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

A Unified Method for Placing Problems in Polylogarithmic Depth

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

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

Abstract. In this work we consider the term evaluation problem which involves, given a term over some algebra and a valid input to the term, computing the value of the term on that input. This is a classical problem studied under many names such as formula evaluation problem, formula value problem etc.. 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 $(\mathbb{Z}, +, \times)$. Here, we allow the algebra to be completely general and obtain a bound for the term evaluation problem. We consider the problem of deriving upper bounds in terms of polylogarithmically deep circuits. To that end we present a generic term evaluation algorithm that works in polylogarithmic depth.
This efficient term evaluation algorithm over a very general algebra then serves as a tool for showing polylogarithmic time upper bounds for various well-studied problems. To underline the utility of our result we show new bounds and reprove known results using our approach and thereby present a unified proof approach for problems of this nature. The spectrum of problems for which we apply our term evaluation algorithm is wide: in particular, the application 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

Circuits are a natural model of computation closely related to parallel computation. Typically, algorithms which need less than linear time on multiprocessor machines can be modeled by circuits. When a circuit has, say, logarithmic depth and polynomial size, this can be seen as a logarithmic time algorithm using polynomially many parallel processors.

Starting point for our work is the observation that many parallel algorithms (i.e. circuit constructions) have a similar core structure.

ISSN 1433-8092

Our main goal was to identify the similarities between these algorithms and create a uniform framework which can be used to design parallel algorithms.

The kind of problems that we study in this work are the ones which instrisically have a tree-like structure. Often the tree-like structure is not obvious from the statement of the problem, but needs to be extracted out from the problem definition. We then use such a tree-like structure of a given problem and remodel it as a slightly different problem. In particular, we write it as *a term* over *a universal algebra*. In other words, we reduce the original problem to the problem of evaluation of terms, wherein the inputs to the term come from a (possibly infinite) set and operators of the term are those used in the algebra.

One of our primary contribution in this work is a structural theorem that allows us to remodel the term evaluation problem as an efficient parallel computation. Our main theorem (stated below) gives the complexity of evaluating terms over an arbitrary algebra:

Main Theorem
Given a universal algebra $\mathcal{A}$ and domain $\mathbb{D}$, the evaluation of the term over the algebra $\mathcal{A}$ when the inputs of the term are assigned values from the domain $\mathbb{D}$ can be performed in DLOGTIME-uniform $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$, where $\mathcal{F}(\mathcal{A})$ is a slight generalization of the algebra $\mathcal{A}$ and $\mathcal{F}(\mathcal{A})-\mathbf{NC}^1$ is a logdepth $\mathbf{NC}^1$ circuit which over and above uses oracle gates from $\mathcal{F}(\mathcal{A})$.

Informally, the above theorem states the following: Given an algebra, i.e., a set together with some operators over this set, for evaluating terms over this algebra, logarithmic depth suffices. This would be $\mathbf{NC}^1$ but here in particular we need $\mathbf{NC}^1$ together with some oracle gates which perform the algebra functions. We note that there is a tradeoff in the complexity of this: to obtain logarithmic depth, the algebra is appended with a slightly more structure over and above the original operators. So given an algebra $\mathcal{A}$, we design a slightly appended algebra $\mathcal{F}(\mathcal{A})$ and then prove that evaluating terms over $\mathcal{A}$ is possible in $\mathbf{NC}^1$ using $\mathcal{F}(\mathcal{A})$ gates, that is $\mathcal{F}(\mathcal{A}) - \mathbf{NC}^1$.

Our main result is powerful in its own right because it gives a very general framework for changing any sequential tree-like computation

to an efficient parallel (circuit-like) computation. But beyond this, our main result also has many applications. We discuss them below.

1.1 Applications

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 (but not exhaustive) list of problems which have been considered in this literature.

- Boolean and arithmetic term evaluation [8,10].
- Membership for language classes in $\mathbf{NC}^1$, $\mathbf{SAC}^1$ [26,25,16,1,23].
- Circuits of bounded tree width [21].
- Courcelle's Theorem and counting [13,17].
- Maximal cuts in bounded clique-width graphs [35].
- Counting Hamiltonian paths in bounded clique-width graphs [35].

Using our main theorem, 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. The variants we consider here are not considered before our work to the best of our knowledge.

Note that each application is originally a significant result in its own right. Using our approach, all of them become shorter and follow that same proof structure. The proof structure we use consists of three steps:

1. Reduce the problem to a term evaluation problem.
2. Embed the $\mathcal{F}(\mathcal{A})$ algebra in a way that we get a Boolean or arithmetic circuit.
3. Analyze the complexity of the resulting circuit. Usually the overall complexity is the complexity of the oracle gates multiplied by a logarithmic factor for the depth.

We believe this framework is general enough to see many other applications in the future.

1.2 How does term evaluation work?

Our algorithm for the term evaluation problem fits in the long chain of contributions dedicated to the evaluation problem. The origin of which can be vaguely traced back to the investigation of upper bounds for the Boolean formula value problem. In [27] Lynch studied it first and achieved a log-space bound. Subsequently Cook conjectured that this bound is tight [12] which, as we know today, is not (unless log space equals log depth). A way to deal with formulas that are very deep trees, was already investigated by Spira [32]: By a quadratic increase in size, we can balance a Boolean formula. Brent built upon this work [7]. The step from this balancing to obtaining an $\mathbf{NC}^1$ upper bound is not big. If the transformation can be done in $\mathbf{NC}^1$, the evaluation is in $\mathbf{NC}^1$.

Cook and Gupta [19] as well as Ramachandran [30] were the next in line and showed that $\mathcal{O}(\log n \log\log n)$ deep circuits suffice for evaluating. Based on [19], Buss showed an ALOGTIME bound [8] which equals logarithmic depth [31]. His proof utilized a sophisticated two-player-game. Buss' bound is tight, so from there on research went in the direction of broadening the result. This continued research is always rooted in the work of [19] and [8]. This way Dymond showed [16] that the visibly pushdown languages[3] are in $\mathbf{NC}^1$. A different generalization by Buss et al considered arithmetic formulas and showed a $\#\mathbf{NC}^1$-bound [10]. Meanwhile Buss published an alternative presentation of his original proof for Boolean formulas [9]. After some time has passed, in [24] Krebs et al built upon Dymonds approach to show that counting the number of accepting computations in visibly pushdown automata is in $\#\mathbf{NC}^1$. The authors of the present paper again showed upper bounds in a similar way [23] - this time for a quantitative version for VPAs. Through these works, there emerged a pattern which we have followed to exploit which is the topic of this paper.

Our algorithm takes ideas from different steps in the chain of research. Below we give an outline of how it works.

What we want for a log depth algorithm is a recursive approach. Given some term T, if we could just split it in half and evaluate the halves and combine the results, we would be done. However

[3] a.k.a. input driven pushdown languages [28]

if a formula is split, the two parts cannot be assigned any usable semantic. The major part of the task of designing an evaluation algorithm over a general algebra is to determine how to split terms in a meaningful manner. The first tool we borrow is from [8]: We transform the term in a certain kind of post-fix notation. That way we get rid of parentheses. In case of a balanced term, we could find a cutting point near the center and evaluate the halves. However if the term is a degenerated tree, i.e. a list, the algorithm may get stuck. To overcome this hurdle, we use the idea originating in [10], which is to assign te split parts some meaning that may not be terms any more. If we split a term that is a list in half, we get a term and something which is a term *with a hole*. Assigning meaning to terms with holes is what is needed here. For the third key idea consider what happens if we actually want to implement a recursion in uniform log depth: The recursion intervals have to be fixed apriori irrespective of the specific input, but only based on the length of the input. This is in some sense contrast to the larger complexity classes. So we cannot look at the term and decide for a nice place to cut. We have to just cut at predecided fixed places. The third idea is to assign meaning to subintervals of a term. What we e.g. do is to evaluate the larges subterm of some interval that contains the middle. By admitting overlapping recursion interval patterns it is possible to achieve a scheme where the whole term is seamlessly covered.

1.3 Contributions

We give a proof for a general upper bound for formula evaluation over arbitrary algebras where previously only proofs for certain examples of algebras existed; see Section 3. To get a clear formal framework we needed to come up with a way to define circuits which use arbitrary algebras as gates and values. Also we need a very general notion of algebra, that is algebras of more than one domain, which are called many-sorted algebra. These definitions are outlined in Section 2. Finally a large set of applications is provided in Section 4.

2 Preliminaries: Many-sorted terms, circuits, and universal algebras

First we fix some basic notation: The set $\{1, \ldots, n\}$ is abbreviated 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 $\{\bot, \top\}$. Alphabets, which we often call Σ, are 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 i'th letter of a word w we denote by $w(i)$ and the length of w is denoted by $|w|$. The word of length 0 is denoted by ϵ. A language is a subset of Σ^*.

