphs. This is one of the most well-studied problems in theoretical computer science since its introduction in Karp's classical 21 NP-complete problems [30]. In [48] it was shown that it becomes tractable if we impose a restriction on the input graph. This restriction is that the clique-width is bounded. This notion is related to NLC-width [48]: A graph has bounded clique width if and only if it has bounded NLC-width. We show an improvement of the upper bound from **P** to parallel complexity.

The maximum cut problem for width k is the following problem: Given a clique decomposition of an undirected graph $G = (V,E)$ of clique width k. Now let $V_1 \cup V_2$ be a partition of V such that the cardinality of the set $\{\{e_1,e_2\} \in E \mid e_1 \in V_1 \wedge e_2 \in V_2\}$ is maximal. The output is the value of the maximal partition. A partition is also called a cut.

In the following we revisit the proof form [48] and show an upper bound of $\mathbf{SAC}^1$.

Theorem 46. *The maximum cut problem for width k is in* $\mathbf{SAC}^1$.

Proof. **step 1.** We are given a clique decomposition. This is a tree and this tree we can interpret as a term over some algebra that, if evaluated, results in the actual graph. We now assign a new family of algebras $(\mathscr{A}_n)_{n\in\mathbb{N}}$ to that term. If evaluated we get the desired value. So let

$$\mathscr{A}_n = \left(\mathscr{P}\left([n]^{2k+1}\right), (\otimes_l)_{l\colon [k]\to[k]}, (\circledast_S)_{S\subseteq[k]\times[k]}, (\dagger_i)_{i\in[k]}\right).$$

We choose n to be $|V|$. This is formally not a family of algebras but it can be made into one. The way as it is, is however easier to understand.

So each element is a set of vectors of the form $(a_1,\dots,a_k,b_1,\dots,b_k,c)$. The intuition behind this is that each a_i counts how many elements of V_1 are labeled i and each b_i counts how many elements of V_2 are labeled i. The number c then stores the value of the corresponding cut [48].

The operations are defined as follows.

- There are 0-ary operations $\dagger_i$ for $i \in [k]$ is a set containing one tuple corresponding to the graph of one vertex colored i.
- For each total map $l\colon [k] \to [k]$ there is a unary operations $\otimes'_l\colon \mathbb{N}^{2k+1} \to \mathbb{N}^{2k+1}$ with $(a_1,\dots,a_k,b_1,\dots,b_k,c) \mapsto (a'_1,\dots,a'_k,b'_1,\dots,b'_k,c)$ where $a'_i = \sum_{j\in l^{-1}(i)} a_j$ and $b'_i = \sum_{j\in l^{-1}(i)} b_j$. Then for $\mathbb{D}$ being the domain, $\otimes_l\colon \mathbb{D}\to\mathbb{D}$ is derived from $\otimes'$ by the following map: $\{x_1,\dots,x_m\} \mapsto \{\otimes'_l(x_1),\dots,\otimes'_l(x_m)\}$. These unary operations directly correspond to the unary relabeling operations from the clique width definition.
- For each $S \subseteq [k]\times[k]$ there is an operation of the form $\circledast_S\colon \mathbb{D}\times\mathbb{D}\to\mathbb{D}$. It maps $X \circledast_S Y \mapsto \bigcup_{x\in X, y\in Y}\{\circledast'_S(x,y)\}$. Let $x = (a'_1,\dots a'_k,b'_1,\dots b'_k,c)$, $y = (a''_1,\dots a''_k,b''_1,\dots b''_k,c)$ and $\circledast'_S(x,y) = (a_1,\dots a_k,b_1,\dots b_k,c)$. Then $a_i = a'_i + a''_i$ and $b_i = b'_i + b''_i$. Further $c = c' + c'' + \sum_{(i,j\in S)} a'_i \cdot b''_j + b'_i \cdot a''_j$.

The evaluation of this term yields the desired value [48].

step 2. We first give a coding for $\mathscr{A}_n$ and then extend it to $\mathscr{F}(\mathscr{A}_n)$. Consider the domain of $\mathscr{A}_n$ which is $\mathscr{P}\left([n]^{2k+1}\right)$. The set $[n]^{2k+1}$ has polynomial size. Hence we can represent each element of the domain by a word of $\{0,1\}^{n^{2k+1}}$, where each position holds the information whether the corresponding tuple is part of the set. Let $\phi\colon [n]^{2k+1} \to [n^{2k+1}]$ be a bijection and let

$$c\colon \mathscr{P}\left([n]^{2k+1}\right) \to \{0,1\}^{n^{2k+1}}$$

be a code with $c(X)$ being a string of length n^{2k+1} which is 1 in position i if and only if there is an $x \in X$ such that $\phi(x) = i$ and 0 otherwise.

Now, in $\mathscr{F}(\mathscr{A}_n)$ we also have the subdomain $\widetilde{\mathbb{D}}$ which contains functions of the form $f\colon \mathscr{P}\left([n]^{2k+1}\right) \to \mathscr{P}\left([n]^{2k+1}\right)$. We will use the property $f(X) = \bigcup_{x\in X} f(\{x\})$ of these functions which we call *singleton property*. We will now observe that the functions indeed have the singleton property. First, the identity function clearly has it. Further if we are given two functions f,g which have the singleton property, then $f\circ g$ has also does: $(f\circ g)(X) = f(g(X)) = f(\bigcup_{x\in X} g(\{x\})) = \bigcup_{x\in X} f(g(\{x\}))$. For $\widetilde{\otimes}_l f$ we get $\widetilde{\otimes}_l f(X) = \widetilde{\otimes}_l \bigcup_{x\in X} f(\{x\}) = \bigcup_{x\in X} \widetilde{\otimes}_l f(\{x\})$ since $\widetilde{\otimes}_l f(X) \subseteq \widetilde{\otimes}_l f(Y)$ iff $X \subseteq Y$. Lastly

for $\overleftarrow{\circledast}_S$ and $\overrightarrow{\circledast}_S$ we see that the singleton property holds since $\circledast_S$ is already defined as a union over singletons.

