Theoretical Computer Science 850 (2021) 40–97

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

XML navigation and transformation by tree-walking automata and transducers with visible and invisible pebbles

Joost Engelfriet *, Hendrik Jan Hoogeboom, Bart Samwel

LIACS, Leiden University, P.O. Box 9512, 2300 RA Leiden, the Netherlands

ARTICLE INFO

Article history:
Received 19 September 2018
Accepted 23 October 2020
Available online 28 October 2020
Communicated by D. Perrin

Keywords:
Tree transducers
Tree-walking automata
Pebbles
XML document navigation
XML document transformation

ABSTRACT

The pebble tree automaton and the pebble tree transducer are enhanced by additionally allowing an unbounded number of "invisible" pebbles (as opposed to the usual "visible" ones). The resulting pebble tree automata recognize the regular tree languages (i.e., can validate all generalized DTD's) and hence can find all matches of MSO definable patterns. Moreover, when viewed as a navigational device, they lead to an XPath-like formalism that has a path expression for every MSO definable binary pattern. The resulting pebble tree transducers can apply arbitrary MSO definable tests to (the observable part of) their configurations, they (still) have a decidable typechecking problem, and they can model the recursion mechanism of XSLT. The time complexity of the typechecking problem for conjunctive queries that use MSO definable patterns can often be reduced through the use of invisible pebbles.

Contents

1. Introduction . . . 41
2. Preliminaries . . . 43
3. Automata and transducers . . . 45
4. Decomposition . . . 49
5. Typechecking . . . 55
6. Trees, tests and trips . . . 56
7. The power of the I-PTT . . . 60
8. Look-ahead tests . . . 63
9. Document navigation . . . 65
10. Pattern matching . . . 71
11. Pebble forest transducers . . . 74
12. Document transformation . . . 76
13. A TL program in XSLT . . . 85
14. Data complexity . . . 86
15. Variations of decomposition . . . 88
16. Conclusion . . . 95
Declaration of competing interest . . . 96
References . . . 96

* Corresponding author.
E-mail addresses: j.engelfriet@liacs.leidenuniv.nl (J. Engelfriet), h.j.hoogeboom@liacs.leidenuniv.nl (H.J. Hoogeboom), bsamwel@gmail.com (B. Samwel).

https://doi.org/10.1016/j.tcs.2020.10.030
0304-3975/

1. Introduction

Pebble tree transducers, as introduced by Milo, Suciu, and Vianu [41], are a formal model of XML navigation and transformation for which typechecking is decidable. The pebble tree transducer is a tree-walking tree transducer with nested pebbles, i.e., it walks on the input tree, dropping and lifting a bounded number of pebbles that have nested life times, whereas it produces the output tree in a parallel top-down fashion. We enhance the power of the pebble tree transducer by allowing an unbounded number of (coloured) pebbles, still with nested life times, i.e., organized as a stack. However, apart from a bounded number, the pebbles are "invisible", which means that they can be observed by the transducer only when they are on top of the stack (and thus the number of observable pebbles is bounded at each moment in time). We will call V-PTT the pebble tree transducer of [41] (or rather, the one in [20]: an obvious definitional variant), and VI-PTT the enhanced pebble tree transducer. Moreover, I-PTT refers to the VI-PTT that does not use visible pebbles, which can be viewed as a generalization of the indexed tree transducer of [23]. And TT refers to the pebble tree transducer without pebbles, i.e., to the tree-walking tree transducer, cf. [14] and [10, Section 8]. Tree-walking transducers were introduced in [2], where they translate trees into strings.[1]

The navigational part of the V-PTT, i.e., the behaviour of the transducer when no output is produced, is the pebble tree automaton (V-PTA), introduced in [15], which is a tree-walking automaton with nested pebbles. It was shown in [15] that the V-PTA recognizes regular tree languages only. In [8] the important result was proved that not all regular tree languages can be recognized by the V-PTA, and thus [11,54] the navigational power of the V-PTT is below Monadic Second Order (MSO) logic, which is undesirable for a formal model of XML transformation (see, e.g., [46]). One of the reasons for introducing invisible pebbles is that the VI-PTA, and even the I-PTA, recognizes exactly the regular tree languages (Theorem 11). Thus, since the regular tree grammar is a formal model of DTD (Document Type Definition) in XML, the VI-PTA can validate arbitrary generalized DTD's. We note that the I-PTA is a straightforward generalization of the two-way backtracking pushdown tree automaton of Slutzki [51].

Surveys on the use of tree-walking automata and transducers for XML can be found in [45,50]. For a survey on tree-walking automata see [7].

It is easy to show that every regular tree language can be recognized by an I-PTA, just simulating a bottom-up finite-state tree automaton. The proof that all VI-PTA tree languages are regular, is based on a decomposition of the VI-PTT into TT's (Theorem 5), similar to the one for the V-PTT in [20]. Since the inverse type inference problem is solvable for TT's (where a "type" is a regular tree language), this shows that the domain of a VI-PTT is regular, and so even the alternating VI-PTA tree languages are regular. It also shows that the typechecking problem is decidable for VI-PTT's, by the same arguments as used in [41] for V-PTT's. More precisely, we prove (Theorem 8, based on [14, Theorem 3]) that a VI-PTT with k visible pebbles can be typechecked in $(k+3)$-fold exponential time. For varying k the complexity is non-elementary (as in [41]), but it is observed in [42] that "non-elementary algorithms on tree automata have previously been seen to be feasible in practice".

Generalizing the fact that the I-PTA can recognize the regular tree languages, we prove that the VI-PTA and the VI-PTT can perform MSO tests on the observable part of their configuration, i.e., they can check whether or not the observable pebbles on the input tree (i.e., the visible ones, plus the top pebble on the stack) satisfy certain MSO requirements with respect to the current position of the reading head (Theorem 16). If all the observable pebbles are visible this is obvious (drop an additional visible pebble, simulate an I-PTA that recognizes the regular tree language corresponding to the MSO requirements, return to the pebble and lift it), but if the top pebble is invisible (or if there is no visible pebble left) that does not work and a more complicated technique must be used. Consequently, the VI-PTA can match arbitrary MSO definable n-ary patterns, using n visible pebbles to find all candidate matches as in [41, Example 3.5], and using invisible pebbles to perform the MSO test; the VI-PTT can also output the matches. In fact, instead of the n visible pebbles the VI-PTA can use $n-2$ visible pebbles, one invisible pebble (on top of the stack), and the reading head (Theorem 29).

As the navigational part of the VI-PTT, the VI-PTA in fact computes a binary pattern on trees, i.e., a binary relation between two nodes of a tree: the position of the reading head of the VI-PTT before and after navigation. We prove that also as a navigational device the VI-PTA and the I-PTA have the same power as MSO logic: they compute exactly the MSO definable binary patterns (Theorem 15). This improves the result in [17] (where binary patterns are called "trips"), because the I-PTA is a more natural automaton than the one considered in [17].

One of the research goals of Marx and ten Cate (see [30,39,52,53] and the entertaining [40]) has been to combine Core XPath of [31] which models the navigational part of XPath 1.0, with regular path expressions [1] (or caterpillar expressions [9]) which naturally correspond to tree-walking automata. An important feature of XPath is the "predicate": it allows to test the context node for the existence of at least one other node that matches a given path expression. Thus, the path expression $\alpha_1[\beta]/\alpha_2$ takes an α_1-walk from the context node to the new context node v, checks whether there exists a β-walk from v to some other node, and then takes an α_2-walk from v to the match node. For tree automata this corresponds to the notion of "look-ahead" (cf. [23, Definition 6.5]). We prove (Theorem 19) that an I-PTA $\mathcal{A}$ can use another I-PTA $\mathcal{B}$ as look-ahead test, i.e., $\mathcal{A}$ can test whether or not $\mathcal{B}$ has a successful computation when started in the current configuration of $\mathcal{A}$ (and similarly for VI-PTA and VI-PTT). Since XPath expressions can be nested arbitrarily, we even allow $\mathcal{B}$ to use yet another I-PTA as look-ahead test, etcetera (Theorem 20). Due to this "iterated look-ahead" feature, we can use Kleene's classical construction to

[1] In [10, Section 8] the TT is called tree-walking transducer and the transducer of [2] is called tree-walking tree-to-word transducer.

translate the I-PTA into an XPath-like algebraic formalism, which we call *Pebble XPath*, with the same expressive power as MSO logic for defining binary patterns (Theorem 21). In fact, Pebble XPath is the extension of Regular XPath [39,52] with a stack of invisible pebbles. It is proved in [53] that Regular XPath is not MSO complete (see also [40]).[2] Other MSO complete extensions of Regular XPath are considered in [30,52].

To explain another reason for introducing invisible pebbles we consider XQuery-like conjunctive queries of the form

$$\texttt{for } x_1,\dots,x_n \texttt{ where } \varphi_1 \wedge \cdots \wedge \varphi_m \texttt{ return } r,$$

where $x_1,\dots,x_n$ are variables, each φ_ℓ (with $1 \le \ell \le m$) is an MSO formula with two free variables x_i and x_j, and r is an output tree with variables at the leaves. As observed above, such pattern matching queries can be evaluated by a VI-PTT with $n-2$ visible pebbles, even if the `where`-clause contains an arbitrary MSO formula. In many cases, however, a much smaller number of visible pebbles suffices (Theorem 31). This is an enormous advantage when typechecking the query, as for the time complexity every visible pebble counts (viz. it counts as an exponential). For instance if $j = i+1$ for every φ_ℓ, then *no* visible pebbles are needed, i.e., the query can be evaluated by an I-PTT: we use invisible pebbles $p_1,\dots,p_n$ on the stack (in that order), representing the variables, and move them through the input tree in document order, in a nested fashion; just before dropping pebble p_{i+1}, each formula $\varphi_\ell(x_i,x_{i+1})$ can be verified by an MSO test on the observable part of the configuration (which consists of the top pebble p_i and the reading head position).

The pebble tree transducer transforms ranked trees. However, an XML document is not ranked; it is a forest: a sequence of unranked trees. To model XML transformation by PTT's, forests are encoded as binary trees in the usual way. For the input, it does not make much of a difference whether the PTT walks on a binary tree or a forest. However, as opposed to what is suggested in [41], for the output it *does* make a difference, as pointed out in [47] for macro tree transducers. For that reason we also consider pebble *forest* transducers (abbreviated with PFT instead of PTT) that walk on encoded forests, but construct forests directly, using forest concatenation as basic operation. As in [47], PFT are more powerful than PTT, but the complexity of the typechecking problem is the same, i.e., VI-PFT with k visible pebbles can be typechecked in $(k+3)$-fold exponential time (Theorem 34). In fact, PFT have all the properties mentioned before for PTT.

The document transformation languages DTL and TL were introduced in [38] and [37], respectively, as a formal model of the recursion mechanism in the template rules of XSLT, with MSO logic rather than XPath to specify matching and selection. Documents are modeled as forests. The language DTL has no variables or parameters, and its only instruction is `apply-templates`. The language TL is the extension of DTL with accumulating parameters, i.e., the parameters of XSLT 1.0 whose values are "result tree fragments" (and on which no operations are allowed). We prove that every DTL program can be simulated, with forests encoded as binary trees, by an I-PTT (Theorem 37). More importantly, we prove that TL and I-PFT have the same expressive power (Theorem 46). Thus, in its forest version, our new model the VI-PFT can be viewed as the natural combination of the pebble tree transducer of [41] (V-PTT) and the TL program of [37] (I-PFT). Note that V-PTT and TL have incomparable expressive power. As claimed by [37], TL can "describe many real-world XML transformations". We show that it contains all deterministic VI-PFT transformations for which the size of the output document is linear in the size of the input document (Theorem 57). However, the visible pebbles seem to be a requisite for the XQuery-like queries discussed above, and we conjecture that not all such queries can be programmed in TL (though they *can*, e.g., in the case that $j = i+1$ for every ℓ). As shown in [4] (for a subset of MSO), these queries can be programmed in XSLT 1.0 using parameters that have input nodes as values; however, with such parameters even V-PTT's with *non*nested pebbles can be simulated, and typechecking is no longer decidable. In XSLT 2.0 *all* (computable) queries can be programmed [33]. The main result of [37] is that typechecking is decidable for TL programs. Assuming that MSO formulas are represented by deterministic bottom-up finite-state tree automata, the above relationship between TL and I-PFT allows us to prove that TL programs can be typechecked in 4-fold exponential time (Theorem 41), which seems to be one exponential better than the algorithm in [37].

In addition to the time complexity of typechecking a VI-PTT, also the time complexity of evaluating the queries realized by a VI-PTA or a VI-PTT is of importance. The binary pattern (or "trip") computed by a VI-PTA, i.e., the binary relation between two nodes of the input tree, can be evaluated in polynomial time. The same is true for every (fixed) expression of Pebble XPath (see the last two paragraphs of Section 9). Deterministic VI-PTT's have exponential time data complexity, provided that the output tree can be represented by a DAG (directed acyclic graph). To be precise, for every deterministic VI-PTT there is an exponential time algorithm that transforms any input tree of that VI-PTT into a DAG that represents the corresponding output tree (Theorem 47). For the VI-PTT's that match MSO definable n-ary patterns (as discussed above) the algorithm is polynomial time (Theorem 48). Note that V-PTT's have polynomial time data complexity [41, Proposition 3.8].

Apart from the above results that are motivated by XML navigation and transformation, we also prove some more theoretical results. We show that (as opposed to the V-PTT) the I-PTT can simulate the bottom-up tree transducer (Theorem 18). We show that the composition of two deterministic TT's can be simulated by a deterministic I-PTT (Theorem 17). This even holds when the TT's are allowed to perform MSO tests on their configuration, and then also vice versa, every deterministic I-PTT can be decomposed into two such extended TT's (Theorem 53).

We show that every deterministic VI-PTT can be decomposed into deterministic TT's (Theorem 55) and that, for the deterministic VI-PTT, $k+1$ visible pebbles are more powerful than k visible pebbles (Theorem 56). Pebbles have to be lifted

[2] To be precise, it is proved in [53] that Regular XPath with "subtree relativisation" is not MSO complete and has the same power as first-order logic with monadic transitive closure.

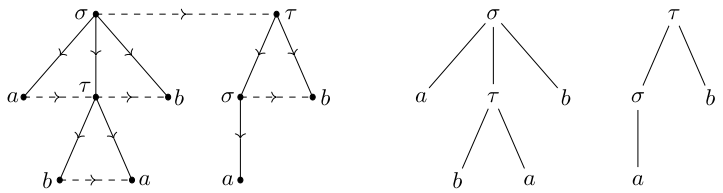

Fig. 1. Picture of the forest $\sigma(a, \tau(b,a), b)\,\tau(\sigma(a), b)$. Formal at the left, with dotted lines for the horizontal edges and solid lines for the vertical edges, and informal at the right.

from the position where they were dropped; however, in [16] it was convenient to consider a stronger type of pebbles that can also be retrieved from a distance. Whereas I-PTT's with strong invisible pebbles can recognize nonregular tree languages, we show that VI-PTT's with strong visible pebbles can still be decomposed into TT's (Theorems 60 and 64) and hence their typechecking is decidable (as already proved for V-PTT's with strong pebbles in [27]). Similarly, deterministic VI-PTT's with strong visible pebbles can be decomposed into deterministic TT's (Theorems 62 and 65).

Some of these theoretical results can be viewed as (slight) generalizations of existing results for formal models of compiler construction (in particular attribute grammars), such as attributed tree transducers [25], macro tree transducers [22], and macro attributed tree transducers [35], see also [26]. As explained in [20, Section 3.2], attributed tree transducers are TT's that satisfy an additional requirement of "noncircularity". Similarly, as observed in [37], macro attributed tree transducers (that generalize both attributed tree transducers and macro tree transducers) are closely related to TL programs, and hence to I-PTT's by Theorem 46. For instance, Theorem 17 slightly generalizes the fact that the composition of two attributed tree transducers can be simulated by a macro attributed tree transducer, as shown in [35].

Most of the results of this paper were announced in the PODS'07 conference [18]. The remaining results are based on technical notes of the authors from the years 2004–2008. This paper has not been updated with the literature of later years (with the exception of [10,14,53]).

