A Unified Method for Placing Problems in Polylogarithmic Depth[*]

Andreas Krebs[1], Nutan Limaye[2], and Michael Ludwig[3]

1 **University of Tübingen, Germany**
2 **IIT Bombay, India**
3 **University of Tübingen, Germany**

Abstract

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

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

1998 ACM Subject Classification F.1.2 Modes of Computation

Keywords and phrases Polylogarithmic depth, Term evaluation, Parallel algorithms

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2017.36

1 Introduction

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

There are many interesting and fundamental computational problems for which the classical algorithms designed were inherently sequential in nature.

Owing to a series of theoretically intriguing and practically relevant discoveries we know **NC** algorithms for some of these problems. That is, a fairly diverse set of problems from the class **P** is known to be in **NC**. This raises a natural question: is **P** the same as **NC**? That is, can any sequential algorithm be turned into an efficient parallel one? In fact, this is one of the very central open questions in computer science. The question has implications to many different practical aspects of computer science such as distributed computing and parallel algorithms for large scale data (see for instance [25, 30, 38]).

[*] A full version of the paper is available at [22], `https://eccc.weizmann.ac.il/report/2017/019/`

A lot of effort has gone into understanding the relative strengths and weaknesses of **P** and **NC**. The study of Boolean and arithmetic circuits and many interesting results proved therein show that some specific subclasses of **NC** are strictly less powerful than **P**. (See for instance [13, 17, 18].) This rich body of work could be thought of as attempts to separate **NC** from **P**. On the other hand, over the last many years, surprising **NC** upper bounds have been proved for problems which were previously believed to be hard to parallelize. (See for instance [19, 28, 1, 16, 12].)

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

- We identify similarities between a large number of parallel algorithms. We observe that if a problem has a core *tree-like* structure, then it is amenable to have an **NC** algorithm. We formally define the notion of the tree-like structure and demonstrate the presence of such a structure in a large collection of problems.
- Our second important contribution is to mechanise the process of coming up with a parallel algorithm for any problem that has this tree-like structure. This can be thought of as an algorithmic contribution stemming from our work. We demonstrate the strength of this algorithmic technique by rediscovering many known **NC** upper bounds.
  This is also a technically challenging part of our work. The difficulty arises because we need an algorithm which is independent of the problem and only dependent on the underlying tree-like structure. The structure itself is dependent on the problem: for two seemingly unrelated problem these structures could be different. Therefore, the algorithm has to be general, which makes as few assumptions about the structure as possible.
  One caveat worth mentioning is that in some cases the tree-like structure is clear from the problem definition and in some cases it requires some work to notice this structure. We assume that an expert working on a specific problem may be able to notice this structure easily for the problem of her interest and then choose to use our approach mechanically to obtain an **NC** algorithm for the problem.

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

- Boolean and arithmetic term evaluation [5, 7].
- Membership for language classes [26, 24, 10, 2, 21].
- Evaluation of circuits with bounded tree-width [20].
- Courcelle's Theorem and counting [9, 11].
- Computation of maximal cuts in bounded clique-width graphs [37].
- Counting Hamiltonian paths in bounded clique-width graphs [3, 37].

Using our techniques, we reprove the above results (see the full version [22]). That is, we give a unified way of proving all the above bounds. Moreover, we also consider variants of the above applications and obtain parallel ($\mathbf{NC}^1$, $\mathbf{NC}^2$, $\mathbf{SAC}^1$, $\mathbf{SAC}^2$) upper bounds.

Techniques. Our approach uses algebra as a tool for obtaining the desired abstraction. An algebra $\mathcal{A}$ consists of a set $\mathbb{D}$ and a finite set of operations $\{\oplus_1, \oplus_2, \ldots, \oplus_k\}$, where k is fixed. As mentioned earlier, we focus on coming up with an efficient parallel algorithm for problems with a tree-like structure. For this, we consider the *term evaluation problem.* A term is simply a tree in which the leaves are labelled by the elements of $\mathbb{D}$ and the internal nodes are labelled by the operations. The term evaluation problem deals with evaluating the value of the term for a given assignment of the leaves from the domain $\mathbb{D}$. The main algorithmic and technical contribution of our work is to rewrite the term T as an equivalent term T' whose inputs are labelled by the elements of $\mathbb{D}$ and the internal nodes (gates) are labelled by the operations of an algebra $\mathcal{F}(\mathcal{A})$, which is an extended algebra derived from $\mathcal{A}$. Moreover, if T has size s then T' has size poly(s) and depth $O(\log s)$[1].