In this section we lay out the framework in which we formulate our main result. Our main result is a meta theorem which can be applied on different problems. In order to state it in a way so that it can be easily used as a template, we need to carefully define abstracted versions of familiar objects like circuits. What we want is a definition of circuits that goes beyond Boolean and arithmetic circuits. In fact we want to be able to plug in arbitrary fitting algebras into it. For example if we have a DAG, then we could either plug in $(\mathbb{B}, \wedge, \vee, \bot, \top)$ or $(\mathbb{N}, +, \times, 0, 1)$. What does it mean that an algebra fits a circuit? This is handled by signatures. Ordinary signatures assign arities to operations. If we have e.g. $\wedge$, we want that this can only to be assigned to nodes in a circuit that have in-degree 2.

Our main theorem is independent of the algebra; it only depends on the signature. However we need a more general notion of signature as the ordinary one. All our definition framework here is lifted to the many-sorted case. This means we have different data sorts. For instance we can have a circuit which has Boolean gates but also gates which add two natural numbers. Then wires transport either Boolean values or natural numbers. Of course a natural number should not be feeded into an $\wedge$-gate. A many-sorted signature will not only assign an arity to operations but also which kind of data.

Definition 1 (Sorts, many-sorted signature). *Let $S \in \mathbb{N}$ be the number of sorts. Given S sorts, a many-sorted signature σ of k operations is an element of $([S]^* \times [S])^k$.*

So, a many-sorted signature is a tuple of pairs of words and letters $\sigma = ((w_1, a_1), \ldots, (w_k, a_k))$. Each word codes the input sorts

of operations. The length of a word $|w_i|$ then is the arity of the i'th operation. We write $\mathsf{In}_\sigma(i)$ to address the word w_i, $\mathsf{Out}_\sigma(i)$ to address the letter a_i and $\sigma(i)$ to address (w_i, a_i). Also, $\mathsf{In}_\sigma(i,j)$ is the j'th letter of $\mathsf{In}_\sigma(i)$.

A (single-sorted) signature σ is one where $|S| = 1$. In this case σ corresponds to the classical notion of signature assigning just an arity to operations.

We will shortly say signature instead of many-sorted signature and define some more many-sorted objects and then also omit mentioning many-sorted in every occasion.

As outlined we want a very general version of circuits which follows the many-sorted concept. It resembles much similarity to a standard definition for Boolean circuit. The main difference is that we use the signature to ensure that the wiring is valid (no natural numbers as inputs for $\wedge$-gates etc.) and that we not yet build the underlying algebra in it.

Definition 2 (Many-sorted circuit). *Given a signature σ, then 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 a input gate,*
- *$\mathsf{Outputgates}\colon \{y_1, \ldots, y_m\} \to V$ promotes gates to output gates,*

such that:

- *If some $v \in V$ has in-degree 0 then $\mathsf{Gatetype}(v) \in \{x_1, \ldots, x_n\}$ or $\mathsf{In}_\sigma(\mathsf{Gatetype}(v)) = \epsilon$, 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. By $\mathsf{In}_C \subseteq [S]^n$ we denote a word which holds the sorts of the input gates and $\mathsf{Out}_C \subseteq [S]^m$ stores the sorts of the 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. Generally speaking: A term is a circuit which is a tree, however in out setting, we do not want a term to have inputs. It rather only has constants. One can think of it as if all variables have already been assigned a value. Also we only want to consider terms with binary operations. Besides terms we also need the notion of terms with an unknown. Such terms come into play when we decompose terms inside our algorithm. Assume a term is to be evaluated over a domain $\mathbb{D}$ then a term with an unknown evaluates to a function $\mathbb{D} \to \mathbb{D}$.

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* σ *and by* $\mathsf{Term}_\sigma[X]$ *we denote the set of terms with an unknown over* σ*.*

Note that in order to get terms over some signature, it has to admit 0-ary operations since we need them for the leafs.We forbid higher arities here which is no restriction as n-ary operations can be simulated by binary operations.

The main topic of this paper is a term evaluation algorithm. An algorithm receives strings as inputs, hence we have to represent terms as strings. Note that such strings still do not possess any semantic.

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 $v \in V$.*

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

In the case of terms with an unknown we set $w(v) = X$ if v is the single input gate and where X is an additional letter.

From now on we will not strictly distinguish between terms and linearizations of terms as a the linearization admits an isomorphism.

Up to now we have defined the objects syntacticly. Now we want to define meanings for terms and circuits. The meaning of a circuit is the realized function and of a term the value it evaluates to. Given a term, which is just a string of parentheses and operation symbols (constants are 0-ary operation symbols), we assign what the domain of values is, on which we operate as well as which operations over the domain are to be assigned the operation symbols. An algebra hence has to conform a signature. Again we are in the many-sorted case.

Definition 5 (Many-sorted universal algebra). *Given a many-sorted signature σ with S sorts, a many-sorted universal algebra is a tuple $\mathcal{A} = (\{\mathbb{D}_1, \ldots, \mathbb{D}_S\}, \circledast_1, \ldots, \circledast_{|\sigma|})$ where $\circledast_i \colon \mathbb{D}_{\mathsf{In}_\sigma(i,1)} \times \ldots \times \mathbb{D}_{\mathsf{In}_\sigma(i,\alpha)} \to \mathbb{D}_{\mathsf{Out}_\sigma(i)}$ where $\alpha = |\mathsf{In}_\sigma(i)|$. 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, 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 a universal algebra $\mathcal{A}$ over signature σ and a word $w \in \mathbb{D}^n$ then the evaluation map $\eta_{\mathcal{A},w} \colon \mathsf{Circ}_{\sigma,n,m} \to \mathbb{D}^m$ 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* $\eta_{\mathcal{A},w}(T(v)) = w_i$ *if* $w_i \in \mathbb{D}_j$ *implies that* $\mathsf{In}_C(i) = j$.
– *Let* α *be the arity* $|\mathsf{In}_\sigma(\mathsf{Gatetype}(v))|$ *and* $v_1, \dots v_k$ *be the predecessors of* v *ordered by the their output wire order, then* $\eta_{A,w}(T(v)) = \circledast_{\mathsf{Gatetype}(v)}(\eta_{\mathcal{A},w}(T(v_1)), \dots, \eta_{\mathcal{A},w}(T(v_\alpha)))$.

Let $v_1, \dots v_m$ *be the output gates and* C *a circuit, then*

$$\eta_{\mathcal{A},w}(C) = (\eta_{\mathcal{A},w}(T(v_1)), \dots, \eta_{\mathcal{A},w}(T(v_m)))$$

if for all $\mathsf{Outputgates}^{-1}(y_i) = v_i$ *holds that* $\mathsf{Out}_C(i) = \mathsf{Out}_\sigma(\mathsf{Gatetype}(v_i))$.

We covered how to evaluate terms and circuits. Terms with an unknown however we want to treat 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 the 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* $\mathcal{A} = (\mathbb{D}, \circledast_1, \dots, \circledast_k)$ *over a single-sorted signature* σ *which only contains operations that are at most binary. Then the functional algebra is* $\mathcal{F}(\mathcal{A}) = (\{\mathbb{D}, \widetilde{\mathbb{D}}\}, F)$ *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* $\mathcal{A}$*:* $\circledast_1, \dots, \circledast_k$.
– $\circ\colon \widetilde{\mathbb{D}} \to \widetilde{\mathbb{D}}$ *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 $\mathcal{F}(\mathcal{A})$ *we denote by* $\sigma(\mathcal{F}(\mathcal{A}))$.

This definition can be lifted to arbitrary arities but we want to keep it simple at this point.

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

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

One circuit accepts words of a fixed length. If we want to handle inputs of arbitrary length we need families of circuits, that is a list $(C_n)_{n\in\mathbb{N}}$ of circuits containing one circuit for each input length. Later we will see cases where it is desirable to have algebras that match a circuit C_i, hence we need also families of algebras.

Definition 9 (Family of algebras). *A family of algebras $(\mathcal{A}_n)_{n\in\mathbb{N}}$ is a sequence of algebras, where $\mathcal{A}_1 = (\{\mathbb{D}_1, \ldots, \mathbb{D}_S\}, \circledast_1^1, \ldots, \circledast_k^1)$ and*

$$\mathcal{A}_i = (\{D(i,1), \ldots, D(i,S)\}, \circledast_1^i, \ldots, \circledast_k^i)$$

where $D(i,j)$ is either $\mathbb{D}_j$ or $(\mathbb{D}_j)^{p(i)}$, where p is some polynomial and for $i \geq 2$ it holds $D(i,j) = \mathbb{D}_j$ iff $D(i+1,j) = \mathbb{D}_j$.

For now we focus on circuits having one output gate and where all input gates have the same sort. Given an algebra $\mathcal{A}$ let $\mathbb{D}_I$ and $\mathbb{D}_O$ address the two subdomains that correspond to the inputs resp. output values. Then a circuit C_n of n inputs realizes a function $F_{\mathcal{A}}(C_n) \colon \mathbb{D}_I^n \to \mathbb{D}_O$. Given a family of circuits C we then get a a function $F_{\mathcal{A}}(C) = \mathbb{D}_I^* \to \mathbb{D}_O$. In this case also a family of algebras can be given such that a circuit C_i is interpreted over an algebra $\mathcal{A}_i$. Note that by considering families of circuits and algebras, we also need families of signatures.