The consequence of the singleton property is that each map can be represented by only considering the image of singleton inputs. So a coding of f becomes a table:

$$c\colon \left(\mathscr{P}\left([n]^{2k+1}\right)^{\mathscr{P}\left([n]^{2k+1}\right)}\right) \to \left(\{0,1\}^{n^{2k+1}}\right)^{n^{2k+1}}.$$

An element is a table, wherein the i'th line holds $c(f(\phi^{-1}(i)))$, hence $c(f) = c(f(\phi^{-1}(1)))\ldots c(f(\phi^{-1}(n^{2k+1})))$. The definition of the coded functions of $\mathscr{F}(\mathscr{A})$ follow immediately.

step 3. We now are interested in the complexity of the operations of

$$c(\mathscr{F}(\mathscr{A}_n)) = (\{c(\mathbb{D}), c(\widetilde{\mathbb{D}})\}, (\otimes_l^c)_{l\colon [k]\to[k]}, (\circledast_S^c)_{S\subseteq[k]\times[k]}, \circ^c, \odot^c,$$
$$(\widetilde{\otimes}_l^c)_{l\colon [k]\to[k]}, (\overleftarrow{\circledast}_S^c)_{S\subseteq[k]\times[k]}, (\overrightarrow{\circledast}_S^c)_{S\subseteq[k]\times[k]}),$$

as well as the complexity of the multiplexer operations for the two subdomains. The multiplexer operations can be implemented by constant size Boolean circuits with regard to one output bit.

- $\otimes_l^c$: Consider a string $c(d) \in c(\mathbb{D})$ and the result $\otimes_l^c c(d)$. For each $x \in c^{-1}(\otimes_l^c c(d))$ there exists a set Y containing all y such that $x \in \otimes_l(\{y\})$. Now to compute $\otimes_l^c c(d)$, if a bit corresponds to x then it is the result of a disjunction of all positions in $c(d)$ that correspond to an element of Y. This is a $\mathbf{SAC}^0$-construction.

- $\circledast_S^c$: Consider $X \circledast_S Y \mapsto \bigcup_{x\in X, y\in Y}\{\circledast'_S(x,y)\}$. So for each element in $z \in (X \circledast_S Y)$ there exists a number of pairs x_i, y_i such that $\{x_i\} \circledast_S \{y_i\} = \{z\}$. Now in the coded version where we have strings instead of sets, each position in the output string becomes a disjunction over all these pairs and each pair is a conjunction of two. Hence this operation can also be implemented in $\mathbf{SAC}^0$.

- $\circ^c$: To compute $c(f) \circ^c c(g)$ we have to build a table which represents the function. To that end, define a table $t(d_i)$, where d_i is the i'th row of $c(g)$. Then the j'th row of $t(d_i)$ is the $j'th$ row of the pointwise conjunction of $c(f)$ with $d_j(i)$. Now k'th letter of the i'th row of $c(f) \circ^c c(g)$ is the disjunction of the k'th column of $t(d_j)$. This construction needs fan-in two conjunctions and unbounded fan-in disjunctions, hence it it $\mathbf{SAC}^0$.

- $\odot^c$: The computation of $c(f) \odot^c c(d)$ can be reduced to $c(f) \circ^c c(d')$ where d' is a constant function with $d'(x) = d$. The table $c(d')$ we get by filling all rows with $c(d)$. Then in $c(f) \circ^c c(d')$ also each row is identical since it codes a constant function. Take one of the rows as output for $c(f) \odot^c c(d)$.

- $\widetilde{\otimes}_l^c$: This case is similar to $\otimes_l^c$ with the difference that the input is a coded function, hence a table. We apply $\otimes_l^c$ to all rows of the table.

- $\overleftarrow{\circledast}_S^c$: To compute the table $c(f)\overleftarrow{\circledast}_S^c c(d)$, we can use $\circledast^c$. Let r_i be the i'th row of $c(f)$. Then the i'th row of $c(f)\overleftarrow{\circledast}_S^c c(d)$ is $r_i \circledast^c c(d)$.

- $\overrightarrow{\circledast}_S^c$: This case is similar to $\overleftarrow{\circledast}_S^c$.

Since all operations are in $\mathbf{SAC}^0$, the whole problem is in $\mathbf{SAC}^1$.

□

4.9 Counting Hamiltonian paths and Euler tours in graphs of bounded clique width

Besides computing maximal cuts, the problem of computing the Hamiltonian circuits in graphs was also considered in [48]. They showed a poly-time upper-bound for this problem for bounded tree width graphs. It is also possible to count the number of Hamiltonian circuits. In [5] a $\#\mathbf{SAC}^1$ upper bound has been shown for the case of bounded tree width. Here we will prove the same $\#\mathbf{SAC}^1$ upper bound in the cae of bounded clique width graphs.

A path is a sequence of vertices $p = p_1 \dots p_m$ such that no vertex appears more than once and $\{p_i, p_{i+1}\} \in E$ for $1 \leq i < m$. A Hamiltonian circuit is a union of the following two things: a path of length $|V|$ and an edge between the last vertex in the path and the first vertex. The Hamiltonian circuit problem for width k is the following: Given a clique decomposition of an undirected graph $G = (V,E)$ of width k check whether there exists a Hamiltonian circuit in G.

Theorem 47. *Given a natural number k, computing the number of Hamiltonian circuits in clique width k graphs where the clique decomposition is given in the input is in* $\#\mathbf{SAC}^1$.

Proof. **step 1.** As in the case of the maximum cut problem, we are given a tree decomposition as a term and we assign an algebra to it such that the evaluation yields the desired result. Actually we assign a family of algebras $(\mathscr{A}_n)_{n\in\mathbb{N}}$:

$$\mathscr{A}_n = \left(\mathbb{N}^{\left([n]^{k(k+1)/2}\right)}, (\otimes_l)_{l\colon [k]\to[k]}, (\circledast_S)_{S\subseteq[k]\times[k]}, (\dagger_i)_{i\in[k]}\right)$$

The variable n can be chosen as $|V|$, as in the case of the maximum cut problem. This algebra is rooted in the construction for the Boolean version in [48] where they used $\mathscr{P}([n]^{k(k+1)/2})$ as domain. Instead of holding the information whether a tuple is in a set, we count how often is has been occurring. Now an element of $[n]^{k(k+1)/2}$ corresponds to a subset of the edges covering the vertices. We can understand this as a path coverage of V. We have many paths and each vertex is present in exactly one. Now the information the tuple actually holds is how many such paths go between two colors. See [48] for further details. The domain we chose now counts how many such path coverings result in a certain tuple.

