algorithms

MDPI

Article

Parsing Unranked Tree Languages, Folded Once †

Martin Berglund *,‡, Henrik Björklund ‡ and Johanna Björklund *,‡

Deptartment of Computing Science, Umeå University, 90836 Umeå, Sweden; henrikb@cs.umu.se
* Correspondence: mbe@cs.umu.se (M.B.); johanna@cs.umu.se (J.B.)
† This paper is an extended version of our paper published in International Symposium on Fundamentals of Computation Theory, Trier, Germany, 18–21 September.
‡ These authors contributed equally to this work.

Abstract: A regular unranked tree folding consists of a regular unranked tree language and a folding operation that merges (i.e., *folds*) selected nodes of a tree to form a graph; the combination is a formal device for representing graph languages. If, in the process of folding, the order among edges is discarded so that the result is an unordered graph, then two applications of a fold operation are enough to make the associated parsing problem NP-complete. However, if the order is kept, then the problem is solvable in non-uniform polynomial time. In this paper, we address the remaining case, where only one fold operation is applied, but the order among the edges is discarded. We show that, under these conditions, the problem is solvable in non-uniform polynomial time.

Keywords: transducers; trees; graphs; vector addition systems

Citation: Berglund, M.; Björklund, H.; Björklund, J. Parsing Unranked Tree Languages, Folded Once. *Algorithms* **2024**, *17*, 268. https://doi.org/10.3390/a17060268

Academic Editor: Qianping Gu

Received: 31 January 2024
Revised: 14 May 2024
Accepted: 5 June 2024
Published: 19 June 2024

1. Introduction

Graphs are one of the most commonly used data structures in computer science. Whether we are conducting social network analysis [1], defining the semantics of programming languages [2], or devising a better method for training deep neural networks [3], we are likely to operate on some form of graph representation. Practical applications of formal graph languages typically require that the parsing problem is efficiently solvable. This means that given a graph g, we can decide whether the graph adheres to the formalism in polynomial time, and also produce a certificate attesting the veracity of our decision. In the case of so-called order-preserving DAG grammars (OPDGs), for example, we can decide in linear time if a given graph g is well-formed with respect to a particular grammar G, and provide a unique derivation tree for g in G as proof [4].

Significant effort has been devoted to finding graph formalisms that combine expressiveness with parsing efficiency (see, e.g., [5–7]). Most of these are restrictions of hyperedge replacement grammars (HRGs) [8], a natural generalisation of context-free grammars, in which nonterminals are replaced by labelled hyperedges that provide restricted access to the intermediate graphs. The previously mentioned OPDGs are one of the most easily parsed restrictions of HRGs. Like many other graph formalisms, they were designed to be just strong enough to represent so-called abstract meaning representations (AMRs), a semantic representation based on a limited type of directed acyclic graphs. The principles underlying AMRs were introduced by Langkilde and Knight [9] based on a semantic abstraction language by Kasper [10]. The notion was refined and popularized by Banarescu et al. [11] and instantiated for a limited domain by Braune et al. [12].

At the more powerful end of the spectrum, we have s-grammars [7]. In this formalism, the terms over a small set of operators and a finite set of elementary graphs are evaluated in the domain of node-labelled graphs. The operators are defined modulo an auxiliary alphabet, and can merge nodes with identical labels, injectively rename nodes, and clear node names. The membership problem for HRGs and s-grammars require exponential time in general, and HRGs can generate languages for which the associated parsing problem

is NP-complete [13]. However, recent work has provided a parser generator [14–16] that, for certain subclasses of HRGs, yield a parser that runs in quadratic (and, in the common case, linear) time in the size of its input graph. The graph-parsing tool Grappa is available online [17].

In [18], the authors introduced *regular tree foldings*, a generative device consisting of a (finite representation of a) regular tree language and a new type of folding operation. The tree language is non-standard in that there is an auxiliary set of symbols Δ, which can be used to mark already labelled nodes, or to label nodes under which sits a single subtree. The folding operation then translates each tree t in the language into a graph by processing t bottom up: every time the evaluation reaches a node v with a label $\alpha \in \Delta$, it merges all nodes in the subtree sitting below v, which carries the mark α into a single node, and clears it from the mark α (see Figure 1). Just like OPDGs, regular tree-foldings are suitable for modelling AMRs, but unlike OPDGs, they can also accommodate cyclic graphs and more varied types of node-sharing. On the down-side, the parsing problem is more difficult: If the relative order among the edges attaching to a node is preserved by the folding, then parsing can be performed in polynomial time in the size of the input graph [19]. If, however, this order is relaxed so that the output graph is considered to be unordered, then the parsing problem is NP-complete [19].

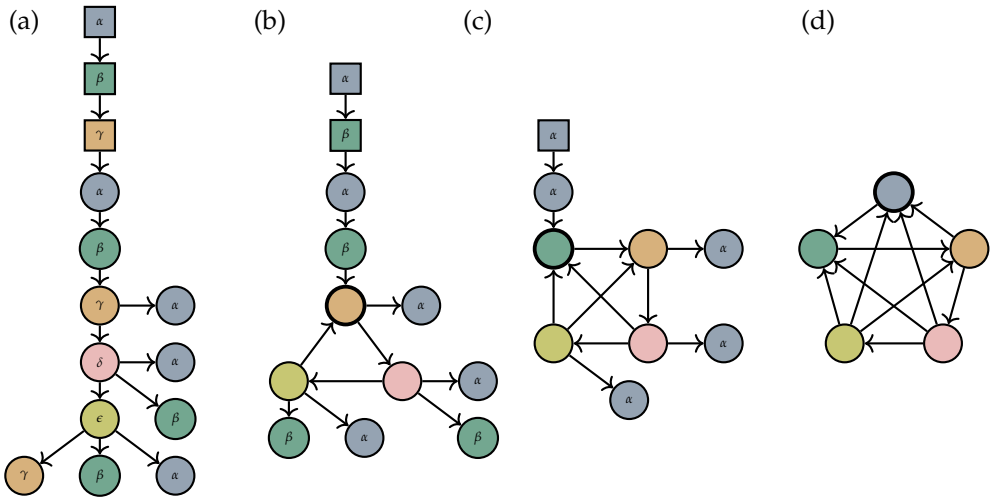

Figure 1. In the tree in Subfigure (**a**), round nodes with and without annotation denote a label in $\Sigma \times \Delta$ and Σ, respectively, and square nodes denote a label in $\Delta = \{\alpha, \beta, \gamma, \delta, \epsilon\}$. Arrows indicate edges, pointing from the source node of each edge to its target. When the tree is evaluated bottom-up, nodes with labels in $\Sigma \cup (\Sigma \times \Delta)$ are copied to the output graph until the transformation reaches a node with a label $\alpha \in \Delta$. Here, all nodes below with a label in $\Sigma \times \{\alpha\}$ are merged into a single node, which is assigned a label in Σ, and the square node labelled α is removed. The result is the graph in Subfigure (**b**). The process continues upwards until all Δ-labelled nodes have been cleared, yielding the graph in Subfigure (**d**). In each of the subfigures (**b**–**d**), the node arising from the most recent merger is indicated with a bold outline.

