A universal tree balancing theorem

MOSES GANARDI and MARKUS LOHREY, University of Siegen, Germany

We present a general framework for balancing expressions (terms) in form of so called tree straight-line programs. The latter can be seen as circuits over the free term algebra extended by contexts (terms with a hole) and the operations which insert terms/contexts into contexts. In [16] it was shown that one can compute for a given term of size n in logspace a tree straight-line program of depth $O(\log n)$ and size $O(n/\log n)$. In the present paper, it is shown that the conversion can be done in DLOGTIME-uniform TC^0. This allows reducing the term evaluation problem over an arbitrary algebra $\mathcal{A}$ to the term evaluation problem over a derived two-sorted algebra $\mathcal{F}(\mathcal{A})$. Three applications are presented: (i) an alternative proof for a recent result by Krebs, Limaye and Ludwig [25] on the expression evaluation problem is given, (ii) it is shown that expressions for an arbitrary (possibly non-commutative) semiring can be transformed in DLOGTIME-uniform TC^0 into equivalent circuits of logarithmic depth and size $O(n/\log n)$, and (iii) a corresponding result for regular expressions is shown.

CCS Concepts: • **Theory of computation → Circuit complexity**;

Additional Key Words and Phrases: depth reduction, tree evaluation, tree contraction

ACM Reference Format:
Moses Ganardi and Markus Lohrey. 2018. A universal tree balancing theorem. *ACM Trans. Comput. Theory* 1, 1, Article 1 (January 2018), 25 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

1.1 Depth reduction for algebraic expressions

Reducing the depth of algebraic expression by some form of tree balancing is an important algorithmic technique in the area of parallel algorithms and circuit complexity. The goal is to compute from a given tree that represents an algebraic expression an equivalent expression of logarithmic depth, which then can be evaluated in time $O(\log n)$ on a parallel computation model such as a PRAM. Equivalence of expressions usually means that the expressions evaluate to the same element in an underlying algebraic structure. For the more general case of expressions with variables, equivalence means that for all possible values of the variables, the expressions evaluate to the same element. A widely studied example in this context is the *Boolean expression balancing problem*, where the underlying algebraic structure is the Boolean algebra $(\{0,1\}, \vee, \wedge, \neg)$. Spira [33] proved that for every Boolean expression e with variables there exists an equivalent Boolean expression of depth $O(\log n)$, where n is the length of e. This also ensures that the size of the resulting expression is polynomially bounded in n. Brent [6] extended Spira's theorem to arithmetic expressions. Further improvements on the size of the balanced output expression where obtained in [5, 7].

Instead of computing an equivalent balanced expression, it is often more natural to compute an equivalent circuit (or directed acyclic graph, dag for short). A circuit can be seen as a succinct

Authors' address: Moses Ganardi, ganardi@eti.uni-siegen.de; Markus Lohrey, lohrey@eti.uni-siegen.de, University of Siegen, Lehrstuhl für theoretische Informatik, Germany.

1942-3454/2018/1-ART1 $15.00
https://doi.org/0000001.0000001

representation of a tree, where identical subtrees are represented only once. The size of a circuit is the number of nodes (or gates) and the depth of a circuit is the length of a longest path from an input gate to the output gate. Brent [6] in fact proved that a given arithmetic expression of size n can be transformed into an equivalent arithmetic circuit of depth $O(\log n)$ and size $O(n)$.[1] Note that a fan-in two circuit of depth $O(\log n)$ can be unfolded into an expression of the same depth and polynomial size.

In our recent paper [16] we developed a new approach for the construction of logarithmic depth circuits from expressions that works in two steps. The first step is purely syntactic and is motivated by the problem of *tree compression*.

1.2 Step 1: tree compression.

Here, the goal is to construct a compact representation of a given tree. In recent years, the data compression community became more and more interested in so called computation-friendly compression methods, where the compressed objects can be further processed without prior decompression. Dags are one such compression method, but they fail to compress trees with long chains. To overcome this restriction, so called *tree straight-line programs* were introduced, see [27] for a survey. Tree straight-line programs are usually defined as context-free tree grammars that generate a unique tree. They generalize context-free string grammars that generate a unique string, which are known as *straight-line programs* in the string compression community,[2] and this led to the term "tree straight-line program". In order to homogenize the definitions in this paper, we prefer an equivalent definition of tree straight-line programs in terms of circuits over an extension of the free term algebra. Let us give some definitions: Consider a fixed set Σ of ranked symbols, meaning that every symbol $f \in \Sigma$ has an associated rank (a natural number) which determines the number of children of an f-labelled node in a tree. Symbols in Σ are also called function symbols. The free term algebra over Σ consists of the set $T(\Sigma)$ of all rooted trees (or terms) over Σ. In such a tree, every node v is labelled with a symbol $f \in \Sigma$, and if the rank of f is r, then v has exactly r children that are ordered from left to right. Elements of $T(\Sigma)$ can be conveniently described as expression. For instance the tree from Figure 1 is represented by the expression $g(f(a, f(a, f(a, f(a, a)))), f(a, f(a, f(a, f(a, b)))))$. In the free term algebra over Σ one takes the set $T(\Sigma)$ as the universe and interprets every symbol $f \in \Sigma$ of rank r by the mapping $(t_1, t_2, \ldots, t_r) \mapsto f(t_1, t_2, \ldots, t_r)$. A dag for a tree $t \in T(\Sigma)$ is nothing else than a circuit over the free term algebra. To go from dags to tree straight-line programs, one has to consider a two-sorted extension $\mathcal{A}(\Sigma)$ of the free term algebra, where the two sorts are (i) the set $T(\Sigma)$ of all trees over Σ and (ii) the set $C(\Sigma)$ of all *contexts* over Σ. A context is a tree with a distinguished leaf that is labelled with a special parameter symbol $x \notin \Sigma$. This allows to do composition of a context s with another context or tree t by replacing the x-labelled leaf in s with t; the result is denoted by $s(t)$. In case t is a tree, one also speaks of substitution. The algebra $\mathcal{A}(\Sigma)$ is the extension of the free term algebra by the following additional operations:

- for all $f \in \Sigma$ of rank $r \geq 1$ and $1 \leq i \leq r$ the $(r-1)$-ary operation $\hat{f}_i : T(\Sigma)^{r-1} \to C(\Sigma)$ with $\hat{f}_i(t_1, \ldots, t_{r-1}) = f(t_1, \ldots, t_{i-1}, x, t_i, \ldots, t_{r-1})$,
- the *substitution operation* sub that maps a pair $(s, t) \in C(\Sigma) \times T(\Sigma)$ to the tree $s(t) \in T(\Sigma)$,
- the *composition operation* $\circ$ that maps a pair $(s, t) \in C(\Sigma) \times C(\Sigma)$ to the context $s(t) \in C(\Sigma)$.

A tree straight-line program is then a circuit over the structure $\mathcal{A}(\Sigma)$ that evaluates to an element of $T(\Sigma)$. Figure 1 (right) shows a tree straight-line program for the tree on the left. Note that

[1] Brent does not explicitly state this result in [6], but Bshouty, Cleve and Eberly notice in [7] that Brent proves this fact.
[2] One should be aware of the fact that the term "straight-line program" has a different meaning in algebraic complexity theory.

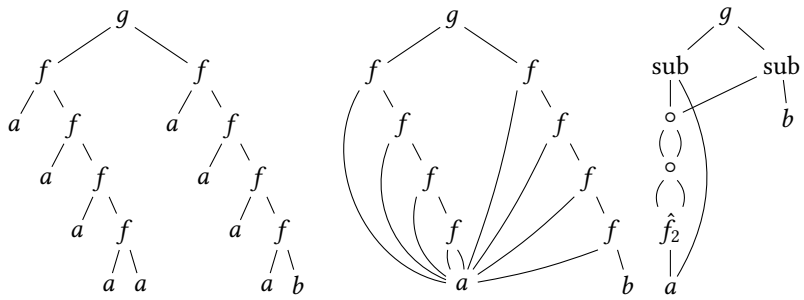

Fig. 1. A tree, its minimal dag and a tree straight-line program for the tree.

minimal dag for this tree (which is shown in the middle of the figure) is not much smaller than the original tree (only leaves can be shared). Take for instance the topmost $\circ$-labelled node in the tree straight-line program on the right. It evaluates to the context

$$\begin{aligned}\circ(\circ(\hat{f}_2(a),\hat{f}_2(a)),\circ(\hat{f}_2(a),\hat{f}_2(a))) &= \circ(\circ(f(a,x),f(a,x)),\circ(f(a,x),f(a,x)))\\ &= \circ(f(a,f(a,x)),f(a,f(a,x)))\\ &= f(a,f(a,f(a,f(a,x)))).\end{aligned}$$

This pattern appears twice in the tree on the left.

In [16], we proved that from a given tree $t \in T(\Sigma)$ of size n one can construct in logarithmic space (or, alternatively, in linear time) a tree straight-line program that evaluates to t, has depth $O(\log n)$ and size $O(n/\log n)$. We call this a *universal tree balancing result* since it is purely syntactic and does not refer to an interpretation of the symbols from Σ. In other words, it works for every interpretation of the symbols in Σ.

The size bound $O(n/\log n)$ in the above universal tree balancing theorem is important in the context of tree compression since it achieves the information theoretic lower bound. It is not hard to see that a tree straight-line program of size m can be encoded by a bit string of length $O(m \log m)$. Since $T(\Sigma)$ contains $2^{\Theta(n)}$ many trees of size n, the optimality of the $O(n/\log n)$ bound follows from a simple counting argument. The $O(\log n)$ bound on the height of the tree straight-line program is important if tree balancing is the main goal. This leads us to the second step of our tree balancing approach from [16] that we describe next.

1.3 Step 2: from tree straight-line programs to dags.

A dag can be trivially viewed as tree straight-line program (since the free term algebra is contained in the extended algebra $\mathcal{A}(\Sigma)$. Going from a tree straight-line program to a dag leads in general to an exponential blow-up: a unary tree of the form $f^n(a)$ can be represented by a tree straight-line program of size $O(\log n)$ but is incompressible with dags. The main observation for the second step in [16] is that if the algebraic structure $\mathcal{A}$ that yields the interpretation of the symbols from Σ belongs to a "nice" class, then a tree straight-line program can be transformed into a dag of the same size and depth. Moreover, this dag evaluates in $\mathcal{A}$ to the same element as the tree straight-line program. The basic idea for this is that contexts, i.e. elements of $C(\Sigma)$ cannot be evaluated to elements of $\mathcal{A}$, but they naturally evaluate to unary linear term functions on $\mathcal{A}$. For many classes of algebraic structures it is possible to represent these unary functions by tuples over $\mathcal{A}$. For example, if $\mathcal{A}$ is a (not necessarily commutative) semiring, then a unary linear term function is an affine mapping $x \mapsto axb + c$, which can be encoded by the tuple (a, b, c). For structures $\mathcal{A}$ that allow such a representation of unary linear term functions one can transform the tree straight-line program

obtained from the first step into a circuit over the structure $\mathcal{A}$ that is equivalent to the initial tree t. This transformation does not increase the size and depth (up to constant factors) and is of very low complexity; more precisely it can be accomplished in TC^0 (we always refer to the DLOGTIME-uniform variant of TC^0).

Let us mention that the idea of evaluating expressions via unary linear term functions can be also found in [29, 30], where the main goal is to develop optimal parallel circuit and term evaluation algorithms for the EREW PRAM model.

1.4 Main result

The main complexity bottleneck in the approach sketched above is the first step, i.e., the construction of the tree straight-line program from the input tree. In [16] we presented two algorithms, one working in linear time (which is the gold standard for data compression) and the other one working in logarithmic space. Logarithmic space is for many applications in circuit complexity too high. A good example is Buss' seminal result stating that the Boolean expression evaluation problem belongs to NC^1 [9]. For this result it is crucial that the Boolean expression is given as a string (for instance its preorder notation) and not as a tree in pointer representation (for the latter representation, the evaluation problem is logspace-complete). For Boolean expressions of logarithmic depth, the evaluation problem can be easily solved in logarithmic time on an alternating Turing machine with a random access tape, which shows membership in $\mathrm{ALOGTIME} = \mathrm{NC}^1$. Hence, one can obtain an alternative proof of Buss' result by showing that Boolean expressions can be balanced in NC^1.

In this paper, we achieve this goal as a corollary of our main result. We show that the first step of our balancing procedure can be carried out in TC^0 (using an algorithm different from the one in [16]). More precisely, we show that from a given expression of size n one can construct in TC^0 a tree straight-line program of depth $O(\log n)$ and size $O(n/\log n)$. The tree straight-line program is given in the *extended connection representation*, which is crucial for the applications. Our approach uses the tree contraction procedure of Abrahamson et al. [1]. Buss [10] proved that tree contraction can be implemented in NC^1. Elberfeld et al. [15] improved this result to TC^0, thereby showing that one can compute a tree decomposition of width three and logarithmic height from a given tree in TC^0.

We follow the ideas from [10, 15] but have to do several modifications, in particular in order to achieve the size bound $O(n/\log n)$. To avoid the usually very technical considerations concerning DLOGTIME-uniformity, we use the characterization of TC^0 by FOM (first-order logic with the majority quantifier). This is quite common in circuit complexity, see also [15, 21]. In a first step, we show how to define in FOM for a given expression a hierarchical decomposition into subexpressions and contexts, where the depth of the composition is logarithmic in the size of the expression. In a second step, this decomposition is then transformed into a tree straight-line program. To achieve the size bound of $O(n/\log n)$ we use a preprocessing of the tree that is based on the tree contraction approach from [19].