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. [33] or [34] for basics in circuit complexity.

We use our frame work to define the classical Boolean version of $\mathbf{NC}^i$. Here $\mathcal{B}$ is the Boolean algebra $(\mathbb{B}, \wedge, \vee, \neg, \bot, \top)$.

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

Note that the bounded fan-in of the gates is ensured through the signature of $\mathcal{B}$. 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 $\mathcal{B}^* = (\mathcal{B}_n)_{n\in\mathbb{N}}$ where $\mathcal{B}_i = (\mathbb{B}, \wedge, \vee, \neg, \bot, \top, \wedge_i, \vee_i)$, $\wedge_i \colon \mathbb{B}^i \to \mathbb{B}$ 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 $F_{(\mathcal{B}_n)_{n\in\mathbb{N}}}(C)$, where $C = (C_n)_{n\in\mathbb{N}}$ is a family of circuits that contains circuits of polynomial size, $\mathcal{O}(\log^i n)$ depth and where C_n has the same signature as $\mathbb{B}_n$.*

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

Next we want to enrich circuits of the previously defined classes by some algebra $\mathcal{A}$ such that we get circuits that have Boolean gates as well as $\mathcal{A}$-gates. Boolean values and values of $\mathcal{A}$ interact via multiplexer gates.

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 $\mathcal{B}$ and some other algebra $\mathcal{A}$ and they interact via multiplexer operations. We can then actually add arbitrary many algebras to a composition:

Definition 13 (Composition of algebras). *Given an algebra* $\mathcal{A} = (\{\mathbb{D}_1, \ldots, \mathbb{D}_S\}, \circledast_1, \ldots, \circledast_k)$*, by* $(\mathcal{B}, \mathcal{A})$ *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 $(\mathcal{B}, \mathcal{A})$ *and an algebra* $\mathcal{A}' = (\{\mathbb{D}'_1, \ldots, \mathbb{D}'_{|S'|}\}, \circledast'_1, \ldots, \circledast'_{k'})$*, by* $((\mathcal{B}, \mathcal{A}), \mathcal{A}')$ *we denote the algebra*

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

Hence we may write $(\mathcal{B}, \mathcal{A}_1, \ldots, \mathcal{A}_k) = ((\mathcal{B}, \mathcal{A}_1, \ldots, \mathcal{A}_{k-1}), \mathcal{A}_k)$ *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 ($\mathcal{A}$-$\mathbf{NC}^i$, $\mathcal{A}$-$\mathbf{NC}^i_{\mathbb{D}}$). *The set* $\mathcal{A}$-$\mathbf{NC}^i_{\mathbb{D}}$ *contains all functions* $F_{(\mathcal{B},\mathcal{A})}(C)$*, where* C *is a family of circuits having the same family of signatures as* $(\mathcal{B}, \mathcal{A})$ *that contains circuits of polynomial size, depth* $\log^i n$*, inputs of* $\mathbb{D}$ *and one output of a subdomain of* $\mathcal{A}$*. For the special case of Boolean inputs we set* $\mathcal{A}$-$\mathbf{NC}^i = \mathcal{A}$-$\mathbf{NC}^i_{\mathbb{B}}$*.*

For the class $(\mathbb{N}, +, \times, 0, 1)$-$\mathbf{NC}^1$ there is the shortcut $\#\mathbf{NC}^1$ and $(\mathbb{Z}, +, \times, 0, 1)$-$\mathbf{NC}^1$ has the shortcut Gap$\mathbf{NC}^1$. The $\mathcal{A}$ -$\mathbf{NC}^i$ and $\mathcal{A}$-$\mathbf{NC}^i_{\mathbb{D}}$ definition naturally carries over to other classes than $\mathbf{NC}^i$. The idea of $\mathcal{A}$-$\mathbf{NC}^i_{\mathbb{D}}$ is that we allow not only Boolean inputs which then allows to combine such circuits, i.e. the output of one circuit is the input of another one.

The following notions we later need for applying the main theorem. It helps to embed some class $\mathcal{A}$-$\mathbf{NC}^1$ in e.g. a purely Boolean one.

Definition 15 (Codings of algebras). *Given algebras* $\mathcal{A} = (\{\mathbb{D}_1, \ldots, \mathbb{D}_S\}, F)$ *and* $\mathcal{A}' = (\{\mathbb{D}'_1, \ldots, \mathbb{D}'_{S'}\}, F')$*, where* F *resp.* F' *contain the operations. We say* $\mathcal{A}'$ *is a coding of* $\mathcal{A}$ *if there exists a relation* $c \subseteq \mathbb{D} \times \mathbb{D}'$ *on the domains of* $\mathcal{A}$ *and* $\mathcal{A}'$ *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* $\circledast(x_1, \ldots, x_n) \in c^{-1}(c(y)) \Leftrightarrow \circledast'(x'_1, \ldots, x'_n) \in c(y)$ *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 $\mathcal{A} \preceq \mathcal{A}'$.

Note that $\preceq$ is transitive. Also if we have two families $(\mathcal{A}_n)_{n\in\mathbb{N}}$ and $(\mathcal{A}'_n)_{n\in\mathbb{N}}$, we write $(\mathcal{A}_n)_{n\in\mathbb{N}} \preceq (\mathcal{A}_n)_{n\in\mathbb{N}}$ if $\mathcal{A}_n \preceq \mathcal{A}'_n$ 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 the definition ensures that codings preserve all the information. This property can be thought of as injectivity for relations.

By taking $\mathcal{A} = (\{\mathbb{D}_1, \ldots, \mathbb{D}_S\}, \circledast_1, \ldots, \circledast_k)$ and c one can actually construct an algebra $c(\mathcal{A}) = (\{c(\mathbb{D}_1), \ldots, c(\mathbb{D}_S)\}, \circledast_1^c, \ldots, \circledast_k^c)$ such that $\mathcal{A} \preceq c(\mathcal{A})$. This is slight abuse of notation since an algebra $c(\mathcal{A})$ is not already defined by c and $\mathcal{A}$. The operations $\circledast_i^c$ can be defined in different ways. However it will 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.

3 An algorithm for evaluating terms

Given some term and an algebra $\mathcal{A}$ of the same signature, what does the term evaluate to over $\mathcal{A}$? This problem is the term evaluation problem. The purpose of this section is to prove our Main Theorem:

Theorem 16 (Main Theorem). *Given a universal algebra* $\mathcal{A}$ *of single-sorted signature* σ *and domain* $\mathbb{D}$*, then the evaluation function* $\eta_{\mathcal{A}} \colon \mathsf{Term}_\sigma \to \mathbb{D}$ *is in DLOGTIME-uniform* $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$.

Note that the theorem and its proof are independent of the actual algebra $\mathcal{A}$. Here we only need to consider its signature which is single-sorted and can be assumed to have at most binary operations. How to deal with the algebras in particular (e.g. how to get Boolean circuits for the evaluation problem) is subject of the following section which shows applications for the Main Theorem.

3.1 Representing terms

Recall that we assume all operations used in terms to be binary. Terms like $(\phi \circledast \psi)$, where ϕ and ψ are also terms, are infix expressions. Instead of infix expressions, we can also consider postfix expressions. If ϕ' and ψ' are the equivalent postfix terms for ϕ and ψ as infix expressions, then $\phi'\psi'\circledast$ is the postfix equivalent of $(\phi \circledast \psi)$. Note that conveniently we do not need parentheses any more.

Definition 17. *A postfix term is in postfix normal form (PNF) if for all subterms $\phi\psi\circledast$, ϕ, and ψ holds that $|\phi| \geq |\psi|$.*

This normal-form is due to Buss and his algorithm [8] for evaluating Boolean formulas. Our algorithm has its roots in this and other related work [10,19]. We are aware of a simplified version [9] of [8] which directly operates on the infix notation, however we found the normal-form to be more convenient.

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

From now on focus on PNF terms without always explicitly calling it PNF. For the algorithm we put emphasis on the fact that terms are trees coded as strings. Also from now on we want to distinguish between open and closed terms. The terms es defined so far a called closed in opposition to open terms:

Definition 18. *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.*

So open terms are suffixes of closed terms. If we think about the tree a term represents then taking a suffix which is an open term corresponds to copping off a left-most subtree; see figure 3.1. Also we can concatenate open terms and get again an open term. An open term concatenated with a closed term results again in a closed term.

Given a term T of length n then for $1 \leq i \leq j \leq n$ we write $i <_T j$ if in the tree j is an ancestor of i. By convenience by $[i, j]$ we ambiguously mean the interval $\{i, \ldots, j\}$ as well the subword $T_i \ldots T_j$ and it will always be clear from the context what we mean. If $i <_T j$ and $[i, j]$ is a term, then we write $i \lhd_T j$.