The proof of NP-completeness in [19] reduces the 3-PARTITION problem into a problem of parsing a regular tree language under two applications of the folding operation. However, the case where only one folding operation is used was left as an open problem. The original construction in [19] does not work, because it requires two fold operations to hide information on how to to partition a multiset into blocks, such that all have three elements and the same sum M. In this paper, we show that the parsing problem for unranked regular tree languages folded once under an order-cancelling fold semantics (see [19] for the contrasting order-preserving case) is solvable in non-uniform polynomial time, i.e., in polynomial time when considering the language as fixed.

Outline

This paper is organised as follows: Section 1 presents the background and motivation for regular tree foldings. Section 2 recalls the necessary parts of tree and graph theory, and fixes the notation. Section 3 presents the folding formalism in more detail. Section 4.1 sets the scene for the tree case by stating and proving a series of helpful lemmas on Vector Addition Systems (VASs). Section 4.2 contains the main result of this work, which is that regular tree languages folded once can be parsed in non-uniform polynomial time, even if the information about the relative order of sibling edges is lost in the folding.

2. Preliminaries

This section recalls relevant formal language theory and fixes notation. Central definitions have been lifted from the introduction of regular tree foldings in [18,19].

The set of non-negative integers is denoted by $\mathbb{N}$. For $i \in \mathbb{N}$, $[i] = \{1, \dots, i\}$ and $[0] = \emptyset$. A *multiset* S' is a set in which elements can have multiple instances.

Graphs: To allow parallel edges, we define our graphs in terms of source and target mappings. An *alphabet* is a finite nonempty set of symbols. Let Σ be an alphabet. A (directed and rooted) *graph* (note that graphs of this type are usually referred to as *rooted multigraphs* in the literature, but since they are the only kind that we will be concerned with, we opt to simply refer to them as 'graphs') over Σ is a tuple $g = (V, E, src, tar, lab, r)$ consisting of: finite sets V and E of *nodes* and *edges*, respectively; *source* and *target mappings* $src\colon E \to V$ and $tar\colon E \to V$ assigning to each edge e its source $src(e)$ and target $tar(e)$; a *labelling* $lab\colon V \to \Sigma$; and a *root* node $r \in V$. The set of *incoming edges* of $u \in V$ is $in(u)$, and the set of *outgoing edges* is $out(u)$. The *in-degree* of u is $|in(u)|$, and the *out-degree* of u is $|out(u)|$. A node with an out-degree 0 is a *leaf*. The *size* of the graph g is $|g| = |V| + |E|$. The set of all graphs with node labels in Σ is denoted by $\mathbb{G}_\Sigma$. We write $root(g)$ for the root node r, and $nodes(g)_\sigma$ for the set of nodes in g that are labelled $\sigma \in \Sigma$.

An *edge-ordered graph* over Σ is a pair $(g, <)$, where $g = (V, E, src, tar, lab, r)$ in $\mathbb{G}_\Sigma$ and $<$ is a binary relation on E that becomes a total order when restricted to any set $in(v)$ or $out(v)$, for $v \in V$. From here on, we leave the second component implicit and refer to it as $<_g$ when needed. The set of all edge-ordered graphs over Σ is denoted by $\mathbb{G}^<_\Sigma$.

A *path* in the graph $g = (V, E, src, tar, lab, r)$ is a finite and possibly empty sequence $p = e_0 e_1 \cdots e_k$ of edges such that, for each $i \in [k]$, the target of e_{i-1} is the source of e_i. Here, we say that p is a path from $src(e_0)$ to $tar(e_k)$. The path p is a *cycle* if $src(e_0) = tar(e_k)$.

Trees: Let Σ be an alphabet. An *ordered unranked tree over* Σ is a tuple $t = ((V, E, src, tar, lab, r), <) \in \mathbb{G}^<_\Sigma$, such that (i) t is connected, (ii) r has no incoming edges, and (iii) every node except for r has exactly one incoming edge. We denote the set of all ordered unranked trees over Σ by T_Σ.

Let $\sigma \in \Sigma$, $k \in \mathbb{N}$, and $t_1, \dots, t_k$ be trees over Σ. Moreover, for each such $t_i = (g_i, <_i)$, let $g_i = (V_i, E_i, src_i, tar_i, lab_i, v_i)$, where the node sets V_i, $i \in [k]$, are mutually disjoint, and similarly for the edge sets E_i, $i \in [k]$. The *top-concatenation* of $t_1, \dots, t_k$ with σ, denoted by $\sigma[t_1, \dots, t_k]$, is the tree obtained by attaching the trees $t_1, \dots, t_k$ as children underneath a new root node with label σ.

Top-concatenation is analogously defined for single-rooted graphs, so we may write $\sigma[g_1, \dots, g_k]$ without risk of confusion. In the case that the graphs are ordered, so are the edges $e_1, \dots, e_k$ that attach the subgraphs $g_1, \dots, g_k$; otherwise, they are unordered.

Let X be a set of variables, such that $X \cap \Sigma = \emptyset$. A *context* is a tree $c \in \mathrm{T}_{\{x\} \cup \Sigma}$, for $x \in X$, such that c contains exactly one occurrence of x, and this occurrence is a leaf. Given such a context and a tree t, we let $c[\![t]\!]$ denote the tree obtained from c by replacing the node labelled x with t. Formally, $c[\![t]\!] = t$ if $c = x$, and otherwise $c[\![t]\!] = \sigma[s_1, \dots, s_{i-1}, s_i[\![t]\!], s_{i+1}, \dots, s_n]$, where $c = \sigma[s_1, \dots, s_n]$ and $s_i \in \mathrm{T}_{\{x\} \cup S}$ is the unique context among $s_1, \dots, s_n$.

Automata for unranked trees: Unranked tree languages have been studied since the 1960s, most notably as a formal model for the markup language XML [20,21]. In this article, we use Z-automata [22], an extension of step-wise automata [23], to represent such

languages. A *Z-automaton* is a tuple $A = (\Sigma, Q, R, F)$ consisting of a finite *input alphabet* Σ; a finite set Q of *states* which is disjoint from Σ; a finite set R of *transition rules*, each of the form $s \to q$ consisting of a left-hand side $s \in \mathrm{T}_{\Sigma \cup Q}$ and a right-hand side $q \in Q$; and a finite set $F \subseteq Q$ of accepting states. Henceforth, we write m for the number of states $|Q|$.

Let $t \in \mathrm{T}_{\Sigma \cup Q}$. A transition rule r of the form $s \to q$ is *applicable* to t, if t can be written as $t = c[\![\sigma[t_1, \ldots, t_n]]\!]$, such that $s = \sigma[t_1, \ldots, t_k]$ for some $k \leq n$. If so, then there is a *computation step* $t \to_A c[\![q[t_{k+1}, \ldots, t_n]]\!]$. A *computation* of A on a tree $t \in \Sigma$ is a sequence of computation steps $t \to_A^* q$, for some $q \in Q$. The computation is *accepting* if $q \in F$. A tree $t \in \mathrm{T}_\Sigma$ is *accepted*, or *recognised*, by A if there is an accepting computation of A on t. The *language* accepted by A, denoted by $\mathcal{L}(A)$, is the set of all trees in T_Σ that A accepts.

As shown in [22], Z-automata recognise the same class of languages as unranked tree automata [20]. We use a normal form, in which all transition rules are of the form $\sigma \to r$, $\sigma[q] \to r$, or $p[q] \to r$, for $\sigma \in \Sigma$ and $p, q, r \in Q$.