Our main result assumes that the input expression comes from a fixed set $T(\Sigma)$ of trees, i.e., the set Σ of ranked symbols is not part of the input. This excludes important applications, e.g., evaluation of arithmetic expressions where arbitrary integer constants may appear or depth reduction for arithmetic expressions that may contain an arbitrary number of variables. To model this setting, we can take a fixed but countably infinite set Σ of ranked symbols. It may consist for instance of the two binary ring operations and all integer constants. In order to carry over our universal balancing procedure to this setting, we have to assume that the maximal rank of the symbols in Σ is bounded. Under this assumption, we can show that from a given expression of size n that contains only ℓ different symbols from Σ one can construct in TC^0 a tree straight-line program of depth

$O(\log n)$ and size $O(n/\log_\ell n) = O(n \log \ell / \log n)$. The size bound $O(n/\log_\ell n)$ again matches the information theoretic lower bound for compression.

1.5 Applications of the main result

We present three applications of our universal TC^0-balancing result:

- We present an alternative (and hopefully simpler) proof of the main result of [25], which states that the evaluation problem for expressions over an algebra $\mathcal{A}$ can be solved in DLOGTIME-uniform $\mathcal{F}(\mathcal{A})$-NC^1. Here, $\mathcal{F}(\mathcal{A})$ is the extension of $\mathcal{A}$ by $\mathcal{A}[x]$, i.e. all linear unary term functions over $\mathcal{A}$, together with the evaluation operation $\mathcal{A}[x]\times\mathcal{A} \to \mathcal{A}$ and the composition operation $\mathcal{A}[x] \times \mathcal{A}[x] \to \mathcal{A}[x]$. The class $\mathcal{F}(\mathcal{A})$-NC^1 is defined by log-depth circuits of polynomial size over the algebra $\mathcal{F}(\mathcal{A})$ that may also contain Boolean gates (the interplay between Boolean gates and non-Boolean gates is achieved by multiplexer gates). We prove the result from [25] as follows: Using our universal balancing theorem, we transform the input expression over the algebra $\mathcal{A}$ into an equivalent expression over $\mathcal{F}(\mathcal{A})$ of logarithmic depth and polynomial size; see also Theorem 5.4. This first stage of the computation can be done in TC^0 and hence in Boolean NC^1. In a second stage we use a universal evaluator circuit for the algebra $\mathcal{F}(\mathcal{A})$ to evaluate the log-depth expression computed in the first stage. Whereas our proof is based on tree contraction, the proof in [25] is in contrast more in the spirit of Buss' Boolean formula evaluation algorithms [10] and works with so called PLNF-encoded terms that are split at certain break-points and recursively evaluated.
- We show that for any fixed semiring $\mathcal{S}$, one can transform in TC^0 an arithmetic expression of size n into an equivalent arithmetic circuit of size $O(n/\log n)$ and depth $O(\log n)$. This result is used in our recent paper [17], where we proved a dichotomy result for the expression evaluation problem for finite semirings: for every finite semiring, the expression evaluation problem is NC^1-complete or in TC^0 (precise algebraic characterizations of the corresponding semiring classes are given in [17] as well). Our TC^0-balancing procedure is used for the TC^0-part of this dichotomy. This shows that depth reduction can be also used to prove that a problem belongs to TC^0, despite the fact that the circuit after depth reduction has logarithmic depth and not constant depth.
- We show that every regular expression of size n can be transformed in TC^0 into an equivalent circuit (that uses the operators $+$, $\cdot$ and *) of size $O(n/\log n)$ and depth $O(\log n)$. This strengthens a result from [20] stating that every regular expression of size n has an equivalent regular expression of star height $O(\log n)$ (the complexity of this transformation and the total height of the resulting expression are not analyzed in [20]). In [17], we used our depth reduction result for regular expressions (as well as the above stated dichotomy for expression evaluation over finite semirings) in order to prove a dichotomy for the following intersection problem, which is parameterized by a fixed regular language L: given an ϵ-free regular expression e, does $L \cap L(e) = \emptyset$ hold? We proved that this problem is either NC^1-complete or in TC^0, depending on the regular language L.

Some of the above mentioned applications (in particular those from [17]) only use the depth bound $O(\log n)$ from our universal tree balancing result, whereas the size bound $O(n/\log n)$ could be replaced by any polynomial bound. Nevertheless, we believe that the size bound $O(n/\log n)$ is a significant feature. We mentioned already its importance in the context of compression, since it matches the information theoretic lower bound for tree compression. But also in the context of expression evaluation, the size bound $O(n/\log n)$ might be useful. Consider the scenario, where an expression of size n has to be evaluated over a ring with expensive operations (e.g. a matrix ring of large dimension). Classical tree contraction algorithms need $\Theta(n)$ arithmetic operations in total. We

can construct in parallel (TC^0) an equivalent circuit of size $O(n/\log_\ell n)$ and depth $O(\log n)$, where ℓ is the number of different ring constants that appear in the expression. This step is independent of the concrete ring. Finally, the circuit can then be evaluated in parallel time $O(\log n)$ using only $O(n/\log_\ell n)$ ring operations in total. Hence, for expression where only a small number of different ring constants appears (think about the evaluation of a multivariate noncommutative polynomial with a small number of variables), this may save some expensive ring operations.

1.6 Related work

1.6.1 Depth reduction. In this paper, we only consider depth reduction for expressions. For circuits, depth reduction becomes more difficult. A seminal result in this context was shown by Valiant, Skyum, Berkowitz and Rackoff [34]: for any commutative semiring, every circuit of size n and degree d can be transformed into an equivalent circuit of depth $O(\log n \log d)$ and size polynomial in n and d. This result led to many further investigations on depth reduction for bounded degree circuits over various classes of commutative as well as noncommutative semirings; see [2] for an excellent survey. If one drops the restriction to bounded degree circuits, then depth reduction gets even harder. For general Boolean circuits, the best known result states that every Boolean circuit of size n is equivalent to a Boolean circuit of depth $O(n/\log n)$ [31].

1.6.2 Tree compression. Tree straight-line programs of worst case size $O(n/\log n)$ have been recently used in the context of universal tree source coding [22]. Here the term "universal" has an information-theoretic meaning. Roughly speaking, it means that the gap between the average compression ratio and the normalized entropy converges to zero. Our TC^0-construction of tree straight-line programs of worst case size $O(n/\log n)$ therefore opens up a road for parallel tree compression.

Apart from the algorithms in this work and its predecessor from [16] several other so called grammar-based tree compressors can be found in the literature [28]. These algorithms compute from a given input tree a (hopefully) small tree straight-line program. In practice, the TreeRePair algorithm from [28] compresses quite well. Whereas the computation of a smallest tree straight-line program from a given input tree is not possible in polynomial time unless P = NP (the same result is already true for strings [11]), a linear time approximation algorithm has been presented in [24]. This algorithm computes from a given input tree t of size n a tree straight-line program that is only by a factor of $O(\log n)$ larger than the smallest tree straight-line program for t. No polynomial time grammar-based tree compressor with a better approximation ratio is known (again, this holds also for strings).

Recently, several generalizations of tree straight-line programs to unranked trees (i.e., trees where the number of children of a node is arbitrary and not determined by the node label) have been proposed [4, 14, 18] and worst-case upper bound of the form $O(n/\log n)$ have been shown [14, 18].

2 PRELIMINARIES

2.1 Terms over algebras

The definitions introduced in this section are standard in universal algebra (see e.g. [36, Section 4.1.1]) and term rewriting [13]. Let S be a finite set of *sorts*. An *S-sorted set* X is a family of sets $\{X_s\}_{s\in S}$; it is finite if every X_s is finite. An *S-sorted signature* Σ is a finite F-sorted set Σ of *function symbols*, where $F \subseteq (S^* \times S)$ is finite. If f has sort $(s_1 \cdots s_r, s)$, we simply write $f : s_1 \times \cdots \times s_r \to s$ and call $r \in \mathbb{N}$ the *rank* of f. An *S-sorted algebra* $\mathcal{A}$ over Σ consists of an S-sorted non-empty *domain* $A = \{A_s\}_{s\in S}$ and *operations* $f^{\mathcal{A}} : A_{s_1} \times \cdots \times A_{s_r} \to A_s$ for each function symbol $f : s_1 \times \cdots \times s_r \to s$ in Σ. Instead of S-sorted signatures and algebras we also speak of $|S|$-sorted signatures and algebras.

A one-sorted signature Σ is a *ranked alphabet* and a one-sorted algebra is simply called an *algebra*. A classical example for a two-sorted algebra is a vector space consisting of the sort of vectors and the sort of scalars.

We define the S-sorted set $T(\Sigma)$ of *terms* over Σ inductively: If $f : s_1 \times \cdots \times s_r \to s$ is a function symbol in Σ where $r \geq 0$ and $t_1, \ldots, t_r$ are terms over Σ of sorts $s_1, \ldots, s_r$, respectively, then $f(t_1, \ldots, t_r)$ is a term over Σ of sort s. A term t over an algebra $\mathcal{A}$ is a term over its signature Σ and we will also write $T(\mathcal{A})$ for $T(\Sigma)$. The *value* $t^{\mathcal{A}} \in A$ of a term $t \in T(\mathcal{A})$ is defined inductively: If $t = f(t_1, \ldots, t_r)$, then $t^{\mathcal{A}} = f^{\mathcal{A}}(t_1^{\mathcal{A}}, \ldots, t_r^{\mathcal{A}})$. If a term t is part of the input for a computational problem then we view t as a string over the finite alphabet consisting of Σ, brackets and comma.

We will also consider terms with a hole, also known as *contexts*, which we will need only for the case $|S| = 1$, i.e., a ranked alphabet Σ. Let us fix a special symbol $x \notin \Sigma$ of rank 0 (the parameter). A context is obtained from a term $t \in T(\Sigma)$ by replacing an arbitrary subterm t' by the symbol x. The set of all contexts over Σ is denoted with $C(\Sigma)$, and if $\mathcal{A}$ is an algebra over Σ, we also write $C(\mathcal{A})$ for $C(\Sigma)$. Formally, $C(\Sigma)$ is inductively defined as follows: $x \in C(\Sigma)$ and if $f \in \Sigma$ has rank r, $t_1, \ldots, t_{r-1} \in T(\Sigma)$, $s \in C(\Sigma)$ and $1 \leq i \leq r$, then $f(t_1, \ldots, t_{i-1}, s, t_i, \ldots, t_{r-1}) \in C(\Sigma)$. Given a context s and a term (resp., context) t, we can obtain a term (resp., context) $s(t)$ by replacing the unique occurrence of x in s by t.

In an algebra $\mathcal{A}$, a context $t \in C(\mathcal{A})$ defines a (unary) *linear term function* $t^{\mathcal{A}} : A \to A$ in the natural way:

- If $t = x$ then $t^{\mathcal{A}}$ is the identity function.
- If $t = f(t_1, \ldots, t_{i-1}, s, t_i, \ldots, t_{r-1})$ with $t_1, \ldots, t_{r-1} \in T(\Sigma)$, $s \in C(\Sigma)$, then for every $a \in A$: $t^{\mathcal{A}}(a) = f^{\mathcal{A}}(t_1^{\mathcal{A}}, \ldots, t_{i-1}^{\mathcal{A}}, s^{\mathcal{A}}(a), t_i^{\mathcal{A}}, \ldots, t_{r-1}^{\mathcal{A}})$.

2.2 Logical structures and graphs

We will view most objects in this paper as logical structures in order to describe computations on them by formulas of (extensions of) first-order logic in the framework of descriptive complexity [23]. A *vocabulary* τ is a tuple $(R_1, \ldots, R_k)$ of relation symbols R_i with a certain *arity* $r_i \in \mathbb{N}$. A *τ-structure* $\mathcal{G} = (V, R_1^{\mathcal{G}}, \ldots, R_k^{\mathcal{G}})$ consists of a non-empty *domain* $V = V(\mathcal{G})$ and relations $R_i^{\mathcal{G}} \subseteq V^{r_i}$ for $1 \leq i \leq k$. The relation symbols are usually identified with the relations themselves. All structures in this paper are defined over finite domains V, and the size $|V|$ is also denoted by $|\mathcal{G}|$.

A *graph* $\mathcal{G}$ is a structure of the form $\mathcal{G} = (V, (E_i)_{1 \leq i \leq k}, (P_a)_{a \in A})$, where all E_i are binary edge relations and all P_a are unary relations. The elements of A can be viewed as node labels. If $\bigcup_{i=1}^k E_i$ is acyclic, $\mathcal{G}$ is called a *dag*. A graph $\mathcal{G}$ is *k-ordered* if for all $u \in V$ and all $1 \leq i \leq k$ there exists at most one $v \in V$ with $(u, v) \in E_i$. If $(u, v) \in E_i$ exists, we call v the *i-th successor* of u. A *tree* $\mathcal{T} = (V, (E_i)_{1 \leq i \leq k}, (P_a)_{a \in A})$ is a graph such that $(V, \bigcup_{i=1}^k E_i)$ is a rooted tree in the usual sense and the edge relations E_i are pairwise disjoint. We write $u \preceq_{\mathcal{T}} v$ if u is an ancestor of v in the rooted tree $(V, \bigcup_i E_i)$. The *depth-first (left-to-right) order* on V defines v to be smaller than w if and only if v is an ancestor of w or there exists a node u and numbers $1 \leq i < j \leq k$ such that the i-th child of u is ancestor of v and the j-th child of u is ancestor of w.

2.3 Circuits

We also make use of a more succinct representation of terms as defined in Section 2.1, namely as circuits. Let Σ be an S-sorted signature with maximal rank k. A *circuit* over Σ is a k-ordered dag C whose nodes are called *gates*. The set of gates V is implicitly S-sorted. Each gate v is labelled with a function symbol $f : s_1 \times \cdots \times s_r \to s$ from Σ such that v has sort s and exactly r successors where the i-th successor of A has sort s_i for all $1 \leq i \leq r$. Furthermore C has a distinguished *output gate* $v_{\text{out}} \in V$ (labelled by some special symbol). The *depth* of C is the maximal path length in C.