The operations of the algebra are defined as follows.

- The 0-ary operation $\dagger_i$ is the characteristic function of the set containing the single tuple corresponding to a graph with a single node colored i.
- For each total map $l\colon [k] \to [k]$ there is a unary operation

  $$\otimes_l\colon \mathbb{N}^{\left([n]^{k(k+1)/2}\right)} \to \mathbb{N}^{\left([n]^{k(k+1)/2}\right)}$$

  which is defined using a unary operation $\otimes'\colon [n]^{k(k+1)/2} \to [n]^{k(k+1)/2}$ with

  $$\otimes'_l(v)_{i,j} = \sum_{i'\in l^{-1}(i), j'\in l^{-1}(j)} v_{i',j'}$$

  as defined in [48]. Now

  $$\otimes_l(f)(v) = \sum_{v'\in \otimes_l'^{-1}(v)} f(v').$$

- For each $S \subseteq [k] \to [k]$ there is an operation

  $$\circledast_S\colon \mathbb{N}^{\left([n]^{k(k+1)/2}\right)} \times \mathbb{N}^{\left([n]^{k(k+1)/2}\right)} \to \mathbb{N}^{\left([n]^{k(k+1)/2}\right)}.$$

  This operation is a counting version of the corresponding operation described in [48]. There it is defined by a procedure which generates new elements based on present elements. In our case

we also have to keep track of the count of paths generating a certain element. Given two vectors $v_1, v_2 \in [n]^{k(k+1)/2}$ a new set of vectors is generated. This is done by defining tuples (A,B,C) the initial tuple being $(v_1,0,v_2)$.

We want to define $(f \circledast_S g)(v)$ for all $v \in [n]^{k(k+1)/2}$ and define a procedure which yields the value. First assume the values $(f \circledast_S g)(v)$ to be 0 for all v. Then for all $v_1, v_2 \in [n]^{k(k+1)/2}$ do the steps of [48] for generating a new set of tuples. In each step one new edge is drawn. That way we get a DAG which originates in $(v_1,0,v_2)$. Actually we are only interested in a spanning tree which we get by imposing an order of the elements of S we process. We assign each triple (A,B,C) a number $\#(A,B,C)$. The initial triple $(v_1,0,v_2)$ is assigned $f(v_1)g(v_2)$. Now assume we get from triple (A,B,C) to (A',B',C') in one step. Then $\#(A',B',C') = p \cdot \#(A,B,C)$ where p is the number of possibilities to draw an edge; p is fixed by (A,B,C). Each triple can be made into an element $v \in [n]^{k(k+1)/2}$ as seen in [48]. Let $\#(v,v_1,v_2) = \#(A,B,C)$ where v_1 and v_2 are the origins of (A,B,C) and v is the vector we get from (A,B,C). Now

$$(f \circledast_S g)(v) = \sum_{v_1,v_2 \in [n]^{k(k+1)/2}} \#(v,v_1,v_2).$$

In this sum, every summand has the factor $f(v_1)g(v_2)$ as we can combine every path covering in f which leads to the tuple v_1 with everyone of g which leads to v_2. Then this is multiplied with the number of ways we can draw edges between the two graphs.

For obtaining the Hamilton paths we have to treat the last $\circledast_S$ operation (the root of the term tree) differently. We generate the triples and then, as described in [48], if the situation occurs that a triple (A,B,C) has A and B to only consist of 0 and B has exactly one value which is non-zero then, if S indicates that we can close the loop, we have found a path. That means this would then result in a triple all zero. Now in our counting setting we sum over all those zero-triples generated in that way and hence we get the final resultant term over the algebra whose evaluation is the desired value.

step 2. We proceed as in the case of maximal cuts. We want to code $\mathscr{F}(\mathscr{A}_n)$. The algebra $\mathscr{A}_n$ has the domain $\mathbb{N}^{\left([n]^{k(k+1)/2}\right)}$. We show how to code it and extend the code to $\mathscr{F}(\mathscr{A}_n)$. So let $\phi\colon [n]^{k(k+1)/2} \to [n^{k(k+1)/2}]$ be a bijection and set $c(\mathbb{N}^{\left([n]^{k(k+1)/2}\right)}) = \mathbb{N}^{n^{k(k+1)/2}}$ where $c(f)$ is a sequence of natural numbers where $c(f)_i = f(\phi^{-1}(i))$.

For $\mathscr{F}(\mathscr{A}_n)$ we have to consider the second subdomain which consists of operations F of the form $F\colon \mathbb{N}^{\left([n]^{k(k+1)/2}\right)} \to \mathbb{N}^{\left([n]^{k(k+1)/2}\right)}$. In the case of maximal cuts we mentioned the singleton property these functions possess. A similar singleton property we can find for the present case: The union becomes a sum. So we observe that

$$F(f) = \sum_{v \in [n]^{k(k+1)/2}} F(f(v)\chi_{\{v\}}) = \sum_{v \in [n]^{k(k+1)/2}} f(v)F(\chi_{\{v\}})$$

where the sum is a sum over functions and $\chi_{\{v\}}$ is the characteristic function of $\{v\}$

To verify the presence of the singleton property we begin with the identity function which has it. Further we have to consider the operations of $\mathscr{F}(\mathscr{A}_n)$. For The functional composition we get

$$\begin{aligned}(F \circ G)(f) = F(G(f)) &= F\Big(\sum_{v \in [n]^{k(k+1)/2}} G(f(v)\chi_{\{v\}})\Big) \\ &= \sum_{v \in [n]^{k(k+1)/2}} F(G(f(v)\chi_{\{v\}})) \\ &= \sum_{v \in [n]^{k(k+1)/2}} f(v)F(G(\chi_{\{v\}})).\end{aligned}$$

For $\widetilde{\otimes}_l F$ we get

$$\begin{aligned}\widetilde{\otimes}_l F(f) =& \widetilde{\otimes}_l \sum_{v\in[n]^{k(k+1)/2}} F(f(v)\chi_{\{v\}}) \\ =& \sum_{v\in[n]^{k(k+1)/2}} \widetilde{\otimes}_l F(f(v)\chi_{\{v\}}) \\ =& \sum_{v\in[n]^{k(k+1)/2}} f(v)\widetilde{\otimes}_l F(\chi_{\{v\}})\end{aligned}$$

by a similar argument as in the corresponding case for the maximal cuts result. Also similarly $\circledast_S$ is already defined in a way which is a sum in the desired form.