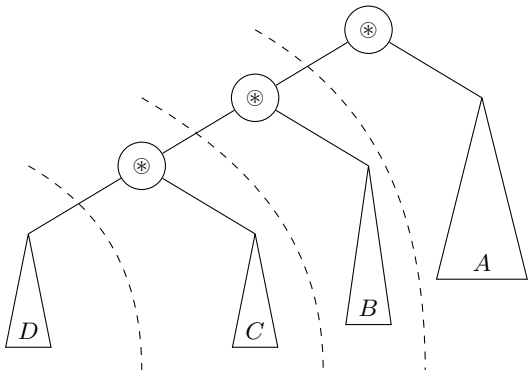

Fig. 1. The figure shows a PNF term T with the first three most-left operation symbols from the top pointed out. The term T is of the form $DC \circledast B \circledast A\circledast$, where A, B, C, and D are again terms. Note that $|A| \geq |DC \circledast B \circledast|$, $|B| \geq DC\circledast$, 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.

3.2 Dividing terms

For the following, T is closed PNF term. It ranges from 1 to n. We want to evaluate subintervals $[l, r]$. The size of the interval is $s = r - l + 1$. Such an interval also has a middle point. Depending on whether s is even or odd we could define a middle by rounding up or down but actually we need to be flexible there and take the middle position as given from outside. So usually we are given not only l and r but also a position m which is the middle position. It can be $\lfloor l + \frac{s}{2} \rfloor$ or $\lceil l + \frac{s}{2} \rceil$. Interval borders we will use for the recursion intervals are $l' = \lfloor l + \frac{s}{3} \rfloor$ and $r' = \lceil l + \frac{2s}{3} \rceil$. This divides the interval $[l, r]$ in thirds. We not only consider the three thirds but also the 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 input:

- The largest closed or open subterm in $[l, r]$ that contains m. This interval is denoted as $\mathcal{M}(l, m, r) = [\mathcal{M}^1(l, m, r), \mathcal{M}^2(l, m, r)]$.
- 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 $\mathcal{N}(l, m, r) = [\mathcal{N}^1(l, m, r), \mathcal{N}^2(l, m, r)]$.

– The largest open subterm in $[l, r]$ that precedes $\mathcal{M}(l, m, r)$. This interval is denoted as $\mathcal{L}(l, m, r) = [\mathcal{L}^1(l, m, r), \mathcal{L}^2(l, m, r)]$. It is $\mathcal{L}^2(l, m, r) + 1 = \mathcal{M}^1(l, m, r)$.
– The largest open subterm in $[l, r]$ that follows $[\mathcal{M}^1(l, m, r), \mathcal{M}^2(l, m, r) + 1]$. This interval is denoted as $\mathcal{R}(l, m, r) = [\mathcal{R}^1(l, m, r), \mathcal{R}^2(l, m, r)]$. It is $\mathcal{R}^1(l, m, r) - 2 = \mathcal{M}^2(l, m, r)$ and $\mathcal{M}^2(l, m, r) + 1$ is a binary operation symbol. If this operation exists, we denote the set containing its position by $O(l, m, r) = \{\mathcal{M}^2(l, m, r) + 1\}$.

If it is clear that the interval is given by (l, m, r), we drop it in the notation and write $\mathcal{M}$, $\mathcal{N}$, $\mathcal{L}$, $\mathcal{R}$, and O.

Note that a $\mathcal{M}$ interval could correspond to an open or a closed term. A $\mathcal{N}$ interval always corresponds by definition to an open term since late we only need it the open ones. The $\mathcal{L}$ interval is also defined to be open however even if we allowed it to be closed, it would still be always open because it is shorter as $\mathcal{M} \cup \mathcal{N}$ and so cannot be the complete second operand of O. In the case of $\mathcal{R}$ we again are only interested in open terms due to the way how we use it. Figure 3.2 shows the considered intervals.

The intervals might be empty however importantly they are unambiguous, which is immediately clear but for $\mathcal{M}$. This can be seen by maximality and the following lemma:

Lemma 19. *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.*

Proof. 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. □

Lemma 20. *It holds $\mathcal{M}^2 = \mathcal{N}^2$ and $\mathcal{M}^2(l, l', r' - 1) + 1 = \mathcal{N}^1(l' + 1, r', r)$.*

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

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

The key lemmas which later constitute the recursive evaluation algorithm are the following ones. They show how to actually compose a term by subterms coming from static subintervals. Figure 3.2 shows the involved subintervals for the next lemma.

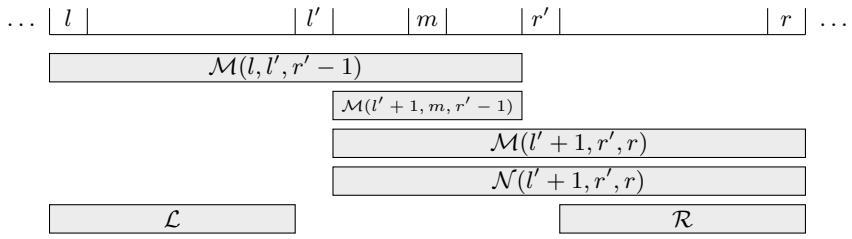

Fig. 2. The figure shows how an recursion interval is subdivided into smaller recursion intervals. In this case the subdivision for computing $\mathcal{M}(l, m, r)$ is shown. The six intervals yields recursively six values which may be used to be combined in order to get the evaluation of the $\mathcal{M}(l,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

Lemma 21. *Given a term T and subinterval $[l, r]$ with middle m, $\mathcal{M}$ equals one of the following intervals:*

1. $\mathcal{M}(l, l', r' - 1)$

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

Further the sets involved in the unions of case 4 and 5 are disjoint unions.

Proof. If $\mathcal{M}$ 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 $\mathcal{M}(l, l', r'-1) \cup \mathcal{N}(l'+1, r', r)$. By Lemma 20 we know this interval is a disjoint union. Further A is a closed or open term contained in $\mathcal{M}$ which contains m. If $A = \mathcal{M}$ we are done as case 4 holds.

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

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 $\mathcal{L}$ coincides with B'. The subterm $B'A\circledast = \mathcal{L}(l, m, r) \cup \mathcal{M}(l, l', r'-1) \cup \mathcal{N}(l'+1, r', r) \cup O(l, m, r)$ 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 $\mathcal{R}(l, m, r)$. Note that $\mathcal{L}(l, m, r)$ and $\mathcal{R}(l, m, r)$ might be empty. This concludes the fifth case. □

Figures 3.2 and 3.2 show how the interval is subdivided in case five.

In a very similar way we can treat $\mathcal{N}$:

Lemma 22. *Given a term T and subinterval $[l, r]$ with middle m, $\mathcal{N}$ equals one of the following intervals:*

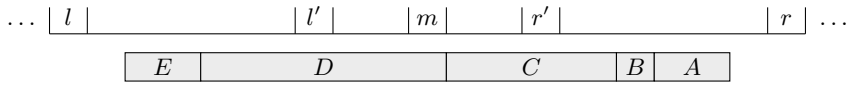

Fig. 3. In case five for $\mathcal{M}$ as shown in Lemma 21, the interval is subdivided into five parts. We see that DC is a closed term where $D = \mathcal{M}(l, l', r'-1)$ and $C = \mathcal{N}(l'+1, r', r)$. Further, B consists of a single position which is a operation symbol and A and E are open terms.

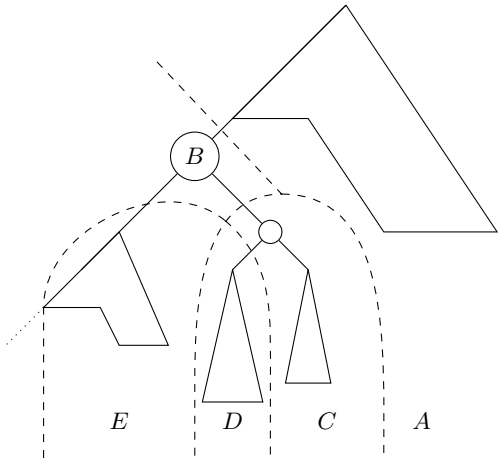

Fig. 4. A graphical representation of case five for $\mathcal{M}$; see Lemma 21 and also figure 3.2. Note that A, C, and E represent open terms and D a closed one. The term DCB then is open again.

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

Further the sets involved in the unions of case 4 and 5 are disjoint.

Proof. 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 $\mathcal{N}(l, l', r' - 1)$, $\mathcal{N}(l' + 1, r', r)$, or $\mathcal{N}(l' + 1, m, r' - 1)$ respectively.

Otherwise let $A = \mathcal{N}(l, l', r' - 1) \cup \mathcal{N}(l' + 1, r', r)$, similar to the previous proof. Note that by Lemma 20 we get that $\mathcal{N}(l, l', r' - 1) \cup \mathcal{N}(l' + 1, r', r)$ 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 $\mathcal{N}$ due to maximality of $\mathcal{N}^1$. 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 $\mathcal{R}(l, m, r)$. This is possible since $\mathcal{M}^2 = \mathcal{N}^2$, which we know from Lemma 20. □

The intervals $\mathcal{M}$ and $\mathcal{N}$ are built around the property of containing a middle position m. The intervals $\mathcal{L}$ and $\mathcal{R}$ are different. They can lie arbitrarily in $[l, l' - 1]$ resp. $[r' + 1, r]$ and we initially do know nothing about the location of the middle points. Our goal is to reduce $\mathcal{L}$ and $\mathcal{R}$ to some $\mathcal{M}(\bar{l}, \bar{m}, \bar{r})$ where find $\bar{l}$, $\bar{m}$, and $\bar{r}$ using a binary search.