A run of a Z-automaton $A = (\Sigma, Q, R, F)$ in normal form on a tree t can be seen as assigning states to $nodes(t)$. For each state (q, σ) in $Q \times \Sigma$, there is a regular language $r_{q,\sigma}$ such that in a run, the sequence w of states assigned to the children of a node that has label σ and is assigned q must be a string in $r_{q,\sigma}$ [22]. We denote by $parikh(q, \sigma)$ the Parikh image [24] of the language $r_{q,\sigma}$.

3. Regular Tree Foldings

The purpose of the folding operation is to turn an unranked tree t over an alphabet Σ into a graph g by merging nodes. The folding is performed by marking nodes with symbols from an auxiliary alphabet Δ, meaning that some nodes will have labels in $\Sigma \times \Delta$, and then, for each $\alpha \in \Delta$ merging all nodes marked with symbols in $\Sigma \times \{\alpha\}$ into a single node with a label in Σ. The reason for removing the folding symbol of the node resulting from a merging is to prevent it from being folded together with other nodes later on in the process, which would complicate parsing. By itself, this formalism can only produce output graphs where, at most, $|\Delta|$ nodes have more than one incoming edge. For this reason, the merging is divided into *scopes* by allowing for nodes in t that have labels from Δ: a node u in t labelled $\alpha \in \Delta$ is an instruction that all nodes with labels in $\Sigma \times \{\alpha\}$ *below it* are to be merged, whereupon the node u itself is to be deleted. The tree is then evaluated bottom-up, so that Δ-labelled nodes lower in the tree have their corresponding operations performed earlier. To keep the result well-defined, the tree language must force nodes with labels in Δ to have exactly one direct subtree. The combination of a regular unranked tree language over $\Sigma \cup (\Sigma \times \Delta) \cup \Delta$ and a fold operation over Δ is called a *regular tree folding*.

The folding operation is illustrated in Figure 1. In the original definition [18], the label of the merged node v is chosen non-deterministically based on the labels of the nodes that went into the merger. However, there is a (less compact) normal form of regular tree foldings, which only merges nodes if they share the same label $(\sigma, \alpha) \in \Sigma \times \Delta$, and the resulting node in the output graph is then labelled σ [19]. This paper is only concerned with the case where there is a single folding symbol, and the normal form allows us to assume that there is a unique $\langle \sigma, \alpha \rangle \in \Sigma \cup \Delta$ that is allowed by the regular tree language.

Thus, throughout this paper, let Σ be an alphabet, let $\sigma \in \Sigma$, and let α be a special symbol not in Σ. We write Γ for the alphabet $\Sigma \cup \{\langle \sigma, \alpha \rangle, \alpha\}$.

Definition 1 (The fold operation F)**.** *The function* $[\![\alpha]\!] \colon \mathbb{G}_\Gamma \to \mathbb{G}_\Gamma$ *takes a single-rooted input graph* $g = (V, E, src, tar, lab, r) \in \mathbb{G}_\Gamma$ *and computes an output graph* $h = [\![\alpha]\!](g)$ *by merging the set of nodes* $nodes(g)_{\langle \sigma, \alpha \rangle}$ *into a single node* u*, and assigning* u *the label* σ*. If* $root(g) \in nodes(g)_{\langle \sigma, \alpha \rangle}$*, then* $root(h) = u$*; otherwise* $root(h) = root(g)$*. The* fold operation $F : \mathrm{T}_\Gamma \to \mathbb{G}_\Gamma$ *is defined for every tree* $t = \gamma[t_1, \ldots, t_k] \in \mathrm{T}_\Gamma$ *by* $F(t) = [\![\gamma]\!](F(t_1))$ *if* $\gamma = \alpha$ *(when* k *is always* 1*), and* $\gamma[F(t_1), \ldots, F(t_k)]$ *otherwise. It is extended to sets of trees in the expected way: for* $L \subseteq \mathrm{T}_\Gamma$*,* $F(L) = \bigcup_{t \in L} \{F(t)\}$.

By combining the fold operation with a regular tree language, we reach our formalism of interest.

Definition 2 (Regular tree folding (with one folding symbol))**.** *A regular tree folding (RTF) over* Γ *is defined through a Z-automaton* A *over the same alphabet, such that, for every* $t \in \mathcal{L}(A)$ *and every node* v *in* t*, it holds that if the label of* v *is* α*, then* v *has exactly one direct subtree. The folded graph language with respect to* A *is* $\mathcal{L}_F(A) = F(\mathcal{L}(A))$*.*

Example 1. *Figure 2 shows two trees annotated with folding symbols, along with the corresponding graphs they fold into. In the first tree (located on the left in the illustration), the two nodes labelled* $\alpha \in \Delta$ *(represented as blue squares) appear side-by-side. There is no interaction between their scopes: removing either of these nodes would not affect the outcome of the application of the other. Moving to the second tree (two steps to the right in the same illustration), the lower square node labelled* α *shadows the scope of the upper one. Had it not been present, then all blue round nodes would have been merged into a single node.*

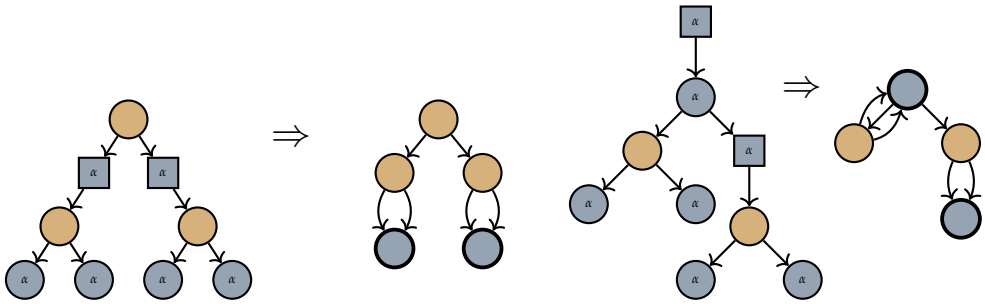

Figure 2. In the above figure, we see two examples of trees decorated with folding symbols and the graphs they fold into. As in Figure 1, round nodes with and without annotations denote a label in $\Sigma \times \Delta$ and Σ, respectively, and square nodes denote a label in Δ. Nodes that result from mergings are indicated with a bold outline.

The remainder of this paper is devoted to the membership problem for a fixed RTF represented by a Z-automaton A over Γ. It asks: *given a graph* g*, is* g *in* $\mathcal{L}_F(A)$*?* In the special case where folding is only applied once, the problem can be restated as one of combining tree fragments into a single tree in a target language. From here on, $x \notin \Gamma$ is a fixed but arbitrary variable symbol.

A *tree fragment* is an unordered, unranked tree t with the following properties: (i) The root has exactly one child. (ii) Some leafs may have label x, while all other nodes have labels from Γ. We call the unique child of the root the *prior* of t, denoted by $prior(t)$.

A *substitution* is an operation that takes a tree or a tree fragment t and $k = |nodes(t)_x|$ sets $T_1, \ldots, T_k$ of tree fragments. It assigns a unique set T_i to each $v_i \in nodes(t)_x$. For each i, the roots of the fragments in T_i are then identified with v_i. Finally, v_i is labelled $\langle \sigma, \alpha \rangle$.