Now by using this property when coding an operation F, we only need to store all maps of the form $\chi_{\{v\}}$. Hence:

$$c\colon \left(\mathbb{N}^{\left([n]^{k(k+1)/2}\right)^{\mathbb{N}^{\left([n]^{k(k+1)/2}\right)}}}\right) \to \left(\mathbb{N}^{n^{k(k+1)/2}}\right)^{n^{k(k+1)/2}}.$$

The elements of this can be understood as a table where the i'th line is $c(F(\chi_{\{\phi^{-1}(i)\}}))$. The definition of the operations of $\mathscr{F}(\mathscr{A}_n)$ follows.

step 3. We analyze the complexity of the operations of

$$\begin{aligned}c(\mathscr{F}(\mathscr{A}_n)) = (\{c(\mathbb{D}), c(\widetilde{\mathbb{D}})\}, (\otimes_l^c)_{l\colon [k]\to[k]}, (\circledast_S^c)_{S\subseteq[k]\times[k]}, \circ^c, \odot^c,\\ (\widetilde{\otimes}_l^c)_{l\colon [k]\to[k]}, (\overleftarrow{\circledast}_S^c)_{S\subseteq[k]\times[k]}, (\overrightarrow{\circledast}_S^c)_{S\subseteq[k]\times[k]}),\end{aligned}$$

and the complexity of multiplexers as well. Multiplexing elements of $c(\mathbb{D})$, resp. $c(\widetilde{\mathbb{D}})$ which are just sequences of naturals can be done in $\#\mathbf{NC}^0$. For the rest many ideas are very similar to the proof for maximal cuts, so we keep similar constructions short.

- $\otimes_l^c$: We want to compute $\otimes_l^c c(F)$ which is a table. For convenience assume a continuation of l to $l\colon [n]^{k(k+1)/2} \to [n]^{k(k+1)/2}$ defined as $l(v_{i,j}) = \sum_{i'\in l^{-1}(i), j'\in l^{-1}(j)} v_{i',j'}$. The i'th row of the table consists of $\otimes_l^c c(F(\chi_{\{\phi^{-1}(i)\}}))$ which we can compute from the $c(F(\chi_{\{\phi^{-1}(i)\}}))$ which again is a row of a table for $c(F)$. Now we get the row as $\otimes_l^c c(F(\chi_{\{\phi^{-1}(i)\}})) = c(F(\chi_{l^{-1}(\{\phi^{-1}(i)\})}))$. In terms of complexity this translates into the need for unbounded fan-in summation gates and yields a $\#\mathbf{SAC}^0$-bound for $\otimes_l^c$.

- $\circledast_S^c$: We want to compute $c(f)\circledast_S^c c(f) = c(f\circledast_S g)$ for $f,g\colon [n^{k(k+1)/2}] \to \mathbb{N}$. This is a sequence of naturals and the i'th position is $(f\circledast_S g)(\phi^{-1}(i)) = \sum_{v_1,v_2\in[n]^{k(k+1)/2}} \#(v,v_1,v_2)$ where $v = \phi^{-1}(i)$. So given v, v_1, v_2 we basically have to compute $\#(v,v_1,v_2)$. Keep in mind how we defined $\#(v,v_1,v_2)$ by constructing a tree of triples (A,B,C). This tree has at most depth nk^2. By adjusting the construction we can get a tree of depth k^2 by choosing the number edges for a certain pair of S in parallel. Instead of investing one step in depth for every singe edge. All edges which correspond to one pair of S are inserted at once. The corresponding number $\#(A,B,C)$ consists of factors $f(v_1)$, $g(v_2)$ and factors we get for each edge in the tree. These factors can be hard-coded. By then picking the right number we obtain $\#(v,v_1,v_2)$ and can do the summation $\sum_{v_1,v_2\in[n]^{k(k+1)/2}} \#(v,v_1,v_2)$. As the depth of the trees we construct is constant in n we need only bounded fan-in multiplication gates. Further we need an unbounded addition gate. This gives us a $\#\mathbf{SAC}^0$ bound for $\circledast_S^c$.

- $\circ^c$: We want to compute $c(F\circ G)$ which is a table and $c((F\circ G)(\chi_{\phi^1(i)}))$ is the i'th row of the table. We are given the tables for $c(F)$ and $c(G)$. To compute the i'th row of $c(F\circ G) = c(F)\circ^c c(G)$, take the i'th row of $c(G)$; let r_i denote this row. Now multiply the j'th row of $c(F)$ by the j'th

element in r_i, that is $r_{i,j}$; let the resulting row be $r'_{i,j}$. Now the i'th row of $c(F) \circ^c c(G)$ is the point-wise sum of $r'_{i,j}$ for all j. In the construction we had to multiply pairs of numbers. Further, addition gates with fan-in of at most $n^{k(k+1)/2}$ were needed which gives us an $\#\mathbf{SAC}^0$ bound for computing the composition.

- $\odot^c$: Computing $c(F) \odot^c c(d)$ can be reduced by using $\circ^c$: $c(F) \odot^c c(d) = c(F) \circ^c c(d')$ where d' is a constant function of value d.
- $\widetilde{\otimes}^c_l$: By invoking $\otimes^c_l$ and applying it on all rows of the argument we get the result.
- $\overleftarrow{\circledast}^c_S$: If we want to compute $c(F)\overleftarrow{\circledast}^c_S c(d)$ we apply $c(d)$ on each row by $\overleftarrow{\circledast}^c_S$ and by that have a reduction.
- $\overrightarrow{\circledast}^c_S$: This case is similar to $\overleftarrow{\circledast}^c_S$.

All operations are in the bounds of $\#\mathbf{SAC}^0_{\mathbb{N}}$, so the original problem is in $\#\mathbf{SAC}^1$.

□ □

If we are only interested in whether a Hamiltonian circuit exists then we see that the previous construction can be easily made Boolean to yield the following result:

Theorem 48. *The Hamiltonian circuit problem for width k is in* $\mathbf{SAC}^1$.