Lemma 23. *Given a term T and subinterval $[l, r]$ with middle m then for $\mathcal{L}$ there is an interval $[\bar{l}, \bar{r}] \subseteq [l, l' - 1]$ with middle $\bar{m}$ that can be found by binary search from l, m, r such that $\mathcal{L} = \mathcal{M}(\bar{l}, \bar{m}, \bar{r})$.*

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

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

□

Lemma 24. *Given a term T and subinterval $[l, r]$ with middle m then for $\mathcal{R}$ there is an interval $[\bar{l}, \bar{r}] \subseteq [r' + 1, r]$ with middle $\bar{m}$ that can be found by binary search from l, m, r such that $\mathcal{R} = \mathcal{M}(\bar{l}, \bar{m}, \bar{r})$.*

Proof. This proof is similar to the previous one. First note that $\mathcal{R} \subseteq \mathcal{M}(\bar{l}, \bar{m}, \bar{r})$ and $\mathcal{R}^2 = \mathcal{M}^2(\bar{l}, \bar{m}, \bar{r})$ because of maximality. Further $\mathcal{R}$ is an open term. Now if $\mathcal{M}(\bar{l}, \bar{m}, \bar{r})$ 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. □

3.3 The evaluation algorithm

The algorithm we present is a recursive one which is given in terms of circuits. Lemmas 21, 22, 23, and 24 directly suggest how the recursive evaluation will work: To evaluate an interval we compute smaller fixed subintervals and then use the results to obtain the overall result.

In particular we need the following parts:

– Conversion of the term into PNF.
– Decision procedures determining for given intervals which case holds. By case we mean the ones from Lemma 21 and 22.
– The actual evaluation using the PNF and the computed cases.

The first step is the PNF conversion for which we refer to [8]. The conversion is of complexity $\mathbf{TC}^0$. The resulting term is T.

For the evaluation we need to implement circuits which on a given interval $[l, r]$ evaluate the intervals $\mathcal{M}$, $\mathcal{N}$, L, and R. In the case of $\mathcal{M}$ we need to distinguish whether the evaluation is a value or a univariate function. In the other cases the result is always a function.

– $\textsc{Eval}_{\text{closed}}(\mathcal{M}(l, m, r))$
– $\textsc{Eval}(\mathcal{M}(l, m, r))$
– $\textsc{Eval}(\mathcal{N}(l, m, r))$

– $\textsc{Eval}(\mathcal{L}(l,m,r),\bar{l},\bar{m},\bar{r})$
– $\textsc{Eval}(\mathcal{R}(l,m,r),\bar{l},\bar{m},\bar{r})$

The variables $\bar{l},\bar{m},\bar{r}$ exist to serve the binary search as mentioned in Lemma 23 and 24.

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 feed into a multiplexer-gate which chooses the right output.

The circuits determining the cases are called:

– $\textsc{Case}(\mathcal{M}(l,m,r))$
– $\textsc{Case}(\mathcal{N}(l,m,r))$
– $\textsc{Case}(\mathcal{L}(l,m,r),\bar{l},\bar{m},\bar{r})$
– $\textsc{Case}(\mathcal{R}(l,m,r),\bar{l},\bar{m},\bar{r})$

In the end, $\textsc{Eval}_{\text{closed}}(\mathcal{M}(1,\lfloor n/2\rfloor,n))$ is the circuit evaluating the whole term. For all recursive definitions of circuits, assume some look-up table construction if the interval becomes smaller than some constant. Also if we evaluate an open interval and the intervals happens to be empty, then we output the identity function.

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

This part is based on Lemma 21. Consider its five cases:

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

The circuit $\textsc{Case}(\mathcal{M}(l,m,r))$ 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 $\textsc{Case}_i(\mathcal{M}(l,m,r))$. 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 [8].

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

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

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

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

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

Now that we have the means of deciding the case of Lemma 21 for a given interval, we can actually evaluate the interval. Recursively we get the results for the intervals $\mathcal{M}(l, l', r' - 1)$, $\mathcal{M}(l' + 1, r', r)$, $\mathcal{M}(l' + 1, m, r' - 1)$, $\mathcal{N}(l' + 1, r', r)$, $\mathcal{L}(l, m, r)$, and $\mathcal{R}(l, m, r)$. 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 $\textsc{Eval}_{\text{closed}}(\mathcal{M}(l,m,r))$ we use a functional application gate ($\odot$) which gets the results from $\textsc{Eval}_{\text{closed}}(\mathcal{M}(l,l',r'-1))$ and $\textsc{Eval}(\mathcal{N}(l'+1,r',r))$. For the output of $\textsc{Eval}(\mathcal{M}(l,m,r))$ we use a composition gate ($\circ$) which gets the outputs of $\textsc{Eval}(\mathcal{M}(l,l',r'-1))$ and $\textsc{Eval}(\mathcal{N}(l'+1,r',r))$.

– Case five is composed as $\mathcal{L}(l,m,r) \cup \mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r) \cup O(l,m,r) \cup \mathcal{R}(l,m,r)$. The subinterval $\mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r)$ is a term an can be obtained like in case four. For interval $\mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r) \cup O(l,m,r)$ we use that result and feed it together with a 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 $\textsc{Eval}(\mathcal{R}(l,m,r))$ and feed it into a composition gate which then yields the value for $\mathcal{M}(l,l',r'-1) \cup \mathcal{N}(l'+1,r',r) \cup O(l,m,r) \cup \mathcal{R}(l,m,r)$. Finally we take this value and compose it with the result of $\textsc{Eval}(\mathcal{L}(l,m,r))$ to get the value for the whole interval; see figure 3.4.

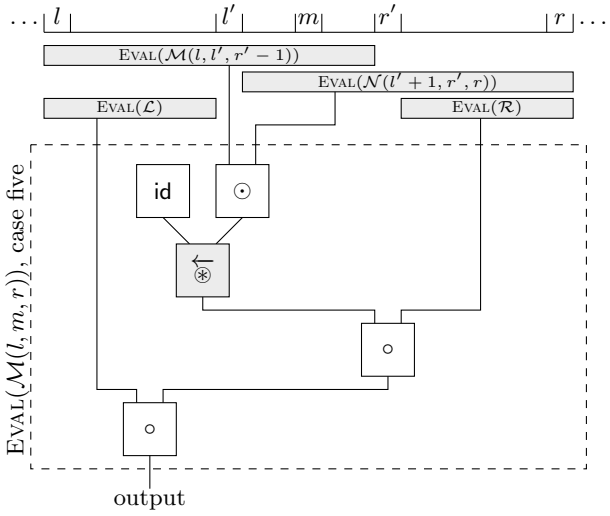

Fig. 5. The dashed box represents the subcircuit of $\textsc{Eval}(\mathcal{M}(l,m,r))$ which performs the combination in case fife. Note that the box $\circledast$ corresponds to the operation symbol in position B in figures 3.2 and 3.2. This box actually is not a single gate but also a construction which is shown in figure 3.4.

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:

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

This is the same as the formula for $\textsc{Case}_5(\mathcal{M}(l,m,r))$ but it also checks whether $\circledast$ is in the place of $O(l,m,r)$; see figure 3.4.

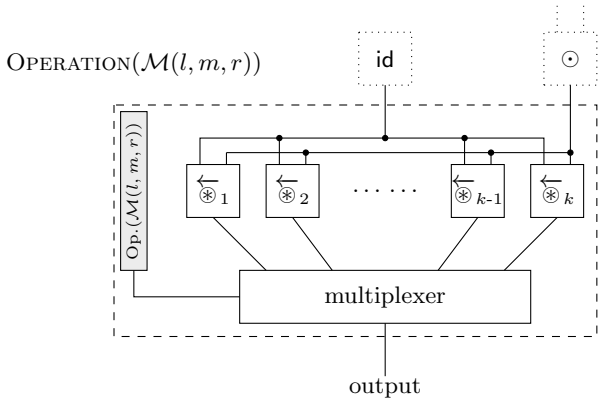

Fig. 6. In case five of the computation of $\textsc{Eval}(\mathcal{M}(l,m,r)$, the operator has to be computed and used. Figure 3.4 shows where the operator circuit shown here has to be inserted.

Finally we have these five possible combinations, we use a multiplexer gate and the results of $\textsc{Case}_i(\mathcal{M}(l,m,r))$ to select the right one as output; see figure 3.4.