We can thus view any tree as composed from a tree and a number of tree fragments through substitution. Taking this idea further, we note that a single application of a tree folding has the effect of turning the input tree into a set of tree fragments, with all but at most one rooted at the merged node, and some of which have leaves attached to the merged node (see Figure 3). The merged nodes hide how these tree fragments originally fit together, and solving the membership problem is tantamount to recapturing this information. In the following specialisation of the membership problem for the case of single foldings, we denote by $order(t)$ the set of ordered trees that can be obtained from an unordered tree t by attaching an order to its edges.

Remark 1. *Constructing the tree fragments is trivial in the interesting cases, i.e., the graph will have a single node, which is obviously the merged node, as it has more than one incoming edge. The tree fragments are then obtained by giving each edge incident with the merged node its own copy of that node, as is shown in Figure 3. The other cases, where zero or one node is "merged",*

can easily be avoided by rewriting the automaton. That is, we use the states to track and verify a nondeterministic guess whether zero, one, or more than one, node(s) will become merged by a folding operator. For zero, we skip generating the folding operator node (which would do nothing); for one node, we instead generate it with its resulting label and *inhibit generating the folding operator node. For two or more, we simply operate the same as the original automaton (but check the guess). Refer to Lemma 4.1 in [19] for a detailed construction easily modified for this case. We begin by dealing with the case of only a single fold, but in Corollary 1, we sketch the straightforward steps needed to reintroduce multiple folds.*

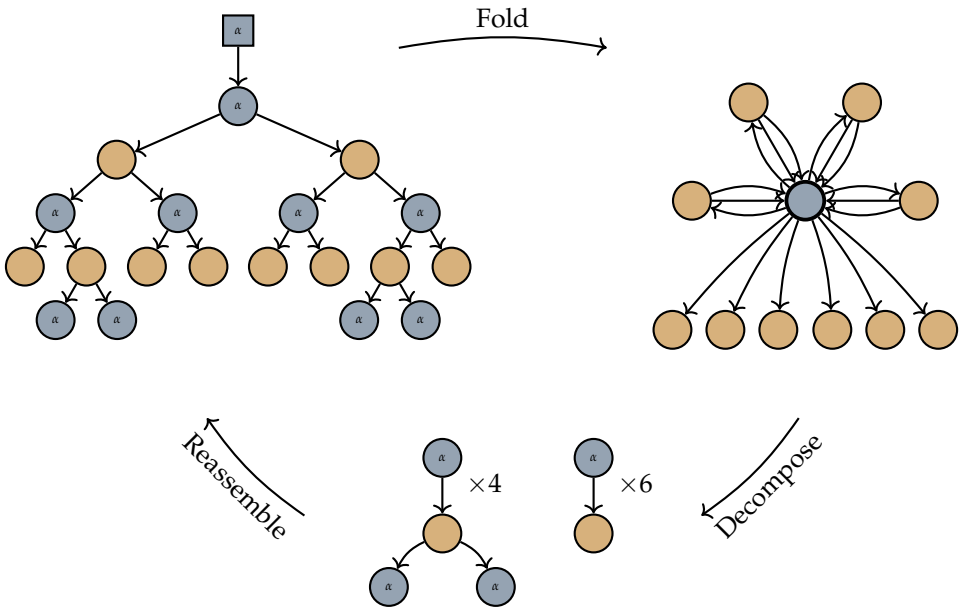

Figure 3. To parse a folded graph (**top right**), we first decompose it into a number of tree fragments attaching to the merged node (**bottom row**), and then search for a way of reassembling the fragments into a tree in the folded tree language (**top left**). The single node arising from merging is indicated with a bold outline.

Given the above, we can give a slightly different (and, for our, proofs more convenient) statement of the membership problem we are investigating the following.

Definition 3 (Membership problem for one folding)**.** *Let A be a fixed but arbitrary Z-automaton over Γ. Given a multiset of unordered tree fragments $\{T_1, \ldots, T_n\}$, is there a sequence of substitutions that uses each tree exactly once, and produces a tree t, such that $\alpha[t'] \in \mathcal{L}(A)$ for some $t' \in order(t)$?*

4. Unfolding Folded Trees

Since the input trees we are working with are unordered (see Definition 3), we extend the behaviour of the Z-automaton A to the unordered case.

Definition 4 (Unordered runs)**.** *A run of a Z-automaton $A = (\Sigma, Q, R, F)$ on an unranked, unordered tree $t = (V, E, src, tar, lab, r)$ is a mapping $\rho : nodes(t) \to Q$ such that for each node v, the states assigned to the children of the node v, when viewed as a multiset, belongs to $parikh(\rho(v), lab(v))$. For a tree fragment s, a* partial run *is a run, except that the condition on the children does not apply to the nodes in $nodes(s)_x$.*

We can now define the *signature* of a tree fragment in terms of which partial runs it can realize. The intuition is that, for each x-labelled node and possible assignment of a state to it, we have to find a set of trees to attach that can evaluate to states in the appropriate Parikh image.

Definition 5 (Signature). *Let $A = (\Sigma, Q, R, F)$ be a Z-automaton with $m = |Q|$, and let t be a tree fragment with $k = |nodes(t)_x|$. The* signature *of t with respect to A, denoted by $sig(t)$, is a set of tuples of the form (q, S), where $q \in Q$ and S is a multiset of elements from Q, defined as follows. Let $v_1, \ldots, v_k$ be the nodes in $nodes(t)_x$. Then, (q, S) belongs to the signature iff there is a partial run ρ on t and a partitioning of S into $S_1, \ldots, S_k$, such that $\rho(prior(t)) = q$ and, for each i, $S_i \in parikh(\rho(v_i), \langle \sigma, \alpha \rangle)$.*

The intuition of the above definition is that (q, S) is in the signature of t if t can "accept" $|S|$ tree fragments whose priors have been assigned the states in S and then "deliver" a state q at its prior.

Given the input set $\{T_1, \ldots, T_n\}$, we only need to consider a polynomial number of signatures. Since there are n input trees, we only consider signatures where $|S| \leq n$. The number of such signatures is bounded from above by mn^m. In other words, the number of possible signatures for all input trees is polynomial. The signature for each input fragment can be computed in polynomial time using a CYK-like dynamic programming algorithm.

Given a set of tree fragments, we compute their signatures, and then reassemble them as in Figure 3, leading to the final theorem.

Theorem 1. *The non-uniform membership problem for tree languages folded once under an order-cancelling semantics is decidable in polynomial time.*

Before proving Theorem 1 in Section 4.2, we recall the definition of vector addition systems and prove some properties of these that will turn out to be useful.

4.1. Reassembly Sequences by Vector Addition Systems

To prove Theorem 1, we rely on the signatures to tell us what multisets of states each tree fragment can "consume", and what state it then "produces". Finding a way of puzzling the fragments together consistently is a combinatorial problem which we will solve by reducing it to reachability in a restricted form of vector addition systems [25] (the restrictions are key as vector addition systems are in general very powerful [26]). We next present these systems and prove the relevant complexity bound. We then explain the reduction in the proof of Theorem 1.