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

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

Related work. Our algorithm for the term evaluation problem fits in the long chain of contributions dedicated to the term evaluation problem. The origin of which can be vaguely traced back to the investigation of upper bounds for the Boolean formula value problem. In [27] Lynch studied it first and achieved a log-space upper bound. Subsequently Cook conjectured that this bound is tight [8] which, as we know today, is not (unless log space equals log depth). Earlier, a way to deal with formulas that are very deep trees was investigated by Spira [33]: by a quadratic increase in size, we can balance a Boolean formula. Brent built upon this work [4]. Going from balancing to obtaining an **NC** (in fact $\mathbf{NC}^1$, i.e. log-depth) upper bound is not tough. It is known that if the transformation can be done in $\mathbf{NC}^1$, the evaluation is in $\mathbf{NC}^1$.

Cook and Gupta [15] as well as Ramachandran [31] were the next in line who showed that $\mathcal{O}(\log n \log\log n)$ deep circuits suffice for evaluating formulas. Based on [15], Buss showed an ALOGTIME bound [5] which equals logarithmic depth [32] and is known to be tight. His proof utilized a sophisticated two-player pebbling game. From there on the research proceeded in the direction of broadening the scope of the result. This continued research is always rooted in the work of [15] and [5]. Many other interesting works have contributed to this rich line of research, each solving the term evaluation problem over a specific algebra [29], [10], [23], [21].

Subsequent work. Closely related to our work is a very recent work of Ganardi and Lohrey [14] which uses the notion of algebras, terms and extended algebras (i.e. $\mathcal{A}, T, \mathcal{F}(\mathcal{A})$)

[1] Please refer to [36] for definitions of circuits, size and depth notions for terms and circuits and circuits with gates coming from an algebra. Also, the size can in fact be bounded by $O(s)$, but that is not crucial here.

and shows optimal upper bounds on the size of the **NC** circuits obtained for some of the problems considered here. This is an interesting piece of follow up work, which improves on the size of the circuits (to $O(n/\log n)$) for some of the problems and uses some of the machinery developed here.

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

2 Preliminaries: notations and definitions

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

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

2.1 Notation

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

2.2 Many-sorted signatures, circuits and terms

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

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

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

A many-sorted signature $\boldsymbol{\sigma}$ is a pair of a word and a letter (w, a). The word codes the input sorts of the operation. The length of a word $|w|$ is the arity of the operation. E.g. the operation f defined above has the signature $(1122, 3)$, where the three sorts are $\mathbb{B}, \mathbb{N}, \mathbb{Q}$

(numbered in that order). For a operation f with signature $\boldsymbol{\sigma}$, we write $\mathsf{In}_{\boldsymbol{\sigma}}(f)$ to denote w and $\mathsf{Out}_{\boldsymbol{\sigma}}(f)$ to denote a. If we have a set of operations $f_1, \ldots, f_\ell$, with signatures $\boldsymbol{\sigma_1}, \ldots, \boldsymbol{\sigma_\ell}$ respectively, we use σ to denote the tuple $(\boldsymbol{\sigma_1}, \ldots, \boldsymbol{\sigma_\ell})$ and $\sigma(i)$ to address (w_i, a_i). Also, $\mathsf{In}_\sigma(i,j)$ is the jth letter of $\mathsf{In}_\sigma(f_i)$. If we have an ordering on the operations $f_1, \ldots, f_\ell$ then we simply use $\mathsf{In}_\sigma(i)$ instead of $\mathsf{In}_\sigma(f_i)$. We use $|\sigma|$ to denote the number of different operations, i.e. ℓ.