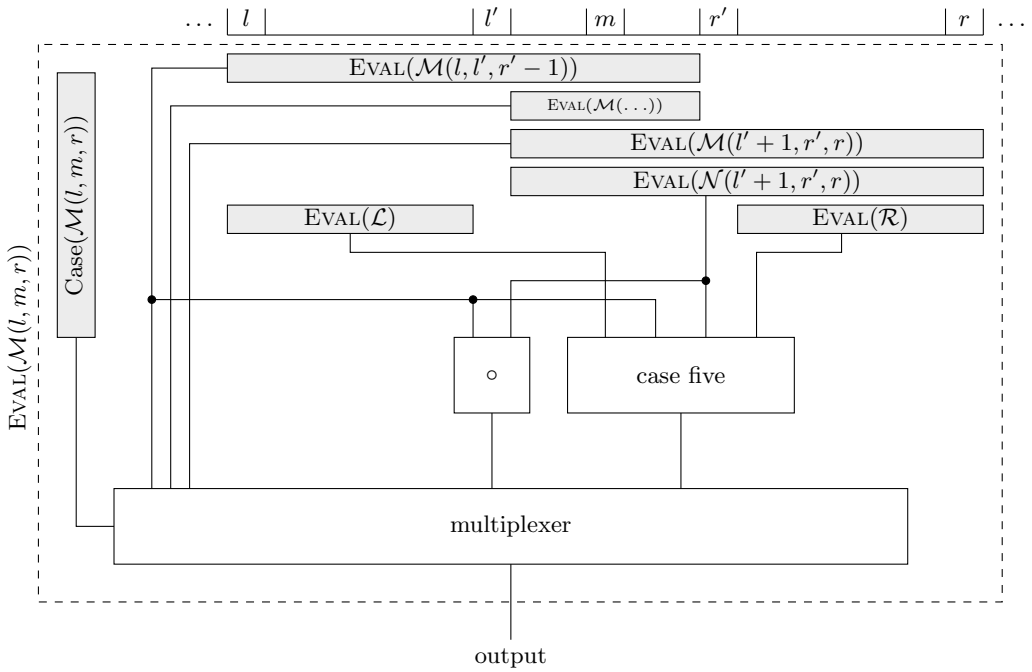

Fig. 7. Construction for the $\textsc{Eval}(\mathcal{M}(l, m, r))$ circuit. It consists of 5 recursive calls, a circuit for determining the case and a subcircuit performing the combination for case five as shown in figure 3.4.

3.5 The evaluation algorithm - Case($\mathcal{N}(l,m,r)$) and Eval($\mathcal{N}(l,m,r)$)

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

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

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

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

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

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

Now by applying Lemma 22 we can build $\textsc{Eval}(\mathcal{N}(l,m,r))$. Consider the cases:

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

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

3.6 The evaluation algorithm - Case($\mathcal{L}(l,m,r), \bar{l}, \bar{m}, \bar{r}$) and Eval($\mathcal{L}(l,m,r), \bar{l}, \bar{m}, \bar{r}$)

I 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 $\mathcal{L}$ interval we face the problem that is can lie arbitrarily in the considered interval. So the idea is that we do a binary search in order to find a interval whose middle is part of $\mathcal{L}$; see Lemma 23.

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

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

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

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

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

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

3.7 The evaluation algorithm - Case($\mathcal{R}(l, m, r), \bar{l}, \bar{m}, \bar{r}$) and Eval($\mathcal{R}(l, m, r), \bar{l}, \bar{m}, \bar{r}$)

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

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

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

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

3.8 Complexity and correctness: Proof of Theorem 16

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

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

The construction also stays 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 $\textsc{Eval}(\mathcal{M}(l, m, r))$. 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 $\textsc{Eval}(\mathcal{M}(l, m, r))$. An addressing scheme can look like this: We assign each $\textsc{Eval}(\mathcal{M}(l, m, r))$ 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 applies 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 a input $w\$x$ it is only a look-up to which kind of gate x corresponds. The wiring between gates can also expressed: For a pair of combination gates of some $\textsc{Eval}(\mathcal{M}(l, m, r))$, 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 $\textsc{Eval}(\mathcal{M}(l, m, r))$ 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 $\textsc{Case}(\mathcal{M}(l, m, r))$ which are given in terms of MAJ[<] formulas. By [5] we know that these are also in DLOGTIME-uniform $\mathbf{NC}^1$.

4 Application

Our meta theorem can be used for many different applications. All applications follow the same line of proof. This recipe we will illustrate in detail. It consists of the following steps:

1. **Find an algebra $\mathcal{A}$.** Given a problem P, which could be a language or a function, find an algebra $\mathcal{A} = (\mathbb{D}, \circledast_1, \ldots, \circledast_k)$, such that P reduces to term evaluation over $\mathcal{A}$.
2. **Find a coding for $\mathcal{F}(\mathcal{A})$.** Now we know by our main theorem that P is in $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$. However what we want is a "real" class like $\mathbf{NC}^1$ or $\#\mathbf{SAC}^7$. Hence we have to code $\mathcal{F}(\mathcal{A})$ in a way that we end up with a Boolean or an arithmetic class. So find a code c mapping into an algebra resp. a family of algebras, that have domains based on $\mathbb{B}$, $\mathbb{N}$, or $\mathbb{Z}$, depending on whether you want to prove Boolean or arithmetic circuit upper bounds.
3. **Analyze the complexity of the operations used in $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$.** Now we know that P is in $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$ since the coding admits a reduction. If we have chosen c well, we can implement the operations of $c(\mathcal{F}(\mathcal{A}))$ efficiently. Note that $c(\mathcal{F}(\mathcal{A}))$ could be a family. The members of the family all contain the following coded operations:
   - The operations of $\mathcal{A}$: $\circledast_1^c \ldots \circledast_k^c$.
   - 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 $\mathcal{F}(\mathcal{A})$: $\circ^c \colon \mathbb{D}^{\mathbb{D}} \times \mathbb{D}^{\mathbb{D}} \to \mathbb{D}^{\mathbb{D}}$.
   - The function application operation of $\mathcal{F}(\mathcal{A})$: $\odot^c \colon \mathbb{D}^{\mathbb{D}} \times \mathbb{D} \to \mathbb{D}$.

An algebra usually also has 0-ary operators, but here there is no complexity to analyze.
The following is not part of the algebra but comes into play in the construction of the $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$ circuits:
– Multiplexer operations for all subdomains $\mathbb{D}$ of $c(\mathcal{F}(\mathcal{A}))$: $\mathrm{mp}_{\mathbb{D}}$.
These operations are used in the $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$ 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. If, say, all the operations are in $\#\mathbf{NC}^1_{\mathbb{D}}$, then $c(\mathcal{F}(\mathcal{A})) \subseteq \#\mathbf{NC}^2$. Actually we can subsume all cases we come across later in table 3:

Complexity of operations				Overall complexity		
$\mathbf{NC}^i$	$\mathbf{AC}^i$	$\mathbf{SAC}^i$	$\rightsquigarrow$	$\mathbf{NC}^{i+1}$	$\mathbf{AC}^{i+1}$	$\mathbf{SAC}^{i+1}$
$\#\mathbf{NC}^i$	$\#\mathbf{AC}^i$	$\#\mathbf{SAC}^i$		$\#\mathbf{NC}^{i+1}$	$\#\mathbf{AC}^{i+1}$	$\#\mathbf{SAC}^{i+1}$
$\mathrm{Gap}\mathbf{NC}^i$	$\mathrm{Gap}\mathbf{AC}^i$	$\mathrm{Gap}\mathbf{SAC}^i$		$\mathrm{Gap}\mathbf{NC}^{i+1}$	$\mathrm{Gap}\mathbf{AC}^{i+1}$	$\mathrm{Gap}\mathbf{SAC}^{i+1}$

Table 1. The depth increases by a logarithmic factor when comparing the complexity of the operations and the overall circuit. Note that arithmetic circuits can simulate Boolean gates.

All the applications we show follow this scheme. The first classical ones are the easiest ones and so best fit to exemplify the recipe.

4.1 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 $\mathcal{B} = (\mathbb{B}, \wedge\vee, \neg, \bot, \top)$.

Theorem 25 ([8]). *The Boolean formula value problem is in* $\mathbf{NC}^1$.

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

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

3. step. Consider the algebra $c(\mathcal{F}(\mathcal{B}))$. The operations of $c(\mathcal{B}) = \mathcal{B}$ 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 $c(\mathcal{F}(\mathcal{B}))\text{-}\mathbf{NC}^1 \subseteq \mathbf{NC}^1$. □

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

4.2 Evaluating arithmetic terms and distributive algebras

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

Theorem 27 ([10]). *Evaluating terms over $\mathcal{N}$ is in $\#\mathbf{NC}^1$.*

Proof. **1. step.** The problem is directly a term evaluation problem, hence no reduction is needed and we stick to $\mathcal{N}$.

2. step. Consider the algebra

$$\mathcal{F}(\mathcal{N}) = (\{\mathbb{N}, \widetilde{\mathbb{N}}\}, +, \times, 0, 1, \overleftarrow{+}, \overleftarrow{\times}, \overrightarrow{+}, \overrightarrow{\times}, \circ, \odot).$$

Here $\widetilde{\mathbb{N}} \subseteq \mathbb{N}^{\mathbb{N}}$. 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 have to show that the operations of $\mathcal{F}(\mathcal{N})$ leave 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.