5 Discussion

We have seen that many problems that have a tree-like structure can be solved in parallel using our term evaluation algorithm. The list of problem we covered here should indicate the potential of the framework. We expect that there will be many more applications, which can be derived using the unifying framework presented here.

References

[1] Manindra Agrawal, Thanh Minh Hoang, and Thomas Thierauf. The polynomially bounded perfect matching problem is in nc^ 2. In *STACS*, volume 4393, pages 489–499, 2007.

[2] Eric Allender and Ian Mertz. Complexity of regular functions. In Adrian Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, volume 8977 of *Lecture Notes in Computer Science*, pages 449–460. Springer, 2015. URL: `http://dx.doi.org/10.1007/978-3-319-15579-1_35`, `doi:10.1007/978-3-319-15579-1_35`.

[3] Rajeev Alur, Loris D'Antoni, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 13–22. IEEE Computer Society, 2013. URL: `http://dx.doi.org/10.1109/LICS.2013.65`, `doi:10.1109/LICS.2013.65`.

[4] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 202–211. ACM, 2004. URL: `http://doi.acm.org/10.1145/1007352.1007390`, `doi:10.1145/1007352.1007390`.

[5] Nikhil Balaji, Samir Datta, and Venkatesh Ganesan. Counting euler tours in undirected bounded treewidth graphs. In Prahladh Harsha and G. Ramalingam, editors, *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India*, volume 45 of *LIPIcs*, pages 246–260. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. URL: http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2015.246, doi:10.4230/LIPIcs.FSTTCS.2015.246.

[6] Christoph Behle and Klaus-Jörn Lange. Fo[<]-uniformity. In *21st Annual IEEE Conference on Computational Complexity (CCC 2006), 16-20 July 2006, Prague, Czech Republic*, pages 183–189. IEEE Computer Society, 2006. URL: http://dx.doi.org/10.1109/CCC.2006.20, doi:10.1109/CCC.2006.20.

[7] Mikołaj Bojańczyk and Igor Walukiewicz. Forest algebras. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*., volume 2 of *Texts in Logic and Games*, pages 107–132. Amsterdam University Press, 2008.

[8] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974. URL: http://doi.acm.org/10.1145/321812.321815, doi:10.1145/321812.321815.

[9] Samuel R. Buss. The boolean formula value problem is in ALOGTIME. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 123–131. ACM, 1987. URL: http://doi.acm.org/10.1145/28395.28409, doi:10.1145/28395.28409.

[10] Samuel R. Buss. Algorithms for boolean formula evaluation and for tree contraction. In *Arithmetic, Proof Theory and Computational Complexity*, pages 96–115. Oxford University Press, 1993.

[11] Samuel R. Buss, Stephen A. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM J. Comput.*, 21(4):755–780, 1992. URL: http://dx.doi.org/10.1137/0221046, doi:10.1137/0221046.

[12] Mathieu Caralp, Pierre-Alain Reynier, and Jean-Marc Talbot. Visibly pushdown automata with multiplicities: Finiteness and k-boundedness. In Hsu-Chun Yen and Oscar H. Ibarra, editors, *Developments in Language Theory - 16th International Conference, DLT 2012, Taipei, Taiwan, August 14-17, 2012. Proceedings*, volume 7410 of *Lecture Notes in Computer Science*, pages 226–238. Springer, 2012. URL: http://dx.doi.org/10.1007/978-3-642-31653-1_21, doi:10.1007/978-3-642-31653-1_21.

[13] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–21, 1985. URL: http://dx.doi.org/10.1016/S0019-9958(85)80041-3, doi:10.1016/S0019-9958(85)80041-3.

[14] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990. URL: http://dx.doi.org/10.1016/0890-5401(90)90043-H, doi:10.1016/0890-5401(90)90043-H.

[15] Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1-3):77–114, 2000. URL: http://dx.doi.org/10.1016/S0166-218X(99)00184-5, doi:10.1016/S0166-218X(99)00184-5.

[16] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[17] Patrick W. Dymond. Input-driven languages are in log n depth. *Inf. Process. Lett.*, 26(5):247–250, 1988. URL: http://dx.doi.org/10.1016/0020-0190(88)90148-2, doi:10.1016/0020-0190(88)90148-2.

[18] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of bodlaender and courcelle. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 143–152. IEEE Computer Society, 2010. URL: http://dx.doi.org/10.1109/FOCS.2010.21, doi:10.1109/FOCS.2010.21.

[19] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Algorithmic meta theorems for circuit classes of constant and logarithmic depth. In Christoph Dürr and Thomas Wilke, editors, *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, February 29th - March 3rd, 2012, Paris, France*, volume 14 of *LIPIcs*, pages 66–77. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012. URL: http://dx.doi.org/10.4230/LIPIcs.STACS.2012.66, doi:10.4230/LIPIcs.STACS.2012.66.

[20] Stephen A. Fenner, Rohit Gurjar, and Thomas Thierauf. Guest column: Parallel algorithms for perfect matching. *SIGACT News*, 48(1):102–109, 2017. URL: http://doi.acm.org/10.1145/3061640.3061655, doi:10.1145/3061640.3061655.

[21] Merrick Furst, James B Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Theory of Computing Systems*, 17(1):13–27, 1984.

[22] Moses Ganardi and Markus Lohrey. A universal tree balancing theorem. *CoRR*, abs/1704.08705, 2017. URL: http://arxiv.org/abs/1704.08705.

[23] A. Gupta. A fast parallel algorithm for recognition of parenthesis languages. Master's thesis, 1985.

[24] Ankit Gupta, Pritish Kamath, Neeraj Kayal, and Ramprasad Saptharishi. Arithmetic circuits: A chasm at depth 3. *SIAM J. Comput.*, 45(3):1064–1079, 2016. URL: https://doi.org/10.1137/140957123, doi:10.1137/140957123.

[25] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8(1):171–186, 1976. URL: http://dx.doi.org/10.1007/BF01917434, doi:10.1007/BF01917434.

[26] Johan Hastad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 6–20. ACM, 1986.

[27] Johan Håstad. On the correlation of parity and small-depth circuits. *SIAM J. Comput.*, 43(5):1699–1708, 2014. URL: https://doi.org/10.1137/120897432, doi:10.1137/120897432.