2. Preliminaries

Sets, strings, and relations. The set of natural numbers is $\mathbb{N} = \{0, 1, 2, \dots\}$. For $m, n \in \mathbb{N}$, we denote the interval $\{k \in \mathbb{N} \mid m \le k \le n\}$ by $[m, n]$. The cardinality or size of a set A is denoted by $\#(A)$, and its powerset, i.e., the set of all its subsets, by 2^A. The set of strings over A is denoted by A^*. It consists of all sequences $w = a_1 \cdots a_m$ with $m \in \mathbb{N}$ and $a_i \in A$ for every $i \in [1, m]$. The length m of w is denoted by $|w|$. The empty string (of length 0) is denoted by ε. The concatenation of two strings v and w is denoted by $v \cdot w$ or just vw. Moreover, $w^0 = \varepsilon$ and $w^{n+1} = w \cdot w^n$ for $n \in \mathbb{N}$. The composition of two binary relations $R \subseteq A \times B$ and $S \subseteq B \times C$ is $R \circ S = \{(a, c) \mid \exists b \in B : (a, b) \in R,\ (b, c) \in S\}$. The inverse of R is $R^{-1} = \{(b, a) \mid (a, b) \in R\}$, and if $A = B$ then the transitive-reflexive closure of R is $R^* = \bigcup_{n \in \mathbb{N}} R^n$ where $R^0 = \{(a, a) \mid a \in A\}$ and $R^{n+1} = R \circ R^n$. The composition of two classes of binary relations $\mathcal{R}$ and $\mathcal{S}$ is $\mathcal{R} \circ \mathcal{S} = \{R \circ S \mid R \in \mathcal{R},\ S \in \mathcal{S}\}$. Moreover, $\mathcal{R}^1 = \mathcal{R}$ and $\mathcal{R}^{n+1} = \mathcal{R} \circ \mathcal{R}^n$ for $n \ge 1$.

Trees and forests. An alphabet is a finite set of symbols. Let Σ be an alphabet, or an arbitrary set. Unranked trees and forests over Σ are recursively defined to be strings over the set $\Sigma \cup \{(,)\}$ consisting of the elements of Σ, the left parenthesis, and the right parenthesis, as follows. If $\sigma \in \Sigma$ and $t_1, \dots, t_m$ are unranked trees, with $m \in \mathbb{N}$, then their concatenation $t_1 \cdots t_m$ is a forest, and $\sigma(t_1 \cdots t_m)$ is an unranked tree. For $m = 0$, $t_1 \cdots t_m$ is the empty forest ε. For readability we also write the tree $\sigma(t_1 \cdots t_m)$ as $\sigma(t_1, \dots, t_m)$, and even as σ when $m = 0$. Obviously, the concatenation of two forests is again a forest. It should also be noted that every nonempty forest can be written uniquely as $\sigma(f_1) f_2$ where σ is in Σ and f_1 and f_2 are forests. The set of forests over Σ is denoted F_Σ. For an arbitrary set A, disjoint with Σ, we denote by $F_\Sigma(A)$ the set of all forests f over $\Sigma \cup A$ such that every node of f that is labeled by an element of A, is a leaf.

As usual trees and forests are viewed as directed labeled graphs. Here we distinguish between two types of edges: "vertical" and "horizontal" ones. The root of the tree $t = \sigma(t_1, \dots, t_m)$ is labeled by σ. It has vertical edges to the roots of subtrees $t_1, \dots, t_m$, which are the children of the root of t and have child number 1 to m. The root of t is their parent. The roots of $t_1, \dots, t_m$ are siblings, also in the case of the forest $t_1 \cdots t_m$. There is a horizontal edge from each sibling to the next, i.e., from the root of t_i to the root of t_{i+1} for every $i \in [1, m-1]$. Thus, the vertical edges represent the usual parent/child relationship, whereas the horizontal edges represent the linear order between children (and between the roots in a forest), see Fig. 1.[3] For a tree t, its root is denoted by root_t, which is given child number 0 for technical convenience. Its set of nodes is denoted by $N(t)$. For a forest $f = t_1 \cdots t_m$, the set of nodes $N(f)$ is the disjoint union of the sets $N(t_i)$, $i \in [1, m]$. For a node u of a tree t the subtree of t with root u is denoted $t|_u$, and the i-th child of u is denoted ui (and similarly for a forest f instead of t). The nodes of a tree t correspond one-to-one to the positions of the elements of Σ in the string t, i.e., for every $\sigma \in \Sigma$, each occurrence of σ in t corresponds to a node of t with label σ. Since the positions of string t are

[3] In informal pictures the horizontal edges are usually omitted because they are implicit in the left-to-right orientation of the page. Similarly, the arrows of the vertical edges are omitted because of the top-down orientation of the page.

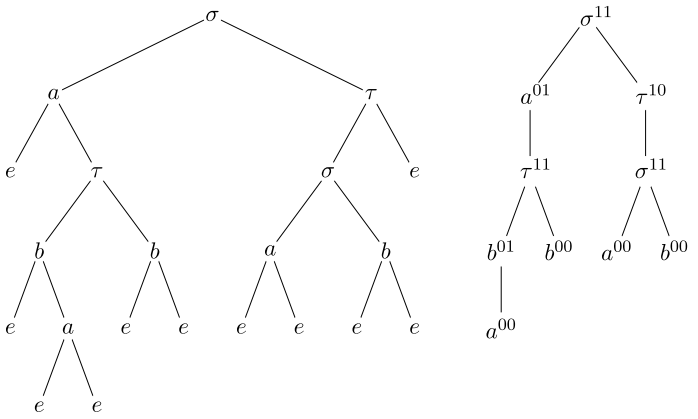

Fig. 2. Encoding of the forest of Fig. 1 by enc (at the left) and by enc′ (at the right).

naturally ordered from left to right, this induces an order on the nodes of t, which is called pre-order (or document order, when viewing t as an XML document). For example, the tree $\sigma(\tau(\alpha,\beta),\gamma))$ has five nodes which have the labels σ, τ, α, β, and γ in pre-order.

A *ranked* alphabet (or set) Σ has an associated mapping $\mathrm{rank}_\Sigma : \Sigma \to \mathbb{N}$. The maximal rank of elements of Σ is denoted mx_Σ. By $\Sigma^{(m)}$ we denote the elements of Σ with rank m. Ranked trees over Σ are recursively defined as above with the requirement that $m = \mathrm{rank}_\Sigma(\sigma)$. The set of ranked trees over Σ is denoted T_Σ. For an arbitrary set A, disjoint with Σ, we denote by $T_\Sigma(A)$ the set $T_{\Sigma\cup A}$ where each element of A has rank 0. We will not consider ranked forests.

Forests over an alphabet Σ can be encoded as binary trees, in the usual way: each node has a label in Σ, a "vertical" pointer to its first child, and a "horizontal" pointer to its next sibling; the pointer is nil if there is no such child or sibling. Such a binary tree can be modeled as a ranked tree over the ranked alphabet $\Sigma \cup \{e\}$ where every $\sigma \in \Sigma$ has rank 2 and e is a symbol of rank 0 that represents the empty forest ε (or nil). Formally, the encoding of the empty forest equals $\mathrm{enc}(\varepsilon) = e$, and recursively, the encoding $\mathrm{enc}(f)$ of a forest $f = \sigma(f_1)f_2$ equals $\sigma(\mathrm{enc}(f_1), \mathrm{enc}(f_2))$. Obviously, enc is a bijection between forests over Σ and ranked trees over $\Sigma \cup \{e\}$. The decoding which is its inverse will be denoted by dec. For an example of $\mathrm{enc}(f)$ see Fig. 2 at the left.

The disadvantage of this encoding is that the tree $\mathrm{enc}(f)$ has more nodes than the forest f, viz. all nodes with label e. That is inconvenient when comparing the behaviour of tree-walking automata on f and $\mathrm{enc}(f)$. Thus, we will also use an encoding that preserves the number of nodes (and thus cannot encode the empty forest). For this we use the ranked alphabet Σ' consisting, for every $\sigma \in \Sigma$, of the symbols σ^{11} of rank 2 (for a binary node without nil-pointers), σ^{01} and σ^{10} of rank 1 (for a binary node with vertical or horizontal nil-pointer, respectively), and σ^{00} of rank 0 (for a binary node with two nil-pointers). The encoding $\mathrm{enc}'(f)$ of a nonempty forest $f = \sigma(f_1)f_2$ equals $\sigma^{11}(\mathrm{enc}'(f_1), \mathrm{enc}'(f_2))$ or $\sigma^{01}(\mathrm{enc}'(f_2))$ or $\sigma^{10}(\mathrm{enc}'(f_1))$ or σ^{00}, where the first (second) superscript of σ equals 0 if and only if $f_1 = e$ ($f_2 = e$). Now, enc′ is a bijection between nonempty forests over Σ and ranked trees over Σ'. The decoding which is its inverse will be denoted by dec′. For an example of $\mathrm{enc}'(f)$ see Fig. 2 at the right. From the point of view of graphs, we assume that $\mathrm{enc}'(f)$ has the same nodes as f, i.e., $N(\mathrm{enc}'(f)) = N(f)$. The label of a node u of f is changed from σ to σ^{ij} where $i = 1$ if and only if u has at least one child, and $j = 1$ if and only if u has a next sibling. If u has children, then its first child in $\mathrm{enc}'(f)$ is its first child in f, and its second child in $\mathrm{enc}'(f)$ is its next sibling (if it has one). If u has no children, then its only child in $\mathrm{enc}'(f)$ is its next sibling (if it has one). Although this encoding is intuitively clear, it is technically less attractive. We will use enc′ for the input forest of automata and transducers, and enc for the output forest of the transducers.

We assume the reader to be familiar with the notion of a *regular tree grammar*. It is a context-free grammar G of which every rule is of the form $X_0 \to \sigma(X_1 \cdots X_m)$ where X_i is a nonterminal and σ is a terminal symbol of rank m. Thus, G generates a set $L(G)$ of ranked trees, which is called a regular tree language. The class of regular tree languages will be denoted REGT. We define a *regular forest grammar* to be a context-free grammar G of which every rule is of the form $X_0 \to \sigma(X_1)X_2$ or $X \to \varepsilon$, where σ is from an unranked alphabet. It generates a set $L(G)$ of (unranked) forests, which is called a regular forest language. Obviously, L is a regular forest language if and only if $\mathrm{enc}(L)$ is a regular tree language, and, as one can easily prove, if and only if $\mathrm{enc}'(L)$ is a regular tree language. The regular tree/forest grammar is a formal model of DTD (Document Type Definition) in XML.[4]

Monadic second-order logic (abbreviated as MSO logic) is used to describe properties of forests and trees. It views each forest or tree as a logical structure that has the set of nodes as domain. As basic properties of a forest over alphabet Σ it uses the atomic formulas $\mathrm{lab}_\sigma(x)$, $\mathrm{down}(x,y)$, and $\mathrm{next}(x,y)$, meaning that node x has label $\sigma \in \Sigma$, that y is a child of x, and that y is the next sibling of x, respectively. Thus, $\mathrm{down}(x,y)$ and $\mathrm{next}(x,y)$ represent the vertical and horizontal edges

[4] In the literature regular forest languages are usually defined in a different way, after which it is proved that L is a regular forest language if and only if $\mathrm{enc}(L)$ is a regular tree language, thus showing the equivalence with our definition, see, e.g., [45, Proposition 1].

of the graph representation of the forest. For a ranked tree over ranked alphabet Σ we could use the same atomic formulas, but it is customary to replace $\mathrm{down}(x, y)$ and $\mathrm{next}(x, y)$ by the atomic formulas $\mathrm{down}_i(x, y)$, for every $i \in [1, mx_\Sigma]$, meaning that y is the i-th child of x. Additionally, MSO logic has the atomic formulas $x = y$ and $x \in X$, where X is a set of nodes. The formulas are built with the usual connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$; both node variables $x, y, \ldots$ and node-set variables $X, Y, \ldots$ can be quantified with $\exists$ and $\forall$. For a forest (or ranked tree) f over Σ and a formula $\varphi(x_1, \ldots, x_n)$ with n free node variables $x_1, \ldots, x_n$, we write $f \models \varphi(u_1, \ldots, u_n)$ to mean that φ holds in f for the nodes $u_1, \ldots, u_n$ of f (as values of the variables $x_1, \ldots, x_n$ respectively).

We will occasionally use the following formulas: $\mathrm{root}(x)$ and $\mathrm{leaf}(x)$ test whether node x is a root or a leaf, and $\mathrm{first}(x)$ and $\mathrm{last}(x)$ test whether x is a first or a last sibling. Also, $\mathrm{child}_i(x)$ tests whether x is an i-th child, $\mathrm{up}(x, y)$ expresses that y is the parent of x, and $\mathrm{stay}(x, y)$ expresses that y equals x. Thus, we define $\mathrm{stay}(x, y) \equiv x = y$ and

$$\mathrm{root}(x) \equiv \neg \exists z(\mathrm{down}(z, x)),\ \mathrm{leaf}(x) \equiv \neg \exists z(\mathrm{down}(x, z)),$$

$$\mathrm{first}(x) \equiv \neg \exists z(\mathrm{next}(z, x)),\ \mathrm{last}(x) \equiv \neg \exists z(\mathrm{next}(x, z)),$$

$$\mathrm{child}_i(x) \equiv \exists z(\mathrm{down}_i(z, x)),\ \mathrm{up}(x, y) \equiv \mathrm{down}(y, x).$$

Patterns. Let Σ be a ranked alphabet and $n \geq 0$. An n-ary *pattern* (or n-ary query) over Σ is a set $T \subseteq \{(t, u_1, \ldots, u_n) \mid t \in T_\Sigma,\ u_1, \ldots, u_n \in N(t)\}$. For $n = 0$ this is a tree language, for $n = 1$ it is a *site* (trees with a distinguished node), for $n = 2$ it is a *trip* [17] (or a binary tree-node relation [5]).

We introduce a new ranked alphabet $\Sigma \times \{0, 1\}^n$, the rank of (σ, ℓ) equals that of σ in Σ. For a tree t over Σ and n nodes $u_1, \ldots, u_n$ we define $\mathrm{mark}(t, u_1, \ldots, u_n)$ to be the tree over $\Sigma \times \{0, 1\}^n$ that is obtained by adding to the label of each node u in t a vector $\ell \in \{0, 1\}^n$ such that the i-th component of ℓ equals 1 if and only if $u = u_i$. The n-ary pattern T is *regular* if its marked representation is a regular tree language, i.e., $\mathrm{mark}(T) \in \mathsf{REGT}$.

An MSO formula $\varphi(x_1, \ldots, x_n)$ over Σ, with n free node variables $x_1, \ldots, x_n$, defines the n-ary pattern $T(\varphi) = \{(t, u_1, \ldots, u_n) \mid t \models \varphi(u_1, \ldots, u_n)\}$. Note that $T(\varphi)$ also depends on the order $x_1, \ldots, x_n$ of the free variables of φ. It easily follows from the result of Doner, Thatcher and Wright [11,54] that a pattern is MSO definable if and only if it is regular (see [5, Lemma 7]).

We will also consider patterns on forests. For an unranked alphabet Σ, a (forest) pattern over Σ is a subset of $\{(f, u_1, \ldots, u_n) \mid f \in F_\Sigma,\ u_1, \ldots, u_n \in N(f)\}$. As for ranked trees, an MSO formula $\varphi(x_1, \ldots, x_n)$ over Σ, defines the n-ary (forest) pattern $\{(f, u_1, \ldots, u_n) \mid f \models \varphi(u_1, \ldots, u_n)\}$.

3. Automata and transducers

In this section we define tree-walking automata and transducers with pebbles, and discuss some of their properties.

Automata. A *tree-walking automaton with nested pebbles* (pebble tree automaton for short, abbreviated PTA) is a finite state device with one reading head that walks from node to node over its ranked input tree following the vertical edges in either direction. Additionally it has a supply of *pebbles* that can be used to mark the nodes of the tree. The automaton may drop a pebble on the node currently visited by the reading head, but it may only lift any pebble from the current node if that pebble was the last one dropped during the computation. Thus, the life times of the pebbles on the tree are nested. Here we consider two types of pebbles. First there are a finite number of "classical" pebbles, which we here call *visible* pebbles. Each of these has a distinct colour, and at most k visible pebbles (each with a different colour) can be present on the input tree during any computation, where k is fixed. Second there are *invisible* pebbles. Again, these pebbles have a finite number of colours (distinct from those of the visible pebbles), but for each colour there is an unlimited supply of pebbles that can be present on the input tree. Visible pebbles can be observed by the automaton at any moment when it visits the node where they were dropped. An invisible pebble can only be observed when it was the last pebble dropped on the tree during the computation.