3. step. We now have an upper bound of $c(\mathcal{F}(\mathcal{B}))$-$\mathbf{NC}^1$. 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 $c(\mathcal{F}(\mathcal{B}))$-$\mathbf{NC}^1 \subseteq \#\mathbf{NC}^1$. □

The same construction carries over to integers:

Theorem 28. *Evaluating terms over $\mathcal{Z}$ is in Gap$\mathbf{NC}^1$.*

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

4.3 Tree automata

Tree automata are finite state machines and the counter part to finite automata for strings. We are interested in proving upper bounds for membership problems known from [26].

For sake of a clear presentation, in this section we focus on binary trees but all results can be generalized. 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 30. *A nondeterministic bottom-up tree automaton (BUTA) is a tuple $\mathcal{T} = (Q, \Sigma, Q_0, F, \delta)$ 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 leafs 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.

In the following we also want to consider counting problems. A tree automaton accepts a word if there exists an accepting run. Further, in reference to [26] we consider the uniform membership problem where the automaton is not fixed but part of the input. However we first show a more general result.

Theorem 31. *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. **1. step.** Let Σ be a fixed alphabet. The input tree we interpret as a term over a family of algebras $(\mathcal{A}_n)_{n\in\mathbb{N}}$ with

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

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.

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

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

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 show that all functions of the algebra confirm with 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.

3. step. 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 $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$ 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:

Theorem 32 ([26]). *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 33. *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 34 ([26]). *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 class of context-free languages containing the regular ones. They were first covered under the name of input-driven pushdown languages in [28] and later rediscovered and popularized by Alur and Madhusudan in [3]. 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 restriction: 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 a unranked tree.

Definition 35 (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 36. *Given a nondeterministic VPA M and a well-matched word $w \in \Sigma^*$ as input, then computing the number of accepting runs of M on w is in $\#\mathbf{SAC}^1$*

Proof. **1. step** 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 $(\mathcal{A}_n)_{n\in\mathbb{N}}$ we will evaluate the term over. We choose

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

Now given as input a well-matched input word, we construct a term. If the input w is either the empty word or a 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 take care building the automaton into the term. We leave is out for readability. Just know that the same automaton is accessible in every step of the evaluation. The definition of the algebra operations in particular is as follows:

- $\dagger_\epsilon$ 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.

2. step The algebra $\mathcal{F}(\mathcal{A}_n)$ 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 $w_1 w_2$ 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 $w_1 w w_2$.

The idea for our coding c is that we 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 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 $c(\mathrm{id}) = I^{n,n}$, 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 $(q_2, \gamma) \in \delta_{\mathrm{call}}(q_1, a)$ and $q_4 \in \delta_{\mathrm{ret}}(q_3, b, \gamma)$ 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.

3. step Up to now we have reduced the problem such that we know it is in $c(\mathcal{F}(\mathcal{A}))$-$\mathbf{NC}^1$. 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 bound $\#\mathbf{SAC}^1$ 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 37. *The uniform membership problem for nondeterministic VPA is in* $\mathbf{SAC}^1$.

Theorem 38 ([25]). *For a fixed nondeterministic VPA, counting the number of accepting runs is in* $\#\mathbf{NC}^1$.

Theorem 39 ([16]). *For a fixed VPA, the membership problem is in* $\mathbf{NC}^1$.

Weighted automata theory is a branch of theory which received ample research. The original concept is based on finite automata however also generalizations to VPA have been investigated [11]. 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 40. *Functions of WVPA over a semiring* $\mathcal{A} = (\mathbb{D}, \oplus, \otimes)$ *are in* $\mathcal{A}$-$\mathbf{NC}^1$.

Proof. **1. step.** In a WVPA for all computations the sum of weights is obtained by $\oplus$ and then those weights are multiplied by $\otimes$. An approach of doing the computation in that order is awkward since there can exist exponentially many runs. But since we have a semiring at hand we can use distributivity. We again consider the input word basically 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 $(q_1, q_2) \to d \in \mathbb{D}$ says what the weight is which is accumulated when going from q_1 to q_2 by reading w. Hence let

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

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: $\dagger_\epsilon$. 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 $w_1 w_2$. 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 is 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 $\mathcal{A}$-term which evaluates to the final output because the maps of $\mathcal{A}'$ can be considered matrices and the matrix operations can be made explicit.

2. and 3. step. By Theorem 29 it follows that those terms over $\mathcal{A}$ can be evaluated in the bounds of $\mathcal{A}$-$\mathbf{NC}^1$ □

Applied, we directly obtain:

Theorem 41. *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 $\mathcal{A}$ in the context of weighted automata is $(\mathbb{N}, +, \min)$, hence:

Theorem 42. *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 40 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) [2] 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 actions on registers. A register holds a value of some algebra. Importantly registers do not have influence on the states which results in a tame and analyzable model. In [1] the complexity of CRA has been analyzed and in [23] 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 $\mathcal{A}$, a finite set of registers X, an initial valuation $v_0 \colon X \to \mathbb{D}$, and a register update function ρ:

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

Here $E(\mathcal{A}, X_{\text{prev}})$ is the set of expressions over the algebra $\mathcal{A}$, which may use variable names of X_{prev} which is a copy of X. In $E(\mathcal{A}, X_{\text{prev}}, X_{\text{prev}})$ there are two copies of the variable set X involved. The final cost function $\mu \colon Q \to E(\mathcal{A}, X)$ 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 $\mathcal{A}$-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 [23].

Theorem 43 ([23]). *Functions realized by CVPA over $(\mathbb{Z}, +)$ are in Gap$\mathbf{NC}^1$.*

Proof. **1. step.** 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 $r(w) \in (\Sigma \cup \{\epsilon\} \times Q \times (\Gamma \cup \{\epsilon\}))^{|w|+1}$ 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 $\gamma \neq \epsilon$ is the symbol which can be seen on the stack. For $r(|w|+1)$ we set (ϵ, q, ϵ). Using Theorem 39, the word $r(w)$ can be computed in $\mathbf{NC}^1$.

Let the algebra be

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

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 $v_1 x_1 + \ldots + v_m x'_m + v_1 x_1 + \ldots + v_m x'_m$ 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 $\dagger_\epsilon$ 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 $r = r_1 r_2$ 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 a internal letter e are assigned a term $\dagger_{e,q}$ for q. If there is a factor which corresponds to a word awb where $w \neq \epsilon$ 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 $\otimes_{a,q_a,b,q_b,\gamma} \dagger_\epsilon$.

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 $\mu(f(v_0))$.

2. step. The algebra $\mathcal{F}(\mathcal{A})$ 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 a 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 $\mathcal{F}(\mathcal{A})$, 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 $x \mapsto A_f x B_f + C_f$ and g is of the form $x \mapsto A_g x B_g + C_g$. Now $c(f \circ g) = c(f) \circ^c c(g)$ is the map $x \mapsto A_f(A_g x B_g + C_g)B_f + C_f = A_f A_g x B_g B_f + A_f C_g B_f + C_f$, so $c(f \circ g) = (A_f A_g, B_g B_f, A_f C_g B_f + C_f)$.
- $\circledcirc^c$: 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 $\dagger^c_\epsilon$ 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 $x \mapsto M_{q_a,a}(AxB + C)M^1_{q_b,b,\gamma} + M^2_{q_b,b,\gamma} = M_{q_a,a} AxB M^1_{q_b,b,\gamma} + M_{q_a,a} C M^1_{q_b,b,\gamma} + M^2_{q_b,b,\gamma}$.

3. step. All operations of the algebra $c(\mathcal{F}(\mathcal{A}))$ 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 $c(\mathcal{F}(\mathcal{A}))$ 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 fix graph 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 [15].

The Tree width [20] is a parameter which has been successfully utilized to bound complexity, where Courcelle's Theorem is a prime example [13].

Definition 44. *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 $\tau^{-1}(r) \cap t^{-1}(s) \subseteq \tau^{-1}(t)$*

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 [14,35]. Both notions are closely related. A set of graphs has bounded

clique width iff it has bounded NLC-width [14]. 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.

Clique decompositions are yielding the property of clique width in the following way. 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 45 (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 [17] that this is possible in log-space. For clique width the complexity is poly-time [29].

4.6 Circuits of bounded tree width

We apply the term evaluation algorithm to a recent result concerning circuits of bounded tree width [21]. It states that Boolean circuit families of polynomial size 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 it in correspondence to the graph 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 46. *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 then 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 $\mathcal{A} = (\mathbb{D}, \circledast_1, \ldots, \circledast_k)$ 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

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

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-are operator values of $\mathcal{A}$ 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 $v_1 \neq \bot$ and $v_2 \neq \bot$ 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 $\bot$.

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 $\mathcal{A}(w)$ while dropping the circuit in the notation.