[28] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.

[29] Maurice J. Jansen and Jayalal Sarma. Balancing bounded treewidth circuits. *Theory Comput. Syst.*, 54(2):318–336, 2014. URL: http://dx.doi.org/10.1007/s00224-013-9519-3, doi:10.1007/s00224-013-9519-3.

[30] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. URL: http://www.cs.berkeley.edu/~luca/cs172/karp.pdf.

[31] Andreas Krebs, Nutan Limaye, and Michael Ludwig. Cost register automata for nested words. In Thang N. Dinh and My T. Thai, editors, *Computing and Combinatorics - 22nd International Conference, COCOON 2016, Ho Chi Minh City, Vietnam, August 2-4, 2016, Proceedings*, volume 9797 of *Lecture Notes in Computer Science*, pages 587–598. Springer, 2016. URL: http://dx.doi.org/10.1007/978-3-319-42634-1_47, doi:10.1007/978-3-319-42634-1_47.

[32] Andreas Krebs, Nutan Limaye, and Michael Ludwig. A unified method for placing problems in polylogarithmic depth. *Electronic Colloquium on Computational Complexity (ECCC)*, 24:19, 2017. URL: https://eccc.weizmann.ac.il/report/2017/019.

[33] Andreas Krebs, Nutan Limaye, and Meena Mahajan. Counting paths in VPA is complete for #nc^1. In My T. Thai and Sartaj Sahni, editors, *Computing and Combinatorics, 16th Annual International Conference, COCOON 2010, Nha Trang, Vietnam, July 19-21, 2010. Proceedings*, volume 6196 of *Lecture Notes in Computer Science*, pages 44–53. Springer, 2010. URL: http://dx.doi.org/10.1007/978-3-642-14031-0_7, doi:10.1007/978-3-642-14031-0_7.

[34] Andreas Krebs, Nutan Limaye, and Meena Mahajan. Counting paths in VPA is complete for #nc 1. *Algorithmica*, 64(2):279–294, 2012. URL: http://dx.doi.org/10.1007/s00453-011-9501-x, doi:10.1007/s00453-011-9501-x.

[35] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, 1994.

[36] Markus Lohrey. On the parallel complexity of tree automata. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2001. URL: http://dx.doi.org/10.1007/3-540-45127-7_16, doi:10.1007/3-540-45127-7_16.

[37] Nancy A. Lynch. Log space recognition and translation of parenthesis languages. *J. ACM*, 24(4):583–590, 1977. URL: http://doi.acm.org/10.1145/322033.322037, doi:10.1145/322033.322037.

[38] Meena Mahajan and V. Vinay. A combinatorial algorithm for the determinant. In Michael E. Saks, editor, *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana.*, pages 730–738. ACM/SIAM, 1997. URL: http://dl.acm.org/citation.cfm?id=314161.314429.

[39] Kurt Mehlhorn. Pebbling moutain ranges and its application of dcfl-recognition. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 422–435. Springer, 1980. URL: http://dx.doi.org/10.1007/3-540-10003-2_89, doi:10.1007/3-540-10003-2_89.

[40] Dan I Moldovan. *Parallel processing from applications to systems*. Elsevier, 2014.

[41] Sang-il Oum and Paul D. Seymour. Approximating clique-width and branch-width. *J. Comb. Theory, Ser. B*, 96(4):514–528, 2006. URL: http://dx.doi.org/10.1016/j.jctb.2005.10.006, doi:10.1016/j.jctb.2005.10.006.

[42] V. Ramachandran. Restructuring formula trees. Unpublished manuscript, 1986.

[43] Walter L. Ruzzo. On uniform circuit complexity. *J. Comput. Syst. Sci.*, 22(3):365–383, 1981. URL: http://dx.doi.org/10.1016/0022-0000(81)90038-6, doi:10.1016/0022-0000(81)90038-6.

[44] P.M. Spira. On time hardware complexity tradeoffs for boolean functions. *Proceedings of the Fourth Hawaii International Symposium on System Sciences*, pages 525–527, 1971.

[45] Howard Straubing. *Finite Automata, Formal Logic, and Circuit Complexity*. Birkhäuser, Boston, 1994.

[46] Heribert Vollmer. *Introduction to Circuit Complexity - A Uniform Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 1999. URL: http://dx.doi.org/10.1007/978-3-662-03927-4, doi:10.1007/978-3-662-03927-4.

[47] Heribert Vollmer. Introduction to circuit complexity: a uniform approach, 2013.

[48] Egon Wanke. k-nlc graphs and polynomial algorithms. *Discrete Applied Mathematics*, 54(2-3):251–266, 1994. URL: http://dx.doi.org/10.1016/0166-218X(94)90026-4, doi:10.1016/0166-218X(94)90026-4.

[49] Barry Wilkinson and Michael Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, 1999.

A Dividing terms: some structural lemmas regarding terms

Proof of Lemma 16. Assume that $p_1 < p_2 < q_1 < q_2$ because otherwise the statement is trivial. The interval $[p_2, q_2]$ has to correspond to an open term since otherwise $p_1 \not<_T q_1$. So as $p_2 <_T q_1$ holds we have that $[p_2, q_1]$ corresponds to an open term and so $[p_1, p_2 - 1]$ is also a term; it could be open or closed. By combining all parts, we get that $[p_1, q_2]$ is a term and it is closed or open depending whether $[p_1, q_1]$ is closed or open. □

Proof of Lemma 17. For the first statement, note that $\mathscr{N} \subseteq \mathscr{M}$ and hence $\mathscr{N}^2 \leq \mathscr{M}^2$ which follows from the previous lemma. Now assume that $\mathscr{N}^2$ is strictly smaller than $\mathscr{M}^2$. Let p be the position such that $[p, \mathscr{N}^2]$ is closed. If $p \in \mathscr{M}$ then we find $q \in \mathscr{M}$ and $q \geq \mathscr{N}^2$ such that $[p, q]$ is an open term. But then $[\mathscr{N}^2 + 1, q]$ is also an open term and so is $\mathscr{N} \cup [\mathscr{N}^2 + 1, q]$. Hence this violates the maximality of $\mathscr{N}^2$. If $p \notin \mathscr{M}$ then $[\mathscr{M}^1, \mathscr{N}^1 - 1]$ is an open term and so is $[\mathscr{M}^1, \mathscr{N}^2]$. But then also $[\mathscr{N}^1, \mathscr{M}^2]$ is an open term and again maximality of $\mathscr{N}^2$ is violated.