The possible actions of the automaton are determined by its state, the label of the current node, the child number of the node, and the set of *observable* pebbles on the current node, that is, visible pebbles and an invisible pebble when it was the last pebble dropped on the tree. Unlike the PTA from [41], our automata do *not* branch (i.e., are not alternating).

The PTA is specified as a tuple $\mathcal{A} = (\Sigma, Q, Q_0, F, C, C_\mathrm{v}, C_\mathrm{i}, R, k)$, where Σ is a ranked alphabet of input symbols, Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, C_v and C_i are the finite sets of visible and invisible colours, $C = C_\mathrm{v} \cup C_\mathrm{i}$, $C_\mathrm{v} \cap C_\mathrm{i} = \varnothing$, R is a finite set of rules, and $k \in \mathbb{N}$. Each rule is of the form $\langle q, \sigma, j, b\rangle \rightarrow \langle q', \alpha\rangle$ such that $q, q' \in Q$, $\sigma \in \Sigma$, $j \in [0, mx_\Sigma]$, $b \subseteq C$ with $\#(b \cap C_\mathrm{v}) \leq k$ and $\#(b \cap C_\mathrm{i}) \leq 1$, and α is one of the following *instructions*:

stay,	
up	provided $j \neq 0$,
down_i	with $1 \leq i \leq \mathrm{rank}_\Sigma(\sigma)$,
drop_c	with $c \in C$, and
lift_c	with $c \in b$,

where the first three are *move instructions* and the last two are *pebble instructions*. Note that, due to the nested life times of the pebbles, at most one pebble c in b can actually be lifted; however, the subscript c of lift_c often increases the readability of a PTA.

A *situation* $\langle u, \pi \rangle$ of the PTA $\mathcal{A}$ on ranked tree t over Σ is given by the position u of the head of $\mathcal{A}$ on t, and the stack π containing the positions and colours of the pebbles on the tree in the order in which they were dropped. Formally, $u \in N(t)$ and $\pi \in (N(t) \times C)^*$. The last element of π represents the top of the stack. The set of all situations of $\mathcal{A}$ on t is denoted $\text{Sit}(t)$, i.e., $\text{Sit}(t) = N(t) \times (N(t) \times C)^*$; note that it only depends on C. A *configuration* $\langle q, u, \pi \rangle$ of $\mathcal{A}$ on t additionally contains the state q of $\mathcal{A}$, $q \in Q$. It is *final* when $q \in F$. An *initial* configuration is of the form $\langle q_0, \text{root}_t, \varepsilon \rangle$ where $q_0 \in Q_0$, root_t is the root of t, and ε is the empty stack. The set of all configurations of $\mathcal{A}$ on t is denoted $\text{Con}(t)$, i.e., $\text{Con}(t) = Q \times N(t) \times (N(t) \times C)^*$.

We now define the computation steps of the PTA $\mathcal{A}$, which lead from one configuration to another. For a given input tree t they form a binary relation on $\text{Con}(t)$. A rule $\langle q, \sigma, j, b \rangle \to \langle q', \alpha \rangle$ is *relevant* to every configuration $\langle q, u, \pi \rangle$ with state q and with a situation $\langle u, \pi \rangle$ that satisfies the tests σ, j, and b, i.e., σ and j are the label and child number of node u, and b is the set of colours of the observable pebbles dropped on the node u. More precisely, b consists of all $c \in C_\text{v}$ such that (u, c) occurs in π, plus $c \in C_\text{i}$ if (u, c) is the topmost (i.e., last) element of π. Application of the rule to such a configuration possibly leads to a new configuration $\langle q', u', \pi' \rangle$, in which case we write $\langle q, u, \pi \rangle \Rightarrow_{t,\mathcal{A}} \langle q', u', \pi' \rangle$. The new state is q' and the new situation $\langle u', \pi' \rangle$ is obtained from the situation $\langle u, \pi \rangle$ by the instruction α. For the move instructions $\alpha = \text{stay}$, $\alpha = \text{up}$, and $\alpha = \text{down}_i$ the pebble stack does not change, i.e., $\pi' = \pi$, and the new node u' equals u, is the parent of u, or is the i-th child of u, respectively. For the pebble instructions the node does not change, i.e., $u' = u$. When $\alpha = \text{drop}_c$, $\mathcal{A}$ drops a pebble with colour c on the current node, thus the node-colour pair (u, c) is pushed onto the pebble stack π, i.e., $\pi' = \pi(u, c)$, unless c is a visible colour and the stack already contains a pebble of that colour or already contains k visible pebbles, in which case the rule is not applicable.[5] When $\alpha = \text{lift}_c$, $\mathcal{A}$ lifts a pebble with colour c from the current node, only allowed if the topmost element of the pebble stack is the pair (u, c), which is subsequently popped from the stack, i.e., $\pi = \pi'(u, c)$; otherwise this rule is not applicable. We will also allow instructions like $\text{lift}_c\,;\text{up}$ with the obvious meaning (first lift the pebble, then move up). In this way we have defined the binary relation $\Rightarrow_{t,\mathcal{A}}$ on $\text{Con}(t)$, which represents the computation steps of $\mathcal{M}$. We will say informally that a computation step of $\mathcal{M}$ *halts successfully* if it leads to a final configuration.

The *tree language* $L(\mathcal{A})$ *accepted by* PTA $\mathcal{A}$ consists of all ranked trees t over Σ such that $\mathcal{A}$ has a successful computation on t that starts in an initial configuration. Formally, $L(\mathcal{A}) = \{t \in T_\Sigma \mid \exists q_0 \in Q_0, q_\infty \in F, \langle u, \pi \rangle \in \text{Sit}(t) : \langle q_0, \text{root}_t, \varepsilon \rangle \Rightarrow^*_{t,\mathcal{A}} \langle q_\infty, u, \pi \rangle\}$. Note that pebbles may remain in the final configuration and that the head need not return to the root. Two PTA's $\mathcal{A}$ and $\mathcal{B}$ are *equivalent* if $L(\mathcal{A}) = L(\mathcal{B})$.

By V_kI-PTA we denote a PTA with last component k, i.e., that uses at most k visible pebbles in its computations, and an unbounded number of invisible pebbles, and by V_kI-PTA we denote the class of tree languages accepted by V_kI-PTA's. For $k = 0$, an automaton that only uses invisible pebbles, we also use the notation I-PTA, and for an automaton that only uses k visible pebbles we use V_k-PTA. Moreover, TA is used for a tree-walking automaton without pebbles, i.e., a V_0-PTA. The lower case d or d is added when we only consider *deterministic* automata, which have a unique initial state, no final state in the left-hand side of a rule, and no two rules with the same left-hand side. Thus we have V_kI-dPTA, V_kI-dPTA, and variants.

Properties of automata. It is natural, and sometimes useful, to extend the V_kI-PTA with the facility to test whether its pebble stack is nonempty, and if so, to test the colour of the topmost pebble. Thus, we define a PTA *with stack tests* in the same way as an ordinary PTA except that its rules are of the form $\langle q, \sigma, j, b, \gamma \rangle \to \langle q', \alpha \rangle$ with $\gamma \in C \cup \{\varepsilon\}$. Such a rule is relevant to a configuration $\langle q, u, \pi \rangle$ if, in addition, the pebble stack π is empty if $\gamma = \varepsilon$, and the topmost pebble of π has colour γ if $\gamma \in C$.[6] All other definitions are the same. Note that, obviously, we may require for the above rule that $\gamma = c$ if $\alpha = \text{lift}_c$, which ensures that relevant rules with a lift-instruction are always applicable.[7]

It is not difficult to see that these new tests do not extend the expressive power of the PTA. Informally we will say that the V_kI-PTA can *perform stack tests*.

Lemma 1. *Let $k \geq 0$. For every V_kI-PTA with stack tests $\mathcal{A}$ an equivalent (ordinary) V_kI-PTA $\mathcal{A}'$ can be constructed in polynomial time. The construction preserves determinism and the absence of invisible pebbles.*[8]

Proof. Let $\mathcal{A} = (\Sigma, Q, Q_0, F, C, C_\text{v}, C_\text{i}, R, k)$. The new automaton $\mathcal{A}'$ stepwise simulates $\mathcal{A}$ and, additionally, stores in its finite state whether or not the pebble stack is nonempty, and if so, what is the colour in C of the topmost pebble. Thus, $Q' = Q \times (C \cup \{\varepsilon\})$, $Q'_0 = Q_0 \times \{\varepsilon\}$, and $F' = F \times (C \cup \{\varepsilon\})$. Moreover, the colour sets of $\mathcal{A}'$ are $C'_\text{v} = C_\text{v} \times (C \cup \{\varepsilon\})$ and $C'_\text{i} = C_\text{i} \times (C \cup \{\varepsilon\})$. In fact, if the pebble stack of $\mathcal{A}$ is $\pi = (u_1, c_1)(u_2, c_2) \cdots (u_n, c_n)$, with (u_n, c_n) being the topmost pebble, then the stack of $\mathcal{A}'$ is $\pi' = (u_1, (c_1, \varepsilon))(u_2, (c_2, c_1)) \cdots (u_n, (c_n, c_{n-1}))$, where ε is viewed as a bottom symbol. Thus, the

[5] To be precise, the rule is not applicable if $c \in C_\text{v}$, $\pi = (u_1, c_1) \cdots (u_n, c_n)$, and there exists $i \in [1, n]$ such that $c = c_i$, or $\#(\{i \in [1, n] \mid c_i \in C_\text{v}\}) = k$.

[6] To be precise, for $\pi = (u_1, c_1) \cdots (u_n, c_n)$ the requirements are the following: If $\gamma = \varepsilon$ then $n = 0$, i.e., $\pi = \varepsilon$. If $\gamma \in C$ then $n \geq 1$ and $c_n = \gamma$.

[7] Additionally, we can require the following: If $\gamma = \varepsilon$ then $b = \varnothing$. If $b \cap C_\text{i} = \{c\}$ then $\gamma = c$.

[8] In other words, the statement of the lemma also holds for V_kI-dPTA, V_k-PTA and V_k-dPTA.

new colour of a pebble contains its old colour together with the old colour of the previously dropped pebble (or ε if there is none). This allows $\mathcal{A}'$ to update its additional finite state component when $\mathcal{A}$ lifts a pebble. More precisely, when $\mathcal{A}$ is in configuration $\langle q, u, \pi\rangle$, the automaton $\mathcal{A}'$ is in configuration $\langle (q,\gamma), u, \pi'\rangle$, where $\gamma = c_n$ if $n \geq 1$ and $\gamma = \varepsilon$ otherwise.

The rules of $\mathcal{A}'$ are defined as follows. Let $\langle q, \sigma, j, b, \gamma\rangle \to \langle q', \alpha\rangle$ be a rule of $\mathcal{A}$, and let b' be (the graph of) a mapping from b to $C \cup \{\varepsilon\}$. If α is a move instruction, then $\mathcal{A}'$ has the rule $\langle (q,\gamma), \sigma, j, b'\rangle \to \langle (q',\gamma), \alpha\rangle$. If $\alpha = \mathrm{drop}_c$, then $\mathcal{A}'$ has the rule $\langle (q,\gamma), \sigma, j, b'\rangle \to \langle (q', c), \mathrm{drop}_{(c,\gamma)}\rangle$. If $\alpha = \mathrm{lift}_c$, $\gamma = c$, and $(c,\gamma') \in b'$, then $\mathcal{A}'$ has the rule $\langle (q,\gamma), \sigma, j, b'\rangle \to \langle (q',\gamma'), \mathrm{lift}_{(c,\gamma')}\rangle$.

It should be clear that the construction of $\mathcal{A}'$ takes polynomial time. Note that k is fixed and $\#(b) \leq k+1$ in the left-hand side of the rule $\langle q, \sigma, j, b, \gamma\rangle \to \langle q', \alpha\rangle$ of $\mathcal{A}$. □

PTA's with stack tests will only be used in Sections 8 and 15. The next two properties of PTA's will not be used in later sections, but are meant to clarify some of the details in the semantics of the PTA.

A rule of a V_kI-PTA $\mathcal{A}$ is *progressive* if it is applicable to every reachable configuration[9] to which it is relevant. The V_kI-PTA $\mathcal{A}$ is progressive if all its rules are progressive. Intuitively this means that $\mathcal{A}$ knows that its instructions can always be executed. Clearly, according to the syntax of a PTA, every rule with a move instruction is progressive. The same is true for rules with a pebble instruction drop_c or lift_c with $c \in C_\mathrm{i}$: an invisible pebble can always be dropped and an observable invisible pebble can always be lifted. Thus, only the dropping and lifting of visible pebbles is problematic. It is easy to see that, for the V_kI-PTA $\mathcal{A}'$ constructed in the proof of Lemma 1, every rule with a lift-instruction is progressive.

A V_kI-PTA $\mathcal{A}$ is *counting* if $C_\mathrm{v} = [1,k]$ and, in each reachable configuration, the colours of the visible pebbles on the tree are $1,\ldots,\ell$ for some $\ell \in [0,k]$, in the order in which they were dropped.[10] Note that in the literature V_k-PTA's are usually counting. We have chosen to allow arbitrarily many visible colours in a V_kI-PTA because we want to be able to store information in the pebbles, as in the proof of Lemma 1. It is straightforward to construct an equivalent counting V_kI-PTA $\mathcal{A}'$ for a given V_kI-PTA $\mathcal{A}$ (preserving determinism and the absence of invisible pebbles). The automaton $\mathcal{A}'$ stepwise simulates $\mathcal{A}$ and, additionally, stores in its finite state the colours of the visible pebbles that are dropped on the tree, in the order in which they were dropped. Thus, the states of $\mathcal{A}'$ are of the form (q,φ) where q is a state of $\mathcal{A}$ and φ is a string over C_v without repetitions, of length at most k. The state (q,φ) is final if q is final. The initial states are (q,ε) where q is an initial state of $\mathcal{A}$. The rules of $\mathcal{A}'$ are defined as follows. Let $\langle q, \sigma, j, b\rangle \to \langle q', \alpha\rangle$ be a rule of $\mathcal{A}$ and let (q,φ) be a state of $\mathcal{A}'$ such that every $c \in b \cap C_\mathrm{v}$ occurs in φ. Moreover, let $b' \subseteq [1,k] \cup C_\mathrm{i}$ be obtained from b by changing every $c \in C_\mathrm{v}$ into i, if c is the i-th element of φ. If α is a move instruction, or a pebble instruction drop_c or lift_c with $c \in C_\mathrm{i}$ then $\mathcal{A}'$ has the rule $\langle (q,\varphi), \sigma, j, b'\rangle \to \langle (q',\varphi), \alpha\rangle$. If $\alpha = \mathrm{drop}_c$ with $c \in C_\mathrm{v}$, c does not occur in φ, and $|\varphi| < k$, then $\mathcal{A}'$ has the rule $\langle (q,\varphi), \sigma, j, b'\rangle \to \langle (q', \varphi c), \mathrm{drop}_{|\varphi|+1}\rangle$. Finally, if $\alpha = \mathrm{lift}_c$ with $c \in C_\mathrm{v}$, and $\varphi = \varphi' c$ for some $\varphi' \in C_\mathrm{v}^*$, then $\mathcal{A}'$ has the rule $\langle (q,\varphi), \sigma, j, b'\rangle \to \langle (q',\varphi'), \mathrm{lift}_{|\varphi|}\rangle$. It should be clear that $\mathcal{A}'$ is counting. Note also that all rules of $\mathcal{A}'$ with a drop-instruction are progressive. Thus, if we first apply the construction in the proof of Lemma 1 and then the one above, we obtain an equivalent progressive V_kI-PTA. Obviously, every progressive V_kI-PTA can be turned into an equivalent V_{k+1}I-PTA by simply changing its last component k into $k+1$, and hence $\mathsf{V}_k\text{I-PTA} \subseteq \mathsf{V}_{k+1}\text{I-PTA}$ and $\mathsf{V}_k\text{I-dPTA} \subseteq \mathsf{V}_{k+1}\text{I-dPTA}$.[11]

Transducers. A *tree-walking tree transducer with nested pebbles* (abbreviated PTT) is a PTA without final states that additionally produces an output tree over a ranked alphabet Δ. Thus, omitting F, it is specified as a tuple $\mathcal{M} = (\Sigma, \Delta, Q, Q_0, C, C_\mathrm{v}, C_\mathrm{i}, R, k)$, where Σ, Q, Q_0, C, C_v, C_i, and k are as for the PTA. The rules of $\mathcal{M}$ in the finite set R are of the same form as for the PTA, except that $\mathcal{M}$ additionally has *output rules* of the form $\langle q, \sigma, j, b\rangle \to \delta(\langle q_1, \mathrm{stay}\rangle, \ldots, \langle q_m, \mathrm{stay}\rangle)$ with $\delta \in \Delta$, and $q_1,\ldots,q_m \in Q$, where m is the rank of δ. Intuitively, the output tree is produced recursively. In other words, in a configuration to which the above output rule is relevant (defined as for the PTA) the PTT $\mathcal{M}$ outputs δ, and for each child $\langle q_i, \mathrm{stay}\rangle$ branches into a new process, a copy of itself started in state q_i at the current node, retaining the same stack of pebbles; thus, the stack is copied m times. Note that a relevant output rule is always applicable. As a shortcut we may replace the stay-instruction in any $\langle q_i, \mathrm{stay}\rangle$ by another move instruction or a pebble instruction, with obvious semantics.