Theorem 47. *Given a family of circuits C of bounded tree width and polynomial size over an algebra $\mathcal{A}$, then we can find an equivalent $\mathcal{F}(\mathcal{A}(w))$-$\mathbf{NC}^1$ 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 46.*

Proof. We take the decomposition of width $w-1$ satisfying the conditions of Lemma 46. We interpret it as a term over the algebra $\mathcal{A}(w)$ 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 $\bot$ 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 $\mathcal{F}(\mathcal{A}(w))$-$\mathbf{NC}^1$ 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 $\mathcal{F}(\mathcal{A}(w))$-$\mathbf{NC}^1$ 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 48 ([21]). *Languages accepted by families of Boolean circuits of polynomial size and bounded tree width are in* $\mathbf{NC}^1$.

Proof. Since $\mathcal{F}(\mathcal{A}(w))$ is finite, $\mathcal{F}(\mathcal{A}(w))$-$\mathbf{NC}^1 \subseteq \mathbf{NC}^1$ follows from Theorem 26 and 47. □

4.7 Courcelle's Theorem

Courcelle's Theorem [13] is a famous example of a so-called 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. [17] 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 [18]. 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 [18] 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 [6]. 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 $\odot_a^V \colon H \to H$ 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 49 ([18]). *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. **1. step**

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 [13] results in 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 constriction 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 $\mathcal{V}$-structures [33]. In this case a $\mathcal{V}$-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 $\mathcal{V}$-structures. And each $\mathcal{V}$-structure belongs to a tree which we get by stripping it of the variable information. In the following we consider the language of $\mathcal{V}$-structures. Given a formula with a free variable and an input tree, we count how many $\mathcal{V}$-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 [13]. 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

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

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 $\mathcal{A}$ 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 $\mathcal{A}$. 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.

2. step

The algebra $\mathcal{F}(\mathcal{A})$ 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.

3. step

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 which belongs to Karp's classical 21 NP-complete problems [22]. In [35] 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 [35]: 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 its value $|\{\{e_1, e_2\} \in E \mid e_1 \in V_1 \wedge e_2 \in V_2\}|$ 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 [35] and show a smaller upper bound.

Theorem 50. *The maximum cut problem for width k is in* $\mathbf{SAC}^1$.

Proof. **1. step.** 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 $(\mathcal{A}_n)_{n\in\mathbb{N}}$ to that term. If evaluated we get the desired value. So let

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

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 $(a_1, \ldots, a_k, b_1, \ldots, b_k, c)$. 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 [35].

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 $(a_1, \ldots, a_k, b_1, \ldots, b_k, c) \mapsto (a'_1, \ldots, a'_k, b'_1, \ldots, b'_k, c)$ 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: $\{x_1, \ldots, x_m\} \mapsto \{\otimes'_l(x_1), \ldots, \otimes'_l(x_m)\}$. These unary operations directly correspond the 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 $x = (a'_1, \ldots a'_k, b'_1, \ldots b'_k, c)$, $y = (a''_1, \ldots a''_k, b''_1, \ldots b''_k, c)$ and $\circledast'_S(x, y) = (a_1, \ldots a_k, b_1, \ldots b_k, c)$. 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 [35].

2. step. We first give a coding for $\mathcal{A}_n$ and then extend it to $\mathcal{F}(\mathcal{A}_n)$. Consider the domain of $\mathcal{A}_n$ which is $\mathcal{P}\left([n]^{2k+1}\right)$. 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

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

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 $\mathcal{F}(\mathcal{A}_n)$ we also have the subdomain $\widetilde{\mathbb{D}}$ which contains functions of the form $f\colon \mathcal{P}\left([n]^{2k+1}\right) \to \mathcal{P}\left([n]^{2k+1}\right)$. We will use the property $f(X) = \bigcup_{x\in X} f(\{x\})$ of these functions which we call *singleton property*. The functions have indeed singleton property: First the identity function clearly has it. Further if we are given two functions f, g which have singleton property, then $f \circ g$ has also: $(f \circ g)(X) = f(g(X)) = f(\bigcup_{x\in X} g(\{x\})) = \bigcup_{x\in X} f(g(\{x\}))$. 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:

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

An element is table where 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 $\mathcal{F}(\mathcal{A})$ follow immediately.

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

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

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 in [35] also computing Hamiltonian circuits in graphs was considered. They showed a poly-time upper-bound for this problem for bounded tree width graph inputs. It is

also possible to count the number of Hamiltonian circuits. In [4] a $\#\mathbf{SAC}^1$ upper bound has been shown for the case of bounded tree width. We will generalize this to bounded clique width.

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 $1 \leq i < m$. The Hamiltonian circuit problem for width k is the following: Given a clique decomposition if an undirected graph $G = (V, E)$ of width k. A Hamiltonian circuit is a path of length $|V|$ where there exists an edge from the first to the last edge.

Theorem 51. *Given a natural 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. **1. step** 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 $(\mathcal{A}_n)_{n\in\mathbb{N}}$:

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

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 [35] where they used $\mathcal{P}([n]^{k(k+1)/2})$ 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 [35] 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

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

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 [35]. 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 [35]. 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 $(v_1, 0, v_2)$. See [35] for the detailed procedure.
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 [35] 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 [35]. 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 [35], 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 that way we get the final result.

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

For $\mathcal{F}(\mathcal{A}_n)$ we have to consider the second subdomain which consists of operations F of the form $F\colon \mathbb{N}^{\binom{[n]^{k(k+1)/2}}{}} \to \mathbb{N}^{\binom{[n]^{k(k+1)/2}}{}}$. 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 $\mathcal{F}(\mathcal{A}_n)$. For The functional composition we get

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

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 $\mathcal{F}(\mathcal{A}_n)$ follows.

3. step We analyze the complexity of the operations of

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

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.
- $\widetilde{\otimes}_l^c$: By invoking $\otimes_l^c$ and applying it on all rows of the argument we get the result.
- $\overleftarrow{\circledast}_S^c$: If we want to compute $c(F)\overleftarrow{\circledast}_S^c c(d)$ we apply $c(d)$ on each row by $\overleftarrow{\circledast}_S^c$ and by that have a reduction.
- $\overrightarrow{\circledast}_S^c$: This case is similar to $\overleftarrow{\circledast}_S^c$.

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

5 Discussion

We have seen that many problems that are tree-like structured 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 are much more applications we do not yet know of. In particular for a variety of theorems we give new unified proofs. Further we show several new results:

- For tree automata we gave an upper bound for counting accepting computations. This applies for the uniform membership problem where the automaton is part of the input as well as for the classical membership problem where the automaton is fixed.
- The same we showed for visibly pushdown automata. Here only the bound for the classical membership problem was known.
- For weighted automata we showed an upper bounds for arbitrary algebras.
- We showed for circuits of bounded tree-width and polynomial size how to balance them for arbitrary algebras.

– We showed an upper bound for counting Hamiltonian cycles in a graph of bounded clique width. Before this was known for the case of bounded tree width and in the clique width case only the Boolean version of the problem was known.

We did not exhaust every single result which is nearby the ones we covered. For example a bound on counting the number of maximal cuts in bounded clique width graphs is certainly achievable.

We would be also interested in examples where we end up with circuits of $\mathcal{O}(\log^i n)$ depth for $i > 1$ or in the reason why it is hard to find such examples.

References

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.
8. 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.
9. 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.

10. 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.
11. 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.
12. Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–21, 1985.
13. Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.
14. Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1-3):77–114, 2000.
15. Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.
16. Patrick W. Dymond. Input-driven languages are in log n depth. *Inf. Process. Lett.*, 26(5):247–250, 1988.
17. 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.
18. 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.
19. A. Gupta. A fast parallel algorithm for recognition of parenthesis languages. Master's thesis, 1985.
20. Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8(1):171–186, 1976.
21. Maurice J. Jansen and Jayalal Sarma. Balancing bounded treewidth circuits. *Theory Comput. Syst.*, 54(2):318–336, 2014.
22. 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.
23. 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.
24. Andreas Krebs, Nutan Limaye, and Meena Mahajan. Counting paths in VPA is complete for $\#\mathrm{nc}^1$. 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.
25. Andreas Krebs, Nutan Limaye, and Meena Mahajan. Counting paths in VPA is complete for #nc 1. *Algorithmica*, 64(2):279–294, 2012.
26. 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.
27. Nancy A. Lynch. Log space recognition and translation of parenthesis languages. *J. ACM*, 24(4):583–590, 1977.
28. 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.
29. Sang-il Oum and Paul D. Seymour. Approximating clique-width and branch-width. *J. Comb. Theory, Ser. B*, 96(4):514–528, 2006.
30. V. Ramachandran. Restructuring formula trees. Unpublished manuscript, 1986.
31. Walter L. Ruzzo. On uniform circuit complexity. *J. Comput. Syst. Sci.*, 22(3):365–383, 1981.
32. 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.
33. Howard Straubing. *Finite Automata, Formal Logic, and Circuit Complexity*. Birkhäuser, Boston, 1994.
34. Heribert Vollmer. *Introduction to Circuit Complexity - A Uniform Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 1999.
35. Egon Wanke. k-nlc graphs and polynomial algorithms. *Discrete Applied Mathematics*, 54(2-3):251–266, 1994.

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il