The *value* of C over an S-sorted algebra $\mathcal{A}$ over Σ is defined naturally: One evaluates all gates of C bottom-up: If all successor gates of a gate v are evaluated then one can evaluate v. Finally, the value of C is the value of the output variable v_{out}.

In a slightly more general definition, we also allow *copy gates* to simplify certain constructions. A copy gate v (labelled by a special symbol) has exactly one successor w and the value of v is defined as the value of w.

2.4 Circuit complexity and descriptive complexity

In Section 2.3 we considered circuits as a succinct representation of input terms. In this section we will use Boolean circuits also as a computational model, which can process circuits over an arbitrary signature as described in Section 2.3.

We use standard definitions from circuit complexity, see e.g. [35]. The main complexity class used in this paper is DLOGTIME-uniform TC^0, which is the class of languages $L \subseteq \{0,1\}^*$ recognized by DLOGTIME-uniform circuit families of polynomial size and constant depth with not-gates and threshold gates of unbounded fan-in. If instead of general threshold gates only and-gates and or-gates (again of unbounded fan-in) are allowed, one obtains DLOGTIME-uniform AC^0. Analogously, one defines AC^0- and TC^0-computable functions $f : \{0,1\}^* \to \{0,1\}^*$ where the circuit outputs a bit string instead of a single bit. The definition of DLOGTIME-uniformity can be found in [3]. The precise definition is not needed in this paper.

Instead of working with DLOGTIME-uniform circuit families, we will use equivalent concepts from descriptive complexity based on the logics FO (first-order logic) and FOM (first-order logic with the majority quantifier) [23]. In this setting we assume that the domain of a structure has the form $\{1,\dots,n\}$. Furthermore, the vocabulary implicitly contains the binary relations $<$ and BIT, where $<$ is always interpreted as the natural linear order on $\{1,\dots,n\}$ and $\mathrm{BIT}(i,j)$ is true iff the j-th bit of i is 1. We will not explicitly list these relations when defining structures. The relations $<$ and BIT allow to access the bits of elements of the domain and to do arithmetic manipulation with these elements. In particular, addition and multiplication on the numbers $\{1,\dots,n\}$ are FO-definable using $<$ and BIT [23, Theorem 1.17]. Furthermore, if Σ is a finite alphabet, in a structure $\mathcal{G}$ of size n we can quantify over sequences $a_1 \cdots a_s \in \Sigma^*$ of length $s = O(\log n)$ by identifying such sequences by numbers of size $n^{O(1)}$, or tuples over $V(\mathcal{G})$ of constant length. Using a suitable encoding, the BIT-predicate allows us to access each symbol a_i in a FO-formula.

An *FO-computable function* (or *FO-query*) maps a structure $\mathcal{G}$ over some vocabulary to a structure $I(\mathcal{G})$ over a possibly different vocabulary which is definable in $\mathcal{G}$ by a *d-dimensional interpretation* I using first-order formulas. That means, the domain $V(I(\mathcal{G}))$ is an FO-definable subset of $V(\mathcal{G})^d$ and each r-ary relation in $I(\mathcal{G})$ is an FO-definable subset of $V(\mathcal{G})^{d\cdot r}$; for precise definitions we refer the reader to [23].

If we additionally allow a majority quantifier in the formulas, we obtain *FOM-computable* functions. Roughly speaking, FOM-logic is the extension of FO-logic with the ability to count. Note that the size of $I(\mathcal{G})$ is polynomially bounded in the size of $\mathcal{G}$.

Notice that, formally one also needs to logically define a linear order $<$ and the BIT-predicate on the output structure $I(\mathcal{G})$. For $<$ one can always use the lexicographical order on $V(I(\mathcal{G})) \subseteq V(\mathcal{G})^d$ whereas the BIT-predicate might not be definable in first-order logic, cf. [23, Remark 1.32]. For example, BIT is FO-definable if the domain formula is valid, i.e. $V(I(\mathcal{G})) = V(\mathcal{G})^d$ for all $\mathcal{G}$. Furthermore, since the BIT-predicate is already FOM-definable from $<$, this technicality vanishes for FOM-computable functions [3, Theorem 11.2].

The connection between descriptive complexity and circuit complexity is drawn as follows. A non-empty word $a_1 \cdots a_n \in \{0,1\}^+$ can be viewed as a *word structure* $(\{1,\dots,n\}, S)$ where the

unary relation S contains those positions i where $a_i = 1$. A structure $\mathcal{G}$ can be encoded by a bit string $\mathrm{bin}(\mathcal{G}) \in \{0,1\}^*$ in such a way that the conversions between $\mathcal{G}$ and the word structure of $\mathrm{bin}(\mathcal{G})$ are FO-computable [23].

It is known that a function $f : \{0,1\}^+ \to \{0,1\}^+$ is FO-computable (respectively, FOM-computable) if and only if it is computable in DLOGTIME-uniform AC^0 (respectively, DLOGTIME-uniform TC^0). Hence we can describe AC^0- and TC^0-computations on the binary encoding of a structure by logical formulas on the structure itself.

3 REPRESENTATIONS FOR TREES AND DAGS

It is known that the circuit complexity of algorithmic problems for trees highly depends on the representation of the trees. For example, for trees given in the standard pointer representation, reachability is complete for deterministic logarithmic space [12]. In the *ancestor representation*, which is the extension of a tree $\mathcal{T}$ by its ancestor relation $\preceq_{\mathcal{T}}$, queries like reachability, least common ancestors and the depth-first order become first-order definable. Note that a term $t \in T(\Sigma)$ can be represented by a k-ordered tree $\mathcal{T} = (V, (E_i)_{1 \le i \le k}, (P_a)_{a \in \Sigma})$ where k is the maximal rank of a symbol in Σ. Furthermore, each node has a unique label which determines the number of its children, i.e. $\mathcal{T}$ is a *ranked tree*.

Lemma 3.1 ([15, Lemma 4.1]). *There is an FOM-computable function which transforms a given term t (viewed as a string with opening and closing parentheses) into the corresponding labelled ordered tree $\mathcal{T}$ in ancestor representation, and vice versa.*

For ordered dags of logarithmic depth, we propose a representation scheme which allows us to access paths of logarithmic length. It is similar to the *extended connection languages* of circuit families in the context of uniform circuit complexity [32].

A path in a k-ordered graph $\mathcal{G}$ can be specified by its start node and a so called *address string* over $\{1, \dots, k\}$. Formally, for a string $\rho \in \{1, \dots, k\}^*$ and a node $u \in V(\mathcal{G})$ we define the node $\rho(u)$ (it may be undefined) inductively as follows: If $\rho = \varepsilon$ then $\rho(u) = u$. Now assume that $\rho = \pi \cdot d$ with $d \in \{1, \dots, k\}$ and the node $v = \pi(u)$ is defined. Then $\rho(u)$ is the d-th successor of v, if it is defined, otherwise $\rho(u)$ is undefined. The *extended connection representation*, briefly *EC-representation*, of $\mathcal{G}$, denoted by $\mathrm{ec}(\mathcal{G})$, is the extension of $\mathcal{G}$ by the relation consisting of all so called *EC-tuples* (u, ρ, v) where $u, v \in V(\mathcal{G})$ and $\rho \in \{1, \dots, k\}^*$ is an address string of length at most $\log_k |\mathcal{G}| - 1$ such that $\rho(u) = v$. Note that there are at most $|\mathcal{G}|$ many such address strings, which therefore can be identified with numbers from 1 to $|\mathcal{G}|$. Hence, we can view the set of EC-tuples as a ternary relation over $V(\mathcal{G})$. As remarked above, we can access any position of the address string in a first-order formula using the BIT-predicate. For trees we have:

Lemma 3.2. *There is an FOM-computable function which converts the ancestor representation of a k-ordered tree $\mathcal{T}$ into its EC-representation* $\mathrm{ec}(\mathcal{T})$.

Proof. Let u, v be nodes and $\rho = d_1 \cdots d_{s-1} \in \{1, \dots, k\}^*$ be an address string of logarithmic length. Then (u, ρ, v) is an EC-tuple of $\mathcal{T}$ if and only if for all $1 \le i \le s-1$ there exist nodes v_i, v_{i+1} (which must be unique) such that

- v_{i+1} is the d_i-th successor of v_i,
- $|\{w \in V \mid u \preceq w \preceq v_i\}| = i$, and
- $|\{w \in V \mid v_{i+1} \preceq w \preceq v\}| = s - i$,

which is FOM-definable using the ancestor relation on $\mathcal{T}$. □

Let $\mathcal{G}$ be a k-ordered dag and v_0 be a node in $\mathcal{G}$. The *unfolding* of $\mathcal{G}$ from v_0, denoted by $\mathrm{unfold}(\mathcal{G}, v_0)$, is defined as follows: Its node set is the (finite) set of paths $(v_0, v_1, \dots, v_n) \in V(\mathcal{G})^+$

starting at v_0. If v_{n+1} is the i-th successor of v_n in $\mathcal{G}$, then $(v_0, v_1, \ldots, v_{n+1})$ is the i-th successor of $(v_0, v_1, \ldots, v_n)$ in the unfolding. The labels of a node $(v_0, v_1, \ldots, v_n)$ in the unfolding are the labels of v_n in $\mathcal{G}$. Note that the size of $\mathrm{unfold}(\mathcal{G}, v_0)$ can be exponential in the depth of $\mathcal{G}$.

Lemma 3.3. *For any $c > 0$ there exists an FOM-computable function which, given a k-ordered dag $\mathcal{G}$ of size n and depth $\leq c \cdot \log n$ in EC-representation, and a node v_0, outputs the ancestor representation of the tree* $\mathrm{unfold}(\mathcal{G}, v_0)$.

Proof. A node in the unfolding is an address string $\rho \in \{1, \ldots, k\}^*$ of length $\leq c \cdot \log |G|$, such that $\rho(v_0)$ exists. By an FO-formula one can test whether $\rho(v_0)$ exists and also compute this node. The i-th successor of an address string ρ is the address string ρi. The ancestor relation is the prefix relation on the set of address strings. □

In combination with Lemma 3.1 this yields:

Lemma 3.4. *For any $c > 0$ there exists an FOM-computable function which, given a circuit C of size n and depth $\leq c \cdot \log n$ in EC-representation, outputs an equivalent term t.*

Vice versa, one can compact an ordered tree $\mathcal{T}$ to its *minimal dag* $\mathrm{dag}(\mathcal{T})$. It is the up to isomorphism unique smallest dag $\mathcal{G}$ such that $\mathcal{T}$ is isomorphic to $\mathrm{unfold}(\mathcal{G}, v)$ for some v. One can identify the nodes of $\mathrm{dag}(\mathcal{T})$ with the isomorphism classes of the subtrees of $\mathcal{T}$.

Lemma 3.5. *There exists an FOM-computable function which maps a k-ordered tree $\mathcal{T}$ in ancestor representation to* $\mathrm{dag}(\mathcal{T})$ *in EC-representation.*

Proof. Using Lemma 3.2 we convert the ancestor representation of $\mathcal{T}$ into its EC-representation. With the help of the depth-first order on $V(\mathcal{T})$, it is FOM-definable whether the subtrees rooted in two given nodes u, v are isomorphic. For a node $v \in V(\mathcal{T})$ let $\min(v)$ be the first node (with respect to the built-in order on $V(\mathcal{T})$) such that the subtrees below v and $\min(v)$ are isomorphic. The mapping min is also FOM-definable. Then the node set of $\mathrm{dag}(\mathcal{T})$ can be identified with $V' = \{\min(v) \mid v \in V(\mathcal{T})\}$. A pair $(u', v') \in V' \times V'$ belongs to $E_i^{\mathrm{dag}(\mathcal{T})}$ if there exists $u \in V(\mathcal{T})$ such that $(u', u) \in E_i^{\mathcal{T}}$ and $v' = \min(u)$. The set of EC-tuples of $\mathrm{dag}(\mathcal{T})$ is the set of tuples (u', ρ, v') such that there exists an EC-tuple (u, ρ, v) of $\mathcal{T}$ with $\min(u) = u'$ and $\min(v) = v'$. □

For an arbitrary FOM-computable function I on k-ordered graphs, it is not clear whether the function $\mathrm{ec}(\mathcal{G}) \mapsto \mathrm{ec}(I(\mathcal{G}))$ is FOM-computable as well. On the other hand, this is possible for so called *guarded transductions*. A (*m-dimensional*) *connector* has the form $\gamma : \{1, \ldots, m\} \to \{1, \ldots, k, =\} \times \{1, \ldots, m\}$. Given a k-ordered graph $\mathcal{G}$ and two tuples $\overline{u} = (u_1, \ldots, u_m), \overline{v} = (v_1, \ldots, v_m) \in V(G)^m$, we say that the connector γ *connects* $\overline{u}$ to $\overline{v}$ if for all $1 \leq j \leq m$ the following holds:

- If $\gamma(j) = (d, i)$ for some $1 \leq d \leq k$, then $(u_i, v_j) \in E_d^{\mathcal{G}}$.
- If $\gamma(j) = (=, i)$, then $u_i = v_j$.

Notice that $\overline{u}$ and γ uniquely determine $\overline{v}$. Also note that if k and m are constants (as in the lemma below), then a connector can be specified with $O(1)$ many bits. Hence, a sequence of connectors of length $O(\log |\mathcal{G}|)$ needs $O(\log |\mathcal{G}|)$ bits and can be identified with a tuple over $V(\mathcal{G})$ of fixed length.