An *output form* of the PTT $\mathcal{M}$ on ranked tree t over Σ is a tree in $T_\Delta(\mathrm{Con}(t))$, where $\mathrm{Con}(t)$ is defined as for the PTA. Intuitively, such an output form consists on the one hand of Δ-labeled nodes that were produced by $\mathcal{M}$ previously in the computation, using output rules, and on the other hand of leaves that represent the independent copies of $\mathcal{M}$ into which the computation has branched previously, due to those output rules, where each leaf is labeled by the current configuration of that copy. Note that $\mathrm{Con}(t) \subseteq T_\Delta(\mathrm{Con}(t))$, i.e., every configuration of $\mathcal{M}$ is an output form.

The computation steps of the PTT $\mathcal{M}$ lead from one output form to another. Let s be an output form and let v be a leaf of s with label $\langle q, u, \pi\rangle \in \mathrm{Con}(t)$. If $\langle q, u, \pi\rangle \Rightarrow_{t,\mathcal{M}} \langle q', u', \pi'\rangle$, where the binary relation $\Rightarrow_{t,\mathcal{M}}$ on $\mathrm{Con}(t)$ is defined as for the PTA (disregarding the output rules of $\mathcal{M}$), then we write $s \Rightarrow_{t,\mathcal{M}} s'$ where s' is obtained from s by changing the label of v into $\langle q', u', \pi'\rangle$. Moreover, for every output rule $\langle q, \sigma, j, b\rangle \to \delta(\langle q_1, \mathrm{stay}\rangle, \ldots, \langle q_m, \mathrm{stay}\rangle)$ that is relevant to configuration $\langle q, u, \pi\rangle$, we write $s \Rightarrow_{t,\mathcal{M}} s'$ where s' is obtained from s by replacing the node v by the subtree

[9] The configuration $\langle q, u, \pi\rangle$ on the tree t is *reachable* if $\langle q_0, \mathrm{root}_t, \varepsilon\rangle \Rightarrow^*_{t,\mathcal{A}} \langle q, u, \pi\rangle$ for some $q_0 \in Q_0$.

[10] To be precise, for $\pi = (u_1,c_1)\cdots(u_n,c_n)$ we require that there exists $\ell \in [0,k]$ such that $(c_{i_1},\ldots,c_{i_m}) = (1,\ldots,\ell)$ where $\{i_1,\ldots,i_m\} = \{i \in [1,n] \mid c_i \in C_\mathrm{v}\}$ and $i_1 < \cdots < i_m$.

[11] In fact, these four classes are equal, as will be shown in Theorem 11.

$\delta(\langle q_1, u, \pi\rangle, \ldots, \langle q_m, u, \pi\rangle)$. In the particular case that $m = 0$, s' is obtained from s by changing the label of v into δ. In that case we will say informally that $\mathcal{M}$ *halts successfully*, meaning that the copy of $\mathcal{M}$ corresponding to the node v of s disappears. In this way we have extended $\Rightarrow_{t,\mathcal{M}}$ to a binary relation on $T_\Delta(\mathrm{Con}(t))$.

The *transduction* $\tau_\mathcal{M}$ *realized by* $\mathcal{M}$ consists of all pairs of trees t over Σ and s over Δ such that $\mathcal{M}$ has a (successful) computation on t that starts in an initial configuration and ends with s. Formally, we define $\tau_\mathcal{M} = \{(t, s) \in T_\Sigma \times T_\Delta \mid \exists q_0 \in Q_0 : \langle q_0, \mathrm{root}_t, \varepsilon\rangle \Rightarrow^*_{t,\mathcal{M}} s\}$. Two PTT's $\mathcal{M}$ and $\mathcal{N}$ are *equivalent* if $\tau_\mathcal{M} = \tau_\mathcal{N}$.

The *domain of* $\mathcal{M}$ is defined to be the domain of $\tau_\mathcal{M}$, i.e., the tree language $L(\mathcal{M}) = \{t \in T_\Sigma \mid \exists s \in T_\Delta : (t, s) \in \tau_\mathcal{M}\}$. When $\mathcal{M}$ is viewed as a recognizer of its domain, it is actually the same as an alternating PTA. Existential states in the alternation correspond to the nondeterminism of the PTT, universal states correspond to the recursive way in which output trees are generated. More precisely, an output rule $\langle q, \sigma, j, b\rangle \to \delta(\langle q_1, \mathrm{stay}\rangle, \ldots, \langle q_m, \mathrm{stay}\rangle)$ corresponds to a universal state q that requires every state q_i to have a successful computation (and the output symbol δ is irrelevant). An ordinary (non-alternating) PTA then corresponds to a PTT for which every output symbol has rank 0; for $m = 0$ the above output rule means that the PTA halts in a final state. We say that the PTT $\mathcal{M}$ is *total* if $L(\mathcal{M}) = T_\Sigma$, i.e., $\tau_\mathcal{M}(t) \neq \emptyset$ for every input tree t.

Similar to the notation V_kI-PTA for tree languages, we use the notation V_kI-PTT for the class of transductions defined by tree-walking tree transducers with k visible nested pebbles and an unbounded number of invisible pebbles, as well as the obvious variants V_k-PTT, and I-PTT. Additionally TT denotes the class of transductions realized by tree-walking tree transducers without pebbles, i.e., V_0-PTT. Such a transducer is specified as a tuple $\mathcal{M} = (\Sigma, \Delta, Q, Q_0, R)$, and the left-hand sides of its rules are written $\langle q, \sigma, j\rangle$, omitting $b = \varnothing$. As for PTA's, lower case d is added for *deterministic* transducers, which have a unique initial state and no two rules with the same left-hand side. Moreover, lower case td is used for *total deterministic* transducers, i.e., transducers that are both total and deterministic. Note that a deterministic PTT realizes a function, and a total deterministic PTT a total function from T_Σ to T_Δ.

Properties of transducers. Stack tests are defined for the PTT as for the PTA, and Lemma 1 and its proof carry over to PTT's. If a given PTT $\mathcal{M}$ with stack tests has the output rule $\langle q, \sigma, j, b, \gamma\rangle \to \delta(\langle q_1, \mathrm{stay}\rangle, \ldots, \langle q_m, \mathrm{stay}\rangle)$, and b' is (the graph of) a mapping from b to $C \cup \{\varepsilon\}$, then the constructed PTT $\mathcal{M}'$ has the rule $\langle (q, \gamma), \sigma, j, b'\rangle \to \delta(\langle (q_1, \gamma), \mathrm{stay}\rangle, \ldots, \langle (q_m, \gamma), \mathrm{stay}\rangle)$.

Progressive PTT's can be defined as for PTA's, based on the notion of a reachable configuration, cf. footnote 9. An output form s of the PTT $\mathcal{M}$ on the input tree t is *reachable* if $\langle q_0, \mathrm{root}_t, \varepsilon\rangle \Rightarrow^*_{t,\mathcal{M}} s$ for some $q_0 \in Q_0$. A configuration of $\mathcal{M}$ on t is *reachable* if it occurs in some reachable output form of $\mathcal{M}$ on t. Note that every I-PTT is progressive.

Also, counting PTT's can be defined as for PTA's. For every V_kI-PTT $\mathcal{M}$ an equivalent counting V_kI-PTT $\mathcal{M}'$ can be constructed, just as for PTA's. If $\langle q, \sigma, j, b, \gamma\rangle \to \delta(\langle q_1, \mathrm{stay}\rangle, \ldots, \langle q_m, \mathrm{stay}\rangle)$ is an output rule of $\mathcal{M}$, and φ and b' are as in the proof for PTA's, then $\mathcal{M}'$ has the rule $\langle (q, \varphi), \sigma, j, b'\rangle \to \delta(\langle (q_1, \varphi), \mathrm{stay}\rangle, \ldots, \langle (q_m, \varphi), \mathrm{stay}\rangle)$. Thus, as for PTA's, every V_kI-PTT can be turned into an equivalent progressive V_kI-PTT, with determinism and the absence of invisible pebbles preserved. That implies that V_kI-PTT $\subseteq \mathsf{V}_{k+1}$I-PTT and V_kI-dPTT $\subseteq \mathsf{V}_{k+1}$I-dPTT.

We end this section with an example of an I-PTT.

Example 2. We want to generate itineraries for a trip along the Trans-Siberian Railway, starting in Moscow and ending in Vladivostok, and optionally visiting some cities along the way. An XML document lists all the stops:

```
<stop name="Moscow" large="1" initial="1">
 ...
  <stop name="Birobidzhan" large="0">
   ...
    <stop name="Vladivostok" large="1" final="1" />
   ...
  </stop>
 ...
</stop>
```

The initial and final stops are marked, and for every stop the `large` attribute indicates whether or not the stop is in a large city. We want to generate a list

```
<result>it-1
        <result>it-2
            ...
                <result>it-n
                        <endofresults />
                </result>
            ...
        </result>
</result>
```

where `it-1,it-2,...,it-n` are all itineraries (i.e., lists of stops) that satisfy the constraint that one does not visit a small city twice in a row. An example input XML document, with the corresponding output XML document is given in Tables 1

Table 1
Input.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="transsiberie.xsl"?>

<stop name="Moscow" large="1" initial="1">
  <stop name="Stop 2" large="0">
    <stop name="Stop 3" large="0">
      <stop name="LargeStop 4" large="1">
        <stop name="Stop 5" large="0">
          <stop name="Vladivostok" large="1" final="1"/>
</stop>^5
```

and 2 (where, e.g., `</stop>^3` abbreviates `</stop></stop></stop>`). A deterministic I-PTT $\mathcal{M}_{\text{sib}}$ is able to perform this XML transformation by systematically enumerating all possible lists of stops, marking each stop in the list (except the initial and final stop) by a pebble. Since the pebbles are invisible, $\mathcal{M}_{\text{sib}}$ constructs a possible list of stops on the pebble stack *in reverse*, so that the stops will appear in the output tree in the correct order.

Since in this example the XML tags are ranked, there is no need for a binary encoding of the XML documents. The input alphabet Σ of $\mathcal{M}_{\text{sib}}$ consists of all `<stop at>` where `at` is a possible value of the attributes. The rank of `<stop at>` is 0 if `final="1"` and 1 otherwise. The output alphabet Δ consists of Σ, the tag $r =$ `<result>` of rank 2, and the tag $e =$ `<endofresults>` of rank 0. The set of pebble colours is $C = C_{\text{i}} = \{0, 1\}$, with $C_{\text{v}} = \varnothing$. The transducer $\mathcal{M}_{\text{sib}}$ will not use the attribute `initial`, as it can recognize the root by its child number 0. Also, it will disregard the attribute `large` of the initial and the final stop, and always consider them as large cities. The set of states of $\mathcal{M}_{\text{sib}}$ is $Q = \{q_{\text{start}}, q_1, q_0, q_{\text{out}}, q_{\text{next}}\}$ with $Q_0 = \{q_{\text{start}}\}$.

In the rules below the variables range over the following values: $\sigma_0 \in \Sigma^{(0)}$, $\sigma_1 \in \Sigma^{(1)}$, $j, c \in \{0, 1\}$, and, for $i \in \{0, 1\}$, $\lambda_i \in \{$`<stop at>` $\in \Sigma \mid$ `large="`i`"`$\}$. The I-PTT $\mathcal{M}_{\text{sib}}$ first walks from Moscow to Vladivostok in state q_{start}:

$$\langle q_{\text{start}}, \sigma_1, j, \varnothing\rangle \to \langle q_{\text{start}}, \text{down}_1\rangle$$
$$\langle q_{\text{start}}, \sigma_0, 1, \varnothing\rangle \to \langle q_1, \text{up}\rangle$$

State q_c remembers whether the most recently marked city is small or large; when a new city is marked with a pebble, it gets the colour c. In states q_0 and q_1 as many cities are marked as possible (in the second rule, $c = 1$ or $i = 1$):

$$\langle q_0, \lambda_0, 1, \varnothing\rangle \to \langle q_0, \text{up}\rangle$$
$$\langle q_c, \lambda_i, 1, \varnothing\rangle \to \langle q_i, \text{drop}_c; \text{up}\rangle$$
$$\langle q_c, \sigma_1, 0, \varnothing\rangle \to r(\langle q_{\text{out}}, \text{stay}\rangle, \langle q_{\text{next}}, \text{down}_1\rangle)$$

In state q_{out} an itinerary is generated as output, while state q_{next} continues the search for itineraries by unmarking the most recently marked city:

$$\langle q_{\text{out}}, \sigma_1, 0, \varnothing\rangle \to \sigma_1(\langle q_{\text{out}}, \text{down}_1\rangle)$$
$$\langle q_{\text{out}}, \sigma_1, 1, \varnothing\rangle \to \langle q_{\text{out}}, \text{down}_1\rangle$$
$$\langle q_{\text{out}}, \sigma_1, 1, \{c\}\rangle \to \sigma_1(\langle q_{\text{out}}, \text{lift}_{\text{c}}; \text{down}_1\rangle)$$
$$\langle q_{\text{out}}, \sigma_0, 1, \varnothing\rangle \to \sigma_0$$
$$\langle q_{\text{next}}, \sigma_1, 1, \varnothing\rangle \to \langle q_{\text{next}}, \text{down}_1\rangle$$
$$\langle q_{\text{next}}, \sigma_1, 1, \{c\}\rangle \to \langle q_c, \text{lift}_{\text{c}}; \text{up}\rangle$$
$$\langle q_{\text{next}}, \sigma_0, 1, \varnothing\rangle \to e$$

Note that this XML transformation cannot be realized by a V-PTT, because the height of the output tree is, in general, exponential in the size of the input tree, whereas it is polynomial for V-PTT's (cf. [20, Lemma 7]). □

4. Decomposition

In this section we decompose every PTT into a sequence of TT's, i.e., transducers without pebbles. This is useful as it will give us information on the domain of a PTT, see Theorem 11, and on the complexity of typechecking the PTT, see Theorem 8.

It is possible to reduce the number of visible pebbles used, by preprocessing the input tree with a total deterministic TT. This was shown in [20, Lemma 9] for transducers with only visible pebbles. The basic idea of that proof can be extended to include invisible pebbles.

Table 2
Output.