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

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

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

- If some $v \in V$ has in-degree 0 then $\mathsf{Gatetype}(v) \in \{x_1, \ldots, x_n\}$ or $\mathsf{In}_\sigma(\mathsf{Gatetype}(v)) = \epsilon$, 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. Also, $\mathsf{In}_C \subseteq [S]^n$ denotes the sorts of the n input gates and $\mathsf{Out}_C \subseteq [S]^m$ denotes the sorts of the m output gates.
- For all $v \in V$, let $v_1, \ldots v_{|\mathsf{In}_\sigma(\mathsf{Gatetype}(v))|}$ be the input gates for v such that $\mathsf{Order}(v_i) \leq \mathsf{Order}(v_j)$ iff $i \leq j$. Then $\mathsf{Out}_\sigma(\mathsf{Gatetype}(v_i)) = \mathsf{In}_\sigma(\mathsf{Gatetype}(v), i)$. If v_i is an input gate then $\mathsf{In}_C(j) = \mathsf{In}_\sigma(\mathsf{Gatetype}(v), i)$ where $\mathsf{Gatetype}(v_i) = x_j$.

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

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

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

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

[2] This is no restriction as an n-ary operations can be simulated by a small set of binary operations.
[3] Note that for a term to be over some signature, the signature has to admit 0-ary operations since we need them for the leafs.

We assume that the term (with or without an unknown) is encoded as a string over a fixed alphabet[4]. We call this a linearization of a term. The details of this are presented in [22]. From now on we will not distinguish between terms and their linearizations.

2.3 Semantics: evaluation of circuits and terms without unknown

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

▶ **Definition 4** (Many-sorted universal algebra)**.** Given a many-sorted signature σ with S sorts, a many-sorted universal algebra is a tuple $\mathcal{A} = (\{\mathbb{D}_1, \dots, \mathbb{D}_S\}, \circledast_1, \dots, \circledast_{|\sigma|})$ where $\circledast_i \colon \mathbb{D}_{\mathsf{In}_\sigma(i,1)} \times \dots \times \mathbb{D}_{\mathsf{In}_\sigma(i,\alpha_i)} \to \mathbb{D}_{\mathsf{Out}_\sigma(i)}$ where $\alpha_i = |\mathsf{In}_\sigma(i)|$. We call the sets $\mathbb{D}_i$ subdomains and the union of all subdomains $\mathbb{D}$, which is the domain.

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

▶ **Definition 5** (Evaluation of many-sorted circuits)**.** Given an algebra $\mathcal{A}$ over signature σ and a word $w \in \mathbb{D}^n$, 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 $\eta_{\mathcal{A},w}(T(v)) = w_i$ provided $w_i \in \mathbb{D}_j$ implies that $\mathsf{In}_C(i) = j$.
- Let α be the arity $|\mathsf{In}_\sigma(\mathsf{Gatetype}(v))|$ and $v_1, \dots v_k$ be the predecessors of v ordered by 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 $\mathsf{Outputgates}^{-1}(y_i) = v_i$ the following holds: $\mathsf{Out}_C(i) = \mathsf{Out}_\sigma(\mathsf{Gatetype}(v_i))$.

2.4 Semantics: evaluation of terms with an unknown

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

▶ **Definition 6** (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, the functional algebra is defined to be $\mathcal{F}(\mathcal{A}) = (\{\mathbb{D}, \widetilde{\mathbb{D}}\}, F)$, 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 $\mathcal{A}$: $\circledast_1, \dots, \circledast_k$.

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

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

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

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

▶ **Definition 7** (Evaluation of terms with an unknown)**.** Let $\mathcal{A} = (\mathbb{D}, \circledast_1, \dots, \circledast_k)$ 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 $\mu_{\mathcal{A}}\colon \mathsf{Term}_\sigma[X] \to \widetilde{\mathbb{D}}$ is a map defined inductively for all $i \in [k]$:

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

2.5 Interpreting classical circuit classes in this framework

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

▶ **Definition 8** (Family of algebras)**.** A family of algebras $(\mathcal{A}_n)_{n\in\mathbb{N}}$ is a sequence of algebras, where $\mathcal{A}_i = (\{\mathbb{D}_1^{p_1(i)}, \dots, \mathbb{D}_S^{p_S(i)}\}, \circledast_1^i, \dots, \circledast_k^i)$ for $i \in \mathbb{N}$ and p_j being polynomials for $j \in [S]$[5].