Lemma 3.6. *Let k and m be constants. Given a k-ordered graph $\mathcal{G}$ in EC-representation, tuples $\overline{u}, \overline{v} \in V(\mathcal{G})^m$ and a sequence $\gamma^{(1)} \cdots \gamma^{(s)}$ of connectors of length $s = O(\log |\mathcal{G}|)$, it is FO-definable whether there exists a (necessarily unique) sequence of tuples $\overline{v}^{(1)}, \ldots, \overline{v}^{(s+1)} \in V(\mathcal{G})^m$ such that $\overline{v}^{(1)} = \overline{u}$, $\overline{v}^{(s+1)} = \overline{v}$, and $\gamma^{(i)}$ connects $\overline{v}^{(i)}$ to $\overline{v}^{(i+1)}$ for all $1 \leq i \leq s$. If so, the tuple sequence is FO-computable in the sense that the $(m+1)$-ary relation $R = \{(t, \overline{v}^{(t)}) \mid 1 \leq t \leq s+1\}$ is FO-computable.*

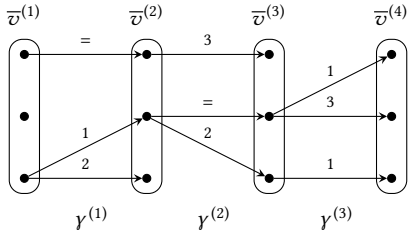

Fig. 2. If I is a guarded transduction, a path in $I(\mathcal{G})$ describes a sequence of connectors.

Proof. Let $\overline{u} = (u_1, \ldots, u_m)$. The FO-formula says that for all $2 \leq t \leq s+1$ there exists a tuple $\overline{w} = (w_1, \ldots w_m) \in V(\mathcal{G})^m$ such that for all $1 \leq j \leq m$ there is a sequence $(j_1, d_1) \cdots (j_{t-1}, d_{t-1}) \in (\{1, \ldots, m\} \times \{1, \ldots, k, =\})^*$ (which is necessarily unique) with the following properties:

- $\gamma^{(i)}(j_{i+1}) = (d_i, j_i)$ for all $1 \leq i \leq t-2$, $\gamma^{(t-1)}(j) = (d_{t-1}, j_{t-1})$ and
- (u_{j_1}, π_t, w_j) is an EC-tuple of $\mathcal{G}$, where the address string $\pi_t \in \{1, \ldots, k\}^*$ is the projection of $d_1 \cdots d_{t-1}$ to the subalphabet $\{1, \ldots, k\}$.

Moreover, in case $t = s+1$ we must have $\overline{w} = \overline{v}$. The relation R from the lemma contains all tuples $(t, \overline{w})$ and $(1, \overline{u})$. □

A *graph transduction* I computes from a k-ordered graph $\mathcal{G}$ a k'-ordered graph $I(\mathcal{G})$ whose node set is a subset of $V(\mathcal{G})^m \times \{1, \ldots, c\}$ for some constants m, c (we can assume that $\{1, \ldots, c\} \subseteq V(\mathcal{G})$). A graph transduction I is *guarded* if for every k-ordered graph $\mathcal{G}$ and every edge $((\overline{u}, a), (\overline{v}, b))$ in $I(\mathcal{G})$ there exists a connector γ which connects $\overline{u}$ to $\overline{v}$. The idea is that for a given path $(\overline{v}^{(1)}, a^{(1)})(\overline{v}^{(2)}, a^{(2)}) \cdots (\overline{v}^{(s)}, a^{(s)})$ in $I(\mathcal{G})$, the connectors $\gamma^{(1)}\gamma^{(2)} \cdots \gamma^{(s-1)}$ describe a forest of paths in $\mathcal{G}$ with its roots in $\overline{v}^{(1)}$, see Figure 2. Based on this forest we can construct the EC-tuples of $I(\mathcal{G})$ from the EC-tuples of $\mathcal{G}$.

Lemma 3.7. *For every FOM-computable guarded graph transduction I there exists an FOM-computable function mapping* $\mathrm{ec}(\mathcal{G})$ *to* $\mathrm{ec}(I(\mathcal{G}))$ *for all k-ordered graphs $\mathcal{G}$.*

Proof. It suffices to compute the EC-tuples of $I(\mathcal{G})$. Assume that the the output vocabulary has k' edge relations $E_1, \ldots, E_{k'}$. Let $(\overline{u}, a)$ and $(\overline{v}, b)$ be two nodes in $I(\mathcal{G})$ and $\rho' = d'_1 \cdots d'_s \in \{1, \ldots, k'\}^*$ be an address string of length at most $\log_{k'} |I(\mathcal{G})| - 1$. We claim that one can express by an FO-formula whether $((\overline{u}, a), \rho', (\overline{v}, b))$ is an EC-tuple of $I(\mathcal{G})$. This is the case if and only if there exist nodes $(\overline{v}^{(1)}, a^{(1)}), \ldots, (\overline{v}^{(s+1)}, a^{(s+1)})$ in $I(\mathcal{G})$ such that $(\overline{v}^{(1)}, a^{(1)}) = (\overline{u}, a)$, $(\overline{v}^{(s+1)}, a^{(s+1)}) = (\overline{v}, b)$, and

$$((\overline{v}^{(t)}, a^{(t)}), (\overline{v}^{(t+1)}, a^{(t+1)})) \in E^{I(\mathcal{G})}_{d'_t} \tag{1}$$

for all $1 \leq t \leq s$. Since $I(\mathcal{G})$ is k'-ordered, these nodes must be unique.

Our FO-formula says that there exists a sequence $a^{(1)} \cdots a^{(s+1)} \in \{1, \ldots, c\}^{s+1}$ and connectors $\gamma^{(1)}, \ldots, \gamma^{(s)}$ such that there exists a (unique) sequence of tuples $\overline{v}^{(1)}, \ldots, \overline{v}^{(s+1)} \in V(\mathcal{G})^m$ with $\overline{v}^{(1)} = \overline{u}$, $\overline{v}^{(s+1)} = \overline{v}$, and $\gamma^{(i)}$ connects $\overline{v}^{(i)}$ to $\overline{v}^{(i+1)}$ for all $1 \leq i \leq s$. The existence of the sequence $\overline{v}^{(1)}, \ldots, \overline{v}^{(s+1)}$ is expressed using Lemma 3.6. Moreover, if this sequence exists we can also express whether (1) holds for all $1 \leq t \leq s$ using the FO-computable relation R from Lemma 3.6. □

Let us remark that in this paper we only need graph transductions where $m = 1$. The more general definition will be used in a forthcoming paper. Finally, we will need the following lemma:

Lemma 3.8. *For any $c > 0$ there exists an FOM-computable function which maps a circuit with copy gates of size n and depth $\leq c \cdot \log n$ to an equivalent circuit without copy gates with the same depth bound, where both circuits are given in EC-representation.*

Proof. Let C be a circuit with copy gates. Let E be the binary relation consisting of all pairs (A, B), where A is a copy gate and B is the unique successor of A. For each copy gate $A \in V$ we define the first non-copy gate on the unique E-path starting in A, which is first-order definable using the EC-representation. By contracting all such paths we can define on all non-copy gates of C an equivalent circuit C' without copy-gates.

The EC-tuples of C' can also be defined in FOM: Let A and B be non-copy gates and $\rho' \in \{1, \dots, k\}^*$ be a string of length at most $\log_k |C'| - 1$ where k is the maximal rank of a function symbol in $\mathcal{B}$. Then (A, ρ', B) is an EC-tuple in C' if and only if there exists an EC-tuple (A, ρ, B) in C such that ρ' is obtained from ρ by omitting those symbols which describe an edge to a non-copy gate on the path (A, ρ, B). Formally, we guess a "bit mask" $z \in \{0, 1\}^*$ with $|z| = |\rho|$ using a single existential quantifier and test whether ρ' is obtained from ρ by removing the positions marked with a 0-bit in z. Then, for each non-empty prefix π of ρ we test whether $\pi(A)$ is a non-copy gate if and only if z has a 1-bit at position $|\pi|$. □

4 HIERARCHICAL TREE DEFINITIONS

In the following we will show how to construct a hierarchical definition of a given tree which has logarithmic depth. Throughout this section all trees are implicitly given in ancestor representation. The idea is to decompose a tree in a well-nested way into (i) subtrees, (ii) contexts (trees with a hole) and (iii) single nodes. From such a decomposition, it is easy to derive a tree straight-line program; this will be done in Section 5. The advantage of hierarchical decompositions over tree straight-line programs is that the former perfectly fit into the descriptive complexity framework: There is a natural representation of a hierarchical decomposition by two relations – a unary one and a binary one – on the node set of the tree.

A *pattern* p in a k-ordered tree $\mathcal{T}$ is either a single node $v \in V(\mathcal{T})$, called a *subtree pattern*, or a pair of nodes $(v, w) \in V(\mathcal{T})^2$, called a *context pattern*, such that w is a proper descendant of v. A subtree pattern v *covers* all descendants of v (including v), whereas a context pattern (v, w) *covers* all descendants of v which are not descendants of w. The set of nodes covered by a pattern p is denoted by $V[p]$ and $\mathcal{T}[p]$ is the subtree of $\mathcal{T}$ induced by $V[p]$. The *root* of p is the root of $\mathcal{T}[p]$ and its *size* is $|\mathcal{T}[p]|$. We call q a *subpattern* of p, denoted by $q \leq p$, if $V[q] \subseteq V[p]$, which partially orders the set of all patterns in a tree. Note that the root of $\mathcal{T}$ is the largest pattern with respect to $\leq$. Two patterns p, q are *disjoint* if $V[p] \cap V[q] = \emptyset$. A set P of patterns in $\mathcal{T}$ is a *hierarchical definition* of $\mathcal{T}$ if

- P contains the largest pattern (the root of $\mathcal{T}$), and
- P is *well-nested*, i.e. any two patterns $p, q \in P$ are disjoint or comparable ($p \leq q$ or $q \leq p$).

The pair $(\mathcal{T}, P)$ is also called a hierarchical definition, which is formally represented as the logical structure $(\mathcal{T}, P \cap V(\mathcal{T}), P \cap V(\mathcal{T})^2)$.

One can view a hierarchical definition itself as a tree where the patterns are its nodes. We say that $q \in P$ is a *direct subpattern* of $p \in P$ in P, denoted by $q \lessdot p$, if $q < p$ and there exists no $r \in P$ with $q < r < p$. The *pattern tree* of P is the tree with node set P where the children of a pattern are its direct subpatterns, ordered by the depth-first order on their roots. The height of the pattern tree is the *depth* of P, denoted by $\mathrm{depth}(P)$. Furthermore, each pattern p in the pattern tree is annotated by its *branching tree*, which is defined as follows: The *boundary* ∂p of p is $\partial p = V[p] \setminus \bigcup_{q \lessdot p} V[q]$, i.e. the set of nodes covered by p but not by any of its (direct) subpatterns. The *branching tree* of a pattern $p \in P$ is obtained from $\mathcal{T}[p]$ by contracting the direct subpatterns to single nodes

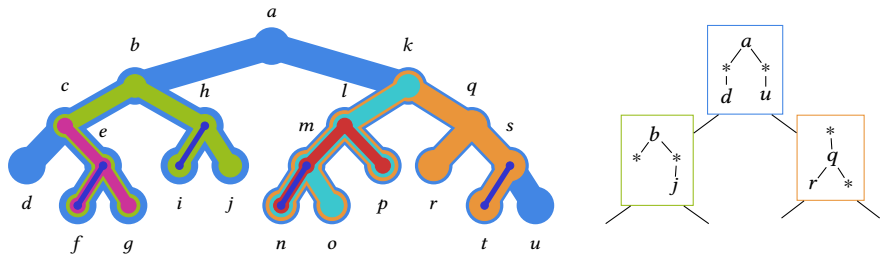

Fig. 3. A hierarchical definition with its pattern tree. Each pattern in the pattern tree is labelled by its branching tree. The symbol $*$ represents a direct subpattern.

labelled by a special symbol (in order to distinguish them from boundary nodes), i.e. its node set is $\partial p \cup \{q \mid q \lessdot p\}$. The *width* of P is the maximal size of a branching tree of a pattern $p \in P$, denoted by $\mathrm{width}(P)$.

Example 4.1. *Figure 3 shows an example of a hierarchical definition P and a top part of its pattern tree, which has height 4. The largest pattern in blue has two direct subpatterns (in green and orange) and its boundary is $\{a, d, u\}$, which are the nodes that are not covered by the green or the orange pattern. Hence the branching tree of the largest pattern has size 5 (2 subpatterns plus 3 boundary nodes), which is also the width of P.*

4.1 Hierarchical definitions via tree contraction

This section is the core of the paper. Using the tree construction technique of Abrahamson et al. [1] we construct a hierarchical definition for a given binary tree. Here, a binary tree is a 2-ordered tree $\mathcal{T} = (V, E_1, E_2, (P_a)_{a\in A})$ where every node $u \in V$ is either a leaf (i.e., there is no v with $(u, v) \in E_1 \cup E_2$) or has a left and a right child (i.e., there exist $v_1, v_2 \in V$ with $(u, v_1) \in E_1$ and $(u, v_2) \in E_2$). Such trees are also called full binary trees.

The unary relations P_a that define the node labels are not important in this section and can be completely ignored; the number of node labels will be relevant only in Section 4.2.

Let $\mathcal{T}$ be a binary tree with at least two leaves. The basic operation of tree contraction is called *prune-and-bypass*. Let w be a leaf node, v its parent node and u be the parent node of v. Applying the prune-and-bypass operation to w means: both v and w are removed and the sibling w' of w becomes a new child of u. We say that the edges (u, v), (v, w) and (v, w') are *involved* in this prune-and-bypass step. In our definition the operation can only be applied to leaves of depth at least 2 so that the root is never removed.