For vectors u and v, let $(u; v)$ denote their concatenation. Let $\mathbb{0}$ denote the set of all vectors of zeros, and for all $k \geq 1$, let $\mathbb{1}_{[k]} = \{(z_1; 1; z_2) \mid z_1, z_2 \in \mathbb{0}, |z_1| = k - 1\}$, i.e., the unit vectors with a 1 in position k. Let $\mathbb{1} = \cup_{k \geq 1} \mathbb{1}_{[k]}$, i.e., all unit vectors. We may treat $\mathbb{0}$ and $\mathbb{1}$ as vectors when the length is implied by context. For a vector s, we write $s \geq \mathbb{0}$ to indicate that s is pointwise greater than or equal to $\mathbb{0}$, i.e., that every element of s is nonnegative.

Definition 6. *A Vector Addition System of dimension $k \in \mathbb{N}$ (a k-VAS) is a finite set $V = \{(p, p') \mid p \in \{-v \mid v \in \mathbb{N}^k\}, p' \in \mathbb{N}^k\}$. We call these the* operations. *An operation sequence in V is denoted $s_0 \to_{(p_1, p_1')} s_1 \to_{(p_2, p_2')} \cdots \to_{(p_n, p_n')} s_n$ for $s_0, \ldots, s_n \in \mathbb{N}^k$, and for each $1 \leq i \leq n$*

$$1.\ (p_i, p_i') \in V, \qquad 2.\ s_{i-1} + p_i \geq \mathbb{0}, \qquad \text{and,} \qquad 3.\ s_i = s_{i-1} + p_i + p_i' \ .$$

A vector s_n is reachable from s_0 if and only if such an operation sequence exists.

For any $k, l \in \mathbb{N}$, we call a $(k + l)$-VAS V a (k, l)-VAS to differentiate the first k elements of the vectors from the rest, writing $((u; v), (u'; v')) \in V$ to signify that $|u| = |u'| = k$ and $|v| = |v'| = l$.

While we define reachability in terms of going from a vector s to a vector t, we are primarily interested in the special case of $\mathbb{0}$ being reachable, since the numbers will represent tree fragments, all of which must be used.

Definition 7. *A (k, l)-VAS V is* metered *if all $((s; b), (s'; b')) \in V$ have $s, b \in -\mathbb{1}$ and $b' \in \mathbb{0}$.*

That is, in a metered (k,l)-VAS, every operation takes precisely one unit from each of the two parts of the vector, and never adds anything to the second part. We will use the first part, the s vector, to represent a *multiset of states*, while the second, the b vector, represents a *budget*. We will informally refer to the vectors accordingly where it makes things more intuitive. This bounded budget then limits the computation in a way similar to a k-bounded Petri net [27].

Definition 8. *For a (k,l)-VAS V an operation sequence $(s_0;b_0) \to_{p_1} \cdots \to_{p_n} (s_n;b_n)$* visits a position $i \in [k]$ *if at least one vector $s \in \{s_0,\ldots,s_n\}$ is nonzero in position i. The set $I \subseteq [k]$ is visited if all elements are visited.*

Visits only concern the first (state) part of the vector, which makes sense for a metered VAS, as a position in the second part is visited iff it is visited in b_0. We will use *visits* to correspond to uses of states in a run of an automaton over a tree.

We next show that if there exists an operation sequence which visits a set I starting from a vector v_0, then there is a *short subsequence* that does the same.

Lemma 1. *For any metered (k,l)-VAS V and operation sequence $v_0 \to_{p_1} \cdots \to_{p_n} v_n$ which visits I, there exists an operation sequence $v'_0 \to_{p'_1} \cdots \to_{p'_\ell} v'_\ell$ such that: (i) this sequence also visits I; (ii) $v'_0 = v_0$; (iii) the sequence $p'_1 \cdots p'_\ell$ forms a subsequence of $p_1 \cdots p_n$; (iv) and $\ell \leq k|I|$.*

Proof. Let $(s_0;b_0) \to_{p_1} \cdots \to_{p_n} (s_n;b_n)$ be an operation sequence visiting I, for an arbitrary $n \in \mathbb{N}$. Let us abbreviate it as $p_1 \cdots p_n$, leaving the vectors implicit. First, consider the singleton case where $I = \{i\}$, when the following procedure constructs the indicated subsequence. Define $P(p_1 \cdots p_n, i)$ recursively as:

1. Let j be the smallest index such that, with $(s_0;b_0) \to_{p_1} \cdots \to_{p_j} (s_j;b_j)$, the vector s_j is nonzero at position i.
2. If $j = 0$, return the empty sequence.
3. Letting $p_j = ((v_1;v_2),(v'_1;v'_2))$, take i' to be the unique (as V is metered) position in v_1, which equals -1.
4. Return $P(p_1 \cdots p_{j-1}, i') \cdot p_j$. That is, the sequence constructed by finding a short visit of i' in steps 1 through $j-1$ (one must exist as step p_j subtracted 1 from that position) followed by the operation p_j (which visits i).

Now, $P(p_1 \cdots p_n, i)$, starting from $(s_0;b_0)$, forms an operation sequence of a length of at most k, which visits i. The visit to i is straight-forward (the final "p_j"), it is of a length of at most k, as each level of the recursion has a distinct i (as the current one is eliminated from the sequence used in the recursive call), and adds one operation. Finally, it is an operation sequence, as each p_j appended in step 4 has its required visit provided by the step immediately prior. The budget part of the vector is unproblematic, as it can only be increased by shortening the sequence.

Generalizing this to an arbitrary set $I \subseteq [k]$ amounts to an iteration of the above argument. Here, elements of I may compete for the same operations, but this can be handled by having Step 1 in the above procedure pick nondeterministically among the k first such indices j, and then letting the computation fail if two elements of i reuse an operation. In the worst case, each $i \in I$ introduces k new operations to the subsequence, for a total of $k|I|$ operations. □

Next, we define a relaxed operation sequence, which contains less information, but the existence of which implies the existence of an operation sequence.

Definition 9. *For a metered (k,l)-VAS V, a vector $v \in \mathbb{N}^{k+l}$, and any $I \subseteq [k]$, a* destructured sequence *is a tuple (v,I,P,S) where $P = p_1 \cdots p_{k|I|} \in V^*$, where*

1. *$v \to_{p_1} \cdots \to_{p_{k|I|}} v'$ is an operation sequence visiting I;*

2. *S is a multiset over V such that for all $((s;b),(s';b')) \in S$ the vectors s and s' are zero in all positions $z \notin I$;*
3. $v + \sum_{(s,s')\in S}(s+s') + \sum_{(p,p')\in P}(p+p') = \mathbb{0}$.

The key then becomes that we only need to consider destructured sequences to demonstrate reachability, as they turn out equivalent to full operation sequences in this setting. That is, past a certain length and ensuring a certain subsequence exists, we can disregard the order of operations.

Lemma 2. *For a metered (k,l)-VAS V and vector $v \in \mathbb{N}^{k+l}$, there exists an operation sequence $v \to \cdots \to \mathbb{0}$ iff there exists a destructured sequence (v, I, P, S).*

Proof. The "only if" direction: Let $v \to_{p_1} \cdots \to_{p_n} \mathbb{0}$ be an operation sequence, and take $I \subseteq [k]$ be the largest set visited by this sequence. Then, apply Lemma 1 to construct P, and let $S = \{p_1, \ldots, p_n\} \setminus P$. Then, (v, I, P, S) is a destructured sequence, fulfilling the requirements of Definition 9: Cond. 1 by Lemma 1, Cond. 2 by construction, and, Cond. 3 as the original sequence reaches $\mathbb{0}$, so the sum across all its operators, plus the initial vector v, has to be zero.