Here, let us assume that the circuits have one output gate and all input gates have the same sort[6]. Given an algebra $\mathcal{A}$ let $\mathbb{D}_I$ and $\mathbb{D}_O$ be the two subdomains that correspond to the input and output values, respectively. Then a circuit C_n of n inputs realizes a function $F_{\mathcal{A}}(C_n)\colon \mathbb{D}_I^n \to \mathbb{D}_O$. Given a family of circuits C we then get a function $F_{\mathcal{A}}(C) = \mathbb{D}_I^* \to \mathbb{D}_O$.

In general, assuming a family of circuits is very powerful, so in complexity it is natural to require that this family is computable in some complexity bound. We then speak of uniformity. Our constructions will be DLOGTIME-uniform. (See e.g. [34] or [35] for basics in circuit complexity.) We use our framework 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 9** ($\mathbf{NC}^i$)**.** 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 10** ($\mathbf{AC}^i$)**.** 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 $\mathcal{B}_n$.

[5] Note that we are assuming a family of signatures here, rather than a single signature.
[6] This is usually the case in the classical circuit complexity models and hence the assumption.

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

2.6 Composition of algebras and extended circuit classes

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

▶ **Definition 11** (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.

▶ **Definition 12** (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 $(\{\mathbb{B}, \mathbb{D}_1, \ldots, \mathbb{D}_S\}, \wedge, \vee, \neg, \circledast_1, \ldots, \circledast_k, (\mathrm{mp}_{\mathbb{D}_i})_{i\in[S]})$. 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

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

Hence we 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 13** ($\mathcal{A}$-$\mathbf{NC}^i$, $\mathcal{A}$-$\mathbf{NC}^i_{\mathbb{D}}$)**.** 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 $\mathcal{A}$-$\mathbf{NC}^i = \mathcal{A}$-$\mathbf{NC}^i_{\mathbb{B}}$.

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

3 Term evaluation algorithm

Given some term and an algebra $\mathcal{A}$ 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 regarding the parallel algorithm for the term evaluation problem.

▶ **Theorem 14** (Main Theorem)**.** *Given an algebra $\mathcal{A}$ of single-sorted signature σ and domain $\mathbb{D}$, the evaluation function $\eta_{\mathcal{A}} \colon \mathsf{Term}_\sigma \to \mathbb{D}$ can be computed in $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$. Moreover, the construction ensures that the we get a DLOGTIME-uniform $\mathcal{F}(\mathcal{A})$-$\mathbf{NC}^1$ circuit family.*

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

The rest of the section is devoted to proving the above theorem. Many details are omitted in the this version due to shortage of space. (See [22] for those details.)

3.1 Representing terms in a normal form

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

A postfix term is in *postfix normal form (PNF)* if for subterms ϕ and ψ of $\phi\psi\circledast$, $|\phi| \geq |\psi|$. This normal-form was defined by Buss to design an **NC** algorithm [5] for evaluating Boolean formulas. Our algorithm has its roots in this and other related works [7, 15]. We are aware of a simplified version [6] of [5] which directly operates on the infix notation, however we use PNF as that is more convenient here. The details of how the given term is converted into the PNF form can be found in [22]. A crucial property of a term in PNF is that it always has suffixes that are *open* terms, which correspond to terms with an unknown.

3.2 Dividing terms: some structural lemmas regarding terms

Let T be a term given as a string of length n in PNF. Let $[l, r]$ be a subinterval. The size of the interval is $s = r - l + 1$. Typically, an interval is specified by l and r and additionally a position m which is approximately the middle position. It is either can be $\lfloor l + \frac{s}{2} \rfloor$ or $\lceil l + \frac{s}{2} \rceil$. Its exact position is not decided a priori, but comes through a recursive evaluation of subintervals of $[l, r]$. We also use *interval borders* $l' = \lfloor l + \frac{s}{3} \rfloor$ and $r' = \lceil l + \frac{2s}{3} \rceil$. This divides the interval $[l, r]$ in approximately thirds. We not only consider the three intervals but also the intervals obtained by concatenating first two thirds and the second two thirds. These five intervals will be our recursion intervals. Based on those static intervals we define some dynamic intervals, i.e. intervals depending on the specific input:

- The largest closed or open subterm in $[l, r]$ that contains m. This interval is denoted as $\mathcal{M}(l, m, r) = [\mathcal{M}^1(l, m, r), \mathcal{M}^2(l, m, r)]$.
- 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 values of (l, m, r) are clear from the context, we drop them. Note that an $\mathcal{M}$ interval could correspond to an open or a closed term. The $\mathcal{N}, R$ intervals always correspond, by definition, to open terms. (That is the only way we use them in the algorithm.) The $\mathcal{L}$ interval is also defined to be open however even if we allowed it to be closed, it can only be an open term. This is because it is strictly shorter than $\mathcal{M} \cup \mathcal{N}$ and does not contain the complete second operand of O. Figure 1 shows the considered intervals. The intervals might

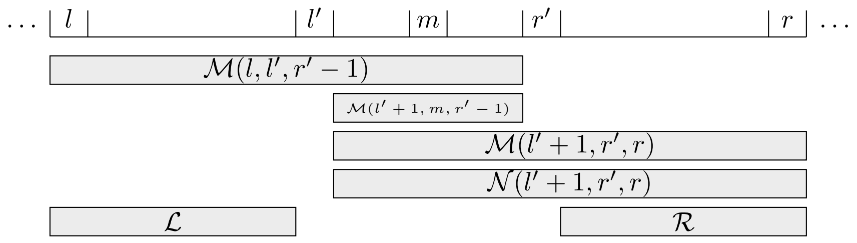
$\dots$
l
l'
m
r'
r
$\dots$
$\mathcal{M}(l, l', r'-1)$
$\mathcal{M}(l'+1, m, r'-1)$
$\mathcal{M}(l'+1, r', r)$
$\mathcal{N}(l'+1, r', r)$
$\mathcal{L}$
$\mathcal{R}$

Figure 1 The figure shows how a 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 yield recursively six values which may be used to be combined in order to get the evaluation of the $\mathcal{M}(l, m, r)$ interval.

be empty. Note that the intervals are well-defined. We now state some properties (see proofs in the full version [22]) regarding these intervals.

▶ **Lemma 15.** *Given intervals $[p_1, q_1] \subseteq [l, r]$ and $[p_2, q_2] \subseteq [l, r]$ which address closed or open terms with $[p_1, q_1] \cap [p_2, q_2] \neq \emptyset$ then $[p_1, q_1] \cup [p_2, q_2]$ is also a closed or open term.*

▶ **Lemma 16.** *It holds $\mathcal{M}^2 = \mathcal{N}^2$ and $\mathcal{M}^2(l, l', r'-1) + 1 = \mathcal{N}^1(l'+1, r', r)$.*

The key lemmas which later constitute the recursive evaluation algorithm are given below. They show how the algorithm actually composes a term by subterms coming from static subintervals. For instance, Figure 1 shows the subintervals relevant for the next lemma.

▶ **Lemma 17.** *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.

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

▶ **Lemma 18.** *Given a term T and subinterval $[l, r]$ with middle m, $\mathcal{N}$ equals one of the following intervals: 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.

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 not know anything 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 19.** *Given a term T and subinterval $[l, r]$ with middle m, there is an interval $[\bar{l}, \bar{r}] \subseteq [l, l'-1]$ with middle $\bar{m}$ such that $\mathcal{L} = \mathcal{M}(\bar{l}, \bar{m}, \bar{r})$ and this $\bar{m}$ can be computed by a binary search inside an appropriate range defined by l, m, r.*

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

3.3 The evaluation algorithm

The algorithm we present is a recursive one which is given in terms of circuits. Lemmas 17, 18, 19, and 20 directly suggest how the recursive evaluation will work: to evaluate an interval compute smaller fixed subintervals and use their evaluations to obtain the overall evaluation.

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

If it is an $\mathcal{M}$ or an $\mathcal{N}$ interval then the procedure first determines which among the five cases applies for the subsequent recursion call. By cases, we mean the five possibilities listed in Lemmas 17, 18. Moreover, if it is an $\mathcal{M}$ interval then it also determines whether the term defined by it is an open term or a closed term and keeps that information in a local flag. (Say the flag is by default set to 0 and then it is toggled to 1 if the term is closed.)