To verify the correctness of our (parallel) tree contraction algorithm, we first present a sequential tree-contraction algorithm. A pattern p in $\mathcal{T}$ is *hidden* in a context pattern (u, v) if p is a subpattern of (u, v) but does not cover u. Starting with $P_0 = \emptyset$ and $\mathcal{T}_0 = \mathcal{T}$, we maintain the following invariants: (1) each pattern $p \in P_i$ is hidden in some edge of $\mathcal{T}_i$ (interpreted as patterns in $\mathcal{T}$) and (2) P_i is well-nested. We obtain $\mathcal{T}_{i+1}$ from $\mathcal{T}_i$ by pruning-and-bypassing an arbitrary leaf node w in $\mathcal{T}_i$ (of depth at least 2). Let u be the grandparent node of w and w' be the sibling of w in $\mathcal{T}_i$. The *contraction pattern* p formed in this prune-and-bypass step is the maximal subpattern p which is hidden in (u, w'). It is the pattern $p = (u', w')$ where u' is the child of u that belongs to the path in $\mathcal{T}$ from u down to w. We add p to P_i to obtain P_{i+1}.

Clearly, property (1) is preserved because the edge (u, w') is introduced in $\mathcal{T}_{i+1}$ and all patterns which are hidden in some involved edge in $\mathcal{T}_i$ are hidden in the edge (u, w') in $\mathcal{T}_{i+1}$. By property (1) every pattern $q \in P_i$ which intersects the contraction pattern p is hidden in one of the three

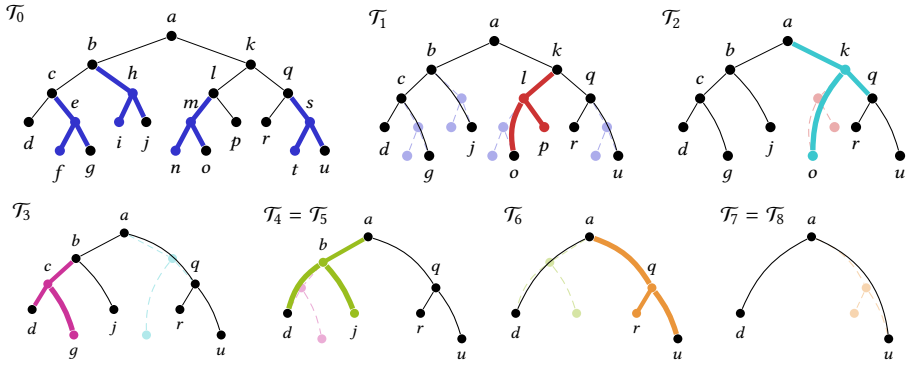

Fig. 4. Example for the tree contraction algorithm. Alternatingly, left and right internal leaves are pruned-and-bypassed. The contraction patterns introduced in $\mathcal{T}_i$ are hidden in the edges of $\mathcal{T}_{i+1}$, e.g. the pattern (m, o) introduced in $\mathcal{T}_0$ is hidden in the edge (l, o) in $\mathcal{T}_1$.

edges involved in the prune-and-bypass operation, therefore q is a subpattern of p. This proves that P_{i+1} is indeed well-nested. By adding the largest pattern in $\mathcal{T}$ to any set P_i, we clearly obtain a hierarchical definition for $\mathcal{T}$.

Now we proceed with the parallel tree-contraction algorithm. Notice that we can apply the prune-and-bypass operation to a set of leaves in parallel if no edge is involved in more than one prune-and-bypass operation. We apply the prune-and-bypass operation only to *internal* leaves, i.e. leaves which are not the left- or the right-most leaf in the tree. This implies that leaves which are children of the root node are not pruned, i.e., every pruned leaf has a grandparent as required above.[3] Let $\mathcal{T}_0$ be the input tree $\mathcal{T}$ with n internal leaves and hence $n + 2$ leaves and $2(n + 2) - 1$ nodes. We label the internal leaves by the numbers $1, \ldots, n$ from left to right. It may be helpful for the reader to think of the leaf numbers in their binary encodings. We construct a sequence of trees $\mathcal{T}_0, \ldots, \mathcal{T}_m$ as follows.

- If $\mathcal{T}_{2i}$ has two leaves, the algorithm terminates.
- If $\mathcal{T}_{2i}$ has at least one internal leaf, we prune-and-bypass all internal leaves in $\mathcal{T}_{2i}$ with an odd number that are left children to obtain the tree $\mathcal{T}_{2i+1}$. Then we prune-and-bypass all internal leaves in $\mathcal{T}_{2i+1}$ with an odd number that are right children and relabel the remaining internal leaves (divide leaf number by 2) to obtain the tree $\mathcal{T}_{2i+2}$.

Notice that $\mathcal{T}_{2i}$ contains exactly those internal leaves whose number in $\mathcal{T}_0$ is divided by 2^i. Hence the algorithm terminates after $m = 2(\lfloor \log_2 n \rfloor + 1)$ rounds. In Figure 4 we illustrate the tree contraction algorithm. The leaves which are pruned and bypassed are colored together with their parent nodes and the involved edges.

Lemma 4.2. *There is an FOM-computable function which maps a binary tree $\mathcal{T}$ and a number $0 \leq i \leq m$ to $\mathcal{T}_i$.*

Proof. Similar proofs are given in [10, 15]. The main observation is that the least common ancestor of two nodes in $\mathcal{T}_i$ is the same as their least common ancestor in $\mathcal{T}_0$. Therefore it suffices to compute the set of leaves of $\mathcal{T}_i$, which directly also yields the inner nodes as the least common ancestors of any two leaves.

[3] Elberfeld et al. [15] enforce this by adding at the very beginning a fresh root with a fresh leaf as its left child, and the original tree as its right subtree. Here, we want to avoid adding new nodes to the tree.

First, the number m is FOM-definable using the BIT-predicate ($\lfloor \log n \rfloor + 1$ is the largest number i with $\mathrm{BIT}(n, i) = 1$). The leaves of the tree $\mathcal{T}_i$ are the left- and rightmost leaf of $\mathcal{T}_0$, together with the internal leaves. The internal leaves of a tree $\mathcal{T}_{2i}$ are the internal leaves of $\mathcal{T}_0$ whose number in $\mathcal{T}_0$ is divided by 2^i. The internal leaves of $\mathcal{T}_{2i+1}$ are the internal leaves of $\mathcal{T}_{2i+2}$ and all internal leaves of $\mathcal{T}_{2i}$ which are right children. □

As in the sequential tree contraction algorithm we obtain a hierarchical definition by taking the set of all contraction patterns which are formed in every prune-and-bypass operation together with the largest pattern. We call this hierarchical definition $\mathrm{CP}(\mathcal{T})$. Figure 3 shows the hierarchical definition obtained from the example in Figure 4. The blue pattern is the largest pattern (the subtree pattern a).

Explicitly written down, we have

$$\mathrm{CP}(\mathcal{T}) = \{a, (e, g), (h, j), (m, o), (s, u), (l, o), (k, q), (c, d), (b, d), (q, u)\}.$$

Proposition 4.3. *There is an FOM-computable function which maps a binary tree $\mathcal{T}$ to $(\mathcal{T}, \mathrm{CP}(\mathcal{T}))$, which is a hierarchical definition of depth $O(\log n)$ and width at most 5.*

Proof. The FOM-definition of $\mathrm{CP}(\mathcal{T})$ follows easily from Lemma 4.2. In every round of the algorithm all new contraction patterns are pairwise disjoint. Furthermore every new contraction pattern is maximal, i.e. it is not a subpattern of a previously introduced contraction pattern, because every previously introduced contraction patterns is hidden in some edge. Hence, the depth of $\mathrm{CP}(\mathcal{T})$ is bounded by the number of rounds, which is $O(\log n)$. It remains to show that $(\mathcal{T}, \mathrm{CP}(\mathcal{T}))$ has width at most 5.

Every edge (u, w') in a tree $\mathcal{T}_i$ which is not contained in $\mathcal{T}$ originates from an earlier prune-and-bypass operation, which implies that $\mathrm{CP}(\mathcal{T})$ contains the maximal subpattern p hidden in (u, w'). Let w be the pruned leaf and v be the parent node of w. Thus, the three edges involved in the prune-and-bypass operation are (u, v), (v, w) and (v, w'). The direct subpatterns of p must be hidden in the three patterns (u, v), (v, w) and (v, w'). This proves that p has at most three direct subpatterns (if say (u, v) is an edge of $\mathcal{T} = \mathcal{T}_0$ then there is no contraction pattern yet hidden in (u, v), and similarly for (v, w) and (v, w'); hence the number of direct subpatterns of p can be smaller than three). Furthermore p has exactly two boundary nodes, namely the leaf w and its parent node v. The largest pattern has three boundary nodes (the root of the tree and the outermost leaves) and at most two direct subpatterns. Hence the width of $\mathrm{CP}(\mathcal{T})$ is bounded by 5. □

4.2 Compression to size $n/\log n$

We improve Proposition 4.3 by constructing a hierarchical definition in which many patterns are equivalent in a strong sense. This will be crucial for proving the size bound $O(n/\log n)$ for tree straight-line programs in Section 5. The result from this section will be only needed for our applications in Section 6.2 (but not Section 6.1).

For a pattern $p \in P$ the set $P[p] = \{q \in P \mid q \leq p\}$ forms a hierarchical definition of $\mathcal{T}[p]$. Two patterns $p_1, p_2 \in P$ are *equivalent* if the structures $(\mathcal{T}[p_1], P[p_1])$ and $(\mathcal{T}[p_2], P[p_2])$ are isomorphic. Alternatively, p_1 is equivalent to p_2 if the subtrees of the pattern tree rooted in p_1 and p_2 are isomorphic. The goal is to construct an FOM-definable hierarchical definition in which there are at most $O(n/\log n)$ inequivalent patterns. We follow the method of [19], in which the authors describe a parallel tree contraction algorithm which uses $O(n/\log n)$ processors on an EREW PRAM. The idea is to decompose the input tree into $O(n/\log n)$ many patterns of size $O(\log n)$.

We briefly summarize the notions and results from [19]. Let $\mathcal{T}$ be a binary tree with n nodes and let $1 < m \leq n$ be an integer. An inner node v in $\mathcal{T}$ is *m-critical* if $\lceil |\mathcal{T}[v]|/m \rceil \neq \lceil |\mathcal{T}[w]|/m \rceil$ for all children w of v, which is equivalent to saying that there exists a multiple m' of m such that

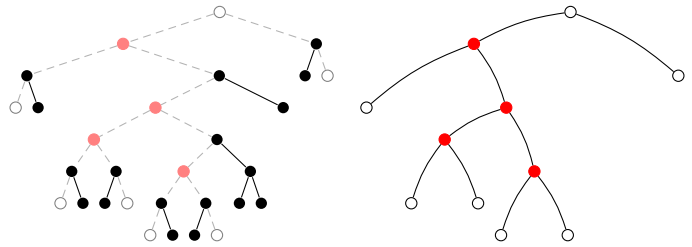

Fig. 5. Removing the 6-critical nodes (in red) and certain auxiliary nodes (in white), yields a disjoint union of patterns (the set B in the proof of Proposition 4.4), which are depicted in black. The binary tree $\mathcal{T}_C$ is obtained by contracting all patterns in B.

$|\mathcal{T}[w]| \leq m' < |\mathcal{T}[v]|$ for all children w of v. Consider the set C of all m-critical nodes and the subgraph of $\mathcal{T}$ induced by $V(\mathcal{T}) \setminus C$. Each of its connected components is a tree $\mathcal{T}[p]$ for some pattern p (this is implicitly stated in [19, Lemma 9.2.1]). These patterns p are called *m-bridges*.[4] It was proven in [19] that each m-bridge has size at most m and that the number of m-critical nodes in $\mathcal{T}$ is at most $2n/m - 1$.

Proposition 4.4. *For every constant ℓ, there is an FOM-computable function which maps a binary tree $\mathcal{T}$ of size n and with ℓ node labels to a hierarchical definition $(\mathcal{T}, P)$ of constant width, depth $O(\log n)$, and with $O(n/\log n)$ inequivalent patterns.*

Proof. Let $m = \Theta(\log n)$, which will be made explicit in the following. The number $m \leq n$ will be FOM-definable, which implies that the set of m-critical nodes will be also FOM-definable. The idea is to contract all m-bridges in $\mathcal{T}$ and then apply Proposition 4.3. However, the resulting tree is not necessarily a binary tree and may not be rooted in the root of $\mathcal{T}$, see Figure 5 for an example. Define C to be the set of all m-critical nodes together with the root of $\mathcal{T}$. We have $|C| = O(n/m)$. Furthermore we add certain leaf nodes to C. If the left (resp., right) subtree below a node $v \in C$ contains no node in C, then we add an arbitrary leaf (e.g. the smallest one with respect to the built-in order on the domain) from the left (resp., right) subtree to C. Since we add at most two nodes for each m-critical node, we still have $|C| = O(n/m)$. Notice that $V(\mathcal{T}) \setminus C$ is a disjoint union of sets $V[p]$ for certain patterns p in $\mathcal{T}$. Let B be the set of all these patterns, which can be seen to be FOM-definable. If we contract all patterns in B to edges we obtain the binary tree $\mathcal{T}_C$ over the node set C of size $O(n/m)$. Since the set of m-critical nodes is FOM-definable, also the set C and the binary tree $\mathcal{T}_C$ are FOM-definable.

We can now apply Proposition 4.3 to obtain a hierarchical definition $\mathrm{CP}(\mathcal{T}_C)$ of depth $O(\log |\mathcal{T}_C|) = O(\log(n/m)) = O(\log n)$ and width 5. Since the size of $\mathrm{CP}(\mathcal{T}_C)$ is at most $O(n/m)$, the number of inequivalent patterns is also bounded by the same number.