The "if" direction: given a destructured sequence (v, I, P, S), we can construct an operation sequence which reaches $\mathbb{0}$ from v. The only thing keeping any particular sequencing of the operations P and S from being an operation sequence from v to $\mathbb{0}$ is Condition 2 in Definition 6. That is, some position in the first (state) part of the vector may turn negative during the application of an operation. Only the first (state) part going negative can cause problems, as the second (budget) part is used up correctly by all orders, and all orders reach $\mathbb{0}$ by the definition of a destructured sequence. We now demonstrate how to intersperse the operations in S in P to create a valid operation sequence reaching $\mathbb{0}$ from v.

This is sketched in Figure 4, and is performed in two phases.

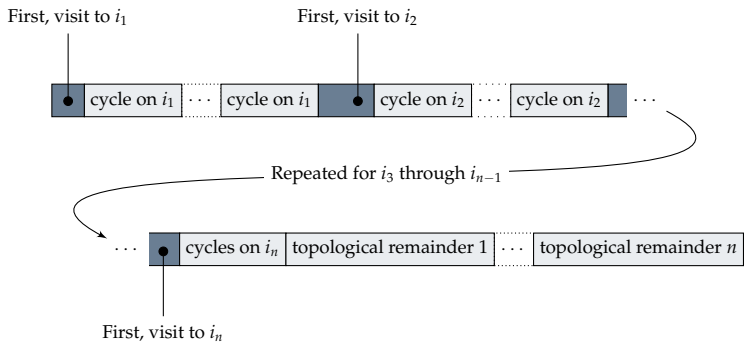

Figure 4. The derivation of an operation sequence from a destructured sequence (v, I, P, S). The shaded parts indicate the initial sequence P. Let $I = \{i_1, \ldots, i_n\}$. After the first visit to a position i in P, all cycles on i are inserted, which can be made using operations in S, iterating this for all positions until the remainder of S is free of cycles. Then, the non-cyclic suffix of S is placed at the end of this sequence in a way determined by a topological sort of operation, iteratively appending all operations that require a visit not made in the remainder of S.

Phase 1: Place cycles at first visit. An ordered submultiset $\{p_1, \ldots, p_n\} \subseteq S$ is a cycle on i if there are vectors $s, t \in \mathbb{N}^k$, $s', t' \in \mathbb{N}^l$, such that $(s;s') \to_{p_1} \cdots \to_{p_n} (t;t')$ with $s \in \mathbb{1}_{[i]}$, $t \geq s$. If such a cycle exists, split $P = P_1 P_2$ such that P_1 ends with the first visit to i (must exist by the definition of a destructured sequence), and construct a new operation sequence $P_1 \cdot p_1 \cdots p_n \cdot P_2$, and a new set of operations $S \setminus \{p_1, \ldots, p_n\}$. Take these to be P and S and iterate this procedure until no cycles remain in S. This retains all the properties of the destructured sequence (v, I, P, S), fulfilling all the conditions except the length of P. Specifically, the spliced sequence is an operation sequence because the visit to i allows it to be applied (the budget as usual irrelevant to the reordering), and as $t \geq s$ it cannot cause any later operation to fail.

Phase 2: Order remainder topologically. If S is cycle-free then there is some i such that every $((s;b),(s';b')) \in S$ has s' zero in position i. Otherwise, a visit to i can be made by some operation that requires a visit to i', but i' can be visited by some operation that requires a visit to i'', etc. But no position may repeat in this chain, as that would be a cycle, and there are only finitely many positions, which causes a contradiction. Pick such an i, let $\{p_1, \ldots, p_n\} \subseteq S$ be all such operations p which have $p = ((-\mathbb{1}_{[i]};b),(s';b'))$ (for any b, s' and b'), and construct a new sequence $P \cdot p_1 \cdots p_n$ and a new multiset $S \setminus \{p_1, \ldots, p_n\}$. Take these to be P and S, and iterate this procedure until S is empty. This produces an operation sequence, since we maintain the destructured sequence invariant that summing P, S and v produces $\mathbb{0}$, and as the i picked in each step is not generated by any operation in S, all needed visits must already be in P. After these steps, S is empty and P is an operation sequence reaching $\mathbb{0}$ from v. □

Theorem 2. *For a metered (k,l)-VAS V and vectors $s_0 \in \mathbb{N}^k$ and $b_0 \in \mathbb{N}^l$, it is decidable in time $\mathcal{O}(|V|^{k^2+1}(\sum b_0))$ as to whether $\mathbb{0}$ is reachable from $(s_0;b_0)$.*

Proof. All relevant tuples (v,I,P,S) can be evaluated to see if they are a destructured sequence, as in Definition 9; if there is one, then by Lemma 2, there exists an operation sequence reaching $\mathbb{0}$ from v, which is the definition of reachability.

First, for a vector v, define the *next smaller* vector v' as the one formed by decrementing the *first* nonzero element of v (for example, for $(0,1,2)$ the next vector is $(0,0,2)$, and the next smaller of that is $(0,0,1)$).

To see that the bound holds, regard what tuples we need to test. There are $|V|^{k^2}$ ways of picking P by the bound on its length, and I is entirely determined by P. Try the following for all such P.

Given v, I and P, compute the v' reached from v using the operations in P. Then, construct S by searching for a path from v' to $\mathbb{0}$ in the following graph.

1. The vector v' is a node in the graph.
2. If the vector $(s;b)$ is a node in the graph and there exists an operation $p = ((u;v),(u';v')) \in V$ such that:
   - $s' = s + u + u'$, $b' = b + v + v'$ (vector elements can be negative);
   - The next smaller vector than b is b';
   - u and v are zero in all positions not in I.

   then there is a node $(s';b')$ and an edge from $(s;b)$ to $(s';b')$ labelled p.

Then, let S be a multiset of operations used (i.e., edges traversed) finding a path from v' to $\mathbb{0}$ in this graph. This procedure is sound and complete.

- An S found this way does make (v,I,P,S) a valid destructured sequence, as by construction $v + \sum_{(s,s')\in S}(s+s') + \sum_{(p,p')\in P}(p+p') = \mathbb{0}$ and S only visits I. All other needed properties derive from an exhaustive enumeration of possible operation sequences P.
- If a destructured sequence does exist, this procedure will find it. Note that the only real pruning happening is requiring the budget to decrease according to the *next smaller* order. This is necessary to limit the effect l has on the size of the graph, but as S is itself unordered requiring the budget to be used in a certain order is not a real restriction.

Finally, all paths in this graph is of length at most $\sum b_0$ and each node has, at most, $|V|$ outgoing edges. Combining trying all P with exhaustive search on the graph gives a bound of $\mathcal{O}(|V|^{k^2})\mathcal{O}(|V|(\sum b_0)) = \mathcal{O}(|V|^{k^2+1}(\sum b_0))$. □

4.2. Combining the Pieces

With all pieces in place, we can prove that the non-uniform membership problem for regular tree languages folded once is efficiently decidable. We also provide, as a corollary, a slight generalisation of the result, showing that it is enough to assume that there is only one folding symbol, even if it occurs more than once in the trees.