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

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

For the evaluation we need to implement circuits which on a given interval $[l, r]$ evaluate the intervals $\mathcal{M}$, $\mathcal{N}$, $\mathcal{L}$, and $\mathcal{R}$. In the case of $\mathcal{M}$ we need to distinguish whether the evaluation is a value of $\mathbb{D}$ or a function of $\widetilde{\mathbb{D}}$. In the other cases the result is always a function. Correspondingly, the following circuits may arise: EVAL($\mathcal{M}(l, m, r)$), EVAL($\mathcal{N}(l, m, r)$), EVAL($\mathcal{L}(l, m, r), \bar{l}, \bar{m}, \bar{r}$), 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 19 and 20.

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

Here we have already assumed that the term is given in the PNF form. To ensure this, as the first step we perform this conversion to the PNF form as done in [5]. The conversion is of complexity $\mathbf{TC}^0$. The resulting term is T as above. The details of the working of various recursive calls and the claimed complexity bounds are presented in the full version [22].

4 Application

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

1. **Find an algebra $\mathcal{A}$.** Given a problem Π, which could be a language or a function, find an algebra $\mathcal{A}_\Pi = (\mathbb{D}, \circledast_1, \ldots, \circledast_k)$, such that Π reduces to term evaluation over $\mathcal{A}_\Pi$.
2. **Find a coding for $\mathcal{F}(\mathcal{A}_\Pi)$.** Now we know by our main theorem that Π is in $\mathcal{F}(\mathcal{A}_\Pi)$-$\mathbf{NC}^1$. However what we want is a "real" class like $\mathbf{NC}^1$ or $\#\mathbf{SAC}^7$. Hence we encode $\mathcal{F}(\mathcal{A}_\Pi)$ in a way that we end up with a Boolean or an arithmetic class. So find a code c mapping (details in [22]) into a family of algebras, that have domains based on $\mathbb{B}$, $\mathbb{N}$, or $\mathbb{Z}$, depending on whether we wish to prove Boolean or arithmetic circuit upper bounds.
3. **Analyze the complexity of the operations used in $c(\mathcal{F}(\mathcal{A}_\Pi))$-$\mathbf{NC}^1$.** Now we know that Π is in $c(\mathcal{F}(\mathcal{A}_\Pi))$-$\mathbf{NC}^1$ since the coding admits a reduction. If we have chosen c

well, we can implement the operations of $c(\mathcal{F}(\mathcal{A}_\Pi))$ efficiently. Note that $c(\mathcal{F}(\mathcal{A}_\Pi))$ could be a family. The members of the family all contain the following coded operations:

- The operations of $\mathcal{A}_\Pi$: $\circledast_1^c \dots \circledast_k^c$.
- 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 $\mathcal{F}(\mathcal{A}_\Pi)$: $\circ^c \colon \mathbb{D}^\mathbb{D} \times \mathbb{D}^\mathbb{D} \to \mathbb{D}^\mathbb{D}$.
- The function application operation of $\mathcal{F}(\mathcal{A}_\Pi)$: $\odot^c \colon \mathbb{D}^\mathbb{D} \times \mathbb{D} \to \mathbb{D}$.

An algebra usually also has 0-ary operations, but here there is no complexity to analyze. The following is not a part of the algebra but comes into play in the construction of the $c(\mathcal{F}(\mathcal{A}_\Pi))$-$\mathbf{NC}^1$ circuits: Multiplexer operations for all subdomains $\mathbb{D}$ of $c(\mathcal{F}(\mathcal{A}_\Pi))$: $\mathrm{mp}_\mathbb{D}$. These operations are used in the $c(\mathcal{F}(\mathcal{A}_\Pi))$-$\mathbf{NC}^1$ circuit as black boxes. In this third step we have to come up with a efficient implementation of all these operations in order to derive a good upper bound. In general, the depth increases by a logarithmic factor when comparing the complexity of the functions and the overall circuit. So, if, say, all the functions are in $\#\mathbf{NC}^1_\mathbb{D}$, then $c(\mathcal{F}(\mathcal{A}_\Pi))$-$\mathbf{NC}^1 \subseteq \#\mathbf{NC}^2$.