```
<result>
  <stop name="Moscow" large="1" initial="1">
    <stop name="Stop 3" large="0">
      <stop name="LargeStop 4" large="1">
        <stop name="Stop 5" large="0">
          <stop name="Vladivostok" large="1" final="1"/>
  </stop>^4
  <result>
    <stop name="Moscow" large="1" initial="1">
      <stop name="Stop 2" large="0">
        <stop name="LargeStop 4" large="1">
          <stop name="Stop 5" large="0">
            <stop name="Vladivostok" large="1" final="1"/>
    </stop>^4
    <result>
      <stop name="Moscow" large="1" initial="1">
        <stop name="LargeStop 4" large="1">
          <stop name="Stop 5" large="0">
            <stop name="Vladivostok" large="1" final="1"/>
      </stop>^3
      <result>
        <stop name="Moscow" large="1" initial="1">
          <stop name="Stop 5" large="0">
            <stop name="Vladivostok" large="1" final="1"/>
        </stop>^2
        <result>
          <stop name="Moscow" large="1" initial="1">
            <stop name="Stop 3" large="0">
              <stop name="LargeStop 4" large="1">
                <stop name="Vladivostok" large="1" final="1"/>
          </stop>^3
          <result>
            <stop name="Moscow" large="1" initial="1">
              <stop name="Stop 2" large="0">
                <stop name="LargeStop 4" large="1">
                  <stop name="Vladivostok" large="1" final="1"/>
            </stop>^3
            <result>
              <stop name="Moscow" large="1" initial="1">
                <stop name="LargeStop 4" large="1">
                  <stop name="Vladivostok" large="1" final="1"/>
              </stop>^2
              <result>
                <stop name="Moscow" large="1" initial="1">
                  <stop name="Stop 3" large="0">
                    <stop name="Vladivostok" large="1" final="1"/>
                </stop>^2
                <result>
                  <stop name="Moscow" large="1" initial="1">
                    <stop name="Stop 2" large="0">
                      <stop name="Vladivostok" large="1" final="1"/>
                  </stop>^2
                  <result>
                    <stop name="Moscow" large="1" initial="1">
                      <stop name="Vladivostok" large="1" final="1"/>
                    </stop>
                    <endofresults/>
                  </result>
                </result>
</result>^8
```

Lemma 3. *Let $k \geq 1$. For every V_kI-PTT $\mathcal{M}$ a total deterministic* TT $\mathcal{N}$ *and a V_{k-1}*I-PTT $\mathcal{M}'$ *can be constructed in polynomial time such that $\tau_{\mathcal{N}} \circ \tau_{\mathcal{M}'} = \tau_{\mathcal{M}}$. If $\mathcal{M}$ is deterministic, then so is $\mathcal{M}'$. Hence, for every $k \geq 1$,*

$$\mathsf{V}_k\text{I-PTT} \subseteq \mathsf{tdTT} \circ \mathsf{V}_{k-1}\text{I-PTT} \textit{ and } \mathsf{V}_k\text{I-dPTT} \subseteq \mathsf{tdTT} \circ \mathsf{V}_{k-1}\text{I-dPTT}.$$

Proof. Let $\mathcal{M} = (\Sigma, \Delta, Q, Q_0, C, C_{\mathrm{v}}, C_{\mathrm{i}}, R, k)$ be a PTT with k visible pebbles. The construction of the TT $\mathcal{N}$ and the PTT $\mathcal{M}'$ with $k-1$ visible pebbles is a straightforward extension of the one in [14, Theorem 5], which slightly differs from the one in the proof of [20, Lemma 9], but uses the same basic idea. For completeness sake we repeat a large part of the proof of [14, Theorem 5], adapted to the current formalism. The simple idea of the proof is to preprocess the input tree $t \in T_\Sigma$ in such a way that the dropping and lifting of the first visible pebble can be simulated by walking into and out of specific areas of the preprocessed input tree $\mathrm{pp}(t)$. This preprocessing is independent of the given pebble tree transducer $\mathcal{M}$. More precisely, $\mathrm{pp}(t)$ is obtained from t by attaching to each node u of t, as an additional (last) subtree, a fresh copy of t in

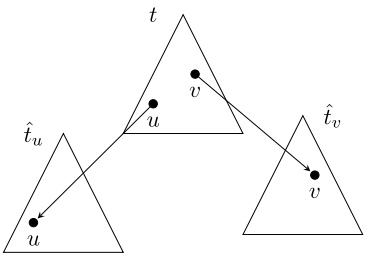

Fig. 3. Output tree pp(t) of the TT $\mathcal{N}$ of Lemma 3 for input tree t.

which (the copy of) node u is marked; let us denote this subtree by t_u. Thus, if t has n nodes, then pp(t) has $n+n^2$ nodes. The subtrees t_u of pp(t) are the "specific areas" mentioned above. As long as there are no visible pebbles on t, $\mathcal{M}'$ stepwise simulates $\mathcal{M}$ on the original nodes of t, which form the "top level" of pp(t). When $\mathcal{M}$ drops the first visible pebble c on node u, $\mathcal{M}'$ enters t_u and walks to the marked node, storing c in its finite state. As long as $\mathcal{M}$ keeps pebble c on the tree, $\mathcal{M}'$ stays in t_u, stepwise simulating $\mathcal{M}$ on t_u rather than t. Since u is marked in t_u, $\mathcal{M}$'s pebble c at u is visible to the transducer $\mathcal{M}'$, not as a pebble but as a marked node. Thus, during this time, $\mathcal{M}'$ only uses $k-1$ visible pebbles. When $\mathcal{M}$ lifts pebble c from u (and hence all visible pebbles are lifted), $\mathcal{M}'$ walks from the copy of u out of t_u, back to the original node u, and continues simulating $\mathcal{M}$ on the top level of pp(t) until $\mathcal{M}$ again drops a visible pebble. There is one problem: how does $\mathcal{M}'$ know whether or not pebble c is on top of the stack when $\mathcal{M}$ tries to lift it? To solve this problem, $\mathcal{M}'$ uses an additional special invisible pebble $\odot$. It drops pebble $\odot$ at the copy of u and thus knows that pebble c is at the top of the stack (for $\mathcal{M}$) when it observes pebble $\odot$. Thus, at any moment of time, $\mathcal{M}'$ has the same pebble stack as $\mathcal{M}$, except that c is replaced by $\odot$ and, moreover, the (invisible) pebbles below $\odot$ are on the top level of pp(t), whereas $\odot$ and the pebbles above it are on t_u.

Unfortunately, this preprocessing cannot be realized by a TT (though it can easily be realized by a v_1-PTT). For this reason we "fold" t_u at the node u, such that (the marked copy of) u becomes its root; let us denote the resulting tree by $\hat{t}_u$. Roughly, $\hat{t}_u$ is obtained from t_u by inverting the parent-child relationship between the ancestors of u (including u), similarly as in the tree traversal algorithm sometimes known as "link inversion" [34, p.562]. Appropriate information is added to the node labels of those ancestors to reflect this inversion. As these changes are local (i.e., each node keeps the same neighbours) and clearly marked in the tree, $\mathcal{M}'$ can easily reconstruct the unfolded t_u, and simulate $\mathcal{M}$ as before. Note also that, with this change of pp(t), dropping or lifting of the first visible pebble can be simulated by $\mathcal{M}'$ in one computation step, because the marked copy of u is the last child of the original u.

Now a TT $\mathcal{N}$ can compute pp(t), as follows.[12] It copies t to the output (adding primes to its labels), but when it arrives at node u it additionally outputs the copy $\hat{t}_u$ of t in a side branch of the computation. Copying the descendants of u "down stream" is an easy recursive task. To invert the parent-child relationship between the nodes on the path from u to root_t, $\mathcal{N}$ uses a single process that walks along the nodes of that path "up stream" to the root, inverting the relationships in the copy. Copies of other siblings of children on the path are connected as in t, and their descendants are copied "down stream". More precisely, if in t the i-th child v of parent w is on the path, then, in the output $\hat{t}_u$, v has an additional (last) child that corresponds to w, and w has the same children (with their descendants) as in t, except that its i-th child is a node that is labeled by the bottom symbol $\perp$ of rank 0. For the sake of uniformity, root_t is also given an additional (last) child, with label $\perp$. Note that the nodes of t correspond one-to-one to the non-bottom nodes of $\hat{t}_u$; in particular, the path in t from u to root_t corresponds to the path in $\hat{t}_u$ from its root to the parent of its rightmost leaf. The bottom nodes of $\hat{t}_u$ will not be visited by $\mathcal{M}'$.

A picture of pp(t) is given in Fig. 3, where $\hat{t}_u$ is drawn for two nodes only. Note that in this picture the root of the copy of t (which is also the root of pp(t)) is the top of the triangle, but the root of $\hat{t}_u$ is u (and, of course, similarly for v). As a concrete example, consider $t = \sigma(\delta(a,b),c)$ where σ, δ have rank 2 and a, b, c rank 0. We will name the nodes of t by their labels. Then

$$\mathrm{pp}(t) = \sigma'(\delta'(a'(\hat{t}_a), b'(\hat{t}_b), \hat{t}_\delta), c'(\hat{t}_c), \hat{t}_\sigma)$$

where

$$\begin{aligned}
\hat{t}_a &= a_{0,1}(\delta_{1,1}(\perp, b, \sigma_{1,0}(\perp, c, \perp))),\\
\hat{t}_b &= b_{0,2}(\delta_{2,1}(a, \perp, \sigma_{1,0}(\perp, c, \perp))),\\
\hat{t}_\delta &= \delta_{0,1}(a, b, \sigma_{1,0}(\perp, c, \perp)),\\
\hat{t}_c &= c_{0,2}(\sigma_{2,0}(\delta(a,b), \perp, \perp)), \text{ and}\\
\hat{t}_\sigma &= \sigma_{0,0}(\delta(a,b), c, \perp).
\end{aligned}$$

[12] See also [41, Example 3.7] where $\hat{t}_u$ occurs as "a complex rotation of the input tree" t, albeit for leaves u only.

The subscripted node labels are on the rightmost paths of the $\hat{t}_u$'s; the subscripts contain "reconstruction" information, to be explained below. As another example, if t is the monadic tree $a(b^m(c(d^n(e))))$ of height $m+n+3$, and u is the c-labeled node, then $\hat{t}_u = c_{0,1}(s_1, s_2)$ with $s_1 = d^n(e)$ and s_2 is the binary tree $b_{1,1}(\bot, b_{1,1}(\bot, \dots b_{1,1}(\bot, a_{1,0}(\bot, \bot))\cdots))$ of height $m+2$. This shows more clearly that $\hat{t}_u$ is obtained by "folding".

We now formally define the deterministic TT $\mathcal{N}$ that, for given ranked alphabet Σ, realizes the preprocessing pp (called EncPeb in [20]). The definition is identical to the one in [14, Section 6]. Since $\mathcal{N}$ has no pebbles, we abbreviate the left-hand side $\langle q, \sigma, j, \varnothing\rangle$ of a rule by $\langle q, \sigma, j\rangle$. To simplify the definition of $\mathcal{N}$ we additionally allow output rules of the form $\langle q, \sigma, j\rangle \to \delta(s_1, \dots, s_m)$ where δ is an output symbol of rank m and every s_i is either the output symbol $\bot$ or it is of the form $\langle q', \varphi\rangle$ where φ is stay, up, or down$_i$ with $i \in [1, m]$. Such a rule should be replaced by the rules $\langle q, \sigma, j\rangle \to \delta(\langle p_1, \text{stay}\rangle, \dots, \langle p_m, \text{stay}\rangle)$ and $\langle p_j, \sigma, j\rangle \to s_j$ for all $j \in [1, m]$, where $p_1, \dots, p_m$ are new states. Obviously this replacement can be done in quadratic time.

We introduce the states and rules of $\mathcal{N}$ one by one; in what follows σ ranges over Σ, with $m = \text{rank}_\Sigma(\sigma)$, j ranges over $[0, mx_\Sigma]$, and i over $[1, m]$. First, $\mathcal{N}$ has an "identity" state d that just recursively copies the subtree of the current node to the output, using the rules $\langle d, \sigma, j\rangle \to \sigma(\langle d, \text{down}_1\rangle, \dots, \langle d, \text{down}_m\rangle)$. Then, $\mathcal{N}$ has initial state g that copies the input tree t to the output (with primed labels) and at each node u of t "generates" a new copy $\hat{t}_u$ of the input tree by calling the state f that computes $\hat{t}_u$ by "folding" t_u. The rules for g are

$$\langle g, \sigma, j\rangle \to \sigma'(\langle g, \text{down}_1\rangle, \dots, \langle g, \text{down}_m\rangle, \langle f, \text{stay}\rangle).$$

Note that σ' has rank $m+1$: the root of $\hat{t}_u$ is attached to u as its last child. The rules for f are

$$\langle f, \sigma, j\rangle \to \sigma_{0,j}(\langle d, \text{down}_1\rangle, \dots, \langle d, \text{down}_m\rangle, \xi_j)$$

where $\xi_j = \langle f_j, \text{up}\rangle$ for $j \neq 0$, and $\xi_0 = \bot$. The "reconstruction" subscripts of $\sigma_{0,j}$ mean the following: subscript 0 indicates that this node is the root of some $\hat{t}_u$, and subscript j is the child number of u in t. Note that $\sigma_{0,j}$ has rank $m+1$: its last child corresponds to the parent of u in t (viewing $\bot$ as the "parent" of root$_t$ in t). The TT $\mathcal{N}$ walks up along the path from u to the root of t using "folding" states f_i, where the i indicates that in the previous step $\mathcal{N}$ was at the i-th child of the current node. The rules for f_i are

$$\begin{aligned}\langle f_i, \sigma, j\rangle \to \sigma_{i,j}(&\\ &\langle d, \text{down}_1\rangle, \dots, \langle d, \text{down}_{i-1}\rangle,\\ &\bot,\\ &\langle d, \text{down}_{i+1}\rangle, \dots, \langle d, \text{down}_m\rangle,\\ &\xi_j)\end{aligned}$$

where ξ_j is as above. If a node (in $\hat{t}_u$) with label $\sigma_{i,j}$ corresponds to the node v in t, then the "reconstruction" subscript i means that its parent corresponds to the i-th child of v in t (and its own i-th child is $\bot$), and, as above, "reconstruction" subscript j is the child number of v. Just as $\sigma_{0,j}$, also $\sigma_{i,j}$ has rank $m+1$: its last child corresponds to the parent of v in t. Note that the copy $\hat{t}_u$ of the input tree is computed by the states f, f_i (for every i) and d, such that f copies node u to the output and the other states walk from u to every other node v of t and copy v to the output. To be precise, $\mathcal{N}$ walks from u to v along the shortest (undirected) path from u to v, from u up to the least common ancestor of u and v (in the states f_i), and then down to v (in the state d). Arriving in a node v from a neighbour of v, the transducer $\mathcal{N}$ branches into a new process for every other neighbour of v.

This ends the description of the TT $\mathcal{N}$. The output alphabet Γ of $\mathcal{N}$ (which will also be the input alphabet of $\mathcal{M}'$) is the union of Σ, $\{\bot\}$, $\{\sigma' \mid \sigma \in \Sigma\}$, and $\{\sigma_{i,j} \mid \sigma \in \Sigma,\ i \in [0, \text{rank}_\Sigma(\sigma)],\ j \in [0, mx_\Sigma]\}$. Thus, $\mathcal{N}$ has $O(n^2)$ output symbols, where n is the size of Σ.[13] So, since $mx_\Gamma = mx_\Sigma + 1$, the size of Γ is polynomial in n. The set of states of $\mathcal{N}$ is $\{d, g, f\} \cup \{f_i \mid i \in [1, mx_\Sigma]\}$, with initial state g. Thus, it has $O(n)$ states and $O(n^3)$ rules; moreover, each of these rules is of size $O(n \log n)$. Hence, the size of $\mathcal{N}$ is polynomial in the size of Σ, and it can be constructed in polynomial time.

We now turn to the description of the v_{k-1}I-PTT $\mathcal{M}'$. It has input alphabet Γ, output alphabet Δ, set of states $Q \cup (Q \times C_\text{v})$, and the same initial states and visible colours as $\mathcal{M}$. Its invisible colour set is $C'_\text{i} = C_\text{i} \cup \{\odot\}$. It remains to discuss the set R' of rules of $\mathcal{M}'$. Let $\langle q, \sigma, j, b\rangle \to \zeta$ be a rule of $\mathcal{M}$ with $\text{rank}_\Sigma(\sigma) = m$. We consider four cases, depending on the variant σ', $\sigma_{0,j}$, $\sigma_{i,j}$ with $i \neq 0$, or σ in Γ of the input symbol $\sigma \in \Sigma$.