For the second statement, first note that $\mathscr{M}(l, l', r' - 1) \cup \mathscr{N}(l' + 1, r', r)$ is indeed an interval, i.e. $[\mathscr{M}^2(l, l', r' - 1) + 1, \mathscr{N}^1(l' + 1, r', r) - 1]$ is empty. This we get though maximality of $\mathscr{M}^2(l, l', r' - 1)$. Also $\mathscr{M}(l, l', r' - 1) \cup \mathscr{N}(l' + 1, r', r)$ is a closed or open term, depending on whether $\mathscr{M}(l, l', r' - 1)$ is closed or open. If not empty, the interval $\mathscr{M}(l, l', r' - 1) \cup \mathscr{N}(l' + 1, r', r)$ has to be an open term and hence $N^1(l' + 1, r', r)$ was not chosen maximal. □

Proof of Lemma 18. If $\mathscr{M}$ is entirely contained in $[l, r' - 1]$, $[l' + 1, r]$ or $[l' + 1, r' - 1]$ then it coincides with one of the first three cases.

If the term stretches from the first third to the last third, it is not entirely contained in one of those three. Let A be $\mathscr{M}(l, l', r' - 1) \cup \mathscr{N}(l' + 1, r', r)$. By Lemma 17 we know this interval is a disjoint union. Further A is a closed or open term contained in $\mathscr{M}$ which contains m. If $A = \mathscr{M}$ we are done as case 4 holds.

The interval $\mathscr{M}(l, l', r' - 1)$ is open iff A is open. But then $\mathscr{M}^1 = \mathscr{M}^1(l, l', r' - 1)$ because of minimality of $\mathscr{M}^1(l, l', r' - 1)$. Similarly it holds that $\mathscr{M}^2 = \mathscr{N}^2(l' + 1, r', r)$. So we get $A = \mathscr{M}$ and case 4 holds.

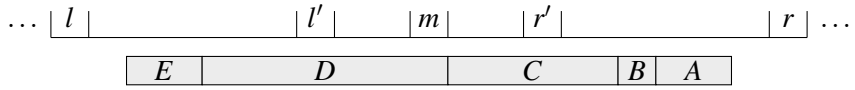

Figure 6: In case five for $\mathscr{M}$ as shown in Lemma 18, the interval is subdivided into five parts. We see that DC is a closed term where $D = \mathscr{M}(l, l', r'-1)$ and $C = \mathscr{N}(l'+1, r', r)$. Further, B consists of a single position which is a operation symbol and A and E are open terms.

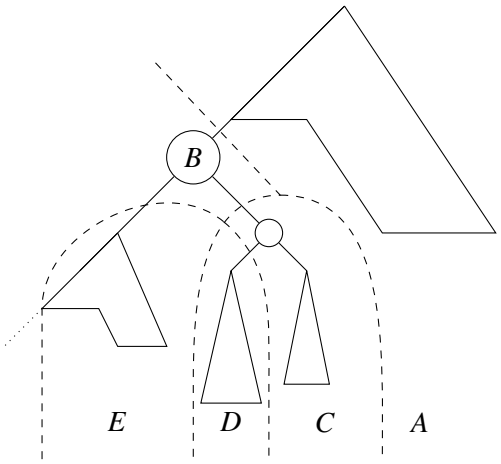

Figure 7: A graphical representation of case five for $\mathscr{M}$; see Lemma 18 and also figure 6. Note that A, C, and E represent open terms and D a closed one. The term DCB then is open again.

Now suppose A is a closed term. The term A is part of a lager possibly open term. It has either the form $AB\circledast$ or $BA\circledast$ where B is a closed term. If $AB\circledast$ is the case then $\circledast$ lies outside $[l,r]$ and case 4 holds which we again get by a maximality argument. If $BA\circledast$ is the case, then $O(l,m,r)$ addresses the operation $\circledast$. Let B' be the largest suffix of B which is an open term and a subset of $[l,r]$. Note that B' is a proper suffix because $|B| \geq |A|$ and $|A|$ is more than one third of $r-l+1$. The interval $\mathscr{L}$ coincides with B'. The subterm $B'A\circledast = \mathscr{L}(l,m,r) \cup \mathscr{M}(l,l',r'-1) \cup \mathscr{N}(l'+1,r',r) \cup O(l,m,r)$ can be followed by an open term and we get again an open term if we unite those. The maximal one in $[l,r]$ is addressed by $\mathscr{R}(l,m,r)$. Note that $\mathscr{L}(l,m,r)$ and $\mathscr{R}(l,m,r)$ might be empty. This concludes the fifth case. □

Figures 6 and 7 show how the interval is subdivided in case five.

Proof of Lemma 19. This proof is similar to the previous one. Only case five slightly differs. Again, either the interval is completely contained in one of the three subintervals for which we fall back to $\mathscr{N}(l,l',r'-1)$, $\mathscr{N}(l'+1,r',r)$, or $\mathscr{N}(l'+1,m,r'-1)$ respectively.

Otherwise let $A = \mathscr{N}(l,l',r'-1) \cup \mathscr{N}(l'+1,r',r)$, similar to the previous proof. Note that by Lemma 17 we get that $\mathscr{N}(l,l',r'-1) \cup \mathscr{N}(l'+1,r',r)$ is a disjoint union and an interval. If we are in the $AB\circledast$ situation, then case 4 holds as $\circledast$ is outside of $[l,r]$. In case of $BA\circledast$, B is not part of $\mathscr{N}$ due to maximality of $\mathscr{N}^1$. If A is closed we can insert $\circledast$ by $O(l,m,r)$ and obtain an open term. Open terms following $O(l,m,r)$ can be appended and are addressed by $\mathscr{R}(l,m,r)$. This is possible since $\mathscr{M}^2 = \mathscr{N}^2$, which we know from Lemma 17. □

Proof of Lemma 20. By definition, the interval $\mathscr{L}$ lies to the left of $\mathscr{M} \cup \mathscr{N}$. The set $\mathscr{M} \cup \mathscr{N}$ is a closed term and $\mathscr{M} \cup \mathscr{N} \cup O$ is an open one. We then want to address the largest term in $[l,l'-1]$ that comes before $\mathscr{M}$. We can use $\mathscr{M}(\bar{l},\bar{m},\bar{r})$ for this. The inclusion $\mathscr{L} \subseteq \mathscr{M}(\bar{l},\bar{m},\bar{r})$ is clear from maximality of $\mathscr{M}(\bar{l},\bar{m},\bar{r})$. On the other hand the converse direction is also true since any position to the right of $\mathscr{L}$ is rooted after l'.