Proof of Theorem 1. By Definition 3, we have a fixed Z-automaton A and are given an input a graph, which we can decompose into a set of tree fragments $T_1, \ldots, T_n$. Assume that the root node was folded, i.e., the graph has no node with zero incoming edges. This causes no loss of generality: if the graph has a distinguished root node r, give it a parent marked by a new symbol "*dummy*", and give that node an incoming edge from the folded node. Then, modify A with the necessary additional transitions (such that where it would have previously accepted $\alpha[t]$, it now accepts $\alpha[\langle\sigma, \alpha\rangle[dummy[t]]]$.) Let $Q = \{q_1, \ldots, q_m\}$ be the states of A, and assume, without loss of generality, that q_m is the only accepting state, and that it occurs on no left-hand-side of a rule in A.

Construct the signatures $\mathrm{sig}(T_1), \ldots, \mathrm{sig}(T_n)$ and, from these, construct a metered $(m, n+1)$-VAS V by giving it precisely the following operations: (i) for each $i \in [n]$ and $(q_j, S) \in \mathrm{sig}(T_i)$, V has the operation $((-\mathbb{1}_{[j]}; -\mathbb{1}_{[i]}), (\bar{S}; \mathbb{0}))$, where $\bar{S}$ is S turned into a vector of length m, letting position k be the number of occurrences of q_k in S; and (ii) for each S, such that $S \in \mathit{parikh}(q_m, \langle\sigma, \alpha\rangle)$ and $|S| \leq n$, V has the operation $((-\mathbb{1}_{[m]}; -\mathbb{1}_{[n+1]}), (\bar{S}; \mathbb{0}))$. Finally, the initial vector is $v = (s; b)$, where $s \in \mathbb{1}_{[m]}$ (with $|s| = m$) and $b = 1 \cdots 1$. Intuitively, V simulates reassembling the tree from the fragments. In each step, there is a current vector $(s'; b')$, where s' describes the multiset of states which still need to be replaced by a tree fragment, and b describes which tree fragments have already been used. Operations of type (i) attach a tree fragment using one of the present states, where type (ii) initializes the multiset of states to one from which A can accept by going to q_m.

Then, $\mathbb{0}$ is reachable from $(s; b)$ if and only if the tree fragments can be reassembled into a tree accepted by A. By induction on the length of a VAS operation sequence $(s; b) \to \cdots \to \mathbb{0}$, relating each step to a part of some final tree t such that $\alpha[t'] \in \mathcal{L}(A)$ for some $t' \in \mathit{order}(t)$. That is, the first step (the only operation of type (ii) by construction) establishes the root and a multiset of states the children must produce. The second operation attaches some tree fragment T_i as one of those children by: picking some $(q, S) \in \mathrm{sig}(T_i)$, removing one q from the state multiset represented, removing the tree fragment itself from the budget, and providing a new set of unaccounted-for children with state multiset S. This maintains the invariant that the part of the tree already constructed can be accepted by A given that the multiset of states currently tracked are provided by the remainder of the procedure. Since $\mathbb{0}$ is reached no further states are needed, and all tree fragments have been placed.

Finally, we can check whether $\mathbb{0}$ is reachable from $(s; b)$ by applying Theorem 2, observing that this $(m, n+1)$-VAS has m constant (as A is assumed fixed) and both n and $|V|$ polynomial. Substituting these into the bound of Theorem 2 yields a polynomial bound. Observe the role Lemma 2 and Theorem 2 play here; in effect, the destructured sequence corresponds to constructing a small tree t, which visits all necessary states without exhibiting any loop. Once this is in place, the remaining tree fragments can be added without keeping record of precisely where they are placed, producing a proper tree. □

We have thus shown that for a fixed regular tree language $\mathcal{L}$, the question of whether a graph g could have been produced by a single application of the order-cancelling fold operation on a tree in $\mathcal{L}$ is solvable in polynomial time. As it turns out, by cutting the graphs up into parts, we can extend this to any number of folding operation applications, as long as they use the same folding symbol.

Corollary 1. *The non-uniform membership problem for tree languages folded using only a single folding symbol under an order-cancelling semantics is decidable in polynomial time.*

Proof. This generalization of Theorem 1 follows from a helpful separability of graphs using a single folding symbol: if the graph contains no edge that would bisect the graph if removed, then the graph contains at most one folded node, i.e., it has at most one node that is the result of the merging two or more tree nodes.

This must be the case because if there is more than one folded nodes, then the corresponding nodes in the tree which was folded must have been in scope of *different instances* of the folding operator (or else they would have *all* been merged into one). Pick (one of)

the folding operators which are the furthest from the root of the tree, and consider the edge which connects it to its parent. Removing that edge from the resulting graph will necessarily bisect the graph, because no node below that folding operator can be merged with a node outside of that subtree.

Observe that, by Remark 1, we can, without loss of generality, assume that it is obvious which nodes in the graph are the result of a merge and which are not (i.e., precisely those with more than one incoming edge are merged). We can therefore assume that the automaton does not produce a tree where a folding operator would merge zero or one nodes. The more general case can then be checked through the following steps.

1. If the graph contains no merged nodes, it is a tree, and the folding having had no effect. In this case, run the tree automaton on the tree and halt with that result.
2. If it contains a single merged node, apply Theorem 1 and halt with that result.
3. If there is no edge e which can bisect the graph in a way that separates two merged nodes, reject it. As argued above, it cannot be in the language.
4. Pick an edge e which bisects the graph into subgraphs g_1 and g_2, which both contain merged nodes, letting g_1 be the subgraph which has e outgoing and g_2 the subgraph that has e incoming. Additionally, pick e such that node that is the source of e has no incoming edge e' which would bisect the graph in this way. Observe that such a choice always exists if any bisecting edge does, as one can then pick e' instead of e, and repeating this argument cannot lead to a cycle (as removing an edge from a cycle would not bisect the graph). Picking e in this way ensures that the folding operator creating the merged node in g_2 is also in g_2.
5. If the graph is in the language, then the edge e was generated by some rule in the tree automaton. For each state q, attempt to parse g_1 and g_2 separately as follows:
   (a) Add e to g_1 letting it go to a new single node labelled σ, where σ is a new symbol not previously in the alphabet. Then, recursively apply this procedure to g_1, modifying the automaton with the new rule $\sigma \to q$.
   (b) Add e to g_2, letting it be outgoing from a new node labelled with a new symbol σ, add a new (unique) accepting state q_f to the automaton, add the rule $\sigma[q] \to q_f$. Then, recursively apply this procedure to g_2 with this modified automaton.
   (c) If 5a and 5b, accept g_1 and g_2, respectively, and accept the graph g.
6. If all states have been tried without accepting, reject.

This procedure is correct, since every graph in the language has to either contain, at most, one application of the folding operator (checked in Steps 1 and 2), or can be bisected by guessing the rule applied (all enumerated by Step 5).

Moreover, the procedure runs in polynomial time. In each recursive call, the polynomial time algorithm of Theorem 1 is applied once, and the number of bisecting edges (across g_1 and g_2) is decreased by one (as e cannot bisect again). □

Author Contributions: Conceptualization, M.B., H.B. and J.B.; methodology, M.B., H.B. and J.B.; formal analysis, M.B., H.B. and J.B.; writing—original draft preparation, M.B., H.B. and J.B.; writing—review and editing, M.B., H.B. and J.B. All authors have read and agreed to the published version of the manuscript.