In the first case, we consider the behaviour of $\mathcal{M}'$ in state q on σ', and we assume that $b \cap C_\text{v} = \varnothing$. If $\zeta = \langle q', \text{drop}_c\rangle$ with $c \in C_\text{v}$, then R' contains the rule $\langle q, \sigma', j, b\rangle \to \langle (q', c), \text{down}_{m+1}; \text{drop}_\odot\rangle$,[14] and otherwise R' contains the rule $\langle q, \sigma', j, b\rangle \to \zeta$. Thus, $\mathcal{M}'$ simulates $\mathcal{M}$ on the original (now primed) part of the input tree t in pp(t), until $\mathcal{M}$ drops

[13] We assume here that the rank of each symbol of the ranked alphabet Σ is specified in unary rather than decimal notation, and thus $mx_\Sigma \leq n$; cf. the last paragraph of [14, Section 2].

[14] To be completely formal, this rule should be replaced by the two rules $\langle q, \sigma', j, b\rangle \to \langle p, \text{down}_{m+1}\rangle$ and $\langle p, \sigma_{0,j}, m+1, \varnothing\rangle \to \langle (q', c), \text{drop}_\odot\rangle$, where p is a new state.

a visible pebble c on node u. Then $\mathcal{M}'$ steps to the root of $\hat{t}_u$ where it drops the invisible pebble $\odot$, and stores c in its finite state.

Next, we let $c \in C_{\mathrm{v}}$ and we consider the behaviour of $\mathcal{M}'$ in state (q, c) on the remaining variants of σ. Let ζ_c be the result of changing in ζ every occurrence of a state q' into (q', c).

In the second case we assume that $c \in b$ (corresponding to the fact that $\sigma_{0,j}$ labels the marked node of some $\hat{t}_u$). If $b = \{c\}$ and $\zeta = \langle q', \mathrm{lift}_c \rangle$, then R' contains the rule $\langle (q, c), \sigma_{0,j}, m+1, \{\odot\} \rangle \to \langle q', \mathrm{lift}_\odot; \mathrm{up} \rangle$.[15] Thus, when $\mathcal{M}$ lifts visible pebble c from node u, $\mathcal{M}'$ lifts invisible pebble $\odot$ and steps from the root of $\hat{t}_u$ back to node u. Otherwise, R' contains the rules

$$\langle (q, c), \sigma_{0,j}, m+1, b \setminus \{c\} \cup \{\odot\} \rangle \to \zeta'_c$$

(provided $b \cap C_{\mathrm{i}} = \varnothing$) and

$$\langle (q, c), \sigma_{0,j}, m+1, b \setminus \{c\} \rangle \to \zeta'_c,$$

where ζ'_c is obtained from ζ_c by changing up into down_{m+1}. These two rules correspond to whether or not the invisible pebble $\odot$ is observable. Note that the child number in $\mathrm{pp}(t)$ of a node with label $\sigma_{0,j}$ is always $m+1$ (and the label of its parent is σ').

In the remaining two cases we assume that $c \notin b$ in the above rule of $\mathcal{M}$. In the third case, we consider $\sigma_{i,j}$ with $i \neq 0$. Then R' contains the rules $\langle (q, c), \sigma_{i,j}, j', b \rangle \to \zeta'_c$ for every $j' \in [1, mx_\Gamma]$, where ζ'_c is now obtained from ζ_c by changing up into down_{m+1}, and down_i into up. In the fourth and final case, we consider σ itself (in Γ). Then R' contains the rule $\langle (q, c), \sigma, j, b \rangle \to \zeta_c$. Thus, $\mathcal{M}'$ stepwise simulates $\mathcal{M}$ on every $\hat{t}_u$.

This ends the description of the v_{k-1}I-PTT $\mathcal{M}'$. It should now be clear that $\tau_{\mathcal{M}'}(\mathrm{pp}(t)) = \tau_{\mathcal{M}}(t)$ for every $t \in T_\Sigma$, and hence $\tau_{\mathcal{N}} \circ \tau_{\mathcal{M}'} = \tau_{\mathcal{M}}$. Each rule of $\mathcal{M}$ is turned into at most $1 + \#(C_{\mathrm{v}}) \cdot (2 + mx_\Sigma(mx_\Sigma + 1))$ rules of $\mathcal{M}'$, of the same size as that rule (disregarding the space taken by the occurrences of c and $m+1$). Thus, $\mathcal{M}'$ can be computed from $\mathcal{M}$ in polynomial time. □

The tree $\mathrm{pp}(t)$ that is used in the previous proof consists of two levels of copies of the original input tree t; on the first level a straightforward copy of t (used until the first visible pebble is dropped) and a second level of copies $\hat{t}_u$ (used to "store" the first visible pebble dropped). It is tempting to add another level, meant as a way to store the next visible pebble dropped. The problem with this is that it would make the first visible pebble effectively unobservable when the next one is dropped. The idea *can* be used for invisible pebbles, for arbitrarily many levels.

Lemma 4. *For every* I-PTT $\mathcal{M}$ *a* TT $\mathcal{N}$ *and a* TT $\mathcal{M}'$ *can be constructed in polynomial time such that* $\tau_{\mathcal{N}} \circ \tau_{\mathcal{M}'} = \tau_{\mathcal{M}}$. *If* $\mathcal{M}$ *is deterministic, then so is* $\mathcal{M}'$. *Hence,* I-PTT $\subseteq$ TT $\circ$ TT *and* I-dPTT $\subseteq$ TT $\circ$ dTT.

Proof. The computation of a PTT $\mathcal{M}$ with invisible pebbles on tree t is simulated by a TT $\mathcal{M}'$ (without pebbles) on tree t'. The input tree t is preprocessed in a nondeterministic way by a TT $\mathcal{N}$ to obtain t'. The top level of t' is a copy of t, as before. On the next level, since the simulating transducer $\mathcal{M}'$ cannot store the colours of all the pebbles in its finite state (as we did for one colour in the proof of Lemma 3), $\mathcal{N}$ does not attach one copy $\hat{t}_u$ of t to each node u of t but $\#(C_{\mathrm{i}})$ such copies, one for each pebble colour. In this way, the child number in t' of the root of $\hat{t}_u$ represents the pebble colour. In fact, in each node u of t the transducer $\mathcal{N}$ nondeterministically decides for each pebble colour c whether or not to spawn a process that copies t into $\hat{t}_u$, and this is a recursive process: in each node in each copy of t it can be decided to spawn such processes that generate new copies.

In this way a "tree of trees" is constructed. For an "artist impression" of such an output tree t', see Fig. 4. The child number in t' of the root of each copy $\hat{t}_u$ indicates an invisible pebble of colour c placed at node u in the original tree t. In each copy only one pebble is observable, the one represented by the child number of its root, exactly as the last pebble dropped in the original computation. In the simulation, moving down or up along the tree of trees corresponds to dropping and lifting invisible pebbles.

In general there is no bound on the depth of the stack of pebbles during a computation of $\mathcal{M}$. The preprocessor $\mathcal{N}$ nondeterministically constructs t'. If t' is not sufficiently deep, the simulating transducer $\mathcal{M}'$ aborts the computation. Conversely, for every computation of $\mathcal{M}$ a tree t' of sufficient depth can be constructed nondeterministically from t.

We now turn to the formal definitions. Let $\mathcal{M} = (\Sigma, \Delta, Q, Q_0, C, C_{\mathrm{v}}, C_{\mathrm{i}}, R, 0)$ be an I-PTT. Without loss of generality we assume that $C = C_{\mathrm{i}}$ and that $C = [1, \gamma]$ for some $\gamma \in \mathbb{N}$. This choice of C simplifies the representation of colours by child numbers.

First, we define the nondeterministic TT $\mathcal{N}$ that preprocesses the trees over Σ. It is a straightforward variant of the one in the proof of Lemma 3. The output alphabet Γ of $\mathcal{N}$ is now the union of $\{\bot\}$, $\{\sigma' \mid \sigma \in \Sigma\}$, and $\{\sigma'_{i,j} \mid \sigma \in \Sigma, i \in$

[15] Again, to be completely formal, this rule should be replaced by the two rules $\langle (q, c), \sigma_{0,j}, m+1, \{\odot\} \rangle \to \langle p, \mathrm{lift}_\odot \rangle$ and $\langle p, \sigma_{0,j}, m+1, \varnothing \rangle \to \langle q', \mathrm{up} \rangle$, where p is a new state.

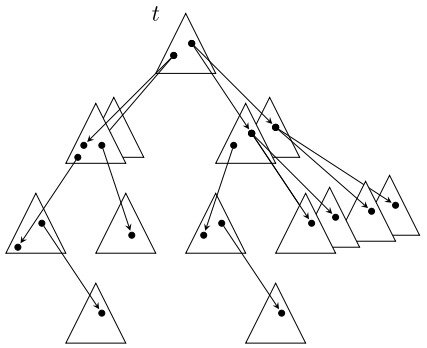

Fig. 4. An output tree t' of the TT $\mathcal{N}$ of Lemma 4 for input tree t.

$[0, \mathrm{rank}_\Sigma(\sigma)]$, $j \in [0, mx_\Sigma]\}$ where, for every $\sigma \in \Sigma$ of rank m, σ' has rank $m+\gamma$ and $\sigma'_{i,j}$ has rank $m+\gamma+1$, because γ processes are spawned at each node, and each of these processes generates, nondeterministically, either a copy $\hat{t}_u$ of t or the bottom symbol $\perp$. The set of states of $\mathcal{N}$ is as before, except that the state d is removed (with its rules). In the rules of $\mathcal{N}$ we will use $\langle f, \mathrm{stay}\rangle^\gamma$ as an abbreviation of the sequence $\langle f, \mathrm{stay}\rangle, \ldots, \langle f, \mathrm{stay}\rangle$ of length γ. The rules for the initial state g are

$$\langle g, \sigma, j\rangle \to \sigma'(\langle g, \mathrm{down}_1\rangle, \ldots, \langle g, \mathrm{down}_m\rangle, \langle f, \mathrm{stay}\rangle^\gamma).$$

The rules for f are

$$\langle f, \sigma, j\rangle \to \perp$$
$$\langle f, \sigma, j\rangle \to \sigma'_{0,j}(\langle g, \mathrm{down}_1\rangle, \ldots, \langle g, \mathrm{down}_m\rangle, \langle f, \mathrm{stay}\rangle^\gamma, \xi_j)$$

where, as before, $\xi_j = \langle f_j, \mathrm{up}\rangle$ for $j \neq 0$, and $\xi_0 = \perp$. Finally, the rules for f_i are

$$\begin{aligned}\langle f_i, \sigma, j\rangle \to \sigma'_{i,j}(&\\ &\langle g, \mathrm{down}_1\rangle, \ldots, \langle g, \mathrm{down}_{i-1}\rangle,\\ &\perp,\\ &\langle g, \mathrm{down}_{i+1}\rangle, \ldots, \langle g, \mathrm{down}_m\rangle,\\ &\langle f, \mathrm{stay}\rangle^\gamma,\\ &\xi_j)\end{aligned}$$

where ξ_j is as above. This ends the definition of $\mathcal{N}$.

Next, we define the simulating TT $\mathcal{M}'$. It has input alphabet Γ (the output alphabet of $\mathcal{N}$), output alphabet Δ, and the same set of states and initial states as $\mathcal{M}$. The set R' of rules of $\mathcal{M}'$ is defined as follows. Let $\langle q, \sigma, j, b\rangle \to \zeta$ be a rule of $\mathcal{M}$ with $\mathrm{rank}_\Sigma(\sigma) = m$. Note that b is either empty or a singleton. We consider three cases, that describe the behaviour of $\mathcal{M}'$ on the symbols σ', $\sigma'_{0,j}$, and $\sigma'_{i,j}$ with $i \neq 0$.

In the first case we assume that $b = \varnothing$ (and hence ζ does not contain a lift-instruction). Then R' contains the rule $\langle q, \sigma', j\rangle \to \zeta'$ where ζ' is obtained from ζ by changing drop_c into down_{m+c} for every $c \in C$.

In the second case we assume that $b = \{c\}$ for some $c \in C$. Then R' contains the rule $\langle q, \sigma'_{0,j}, m+c\rangle \to \zeta'$ where ζ' is now obtained from ζ by changing up into $\mathrm{down}_{m+\gamma+1}$, lift_c into up, and drop_d into down_{m+d} for every $d \in C$. Note that the child number in t' of a node with label $\sigma'_{0,j}$ is always $m+c$ for some $c \in C$ (and the label of its parent is σ' or $\sigma'_{i,j}$ for some $i \in [0, m]$).

In the third case we assume (as in the first case) that $b = \varnothing$. Then R' contains the rule $\langle q, \sigma'_{i,j}, j'\rangle \to \zeta'$ for every $j' \in [1, mx_\Gamma]$, where ζ' is now obtained from ζ by changing up into $\mathrm{down}_{m+\gamma+1}$, down_i into up, and drop_c into down_{m+c} for every $c \in C$.

This ends the definition of $\mathcal{M}'$. It should, again, be clear that for every $t \in T_\Sigma$ and $s \in T_\Delta$, $s \in \tau_{\mathcal{M}}(t)$ if and only if there exists $t' \in \tau_{\mathcal{N}}(t)$ such that $s \in \tau_{\mathcal{M}'}(t')$. Hence $\tau_{\mathcal{N}} \circ \tau_{\mathcal{M}'} = \tau_{\mathcal{M}}$. It is straightforward to show, as in the proof of Lemma 3, that $\mathcal{N}$ and $\mathcal{M}'$ can be constructed in polynomial time from $\mathcal{M}$. Note that $mx_\Gamma = mx_\Sigma + \#(C_\mathrm{i}) + 1$ and so the size of Γ is polynomial in the size of $\mathcal{M}$. □

Combining the previous two results we can inductively decompose tree transducers with (visible and invisible) pebbles into tree transducers without pebbles.

Theorem 5. *For every $k \geq 0$, $\mathsf{V}_k\mathsf{I}\text{-}\mathsf{PTT} \subseteq \mathsf{TT}^{k+2}$. For fixed k, the involved construction takes polynomial time.*

Observe that $\mathsf{V}_k\text{-PTT} \subseteq \mathsf{V}_{k-1}\text{I-PTT}$ as the topmost pebble can be replaced by an invisible one, thus $\mathsf{V}_k\text{-PTT} \subseteq \mathsf{TT}^{k+1}$, which was proved in [20, Theorem 10], also for the deterministic case.

We do not know whether Theorem 5 is optimal, i.e., whether or not $\mathsf{V}_k\text{I-PTT}$ is included in TT^{k+1}. The deterministic version of Theorem 5 (for $k \neq 0$) will be proved in Section 15 (Theorem 55), and we will show that it is optimal (after Theorem 56).

The nondeterminism of the "preprocessing" transducer $\mathcal{N}$ in the proof of Lemma 4 is rather limited. The general form of the constructed tree is completely determined by the input tree, only the depth of the construction is nondeterministically chosen. At the same time it remains nondeterministic even when we start with a deterministic PTT with invisible pebbles: $\mathsf{I\text{-}dPTT} \subseteq \mathsf{TT} \circ \mathsf{dTT}$. However, we can obtain a deterministic transduction if the number of invisible pebbles used by the transducer is bounded (over all input trees), cf. the M. Sc. Thesis of the third author [49] (where visible and invisible pebbles are called global and local pebbles, respectively). In Section 7 we will show that if we start with a deterministic tree transduction, then the inclusions of Lemma 4 also hold in the other direction (Theorem 17). In Section 15 we will show that $\mathsf{I\text{-}dPTT} \subseteq \mathsf{dTT}^3$ (Corollary 54).

5. Typechecking

The *inverse type inference problem* is to construct, for a tree transducer $\mathcal{M}$ and a regular tree grammar G_{out}, a regular tree grammar G_{in} such that $L(G_{\text{in}}) = \tau_{\mathcal{M}}^{-1}(L(G_{\text{out}}))$. The *typechecking problem* asks, for a tree transducer $\mathcal{M}$ and two regular tree grammars G_{in} and G_{out}, whether or not $\tau_{\mathcal{M}}(L(G_{\text{in}})) \subseteq L(G_{\text{out}})$. The inverse type inference problem can be used to solve the typechecking problem, because $\tau_{\mathcal{M}}(L(G_{\text{in}})) \subseteq L(G_{\text{out}})$ if and only if $L(G_{\text{in}}) \cap \tau_{\mathcal{M}}^{-1}(L'_{\text{out}}) = \varnothing$, where L'_{out} is the complement of $L(G_{\text{out}})$.

It was shown in [41] (see also [20, Section 7]) that both problems are solvable for tree-walking tree transducers with visible pebbles, i.e., for V-PTT's, and hence in particular for tree-walking tree transducers without pebbles, i.e., for TT's.[16] This was extended in [14] to compositions of such transducers and, moreover, the time complexity of the involved algorithms was improved, using a result of [3] for attributed tree transducers.

We define a *k-fold exponential* function to be a function of the form $2^{g(n)}$ where g is a $(k-1)$-fold exponential function; a 0-fold exponential function is a polynomial.

Proposition 6. *For fixed $k \geq 0$, the inverse type inference problem is solvable*

(1) *for compositions of k TT's in k-fold exponential time, and*
(2) *for V_k-PTT's in $(k+1)$-fold exponential time.*

Proposition 7. *For fixed $k \geq 0$, the typechecking problem is solvable*

(1) *for compositions of k TT's in $(k+1)$-fold exponential time, and*
(2) *for V_k-PTT's in $(k+2)$-fold exponential time.*

As also observed in [14], one exponential can be taken off the results of Proposition 7 if we assume that G_{out} is a total deterministic bottom-up finite-state tree automaton, because that exponential is due to the complementation of $L(G_{\text{out}})$.

It is immediate from Theorem 5 and Propositions 6(1) and 7(1) that both problems are also solvable for tree-walking tree transducers with invisible pebbles.