All the applications we show follow this template. A simple application regarding arithmetic formula evaluation is presented here. Due to lack of space, the rest of the applications are omitted from this version. (See [22] for the other applications.)

4.1 Evaluating arithmetic terms and distributive algebras

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

▶ **Theorem 21** ([7]). *Evaluating terms over $\mathcal{N}$ ($\mathcal{Z}$) is in $\#\mathbf{NC}^1$ (Gap$\mathbf{NC}^1$ resp.).*

Proof. We give the proof for $\mathcal{N}$. The case of $\mathcal{Z}$ is handled similarly.

step 1. The problem is directly a term evaluation problem, hence no reduction is needed.

step 2. Consider the algebra $\mathcal{F}(\mathcal{N}) = (\{\mathbb{N}, \widetilde{\mathbb{N}}\}, +, \times, 0, 1, \overleftarrow{+}, \overleftarrow{\times}, \overrightarrow{+}, \overrightarrow{\times}, \circ, \odot)$, where $\widetilde{\mathbb{N}} \subseteq \mathbb{N}^\mathbb{N}$. We choose a coding c such that $c(\mathbb{N}) = \mathbb{N}$ and $c(\widetilde{\mathbb{N}}) = \mathbb{N}^2$. The functions in $\widetilde{\mathbb{N}}$ are of the form $x \mapsto ax + b$ for some $a, b \in \mathbb{N}$: We begin with the identity function $x \mapsto 1x + 0$ which is clearly of this form. Now we show that the operations of $\mathcal{F}(\mathcal{N})$ keep functions in this form.

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

This shows that c is indeed a valid coding.

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

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

5 Discussion

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

References

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

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

3 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. `doi:10.4230/LIPIcs.FSTTCS.2015.246`.

4 Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974. `doi:10.1145/321812.321815`.

5 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. `doi:10.1145/28395.28409`.

6 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.

7 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. `doi:10.1137/0221046`.

8 Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–21, 1985. `doi:10.1016/S0019-9958(85)80041-3`.

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

10 Patrick W. Dymond. Input-driven languages are in log n depth. *Inf. Process. Lett.*, 26(5):247–250, 1988. `doi:10.1016/0020-0190(88)90148-2`.

11 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. `doi:10.1109/FOCS.2010.21`.

12 Stephen A. Fenner, Rohit Gurjar, and Thomas Thierauf. Guest column: Parallel algorithms for perfect matching. *SIGACT News*, 48(1):102–109, 2017. `doi:10.1145/3061640.3061655`.

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

14 Moses Ganardi and Markus Lohrey. A universal tree balancing theorem. *CoRR*, abs/1704.08705, 2017. arXiv:1704.08705.

15 A. Gupta. A fast parallel algorithm for recognition of parenthesis languages. Master's thesis, University of Toronto, 1985.

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

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

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

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

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

21 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. doi:10.1007/978-3-319-42634-1_47.

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

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

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

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

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. doi:10.1007/3-540-45127-7_16.

27 Nancy A. Lynch. Log space recognition and translation of parenthesis languages. *J. ACM*, 24(4):583–590, 1977. doi:10.1145/322033.322037.

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

29 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. doi:10.1007/3-540-10003-2_89.

30 Dan I Moldovan. *Parallel processing from applications to systems*. Elsevier, 2014.

31 V. Ramachandran. Restructuring formula trees. Unpublished manuscript, 1986.

32 Walter L. Ruzzo. On uniform circuit complexity. *J. Comput. Syst. Sci.*, 22(3):365–383, 1981. `doi:10.1016/0022-0000(81)90038-6`.

33 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.

34 Howard Straubing. *Finite Automata, Formal Logic, and Circuit Complexity*. Birkhäuser, Boston, 1994.

35 Heribert Vollmer. *Introduction to Circuit Complexity - A Uniform Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 1999. `doi:10.1007/978-3-662-03927-4`.

36 Heribert Vollmer. Introduction to circuit complexity: a uniform approach, 2013.

37 Egon Wanke. k-nlc graphs and polynomial algorithms. *Discrete Applied Mathematics*, 54(2-3):251–266, 1994. `doi:10.1016/0166-218X(94)90026-4`.

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