Let us count the number of non-isomorphic trees $\mathcal{T}[p]$ where $p \in B$. Let ℓ be the number of node labels in $\mathcal{T}$. Since every m-bridge has size at most m, this also holds for all $p \in B$. By inserting a distinguished leaf node to context patterns p, we can instead count the number of binary trees with at most $m+1$ nodes and $\ell+1$ labels. Using the formula for the Catalan number, one can upper-bound the number of such trees by $\frac{4}{3}(4\ell+4)^{m+1} \leq (4\ell+4)^{m+2}$, see e.g. [16, Lemma 1]. Hence by choosing $m = \lfloor 1/2 \cdot \log_{4\ell+4}(n) - 2 \rfloor \in \Theta(\log n)$ (which is indeed FOM-definable since all the involved arithmetic operations can be implemented in DLOGTIME-uniform TC^0), the number of non-isomorphic trees $\mathcal{T}[p]$ for $p \in B$ is bounded by $\sqrt{n} \in o(n/\log n)$.

[4] Our definition slightly deviates from the one given in [19] where a bridge also contains the neighbouring critical nodes as "attachments".

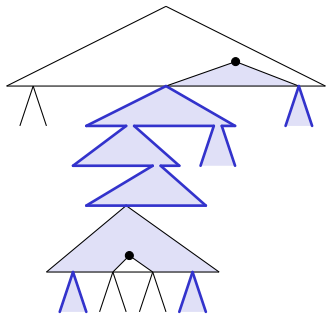

Fig. 6. The shaded pattern contains four maximal subpatterns (framed in blue) which are unions of zones.

For every $p \in B$ we define a canonical well-nested set of patterns Q_p which contains for each covered node $v \in V[p]$ the maximal subpattern $q \leq p$ which is rooted in v. Clearly, Q_p is a hierarchical definition for $\mathcal{T}[p]$ whose size and depth is bounded by $O(\log n)$. Its width is at most 3 because every pattern in Q_p has exactly one boundary node (its root) and at most two direct subpatterns. Furthermore, Q_p is FO-definable from p and canonical in the sense that the isomorphism type of the pattern tree of Q_p is determined by the isomorphism type of $\mathcal{T}[p]$. Hence the number of inequivalent patterns in $Q = \bigcup_{p \in B} Q_p$ is bounded by the number of patterns of size at most m, which by the above calculation is bounded by $o(n/\log n)$.

Now we claim that $P = \mathrm{CP}(\mathcal{T}_C) \cup Q$ is a hierarchical definition for $\mathcal{T}$ with the desired properties. Clearly the largest pattern is contained in P, and both $\mathrm{CP}(\mathcal{T}_C)$ and Q are well-nested. Furthermore, since each pattern $p \in B$ is hidden in some edge of $\mathcal{T}_C$, also P is well-nested. The depth of P is bounded by $O(\log n)$ and the number of inequivalent patterns is $O(n/\log n) + o(n/\log n) = O(n/\log n)$. To prove that the width of P is bounded by some constant, we notice that the patterns $p \in B$ are the maximal subpatterns of $\mathcal{T}$ hidden in some edge of $\mathcal{T}_C$. More precisely, if $p \in B$ is a direct subpattern of a pattern $q \in \mathrm{CP}(\mathcal{T}_C)$ then p must be hidden in an edge of $\mathcal{T}_C[q]$ which is not covered by any subpattern q' of q. Since the branching tree of q has size at most 5, there are at most 4 such possible edges. This proves that the width of P is at most 9. □

4.3 Non-binary trees

Now we extend Proposition 4.4 to arbitrary k-ordered trees, for any constant $k \geq 1$.

Proposition 4.5. *For all constants $k, \ell \geq 1$, there is an FOM-computable function which maps a k-ordered tree $\mathcal{T}$ of size n and with ℓ node labels to a hierarchical definition $(\mathcal{T}, P)$ of depth $O(\log n)$, constant width, and with $O(n/\log n)$ inequivalent patterns.*

Proof. The idea is that one can embed $\mathcal{T}$ into an FOM-definable binary tree $\mathcal{T}'$ and transform a hierarchical definition for $\mathcal{T}'$ into one for $\mathcal{T}$. More precisely, an *embedding* of $\mathcal{T}$ into $\mathcal{T}'$ is an injective function $\varphi : V(\mathcal{T}) \to V(\mathcal{T}')$ such that φ maps the root of $\mathcal{T}$ to the root of $\mathcal{T}'$, $u \preceq_T v$ if and only if $\varphi(u) \preceq_{T'} \varphi(v)$ and φ preserves the depth-first order of nodes in $\mathcal{T}$, see [15]. Nodes which are not in the image of φ are labelled by a fresh symbol to distinguish them from nodes in the image of φ. Each node $v \in V(\mathcal{T})$ defines the *zone*

$$V[\varphi(u)] \setminus \bigcup_{w \text{ child of } v} V[\varphi(w)].$$

The set of all zones form a partition of $V(\mathcal{T}')$. We require that the size of each zone is bounded by a function of the maximum out-degree k, which can be done by embedding a node with r children into a chain of at least $r - 1$ binary nodes.

From Proposition 4.3 we obtain a hierarchical definition $P' = \mathrm{CP}(\mathcal{T}')$ for $\mathcal{T}'$ of depth $O(\log n)$ and width at most 5, and by Proposition 4.4 it has only $O(n/\log n)$ inequivalent patterns. Notice that the patterns in P' can intersect arbitrarily with the zones, as illustrated in Figure 6. We adapt P' in such a way that every pattern is a union of zones. Since φ respects the ancestor relation and the depth-first order, this directly yields a hierarchical definition of $\mathcal{T}$ (by taking the preimages under φ).

For a pattern $p' \in P'$ let $Z(p') \subseteq V[p']$ be the union of all zones which are contained in $V[p']$. Notice that $V[p'] \setminus Z(p')$ has constant size (it is contained in at most two zones) and that $Z(p')$ can be written (uniquely) as a disjoint union of a constant number of maximal subpatterns of p' (the constants only depend on the maximum out-degree of $\mathcal{T}$ and can be set to $1 + 2(k-1)$). We denote the set of these subpatterns by $S(p')$, which are framed in blue in Figure 6. Define the set $P = \bigcup_{p' \in P'} S(p')$, which is clearly FO-definable from P'. We claim that (1) P is a hierarchical definition for $\mathcal{T}'$ of depth $O(\log n)$, (2) its width is bounded by some constant, and (3) the number of inequivalent patterns in P is $O(n/\log n)$.

Clearly the largest pattern is contained in P and we need to verify that P is well-nested. Observe that for all $p', q' \in P'$ we have:

- if $p' \leq q'$ then $Z(p') \subseteq Z(q')$, and
- if $V[p'] \cap V[q'] = \emptyset$ then $Z(p') \cap Z(q') = \emptyset$.

Consider $p \in S(p')$ and $q \in S(q')$. If p' and q' are disjoint, then also p and q are disjoint. If $p' \leq q'$ then p is a subpattern of some pattern $r \in S(q')$. If $r = q$ then $p \leq q$, otherwise q is disjoint from r and therefore also from p. This concludes the proof that P is well-nested. Furthermore these observations imply that, if $p \in S(p')$ and $q \in S(q')$ such that $p < q$, then $p' < q'$. Hence the depth of P is bounded by the depth of P'.

Next we show that the width of P is bounded by some constant. Consider a pattern $p \in P$ and let $p' \in P'$ be the minimal pattern such that $p \in S(p')$. Let $p'_1, \ldots, p'_m \in P'$ be the direct subpatterns of p'. Since the width of P' is constant, m is bounded by a constant. Moreover, also the size of the boundary $\partial p'$ is bounded by a constant. Since $\partial p' = V[p'] \setminus \bigcup_{i=1}^m V[p'_i]$ and $Z(p') \setminus \bigcup_{i=1}^m Z(p'_i)$ only differ by a constant number of nodes, it follows that the size of $Z(p') \setminus \bigcup_{i=1}^m Z(p'_i)$ is bounded by a constant. Note that $V[p] \subseteq Z(p')$. Moreover, by the minimality of p', every pattern in $\bigcup_{i=1}^m S(p'_i)$ is either disjoint or properly contained in $V[p]$. It follows that the boundary ∂p is contained in $V[p] \setminus \bigcup_{i=1}^m Z(p'_i) \subseteq Z(p') \setminus \bigcup_{i=1}^m Z(p'_i)$. Hence, there is a constant that bounds the size of every boundary ∂p for $p \in P$.

To show that P has bounded width, it remains to show that the number of direct subpatterns of a pattern $p \in P$ is bounded by a constant. Consider such a direct subpattern $q \in P$, i.e. $q \lessdot p$. Choose $q' \in P'$ maximal such that $q \in S(q')$ and choose $p' \in P'$ minimal such that $p \in S(p')$. We already know that $q' < p'$ and we claim that in fact $q' \lessdot p'$ holds. Towards a contradiction let $r' \in P'$ with $q' < r' < p'$. By the choice of q' and p' we have $p, q \notin S(r')$. This means that q is a proper subpattern of some pattern $r \in S(r')$. Since both r and p share the subpattern q, the patterns r and p are comparable. Furthermore, since r is a subpattern of some pattern in $S(p')$, we must have $r \leq p$. We conclude that $q \leq r \leq p$ and $q \neq r \neq p$, which contradicts $q \lessdot p$. This proves that the number of direct subpatterns of p is bounded by $|S(p')|$ (a constant) times the number of direct subpatterns of p'. The latter is bounded by the width of P', which is a constant.

It remains to show that P has $O(n/\log n)$ inequivalent patterns. First, if $p', q' \in P'$ are equivalent then there is an isomorphism $\psi : (\mathcal{T}'[p'], P'[p']) \to (\mathcal{T}'[q'], P'[q'])$. This isomorphism induces a bijection between $S(p')$ and $S(q')$ which maps each pattern $p \in S(p')$ to the isomorphic pattern $\psi(p) \in S(q')$. Since each set $S(p')$ has constant size, it suffices to show that ψ yields an isomorphism from $(\mathcal{T}'[p], P[p])$ to $(\mathcal{T}'[\psi(p)], P[\psi(p)])$ for all $p \in S(p')$. Recall that if $r < p$ for a pattern $r \in S(r')$,

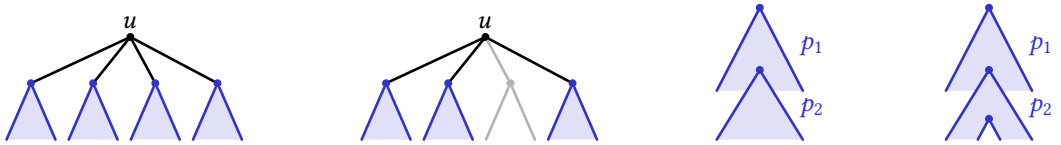

Fig. 7. Four types in which a pattern can be decomposed in a normal form hierarchical definition.

then $r' < p'$ and thus $r' \in P'[p']$, and similarly for $\psi(p)$. It follows that the isomorphism types of $(\mathcal{T}'[p], P[p])$ and $(\mathcal{T}'[\psi(p)], P[\psi(p)])$ are completely determined by the isomorphism types of $(\mathcal{T}'[p'], P'[p'])$ and $(\mathcal{T}'[q'], P'[q'])$, which are equal. □

4.4 Normal form

In a final step, we bring the computed hierarchical definition into a normal form. A hierarchical definition $(\mathcal{T}, P)$ is in *normal form* if for each pattern $p \in P$ one of the following two cases holds:

(1) If u is the root of p, then $\partial p = \{u\}$ and every direct subpattern of p is a subtree pattern (which must be rooted in a child of u).
(2) p has exactly two direct subpatterns p_1 and p_2, and $V[p]$ is the disjoint union of $V[p_1]$ and $V[p_2]$.

Figure 7 illustrates the four types that a pattern can be decomposed into, where the first two patterns have type 1 and the latter two have type 2.

Theorem 4.6. *For all constants $k, \ell \geq 1$, there is an FOM-computable function which maps a k-ordered tree $\mathcal{T}$ of size n and with ℓ node labels to a hierarchical definition $(\mathcal{T}, P)$ in normal form where P has depth $O(\log n)$ and $O(n/\log n)$ inequivalent patterns.*

Proof. Let P be the hierarchical definition from Proposition 4.5. Note that P has constant width. For each pattern $p \in P$ we introduce new subpatterns corresponding to the subtrees of its branching tree: For each node $w \in \partial p$ on the boundary, we add the maximal subpattern of p rooted in w. Furthermore, for each direct subpattern $q \lessdot p$ we add the maximal subpattern of p rooted in the root of q. This ensures that the branching tree of each pattern has height at most 1. The depth of P increases at most by a factor of its width. Now consider a context pattern $(v, w) \in P$ which has a direct subpattern $(v', w) \in P$ such that v' is a child of v. To establish normal form it suffices to introduce the pattern (v, v'). In this step the depth of P increases at most by a factor of two.

Both steps are FO-computable. Finally, notice that for any two equivalent patterns we introduce equivalent new subpatterns, therefore, the number of inequivalent patterns increases by a constant factor. □

5 BALANCING OVER FREE TERM ALGEBRAS AND ARBITRARY ALGEBRAS

In this section, we transform the hierarchical decomposition constructed in the previous section into a so called tree straight-line programs, or *TSLPs* for short. TSLPs are used as a compressed representation of trees, see [27] for a survey. Formally, a *TSLP* $\mathcal{G} = (N, \Sigma, S, P)$ consists of two disjoint ranked alphabets N and Σ, where symbols in N are called *nonterminals* and have rank at most one, a start nonterminal $S \in N$ of rank zero, and a set of productions P. For each nonterminal $A \in N$ there exists exactly one production $(A \to t) \in P$, where $t \in T(\Sigma \cup N)$ if A has rank zero and $t \in C(\Sigma \cup N)$ if A has rank one. Furthermore the relation $\{(A, B) \in N \times N \mid B \text{ occurs in } t \text{ where } (A \to t) \in P\}$ must be acyclic. These properties ensure that the start nonterminal S derives exactly one term $t \in T(\Sigma)$ by applying the productions in any order, starting with S, as long as possible, see [27]

for more details (we will give an alternative circuit based definition below). A TSLP is in *normal form* if every production has one of the following forms:

- $A \to f(A_1, \dots, A_r)$
- $A(x) \to f(A_1, \dots, A_{i-1}, x, A_{i+1}, \dots, A_r)$
- $A \to B(C)$
- $A(x) \to B(C(x))$

We will work here with an alternative definition of TSLPs as circuits over an extended term algebra.