Theorem 8. *For fixed $k \geq 0$, the inverse type inference problem and the typechecking problem are solvable for V_kI-PTT's in $(k+2)$-fold and $(k+3)$-fold exponential time, respectively.*

The main conclusion from Proposition 7(2) and Theorem 8 is that the complexity of typechecking PTT's basically depends on the number of visible pebbles used. Thus we can improve the complexity of the problem by changing visible pebbles into invisible ones as much as possible, see Section 10.

Note that the solvability of the inverse type inference problem for a tree transducer $\mathcal{M}$ means in particular that its domain is a regular tree language, taking $L(G_{\text{out}}) = T_\Delta$ where Δ is the output alphabet of $\mathcal{M}$. Thus, it follows from Theorem 8 that the domains of PTT's are regular, or in other words, that every alternating PTA accepts a regular tree language.

Corollary 9. *For every PTT $\mathcal{M}$, its domain $L(\mathcal{M})$ is regular.*

[16] Note however that our definition of inverse type inference differs from the one in [41], where it is required that $L(G_{\text{in}}) = \{ s \mid \tau_{\mathcal{M}}(s) \subseteq L(G_{\text{out}}) \}$. The reason is that our definition is more convenient when considering compositions of tree transducers.

6. Trees, tests and trips

In this section we show that VI-PTA's recognize the regular tree languages, that they compute the MSO definable binary patterns (or trips), and that they can perform MSO tests on the observable part of their configuration (which consists of the position of the head and the positions of the observable pebbles).

For "classical" tree-walking automata with a bounded number of visible pebbles, i.e., for V-PTA's, it was shown in [15, Section 5] that these automata accept regular tree languages only. However, as proved in [8], they cannot accept all regular tree languages. One of the main reasons for introducing an unbounded number of invisible pebbles is that they can be used to recognize every regular tree language. Recall that REGT denotes the class of regular tree languages.

Lemma 10. REGT $\subseteq$ I-dPTA.

Proof. As the regular tree languages are recognized by deterministic bottom-up finite-state tree automata, it suffices to explain how the computation of such an automaton $\mathcal{A}$ can be simulated by a deterministic PTA $\mathcal{A}'$ with invisible pebbles. The computation of $\mathcal{A}$ on the input tree can be reconstructed by a post-order evaluation of the tree. At the current node u, $\mathcal{A}'$ uses an invisible pebble to store the states in which $\mathcal{A}$ arrives at the first m children of u, for some m. The colour of the pebble represents the sequence of states. For each ancestor v of u the pebble stack contains a similar pebble for the first $i-1$ children of v, where vi is the unique child of v that is also an ancestor of u (or u itself). If u has more than m children, then $\mathcal{A}'$ moves to its $(m+1)$-th child and drops a pebble that represents the empty sequence of states of $\mathcal{A}$. Otherwise, $\mathcal{A}'$ computes the state assumed by $\mathcal{A}$ in u based on the states of the children, lifts the pebble at u, and moves to the parent of u to update its pebble with that state. The post-order evaluation ensures that pebbles are used in a nested fashion.

Formally, let $\mathcal{A} = (\Sigma, P, F, \delta)$ where Σ is a ranked alphabet, P is a finite set of states, $F \subseteq P$ is the set of final states, and δ is the transition function that assigns a state $\delta(\sigma, p_1, \ldots, p_m) \in P$ to every $\sigma \in \Sigma$ and $p_1, \ldots, p_m \in P$ with $m = \mathrm{rank}_\Sigma(\sigma)$. As pebble colours the I-PTA $\mathcal{A}'$ has all strings in P^* of length at most mx_Σ. Its states and rules are introduced one by one as follows, where σ ranges over Σ, j and m range over $[0, \mathrm{rank}(\sigma)]$, and $p, p_1, \ldots, p_m$ range over P. The initial state q_0 does not occur in the right-hand side of any rule. In the initial state, the automaton $\mathcal{A}'$ drops a pebble at the root representing the empty sequence of states of $\mathcal{A}$, and goes into the main state $q_\circ$. The rule is

$$\rho_1 : \langle q_0, \sigma, 0, \varnothing\rangle \to \langle q_\circ, \mathrm{drop}_\varepsilon\rangle.$$

In state $q_\circ$, $\mathcal{A}'$ consults the pebble to see whether or not all children have been evaluated, and acts accordingly. For $m < \mathrm{rank}(\sigma)$ it has the rule

$$\rho_2 : \langle q_\circ, \sigma, j, \{p_1 \cdots p_m\}\rangle \to \langle q_\circ, \mathrm{down}_{m+1}; \mathrm{drop}_\varepsilon\rangle,$$

which handles the case that the state of $\mathcal{A}$ is not yet known for all children of node u. For $m = \mathrm{rank}(\sigma)$ and $p = \delta(\sigma, p_1, \ldots, p_m)$ it has the rules

$$\begin{aligned}
&\rho_3 : \langle q_\circ, \sigma, j, \{p_1 \cdots p_m\}\rangle \to \langle \bar{q}_p, \mathrm{lift}_{p_1 \cdots p_m}; \mathrm{up}\rangle && \text{if } j \neq 0,\\
&\rho_4 : \langle q_\circ, \sigma, 0, \{p_1 \cdots p_m\}\rangle \to \langle q_{\mathrm{yes}}, \mathrm{stay}\rangle && \text{if } p \in F,\\
&\rho_5 : \langle q_\circ, \sigma, 0, \{p_1 \cdots p_m\}\rangle \to \langle q_{\mathrm{no}}, \mathrm{stay}\rangle && \text{if } p \notin F,
\end{aligned}$$

and for $m < \mathrm{rank}(\sigma)$ it has the rule

$$\rho_6 : \langle \bar{q}_p, \sigma, j, \{p_1 \cdots p_m\}\rangle \to \langle q_\circ, \mathrm{lift}_{p_1 \cdots p_m}; \mathrm{drop}_{p_1 \cdots p_m p}\rangle.$$

Thus, if the states $p_1, \ldots, p_m$ of $\mathcal{A}$ at all the children of node u are known, $\mathcal{A}'$ computes the state $p = \delta(\sigma, p_1, \ldots, p_m)$ of $\mathcal{A}$ at u. If u is not the root of the input tree, then $\mathcal{A}'$ stores p in its own state $\bar{q}_p$, lifts the pebble $p_1 \cdots p_m$, and moves up to the parent of u. Since the pebble at the parent is now observable, it can be updated. If u is the root of the input tree, then $\mathcal{A}'$ knows whether or not $\mathcal{A}$ accepts that tree, and correspondingly goes into state q_{yes} or state q_{no}, where q_{yes} is the unique final state of $\mathcal{A}'$. Note that there is one pebble left on the root of the tree. □

Adding an infinite supply of invisible pebbles on the other hand does not lead out of the regular tree languages. It is possible to give a proof of this fact by reducing V_kI-PTA's to the backtracking pushdown tree automata of [51], but here we deduce it from the results of the previous section.

Theorem 11. *For each $k \geq 0$, V_kI-PTA $=$ V_kI-dPTA $=$ REGT.*

Proof. By Lemma 10, REGT $\subseteq \mathsf{V}_k$I-dPTA. Conversely, as observed before, a PTA $\mathcal{A}$ is easily turned into a PTT $\mathcal{M}$ that outputs single node tree δ (with $\mathrm{rank}(\delta) = 0$) for trees accepted by $\mathcal{A}$: for every final state q of $\mathcal{A}$ add all rules $\langle q, \sigma, j, b\rangle \to \delta$. Then $L(\mathcal{A}) = L(\mathcal{M})$, the domain of $\mathcal{M}$, which is regular by Corollary 9. □

Note that an infinite supply of *visible* pebbles could be used to mark a's and b's alternatingly and thus accept the nonregular language $\{a^n b^n \mid n \in \mathbb{N}\}$ (and similarly $\{a^n b^n c^n \mid n \in \mathbb{N}\}$). Note also that the stack of pebbles cannot be replaced by two independent stacks, one for visible and one for invisible pebbles. Then we could accept $\{a^n b^n \mid n \in \mathbb{N}\}$ with just one visible pebble: drop an invisible pebble on each a, and then use the visible pebble on the b's to count the number of a's, by lifting one invisible pebble (in fact, the unique observable one) for each b.

Recall from Section 2 that an n-ary *pattern* over a ranked alphabet Σ is a set $T \subseteq \{(t, u_1, \ldots, u_n) \mid t \in T_\Sigma,\ u_1, \ldots, u_n \in N(t)\}$. Recall also that the pattern T is said to be regular if its marked representation $\mathrm{mark}(T) \subseteq T_{\Sigma \times \{0,1\}^n}$ is a regular tree language. In fact, T is regular if and only if it is MSO definable, which means that there is an MSO formula $\varphi(x_1, \ldots, x_n)$ over Σ such that $T = T(\varphi)$, where $T(\varphi) = \{(t, u_1, \ldots, u_n) \mid t \models \varphi(u_1, \ldots, u_n)\}$. Recall finally that a unary pattern ($n = 1$) is called a *site*, and a binary pattern ($n = 2$) is called a *trip*.

With the help of an unbounded supply of invisible pebbles tree-walking automata can recognize regular tree languages, Lemma 10. Likewise v_nI-PTA's can match arbitrary MSO definable n-ary patterns φ. When n visible pebbles are dropped on a sequence of n nodes, the invisible pebbles can be used to evaluate the tree, and test whether it belongs to the regular tree language $\mathrm{mark}(T(\varphi))$. In Section 10 we will consider pattern matching in detail.

Ignoring the visible pebbles, it is also possible to consider just the position of the head, and test whether the input tree together with that position belongs to a given regular "marked" tree language. We say that a family $\mathcal{F}$ of PTA's (or PTT's) can *perform* MSO *head tests* if, for a regular site T over Σ, an automaton (or transducer) in $\mathcal{F}$ can test whether or not $(t, h) \in T$, where t is the input tree and h the position of the head at the moment of the test. Admittedly, this is a very informal definition. To formalize it we have to define a $\mathrm{PTA}^{\mathrm{MSO}}$ (or a $\mathrm{PTT}^{\mathrm{MSO}}$), i.e., a PTA (or PTT) *with* MSO *head tests*, that has rules of the form $\langle q, \sigma, j, b, T\rangle \to \zeta$ where T is a regular site over Σ (specified in some effective way). Such a rule is relevant to a configuration $\langle q, h, \pi\rangle$ on a tree t if, in addition, $(t, h) \in T$. Since the regular tree languages are closed under complement, the complement T^{c} of T can be tested in a rule with left-hand side $\langle q, \sigma, j, b, T^{\mathrm{c}}\rangle$. Such an automaton (or transducer) is deterministic if for every two distinct rules $\langle q, \sigma, j, b, T\rangle \to \zeta$ and $\langle q, \sigma, j, b, T'\rangle \to \zeta'$, the site T' is the complement of the site T. For a family $\mathcal{F}$ of PTA's (or PTT's), such as the v_kI-PTA or v_kI-DPTT or v_k-PTA, we denote by $\mathcal{F}^{\mathrm{MSO}}$ the corresponding family of $\mathrm{PTA}^{\mathrm{MSO}}$'s (or $\mathrm{PTT}^{\mathrm{MSO}}$'s). With this definition of $\mathrm{PTA}^{\mathrm{MSO}}$ we can formally define that a family $\mathcal{F}$ of PTA's can perform MSO head tests if for every $\mathrm{PTA}^{\mathrm{MSO}}$ in $\mathcal{F}^{\mathrm{MSO}}$ an equivalent PTA in $\mathcal{F}$ can be constructed, and similarly for PTT's.

Obviously, as V-PTA's cannot recognize all regular tree languages, they cannot perform MSO head tests either: for any regular tree language T the set $\{(t, \mathrm{root}_t) \mid t \in T\}$ is a regular site.

The next result shows that any VI-PTA that uses MSO head tests as a built-in feature (i.e., any $\mathrm{VI\text{-}PTA}^{\mathrm{MSO}}$) can be replaced by an equivalent VI-PTA without such tests. The result holds for VI-PTA's with any fixed number of visible pebbles, either deterministic or nondeterministic, and it also holds for the corresponding VI-PTT's.

Lemma 12. *For each $k \geq 0$, the* v_kI-PTA *can perform* MSO *head tests. The same holds for the* v_kI-DPTA, v_kI-PTT, *and* v_kI-DPTT.

Proof. Let $\mathcal{A}_T$ be a deterministic bottom-up finite-state tree automaton recognizing the regular tree language $\mathrm{mark}(T)$ over $\Sigma \times \{0, 1\}$, representing the site T, trees with a single marked node. We show how a deterministic I-PTA $\mathcal{A}'_T$ can test whether or not the input tree with current head position h is accepted by $\mathcal{A}_T$, in a computation starting in configuration $\langle q_0, h, \varepsilon\rangle$ and ending in configuration $\langle q_{\mathrm{yes}}, h, \varepsilon\rangle$ or $\langle q_{\mathrm{no}}, h, \varepsilon\rangle$, where q_0 is the initial state and $\{q_{\mathrm{yes}}, q_{\mathrm{no}}\}$ the set of final states of $\mathcal{A}'_T$. Moreover, it starts the computation by dropping a pebble on h, and it keeps a pebble on h until the final computation step. It should be obvious that this I-PTA $\mathcal{A}'_T$ can be used as a subroutine by any v_kI-PTA or v_kI-PTT $\mathcal{A}$, starting in configuration $\langle(\tilde{q}, q_0), h, \pi\rangle$ and ending in configuration $\langle(\tilde{q}, q_{\mathrm{yes}}), h, \pi\rangle$ or $\langle(\tilde{q}, q_{\mathrm{no}}), h, \pi\rangle$, for every state $\tilde{q}$ and pebble stack π of $\mathcal{A}$. Just replace each rule $\langle q, \sigma, j, b\rangle \to \langle q', \alpha\rangle$ of $\mathcal{A}'_T$ by all possible rules $\langle(\tilde{q}, q), \sigma, j, b \cup b'\rangle \to \langle(\tilde{q}, q'), \alpha\rangle$ where b' is a set of visible pebble colours of $\mathcal{A}$ (except that in the first rule of $\mathcal{A}'_T$, which drops a pebble on h, the set b' possibly contains an invisible pebble colour of $\mathcal{A}$).

The post-order evaluation of Lemma 10 does not work here without precautions. If we mark node h with an invisible pebble the pebble becomes unobservable during the evaluation. In this way we cannot take the special "marked" position of h into account.[17] Instead, we first evaluate the subtree rooted at h, and subsequently the subtrees rooted at the ancestors of h, moving along the path from h to the root of the input tree. At the start of the evaluation of a subtree, we "paint" its root u by adding a special colour to the pebble on u, and preserving that information when the pebble is updated. In this way it is always clear when the painted node is visited. We paint node h with the special additional colour $\odot$ and use the evaluation process of Lemma 10 to compute the state of $\mathcal{A}_T$ at h, viewing the label σ of each node as $(\sigma, 0)$ except for the label σ of h which is treated as $(\sigma, 1)$. We paint each ancestor u of h with an additional colour (j, p) which indicates the child number j of the previous ancestor of h and the state p at which $\mathcal{A}_T$ arrives at that child of u (with h as a marked node). Then we use, again, the evaluation process of Lemma 10 to compute the state of $\mathcal{A}_T$ at u (with every σ viewed as $(\sigma, 0)$), except that the information in the pebble (j, p) is used for the state p of the j-th child of u, which is the unique child that has h as a descendant. Repeating this process for each ancestor, we eventually reach the root of the tree, and know the outcome of the test. Then we return to the original position h picking up the pebbles left on the path from that position to the root.

[17] Marking h with a visible pebble would easily work, showing that VI-PTA's can perform MSO head tests.