Funding: Johanna Björklund was supported by the Swedish Research Council under Grant Number 2020-03852. Martin Berglund, Henrik Björklund and Johanna Björklund were supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Data Availability Statement: The project did not create any new datasets.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Tang, L.; Liu, H. Graph mining applications to social network analysis. In *Managing and Mining Graph Data*; Springer: Boston, MA, USA, 2010; pp. 487–513.
2. Plump, D. The graph programming language GP. In *Algebraic Informatics, Proceedings of the 3rd International Conference on Algebraic Informatics, CAI 2009, Thessaloniki, Greece, 19–22 May 2009*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 99–122.
3. You, J.; Leskovec, J.; He, K.; Xie, S. Graph structure of neural networks. In Proceedings of the International Conference on Machine Learning, PMLR, Virtual, 13–18 July 2020; pp. 10881–10891.
4. Björklund, H.; Björklund, J.; Ericson, P. On the regularity and learnability of ordered DAG languages. In *Implementation and Application of Automata, Proceedings of the 22nd International Conference, CIAA 2017, Marne-la-Vallée, France, 27–30 June 2017*; Springer: Cham, Switzerland, 2017; pp. 27–39.
5. Lautemann, C. The complexity of graph languages generated by hyperedge replacement. *Acta Inform.* **1990**, *27*, 399–421. [CrossRef]
6. Quernheim, D.; Knight, K. DAGGER: A Toolkit for Automata on Directed Acyclic Graphs. In Proceedings of the 10th International Workshop Finite-State Methods and Natural Language Processing, FSMNLP 2012, Donostia-San Sebastian, Spain, 23–25 July 2012; Association for Computational Linguistics: Stroudsburg, PA, USA, 2012; pp. 40–44.
7. Koller, A. Semantic construction with graph grammars. In Proceedings of the 11th International Conference on Computational Semantics, IWCS, London, UK, 14–17 April 2015.
8. Drewes, F.; Kreowski, H.J.; Habel, A. Hyperedge Replacement Graph Grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation*; Rozenberg, G., Ed.; World Scientific: River Edge, NJ, USA, 1997; pp. 95–162.
9. Langkilde, I.; Knight, K. Generation That Exploits Corpus-based Statistical Knowledge. In Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics (Volume 1), Montreal, QC, Canada, 10–14 August 1998; pp. 704–710. [CrossRef]
10. Kasper, R.T. A flexible interface for linking applications to Penman's sentence generator. In Proceedings of the Workshop on Speech and Natural Language, Philadelphia, PA, USA, 21–23 February 1989; Association for Computational Linguistics: Stroudsburg, PA, USA, 1989; pp. 153–158. [CrossRef]
11. Banarescu, L.; Bonial, C.; Cai, S.; Georgescu, M.; Griffitt, K.; Hermjakob, U.; Knight, K.; Koehn, P.; Palmer, M.; Schneider, N. Abstract Meaning Representation for Sembanking. In Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse, Sofia, Bulgaria, 8–9 August 2013; Association for Computational Linguistics: Stroudsburg, PA, USA, 2013; pp. 178–186.
12. Braune, F.; Bauer, D.; Knight, K. Mapping Between English Strings and Reentrant Semantic Graphs. In Proceedings of the Ninth International Conference on Language Resources and Evaluation, LREC 2014, Reykjavik, Iceland, 26–31 May 2014; European Language Resources Association: Paris, France, 2014; pp. 4493–4498.
13. Arnborg, S.; Corneil, D.; Proskurowski, A. Complexity of Finding Embeddings in a *k*-Tree. *SIAM J. Algebr. Discret. Methods* **1987**, *8*, 277–284. [CrossRef]
14. Drewes, F.; Hoffmann, B.; Minas, M. Extending Predictive Shift-Reduce Parsing to Contextual Hyperedge Replacement Grammars. In *Graph Transformation, Proceedings of the 12th International Conference, ICGT 2019, Eindhoven, The Netherlands, 15–16 July 2019*; Guerra, E., Orejas, F., Eds.; Lecture Notes in Computer Science; Springer International Publishing: Cham, Switzerland, 2019; Volume 11629, pp. 55–72.
15. Drewes, F.; Hoffmann, B.; Minas, M. Rule-Based Top-Down Parsing for Acyclic Contextual Hyperedge Replacement Grammars. In *Graph Transformation, Proceedings of the 14th International Conference, ICGT 2021, Virtual Event, 24–25 June 2021*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2021.
16. Drewes, F.; Hoffmann, B.; Minas, M. Acyclic Contextual Hyperedge Replacement: Decidability of Acyclicity and Generative Power. In *Graph Transformation, Proceedings of the 15th International Conference, ICGT 2022, Held as Part of STAF 2022, Nantes, France, 7–8 July 2022*; Behr, N., Strüber, D., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2022; Volume 13349, pp. 3–19.
17. Minas, M. The Graph-Parsing Tool Grappa. Available online: https://www.unibw.de/inf2/grappa (accessed on 18 June 2024).
18. Björklund, J. Tree-to-graph transductions with scope. In *Developments in Language Theory, Proceedings of the 22nd International Conference, DLT 2018, Tokyo, Japan, 10–14 September 2018*; Springer: Cham, Switzerland, 2018; pp. 133–144.
19. Berglund, M.; Björklund, H.; Björklund, J.; Boiret, A. Transduction from trees to graphs through folding. *Inf. Comput.* **2023**, *295*, 105111. [CrossRef]
20. Brüggemann-Klein, A.; Murata, M.; Wood, D. *Regular Tree and Regular Hedge Languages over Unranked Alphabets: Version 1*; Technical Report HKUST-TCSC-2001-0; The Hong Kong University of Science and Technology: Hong Kong, China, 2001.
21. Gécseg, F.; Steinby, M. Tree Automata. *arXiv* **2015**, arXiv:1509.06233. https://doi.org/10.48550/arXiv.1509.06233.
22. Björklund, J.; Drewes, F.; Satta, G. Z-Automata for Compact and Direct Representation of Unranked Tree Languages. In *Implementation and Application of Automata, Proceedings of the 24th International Conference, CIAA 2019, Košice, Slovakia, 22–25 July 2019*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2019; Volume 11601, pp. 83–94.
23. Martens, W.; Niehren, J. Minimizing Tree Automata for Unranked Trees. In *Database Programming Languages, Proceedings of the 10th International Symposium, DBPL 2005, Trondheim, Norway, 28–29 August 2005*; Bierman, G., Koch, C., Eds.; Springer: Berlin/Heidelberg, Germany, 2005; pp. 232–246.

24. Parikh, R. *Language Generating Devices*; Technical Report; Research Laboratory of Electronics, MIT: Cambridge, MA, USA, 1961.
25. Karp, R.M.; Miller, R.E. Parallel program schemata. *J. Comput. Syst. Sci.* **1969**, *3*, 147–195. [CrossRef]
26. Czerwiński, W.; Orlikowski, Ł. Reachability in Vector Addition Systems is Ackermann-complete. In Proceedings of the 2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS), Denver, CO, USA, 7–10 February 2022; pp. 1229–1240. [CrossRef]
27. Jones, N.D.; Landweber, L.H.; Lien, Y.E. Complexity of some problems in Petri nets. *Theor. Comput. Sci.* **1977**, *4*, 277–299. [CrossRef]