Now we can use the binary search inside $[l, l'-1]$. Start with this interval and then recursively do the following: If $\bar{m}$ is the middle position of the current interval then if:

- $\mathscr{L}$ is entirely left of $\bar{m}$ then search in the left part.
- $\mathscr{L}$ is entirely right of $\bar{m}$ then search in the right part.
- $\mathscr{L}$ contains $\bar{m}$ and let $\bar{l}$, $\bar{r}$ be the borders of the current interval, then $\mathscr{L} = \mathscr{M}(\bar{l}, \bar{m}, \bar{r})$.

□

Proof of Lemma 21. This proof is similar to the previous one. First note that $\mathscr{R} \subseteq \mathscr{M}(\bar{l}, \bar{m}, \bar{r})$ and $\mathscr{R}^2 = \mathscr{M}^2(\bar{l}, \bar{m}, \bar{r})$ because of maximality. Further $\mathscr{R}$ is an open term. Now if $\mathscr{M}(\bar{l}, \bar{m}, \bar{r})$ is a strict superset it must contain the operation set $O(l, m, r)$. Inside $[r'+1, r]$ both descendants stay open so there is no open term in $[r'+1, r]$ that contains $O(l, m, r)$.

The binary search is the same as in the previous proof. □

B Applications of the term evaluation algorithm

Before we start decribing the details of the applications, we need one more technical definition, which will we present here.

B.1 Coding of algebras

It helps to embed some class $\mathscr{A}$-$\mathbf{NC}^1$ in e.g. a purely Boolean one.

Definition 49 (Codings of algebras)**.** *Given algebras $\mathscr{A} = (\{\mathbb{D}_1, \ldots, \mathbb{D}_S\}, F)$ and $\mathscr{A}' = (\{\mathbb{D}'_1, \ldots, \mathbb{D}'_{S'}\}, F')$, where F resp. F' contain the operations. We say $\mathscr{A}'$ is a coding of $\mathscr{A}$ if there exists a relation $c \subseteq \mathbb{D} \times \mathbb{D}'$ on the domains of $\mathscr{A}$ and $\mathscr{A}'$ such that:*

- *For all $X \subseteq \mathbb{D}$ holds that $c(c^{-1}(c(X))) = c(X)$.*
- *For all $\circledast \in F$ there exists $\circledast' \in F'$ such that $\circledast(x_1, \ldots, x_n) = y$ if and only if $\circledast(c(x_1), \ldots, c(x_n)) \subseteq c(y)$.*
- *For all $\circledast' \in F'$ there exists $\circledast \in F$ such that $\circledast'(x_1, \ldots, x_n) = y$ if and only if $\circledast(c^{-1}(x_1), \ldots, c^{-1}(x_n)) \subseteq c^{-1}(y)$.*
- *For all $\circledast \in F$ there exists $\circledast' \in F'$ such that $\circledast(x_1, \ldots, x_n) \in c^{-1}(c(y)) \Leftrightarrow \circledast'(x'_1, \ldots, x'_n) \in c(y)$ iff $(x_i, x'_i) \in c$ for all $i \in [n]$.*
- *For all $d, e \in \mathbb{D}$ if $d \neq e$ then $c(d) \cap c(e) = \emptyset$.*

If such a relation exists, we write $\mathscr{A} \preceq \mathscr{A}'$.

Note that $\preceq$ is transitive. Also if we have two families $(\mathscr{A}_n)_{n\in\mathbb{N}}$ and $(\mathscr{A}'_n)_{n\in\mathbb{N}}$, we write $(\mathscr{A}_n)_{n\in\mathbb{N}} \preceq (\mathscr{A}'_n)_{n\in\mathbb{N}}$ if $\mathscr{A}_n \preceq \mathscr{A}'_n$ for all $n \in \mathbb{N}$. An algebra can also be coded into a family of algebras by taking a family of codings. Note that the third condition the definition ensures that codings preserve all the information. This property can be thought of as injectivity for relations.

By taking $\mathscr{A} = (\{\mathbb{D}_1, \ldots, \mathbb{D}_S\}, \circledast_1, \ldots, \circledast_k)$ and c one can actually construct an algebra $c(\mathscr{A}) = (\{c(\mathbb{D}_1), \ldots, c(\mathbb{D}_S)\}, \circledast_1^c, \ldots, \circledast_k^c)$ such that $\mathscr{A} \preceq c(\mathscr{A})$. This is slight abuse of notation since an algebra $c(\mathscr{A})$ is not already defined by c and $\mathscr{A}$. The operations $\circledast_i^c$ can be defined in different ways. However it will always be clear from the context how we define the these; especially since most of the time we assume c to be a function or a family of functions.

As mentioned before, our main theorem shows a connection between evaluating terms over $\mathscr{A}$ and the algebra $\mathscr{F}(\mathscr{A})$. If we want to use codings as outlined before, we need to make sure, that the coding keeps all the relevant information. If $\mathscr{A}$ is single-sorted algebra with domain $\mathbb{D}$, $\mathscr{F}(\mathscr{A}) \preceq \mathscr{A}'$ via coding c then if for all $d_1, d_2 \in \mathbb{D}$ holds that $d_1 = d_2$ iff $c(d_1) = c(d_2)$ we say the coding is evaluation-stable and write $\mathscr{F}(\mathscr{A}) \leq \mathscr{A}'$. For the rest of the paper we are only interested in such codings.

$\mathscr{A}$-evaluation reduces to $\mathscr{A}'$-evaluation and we write $\mathscr{A} \preceq \mathscr{A}'$.

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il