Definition 5.1. *Let Σ be a ranked alphabet. The two-sorted algebra $\mathcal{A}(\Sigma)$ consists of the two sorts $T(\Sigma)$ (all terms) and $C(\Sigma)$ (all contexts) and the following operations:*

- *for all $f \in \Sigma$ of rank $r \geq 0$ the operation $f : T(\Sigma)^r \to T(\Sigma)$ that maps $(t_1, \dots, t_r)$ to $f(t_1, \dots, t_r)$,*
- *for all $f \in \Sigma$ of rank $r \geq 1$ and $1 \leq i \leq r$ the operation $\hat{f}_i : T(\Sigma)^{r-1} \to C(\Sigma)$ with $\hat{f}_i(t_1, \dots, t_{r-1}) = f(t_1, \dots, t_{i-1}, x, t_i, \dots, t_{r-1})$,*
- *the substitution operation $\mathrm{sub} : C(\Sigma) \times T(\Sigma) \to T(\Sigma)$ with $\mathrm{sub}(s, t) = s(t)$,*
- *the composition operation $\circ : C(\Sigma) \times C(\Sigma) \to C(\Sigma)$ with $\circ(s, t) = s(t)$.*

Theorem 5.2 (universal balancing theorem). *Let Σ be a fixed ranked alphabet. Given a term t over Σ of size n, one can compute in TC^0 a normal form TSLP for t of size $O(n/\log n)$ and depth $O(\log n)$. The TSLP is given as a circuit over $\mathcal{A}(\Sigma)$ in EC-representation.*

Proof. In TC^0 we convert t into a Σ-labelled tree $\mathcal{T}$ in ancestor representation (see Theorem 3.1) and apply Theorem 4.6 to obtain a hierarchical definition $(\mathcal{T}, P)$, which has depth $O(\log n)$ and $O(n/\log n)$ many inequivalent patterns. We can translate $(\mathcal{T}, P)$ directly into a tree $\mathcal{T}'$ in ancestor representation over the two-sorted algebra $\mathcal{A}(\Sigma)$ that evaluates to t: Patterns of rank zero (resp., one) are nodes of sort $T(\Sigma)$ (resp., $C(\Sigma)$), and the children of a pattern are its direct subpatterns. The pattern type (whether the pattern is a subtree pattern or a context pattern) determines the operator of the corresponding node. The ancestor relation is FO-definable since pattern p is an ancestor of pattern q if and only if $\mathcal{T}[q] \subseteq \mathcal{T}[p]$. From the tree $\mathcal{T}'$ we can compute in TC^0 by Lemma 3.5 the EC-representation of the minimal dag $\mathcal{C}$, which is a circuit over the structure $\mathcal{A}(\Sigma)$. Since the number of inequivalent patterns of the hierarchical definition is $O(n/\log n)$ and its depth is $O(\log n)$, the circuit $\mathcal{C}$ has size $O(n/\log n)$ and depth $O(\log n)$. □

The following definition can also be found in [25].

Definition 5.3. *For an algebra $\mathcal{A}$ over Σ, the two-sorted algebra $\mathcal{F}(\mathcal{A})$ extends $\mathcal{A}$ by a second sort $\mathcal{A}[x]$ containing all linear term functions $p : A \to A$. The operations of $\mathcal{F}(\mathcal{A})$ are the following:*

- *for all $f \in \Sigma$ of rank $r \geq 0$ the operation $f^{\mathcal{A}} : A^r \to A$,*
- *for all $f \in \Sigma$ of rank $r \geq 1$ and and $1 \leq i \leq r$ the operation $\hat{f}_i : A^{r-1} \to \mathcal{A}[x]$ that maps $(a_1, \dots, a_{r-1}) \in A^{r-1}$ to the linear term function $f^{\mathcal{A}}(a_1, \dots, a_{i-1}, x, a_i, \dots, a_{r-1})$,*
- *the substitution operation $\mathrm{sub} : \mathcal{A}[x] \times A \to A$ with $\mathrm{sub}(p, a) = p(b)$,*
- *the composition operation $\circ : \mathcal{A}[x] \times \mathcal{A}[x] \to \mathcal{A}[x]$ that maps (p, q) to the composition of the mappings p and q.*

From Theorem 5.2 we immediately get:

Theorem 5.4. *Given a term t over $\mathcal{A}$ of size n, one can compute in TC^0 an equivalent circuit over $\mathcal{A}[x]$ in EC-representation of size $O(n/\log n)$ and depth $O(\log n)$.*

By applying Lemma 3.4 we obtain:

Theorem 5.5. *Given a term over $\mathcal{A}$ of size n, one can compute in* TC^0 *an equivalent term over $\mathcal{A}[x]$ of depth $O(\log n)$.*

Theorem 5.2 and 5.4 assume that the number $\ell = |\Sigma|$ of function symbols is a constant. This is certainly true if we work over a fixed algebra with finitely many operations and constants. On the other hand, for many applications it is useful to allow a countably infinite set Σ of ranked symbols. Examples are arithmetic expressions where arbitrary ring constants and/or an arbitrary number of variables may occur.

Hence, for the further considerations we fix a countably infinite set Σ of ranked functions symbols. We need the assumption that the rank of symbols in Σ is bounded by a constant k. In the examples from the previous paragraph this assumption would be not crucial since ring constants and variables would be symbols of rank zero. W.l.o.g. we can identify the elements of Σ with natural numbers. Consider now a term t over Σ of size n. It is represented as a string where the node label $k \in \Sigma$ is represented by the unary string a^k for some fixed symbol a. Let $\Sigma_t \subseteq \Sigma$ be the set of node labels that occur in t, and let $\ell = |\Sigma_t|$. As in Section 3 (first paragraph) we can represent t by the relational structure $\mathcal{T} = (V, (E_i)_{1\leq i\leq k}, (P_a)_{a\in\Sigma_t})$, and the notion of a hierarchical definition can be defined as before (note that the definition of a hierarchical definition does not refer to the edge labels). The problem with this definition is that we do no longer work with relational structures over a fixed vocabulary, which is problematic if we want to use the FOM-framework. The crucial observation is that the algorithm for constructing a hierarchical definition is "almost" independent of the node labels. Only the number of node labels ℓ appears once in the proof Proposition 4.4 when the number m is defined as $m = \lfloor 1/2 \cdot \log_{4\ell+4}(n) - 2 \rfloor$. We can therefore run our algorithm on the unlabelled tree structure $\mathcal{T}_0 = (V, (E_i)_{1\leq i\leq k})$ with the value $m = \lfloor 1/2 \cdot \log_{4\ell+4}(n) - 2 \rfloor \in \Theta(\log_\ell n)$ and obtain a hierarchical definition $(\mathcal{T}, P)$ of depth $O(\log n)$ in which only $O(n/m) = O(n/\log_\ell n)$ many inequivalent patterns exist. The rest of the proof follows the arguments from the proof of Theorem 5.2: From $(\mathcal{T}, P)$ we construct in TC^0 the tree $\mathcal{T}'$ for which the EC-representation of the minimal dag C is computed in TC^0 using Lemma 3.5. For this last step, the node labels of $\mathcal{T}'$ (which are symbols from $\{\mathrm{sub}, \circ\} \cup \{a, \hat{a}_i \mid a \in \Sigma_t, 1 \leq i \leq k\}$) are important. These node labels can be computed easily using the labels of the original term t. Moreover, for the computation of the minimal dag it is only important that one can check whether two nodes have the same label. This is possible in TC^0 also in the case where the node label are not from a fixed alphabet. We have shown the following result:

Theorem 5.6 (universal balancing theorem). *Let Σ be countably infinite set of function symbols such that the rank of symbols in Σ is bounded by a constant. Given a term t over Σ of size n which contains ℓ different symbols from Σ, one can compute in* TC^0 *a normal form TSLP for t of size $O(n/\log_\ell n)$ and depth $O(\log n)$. The TSLP is given as a circuit over $\mathcal{A}(\Sigma)$ in EC-representation.*

6 APPLICATIONS

6.1 Alternative proof of a result by Krebs, Limaye and Ludwig

Recently, Krebs, Limaye and Ludwig presented a similar result to ours [25]. We will state their result and reprove it using our balancing theorem. For an S-sorted algebra $\mathcal{A}$ we define the algebra $(\mathcal{B}, \mathcal{A})$ which extends $\mathcal{A}$ by the Boolean sort $\mathcal{B} = \{0, 1\}$. All operations from $\mathcal{A}$ are inherited to $(\mathcal{B}, \mathcal{A})$, with the addition of the Boolean disjunction $\vee$, conjunction $\wedge$ and negation $\neg$, and for each sort $s \in S$ a multiplexer function

$$\mathrm{mp}_s : \{0, 1\} \times A_s \times A_s \to A_s$$

where $\mathrm{mp}_s(b, d_0, d_1) = d_b$. A circuit family $(C_n)_{n\geq 0}$ of circuits over $(\mathcal{B}, \mathcal{A})$ where C_n has Boolean input gates $x_1, \ldots, x_n$ computes a function $f : \{0, 1\}^* \to A$. The class $\mathcal{A}$-NC^1 denotes the class

of functions computed by a DLOGTIME-uniform circuit family over $(\mathcal{B}, \mathcal{A})$ of constant fan-in, polynomial size and logarithmic depth.

Theorem 6.1. *For every algebra $\mathcal{A}$ there exists a DLOGTIME-uniform $\mathcal{F}(\mathcal{A})$-NC^1 circuit family which computes the value of a given expression over $\mathcal{A}$.*

Proof. For a given input expression one can compute in TC$^0 \subseteq$ NC^1 an equivalent logarithmic depth expression t over $\mathcal{F}(\mathcal{A})$ by Theorem 5.5. It suffices to construct a DLOGTIME-uniform circuit family which evaluates t. Let $n = |t|$ and assume that the depth of t is at most $d = O(\log n)$. Furthermore, let k be the maximal arity of an operation in $\mathcal{F}(\mathcal{A})$ (this is a constant). We can test in TC^0 whether a given string $\rho \in \{1, \ldots, k\}^*$ of length at most d is a valid address string of a path from the root of t to some node and, if so, we can compute the node label in TC^0. Consider the circuit with gates of the form v_ρ, where $\rho \in \{1, \ldots, k\}^*$ is a string of length at most d, and v_ρ computes the value of the addressed node, or computes some arbitrary value if ρ is not a valid address string. With the help of multiplexer gates and the node label information we can clearly compute v_ρ from the gates $v_{\rho \cdot i}$. Clearly, the described circuit has depth $O(\log n)$ and the constructed circuit family can be seen to be DLOGTIME-uniform. □

As shown in [25], many known results on the complexity of expression evaluation problems can be derived from Theorem 6.1. The following list is not exhaustive:

- Buss' theorem [9]: The expression evaluation problem for $(\{0,1\}, \wedge, \vee, \neg, 0, 1)$ belongs to DLOGTIME-uniform NC^1.
- More generally, for every fixed finite algebra $\mathcal{A}$ the expression evaluation problem belongs to DLOGTIME-uniform NC^1 [26].
- Expression evaluation for the semirings $(\mathbb{N}, +, \cdot, 0, 1)$ (resp., $(\mathbb{Z}, +, \cdot, 0, 1)$) belongs to #NC^1 (resp., GapNC^1) [8].

6.2 Regular expressions and semirings

It has been shown in [20, Theorem 6] that from a given regular expression of size n one can obtain an equivalent regular expression of star height $O(\log n)$. Here, we strengthen this result in several directions: (i) the resulting regular expression (viewed as a tree) has depth $O(\log n)$ (and not only star height $O(\log n)$), (ii) it can be represented by a circuit with only $O(n/\log n)$ nodes, and the construction can be carried out in TC^0 (or, alternatively in linear time if we use [16]).

For a finite alphabet Σ, let Reg(Σ) be the set of regular languages over Σ. It forms an algebra with the constants $a \in \Sigma$, $\emptyset$ and $\{\varepsilon\}$, the unary operator * and the binary operations union + and concatenation $\cdot$. It is also known as the free Kleene algebra.

Theorem 6.2. *Given a regular expression t over Σ, one can compute in* TC^0 *an equivalent circuit over* Reg(Σ) *in EC-representation of size $O(n/\log n)$ and depth $O(\log n)$.*

Proof. Let $\mathcal{R} = \mathrm{Reg}(\Sigma)$. We claim the following for any context $t \in C(\mathcal{R})$. If the parameter x is not below any $*$-operator, then the linear term function $t^{\mathcal{R}}$ has the form

$$t^{\mathcal{R}}(x) = axb + c \text{ for some } a, b, c \in \mathcal{R}, \tag{2}$$

otherwise it has the form

$$t^{\mathcal{R}}(x) = \alpha(axb + c)^*\gamma + \delta \text{ for some } a, b, c, \alpha, \gamma, \delta \in \mathcal{R}. \tag{3}$$

The linear term functions (2) and (3) are closed under union and left/right concatenation with constants from $\mathcal{R}$. If a term function is of the form (2) then its star is of the form (3). The only

non-trivial part is to prove that the set of term functions of type (3) are closed under $*$ (this shows then also closure under composition). Let $\beta = axb + c$ and $p(x) = \alpha\beta^*\gamma + \delta$. We claim that

$$p(x)^* = (\alpha\beta^*\gamma + \delta)^* \overset{(\dagger)}{=} \delta^*\alpha(\beta + \gamma\delta^*\alpha)^*\gamma\delta^* + \delta^* = \delta^*\alpha(axb + c + \gamma\delta^*\alpha)^*\gamma\delta^* + \delta^*.$$

Note that this expression is indeed of the form (3). To verify the identity (†), one should consider α, β, γ and δ as letters. The inclusion $\delta^*\alpha(\beta + \gamma\delta^*\alpha)^*\gamma\delta^* + \delta^* \subseteq (\alpha\beta^*\gamma + \delta)^*$ is obvious. For the other inclusion, one considers a word $w \in (\alpha\beta^*\gamma + \delta)^*$. We show that $w \in \delta^*\alpha(\beta + \gamma\delta^*\alpha)^*\gamma\delta^* + \delta^*$. The case that $w \in \delta^*$ is clear. Otherwise, w contains at least one occurrence of α and γ, and we can factorize w uniquely as $w = w_0\alpha w_1\gamma w_2$, where $w_0, w_2 \in \delta^*$. Moreover, we must have $w_1 \in (\beta + \gamma\delta^*\alpha)^*$, which shows that $w \in \delta^*\alpha(\beta + \gamma\delta^*\alpha)^*\gamma\delta^*$.

Note that a term function of type (2) (resp., (3)) can be represented by the three (resp., six) elements $a, b, c \in \mathcal{R}$ (resp., $a, b, c, \alpha, \gamma, \delta \in \mathcal{R}$).

By Theorem 5.4 we compute from the input regular expression t in TC^0 an equivalent circuit C over $\mathcal{R}[x]$ in EC-representation. We partition $V(C)$ into three sets: the set V_0 of nodes which evaluate to elements in $\mathcal{R}$, the set V_1 of nodes that evaluate to a linear term function of type (2), and the set of nodes that evaluate to a linear term function of type (3). The distinction between nodes that evaluate to elements of $\mathcal{R}$ and nodes that evaluate to elements of $\mathcal{R}[x]$ is directly displayed by the node label. Furthermore, if a node $u \in V$ has a descendant $v \in V$ labelled by $\hat{*}$ such that all nodes on the path from u to v are labelled by $\circ$, except from v itself, then u belongs to V_2, otherwise to V_1. This allows to define V_0, V_1, and V_2 in FO using the EC-representation of the circuit.

Using a guarded transduction we keep every node in V_0, replace every node in V_1 by 3 nodes (which compute the three regular languages a, b, c in (2)) and replace every node in V_2 by 6 nodes (which compute the six regular languages $a, b, c, \alpha, \gamma, \delta$ in (2)). Moreover, we can define the wires accordingly.

Let us consider one specific case (the most difficult one) for the definition of the wires. Assume that $A = A_1 \circ A_2$ where $A_1, A_2, A \in V_2$, A_i is replaced by the six nodes $a_i, b_i, c_i, \alpha_i, \gamma_i, \delta_i$ and A was replaced by the six nodes $a, b, c, \alpha, \gamma, \delta$. Then A_i computes the term function $t_i(x) = \alpha(a_i x b_i + c_i)^*\gamma_i + \delta_i$ and A has to compute the composition

$$\begin{aligned} t(x) = t_2(t_1(x)) &= \alpha_2(a_2[\alpha_1(a_1xb_1 + c_1)^*\gamma_1 + \delta_1]b_2 + c_2)^*\gamma_2 + \delta_2 \\ &= \alpha_2\,(\underbrace{a_2\alpha_1}_{\alpha'}\,(\underbrace{a_1xb_1 + c_1}_{\beta'})^*\,\underbrace{\gamma_1 b_2}_{\gamma} + \underbrace{a_2\delta_1 b_2 + c_2}_{\delta'})^*\,\gamma_2 + \delta_2 \end{aligned}$$

Using the above identity (†), the expression in the last line becomes equivalent to

$$\begin{aligned} & \alpha_2\,(\delta'^*\alpha'(\beta' + \gamma'\delta'^*\alpha')^*\gamma'\delta'^* + \delta'^*)\,\gamma_2 \;+\; \delta_2 \\ = \;& \alpha_2\delta'^*\alpha'(\beta' + \gamma'\delta'^*\alpha')^*\gamma'\delta'^*\gamma_2 + \alpha_2\delta'^*\gamma_2 + \delta_2 \\ = \;& \underbrace{\alpha_2\delta'^*\alpha'}_{\alpha}(a_1xb_1 + \underbrace{c_1 + \gamma'\delta'^*\alpha'}_{c})^*\,\underbrace{\gamma'\delta'^*\gamma_2}_{\gamma} + \underbrace{\alpha_2\delta'^*\gamma_2 + \delta_2}_{\delta}. \end{aligned}$$

Hence, we can define $a = a_1$, $b = b_1$ (these are copy gates) and c, α, γ, and δ as shown above. For the latter, we have to introduce a constant number of additional gates to built up the above terms for c, α, γ, and δ. For instance, we have

$$\alpha = \alpha_2\delta'^*\alpha' = \alpha_2(a_2\delta_1 b_2 + c_2)^*\alpha_2\alpha_1,$$

and we need seven more gates to built up this expression from $\alpha_1, \alpha_1, \delta_1, a_2, b_2, c_2$. These seven gates are also produced by the guarded transduction. From Lemma 3.8 we obtain the desired circuit over $\mathcal{R}$. □

A *semiring* $\mathcal{R} = (R, +, \cdot)$ is a structure with two associative binary operations + and $\cdot$, such that $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(a + b) \cdot c = a \cdot c + b \cdot c$ for all $a, b, c \in R$. Notice that we do not require a semiring to have a zero- or a one-element. Using the same strategy as in the proof of Theorem 6.2 one can show the following result:

THEOREM 6.3. *Let $\mathcal{R}$ be a semiring. Given an expression t over $\mathcal{R}$, one can compute in* TC^0 *an equivalent circuit over $\mathcal{R}$ in EC-representation of size $O(n/\log n)$ and depth $O(\log n)$.*

PROOF. For a semiring $\mathcal{R}$ one has to observe that every linear term function $t^{\mathcal{R}}$ can be written as $t^{\mathcal{R}}(x) = axb + c$ for semiring elements $a, b, c \in \mathcal{R}$ where any of the elements a, b, c can also be missing. In other words, the right-hand side of $t^{\mathcal{R}}(x)$ can be of one of the following 8 forms for $a, b, c \in \mathcal{R}$: $a \cdot x \cdot b + c$, $a \cdot x \cdot b$, $a \cdot x + c$, $x \cdot b + c$, $a \cdot x$, $x \cdot b$, $x + c$, x. By Theorem 5.4 we compute from the input semiring expression t in TC^0 an equivalent circuit C over $\mathcal{R}[x]$ in EC-representation. We partition $V(C)$ into $V(C) = V_0 \cup V_1 \cup \cdots \cup V_8$ where V_0 contains all gates which evaluate to a semiring element and $V_1 \cup \cdots \cup V_8$ contain gates which evaluate to a linear term function, grouped by the 8 possible types listed above. It is easy to see that the sets V_i are FO-definable using the EC-representation of C. For example, a gate v carries an a-coefficient, i.e. it computes a term function of the form $a \cdot x$, $a \cdot x \cdot b$, $a \cdot x + c$ or $a \cdot x \cdot b + c$, if and only if there exists a path $v = v_1, v_2, \ldots, v_m$ such all nodes $v_1, \ldots, v_{m-1}$ are labelled by the binary $\mathcal{F}(\mathcal{R})$-operation $\circ$ and v_m is labelled by the unary $\mathcal{F}(\mathcal{R})$-operation $\hat{\cdot}_2$ that maps $a \in \mathcal{R}$ to the linear term function $a \cdot x$. This allows to carry over the arguments from the proof of Theorem 6.2. □

7 FUTURE WORK

An interesting problem in connection with Theorem 5.4 is to determine other classes of algebras $\mathcal{A}$ which admit TC^0-tree balancing algorithms. For this, one has to show that circuits over the algebra $\mathcal{A}[x]$ can be transformed without depth increase into equivalent circuits over the algebra $\mathcal{A}$. For semirings and Kleene algebras (i.e., regular expressions) this is possible. Are there other nice classes of algebras that allow such a reduction?

REFERENCES

[1] Karl R. Abrahamson, Norm Dadoun, David G. Kirkpatrick, and Teresa M. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.

[2] Eric Allender, Jia Jiao, Meena Mahajan, and V. Vinay. Non-commutative arithmetic circuits: Depth reduction and size lower bounds. *Theoretical Computer Science*, 209(1-2):47–86, 1998.

[3] David A. M. Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41:274–306, 1990.

[4] Philip Bille, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Tree compression with top trees. *Information and Computation*, 243:166–177, 2015.

[5] Maria Luisa Bonet and Samuel R. Buss. Size-depth tradeoffs for boolean fomulae. *Information Processing Letters*, 49(3):151–155, 1994.

[6] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[7] Nader H. Bshouty, Richard Cleve, and Wayne Eberly. Size-depth tradeoffs for algebraic formulas. *SIAM Journal on Computing*, 24(4):682–705, 1995.

[8] S. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM Journal on Computing*, 21(4):755–780, 1992.

[9] Samuel R. Buss. The Boolean formula value problem is in ALOGTIME. In *Proceedings of the 19th Annual Symposium on Theory of Computing, STOC 87*, pages 123–131. ACM Press, 1987.

[10] Samuel R. Buss. Algorithms for boolean formula evaluation and for tree-contraction. *Proof Theory, Complexity, and Arithmetic*, pages 95–115, 1993.

[11] M. Charikar, E. Lehman, A. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[12] Stephen A. Cook and Pierre McKenzie. Problems complete for deterministic logarithmic space. *Journal of Algorithms*, 8:385–394, 1987.

[13] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. Elsevier, 1990.
[14] Bartlomiej Dudek and Pawel Gawrychowski. Slowing down top trees for better worst-case compression. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, volume 105 of *LIPIcs*, pages 16:1–16:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
[15] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Algorithmic meta theorems for circuit classes of constant and logarithmic depth. In *Proceedings of the 29th Symposium on Theoretical Aspects of Computer Science, STACS 2012*, volume 14 of *LIPIcs*, pages 66–77. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.
[16] Moses Ganardi, Danny Hucke, Artur Jez, Markus Lohrey, and Eric Noeth. Constructing small tree grammars and small circuits for formulas. *Journal of Computer and System Sciences*, 86:136–158, 2017.
[17] Moses Ganardi, Danny Hucke, Daniel König, and Markus Lohrey. Circuits and expressions over finite semirings, 2018. accepted for publication in *ACM Transactions on Computation Theory*.
[18] Adrià Gascón, Markus Lohrey, Sebastian Maneth, Carl Philipp Reh, and Kurt Sieber. Grammar-based compression of unranked trees. In *Proceedings of 13th International Computer Science Symposium in Russia, CSR 2018*, volume 10846 of *Lecture Notes in Computer Science*, pages 118–131. Springer, 2018.
[19] Hillel Gazit, Gary L. Miller, and Shang-Hua Teng. Optimal tree contraction in an EREW model. In S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, editors, *Concurrent Computations: Algorithms, Architecture and Technology*, pages 139–156, New York, 1988. Plenum Press.
[20] Hermann Gruber and Markus Holzer. Finite automata, digraph connectivity, and regular expression size. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, ICALP 2008, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 2008.
[21] William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65:695–716, 2002.
[22] Danny Hucke and Markus Lohrey. Universal tree source coding using grammar-based compression. In *Proceedings of the IEEE International Symposium on Information Theory, ISIT 2017*, pages 1753–1757. IEEE, 2017.
[23] Neil Immerman. *Descriptive Complexity*. Graduate texts in computer science. Springer, 1999.
[24] Artur Jez and Markus Lohrey. Approximation of smallest linear tree grammar. *Information and Computation*, 251:215–251, 2016.
[25] Andreas Krebs, Nutan Limaye, and Michael Ludwig. A unified method for placing problems in polylogarithmic depth. In *Proceedings of the 37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017*, volume 93 of *LIPIcs*, pages 36:36–36:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
[26] Markus Lohrey. On the parallel complexity of tree automata. In *Proceedings of the 12th International Conference on Rewrite Techniques and Applications, RTA 2001*, volume 2051 of *Lecrture Notes in Computer Science*, pages 201–215. Springer, 2001.
[27] Markus Lohrey. Grammar-based tree compression. In *Proceedings of the 19th International Conference on Developments in Language Theory, DLT 2015*, volume 9168 of *Lecture Notes in Computer Science*, pages 46–57. Springer, 2015.
[28] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. XML tree structure compression using RePair. *Information Systems*, 38(8):1150–1167, 2013.
[29] Gary L. Miller and Shang-Hua Teng. Dynamic Parallel Complexity of Computational Circuits. In *Proceedings of the 19th Annual Symposium on Theory of Computing, STOC 1987*, pages 254–263. ACM Press, 1987.
[30] Gary L. Miller and Shang-Hua Teng. Tree-based parallel algorithm design. *Algorithmica*, 19(4):369–389, 1997.
[31] Mike Paterson and Leslie G. Valiant. Circuit size is nonlinear in depth. *Theoretical Computer Science*, 2(3):397–400, 1976.
[32] Walter L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3):365 – 383, 1981.
[33] Philip M. Spira. On time-hardware complexity tradeoffs for boolean functions. In *Proceedings of the 4th Hawaii International Symposium on System Sciences*, pages 525–527, 1971.
[34] Leslie G. Valiant, Sven Skyum, S. Berkowitz, and Charles Rackoff. Fast parallel computation of polynomials using few processors. *SIAM Journal on Computing*, 12(4):641–644, 1983.
[35] Heribert Vollmer. *Introduction to Circuit Complexity*. Springer, 1999.
[36] Wolfgang Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1992.

Received January 2018; revised January 2018; accepted January 2018