Formally, let $\mathcal{A}_T = (\Sigma \times \{0,1\}, P, F, \delta)$. For convenience we will identify the symbols $(\sigma, 0)$ and σ. The I-PTA $\mathcal{A}'_T$ is an extension of the I-PTA $\mathcal{A}'$ in the proof of Lemma 10. It has the additional states $q_{\downarrow\text{yes}}$ and $q_{\downarrow\text{no}}$, and in addition to the pebble colours $p_1 \cdots p_m$ of $\mathcal{A}'$ it has the pebble colours $(\mu, p_1 \cdots p_m)$ where either $\mu = \odot$ or $\mu = (i, r)$ for some $i \in [1, mx_\Sigma]$ and $r \in P$. The additional pebbles are used to "paint" h (with $\mu = \odot$) and the ancestors of h (with some $\mu = (i, r)$). The automaton $\mathcal{A}'_T$ has all the rules of $\mathcal{A}'$, except that rules ρ_4 and ρ_5 will become superfluous, and rule ρ_1 is replaced by the rule

$$\rho'_1 : \langle q_0, \sigma, j, \varnothing\rangle \to \langle q_\circ, \text{drop}_{((\odot,\varepsilon)}\rangle.$$

Thus, $\mathcal{A}'_T$ starts by evaluating the subtree rooted at h, with h as marked node. For $m < \text{rank}(\sigma)$ and every μ as above, except when $\mu = (m+1, r)$ for some $r \in P$, $\mathcal{A}'_T$ has the rules

$$\rho_2^\mu : \langle q_\circ, \sigma, j, \{(\mu, p_1 \cdots p_m)\}\rangle \to \langle q_\circ, \text{down}_{m+1}; \text{drop}_\varepsilon\rangle$$
$$\rho_6^\mu : \langle \bar{q}_p, \sigma, j, \{(\mu, p_1 \cdots p_m)\}\rangle \to \langle q_\circ, \text{lift}_{(\mu, p_1 \cdots p_m)}; \text{drop}_{(\mu, p_1 \cdots p_m p)}\rangle$$

which intuitively means that the pebble $(\mu, p_1 \cdots p_m)$ is treated in the same way as $p_1 \cdots p_m$ when not all children of the current node have been evaluated: $\mathcal{A}'_T$ moves to the $(m+1)$-th child and calls $\mathcal{A}'$, and when $\mathcal{A}'$ returns with the state p, $\mathcal{A}'_T$ adds p to the sequence of states in the pebble. However, in the exceptional case where $m < \text{rank}(\sigma)$ and $\mu = (m+1, r)$, $\mathcal{A}'_T$ has the rule

$$\rho_{2,6}^\mu : \langle q_\circ, \sigma, j, \{(\mu, p_1 \cdots p_m)\}\rangle \to \langle q_\circ, \text{lift}_{(\mu, p_1 \cdots p_m)}; \text{drop}_{(\mu, p_1 \cdots p_m r)}\rangle$$

which means that for the $(m+1)$-th child $\mathcal{A}'_T$ does not call $\mathcal{A}'$ but uses the state r that was previously computed and stored in μ.

The remaining rules of $\mathcal{A}'_T$ handle the situations that $\mathcal{A}'_T$ has just evaluated the subtrees rooted at the children of h or of one of the ancestors u of h, in state $q_\circ$. The automaton $\mathcal{A}'_T$ computes the state p of $\mathcal{A}_T$ at the marked node h or the unmarked node u, and drops the pebble $((j,p),\varepsilon)$ at its parent v, where j is the child number of h or u, thus indicating that the subtree rooted at the j-th child of v (with h as a marked node) evaluates to p. Then $\mathcal{A}'_T$ evaluates the subtree rooted at v.

For $m = \text{rank}(\sigma)$ and every μ as above, $\mathcal{A}'_T$ has the rules

$$\rho_3^\mu : \langle q_\circ, \sigma, j, \{(\mu, p_1 \cdots p_m)\}\rangle \to \langle q_\circ, \text{up}; \text{drop}_{((j,p),\varepsilon)}\rangle \quad \text{if } j \neq 0,$$
$$\rho_4^\mu : \langle q_\circ, \sigma, 0, \{(\mu, p_1 \cdots p_m)\}\rangle \to \langle q_{\downarrow\text{yes}}, \text{stay}\rangle \quad \text{if } p \in F,$$
$$\rho_5^\mu : \langle q_\circ, \sigma, 0, \{(\mu, p_1 \cdots p_m)\}\rangle \to \langle q_{\downarrow\text{no}}, \text{stay}\rangle \quad \text{if } p \notin F,$$

where $p = \delta((\sigma,1), p_1, \ldots, p_m)$ if $\mu = \odot$ and $p = \delta(\sigma, p_1, \ldots, p_m)$ otherwise.

When $\mathcal{A}'_T$ arrives at the root of the input tree, it knows whether or not $\mathcal{A}_T$ accepts that tree (with h as a marked node), and moves down to h. For the outcome $x \in \{\text{yes}, \text{no}\}$ the rules are

$$\langle q_{\downarrow x}, \sigma, j, \{((i,r), p_1 \cdots p_m)\}\rangle \to \langle q_{\downarrow x}, \text{lift}_{((i,r),p_1 \cdots p_m)}; \text{down}_i\rangle$$
$$\langle q_{\downarrow x}, \sigma, j, \{(\odot, p_1 \cdots p_m)\}\rangle \to \langle q_x, \text{lift}_{(\odot, p_1 \cdots p_m)}\rangle.$$

This ends the description of $\mathcal{A}'_T$. □

This result can easily be extended, using the same proof technique: PTA's and PTT's can test their *visible configuration*, the position of the head together with the positions and colours of the visible pebbles. Later we will show the more complicated result that PTA's and PTT's can even test their *observable configuration*, i.e., the visible configuration plus the topmost pebble (Theorem 16).

Let C be the set of colours of a PTA or PTT. To represent the visible and observable configurations, we introduce a new ranked alphabet $\Sigma \times 2^C$, such that the rank of (σ, b) equals that of σ in Σ. A tree over $\Sigma \times 2^C$ is a "coloured tree". For each pebble stack π on a tree t over Σ we define two coloured trees. The visible configuration tree $\text{vis}(t, \pi)$ is obtained by adding to the label of each node u of t the set $b \subseteq C$ such that b contains c if and only if (u, c) occurs in π and $c \in C_\text{v}$. Similarly for $\text{obs}(t, \pi)$, the observable configuration tree, b contains c if and only if (u, c) occurs in π and c is observable (i.e., $c \in C_\text{v}$ or (u, c) is the top element of π). Note that as long as a PTA does not change its pebble stack by a drop- or lift-instruction, it behaves just as a TA on $\text{obs}(t, \pi)$.

We say that a family $\mathcal{F}$ of PTA's (or PTT's) can *perform* MSO *tests on the visible configuration* if, for a regular site T over $\Sigma \times 2^C$, an automaton (or transducer) in $\mathcal{F}$ can test whether or not $(\text{vis}(t, \pi), h) \in T$, where t is the input tree, π the current pebble stack, and h the current position of the head. A similar definition can be given for MSO *tests on the observable configuration*. These informal definitions could be formalized in a way explained for MSO head tests before Lemma 12.

We now show that the VI-PTA and VI-PTT can perform MSO tests on the visible configuration. Note that for a regular site T over $\Sigma \times 2^C$, $\text{mark}(T)$ is a regular tree language over $\Sigma \times 2^C \times \{0,1\}$.

Lemma 13. *For each $k \geq 0$, the* v_kI-PTA *and* v_kI-DPTA *can perform* MSO *tests on the visible configuration. The same holds for the* v_kI-PTT *and* v_kI-DPTT.

Proof. As in the proof of Lemma 12, let $\mathcal{A}_T$ be a deterministic bottom-up finite-state tree automaton recognizing the regular tree language mark(T) over $\Sigma \times 2^C \times \{0,1\}$, representing the site T, coloured trees with a single marked node. As observed in the first paragraph of that proof the I-PTT $\mathcal{A}'_T$ (of that proof) can be turned into a subroutine for any v_kI-PTA or v_kI-PTT $\mathcal{A}$ with visible colour set C_v by replacing each rule $\langle q, \sigma, j, b\rangle \to \langle q', \alpha\rangle$ of $\mathcal{A}'_T$ (except ρ'_1) by all possible rules $\langle(\tilde{q}, q), \sigma, j, b \cup b'\rangle \to \langle(\tilde{q}, q'), \alpha\rangle$ with $b' \subseteq C_v$. This subroutine can easily be turned into one that tests whether or not $(\mathrm{vis}(t,\pi), h) \in T$ as follows. For the rules corresponding in this way to ρ_3, ρ_4, ρ_5 (in the proof of Lemma 10), change $p = \delta(\sigma, p_1, \ldots, p_m)$ into $p = \delta((\sigma, b', 0), p_1, \ldots, p_m)$. Similarly, for $\rho_3^\mu, \rho_4^\mu, \rho_5^\mu$ change $p = \delta((\sigma, 1), p_1, \ldots, p_m)$ into $p = \delta((\sigma, b', 1), p_1, \ldots, p_m)$ and, again, $p = \delta(\sigma, p_1, \ldots, p_m)$ into $p = \delta((\sigma, b', 0), p_1, \ldots, p_m)$. □

We now turn to the PTA as a navigational device: the *trip* $T(\mathcal{A})$ *computed by a* PTA $\mathcal{A}$ consists of all triples (t, u, v) such that $\mathcal{A}$, on input tree t, started at node u in an initial state without pebbles on the tree, walks to node v, and halts in a final state (possibly leaving pebbles on the tree). Formally, $T(\mathcal{A}) = \{(t,u,v) \in T_\Sigma \times N(t) \times N(t) \mid \exists q_0 \in Q_0, q_\infty \in F, \pi \in (N(t) \times C)^* : \langle q_0, u, \varepsilon\rangle \Rightarrow^*_{\mathcal{A}} \langle q_\infty, v, \pi\rangle\}$. Two PTA's $\mathcal{A}$ and $\mathcal{B}$ are *trip-equivalent* if $T(\mathcal{A}) = T(\mathcal{B})$. Since it is clear that $L(\mathcal{A}) = \{t \in T_\Sigma \mid \exists u \in N(t) : (t, \mathrm{root}_t, u) \in T(\mathcal{A})\}$, trip-equivalence implies (language-)equivalence. A trip T is *functional* if, for every t, $\{(u,v) \mid (t,u,v) \in T\}$ is a function. Note that the trip computed by a deterministic PTA is functional.

It is straightforward to check that Lemma 1 also holds for the PTA as navigational device, replacing equivalence by trip-equivalence. Thus, v_kI-PTA's can perform stack tests also when computing a trip. Similarly, they can perform the MSO tests discussed in Lemmas 12 and 13, and to be discussed in Theorem 16.

In [5, Theorem 8] it is shown that every MSO definable trip (tree-node relation) can be computed by a $\mathrm{TA}^{\mathrm{MSO}}$, i.e., a tree-walking automaton with MSO head tests (and vice versa). Moreover, by (the corrected version of) [5, Theorem 9], if the trip is functional, then the automaton is deterministic. We will also use the fact that, according to the proof of [5, Theorem 8], the MSO definable trips can be computed in a special way.

Proposition 14. *Every* MSO *definable trip can be computed by a tree-walking automaton with* MSO *head tests that has the following two properties:*

(1) *it never walks along the same edge twice (in either direction), and*
(2) *it visits each node at most twice.*

If the trip is functional, then the automaton is deterministic.

The first property means that, when walking from a node u to a node v, the automaton always takes the shortest (undirected) path from u to v, i.e., the path that leads from u up to the least ancestor of u and v, and then down to v. The second property means that the automaton does not execute two consecutive stay-instructions.

The next result provides a characterization of the MSO definable trips by pebble automata that is more elegant than the one in [17], which uses so-called marble/pebble automata, a restricted kind of v_1I-PTA (marbles are invisible pebbles only dropped on the path from the root to the current position of the head; a single visible pebble may only be dropped and picked up on a tree without marbles).

Theorem 15. *For each $k \geq 0$, the trips computed by* v_kI-PTA*'s are exactly the* MSO *definable trips. Similarly for* v_kI-DPTA*'s and functional trips.*

Proof. Consider a trip T computed by v_kI-PTA $\mathcal{A}$. Thus, for any (t,u,v) in T, starting at node u of input tree t, $\mathcal{A}$ walks to node v and halts. Then mark(T) can be recognized by another v_kI-PTA as follows. First it searches (deterministically) for the marked starting node u, then it simulates $\mathcal{A}$, and when $\mathcal{A}$ halts in a final state, verifies that the marked node v is reached. By Theorem 11 this tree language is regular and hence T is MSO definable.

By Proposition 14 every MSO definable trip can be computed by a tree-walking automaton $\mathcal{B}$ with MSO head tests. Since (as observed above) Lemma 12 also holds for the PTA as a navigational device, it can therefore be computed by an I-PTA $\mathcal{B}'$. Moreover, if the trip is functional, then the automata $\mathcal{B}$ and $\mathcal{B}'$ are deterministic. □

Note that the automaton $\mathcal{B}'$ in the above proof always removes all its pebbles before halting. Thus, that requirement could be added to the definition of the trip computed by a v_kI-PTA (implying that not every v_kI-PTA computes a trip). This conforms to the idea that one should not leave garbage after a picknick.

Using the above result, or rather Proposition 14, we are now able to show that the PTA and PTT can perform MSO tests on the observable configuration, i.e., they can evaluate MSO formulas $\varphi(x)$ on the observable configuration tree $\mathrm{obs}(t,\pi)$ with the variable x assigned to the position of the reading head.

Theorem 16. *For each $k \geq 0$, the* V_kI-PTA *and* V_kI-DPTA *can perform* MSO *tests on the observable configuration. The same holds for the* V_kI-PTT *and* V_kI-DPTT.

Proof. Let T be a regular site over $\Sigma \times 2^C$, and let $\mathcal{A}$ be a V_kI-PTA that uses T as a test to find out whether or not $(\text{obs}(t,\pi),h) \in T$. Our aim is to construct a trip-equivalent V_kI-PTA $\mathcal{A}'$ that does not use MSO tests on the observable configuration. The proof is exactly the same for the case where $\mathcal{A}$ and $\mathcal{A}'$ are V_kI-PTT (with equivalence instead of trip-equivalence).

Essentially, $\mathcal{A}'$ simulates $\mathcal{A}$. When $\mathcal{A}$ uses the test T, there are two cases. In the first case, either the pebble stack of $\mathcal{A}$ is empty or the colour of the topmost pebble of $\mathcal{A}$ is visible. Then the observable configuration equals the visible configuration, and so $\mathcal{A}'$ can use the test T too, by Lemma 13. The remaining, difficult case is that the colour d of the topmost pebble of $\mathcal{A}$ is invisible. To implement the test T in this case it seems that $\mathcal{A}'$ cannot use any additional invisible pebbles (as in the proof of Lemma 13), because they make pebble d unobservable. However, this is not a problem as long as the additional pebbles carry sufficient information about the position u of pebble d. The solution is to view T as a trip from u to h (the position of the head), and to keep track of an automaton $\mathcal{B}_d$ that computes that trip. Although $\mathcal{B}_d$ is nondeterministic, it is straightforward for $\mathcal{A}'$ to employ the usual subset construction for finite-state automata.

For every $d \in C_\text{i}$, let T_d be the trip over $\Sigma \times 2^C$ defined by $T_d = \{(s,u,h) \mid (s',h) \in T\}$, where s' is obtained from s by changing the label (σ,b) of u into $(\sigma, b \cup \{d\})$. Then $(\text{obs}(t,\pi),h) \in T$ if and only if $(\text{vis}(t,\pi),u,h) \in T_d$, if (d,u) is the topmost element of π. It should be clear from the regularity of T that T_d is a regular trip. Hence, by Proposition 14, there is a TA with MSO head tests $\mathcal{B}_d$ that computes T_d and that has the special properties mentioned there. Therefore (see the paragraph after Proposition 14), to keep track of the possible computations of $\mathcal{B}_d$, the automaton $\mathcal{A}'$ uses additional invisible pebbles to cover the shortest (undirected) path from u to h. These pebbles will be called *beads* to distinguish them from $\mathcal{A}$'s original pebbles. Each bead carries state information on computations of $\mathcal{B}_d$ that start at position u (in an initial state) and end at position h. More precisely, each bead is a triple (S,δ,d) where S is a set of states of $\mathcal{B}_d$ and $\delta \in \{\text{up}, \text{stay}\} \cup \{\text{down}_i \mid i \in [1, mx_\Sigma]\}$. There is one such bead (S,δ,d) on every node v on the path from u to h (including u and h) where S is the set of states p of $\mathcal{B}_d$ such that $\mathcal{B}_d$ has a computation on $\text{vis}(t,\pi)$ starting at u in an initial state and ending at v in state p. Moreover, δ indicates the node w just before v on the path, which is the parent or i-th child of v if δ is up or down_i, respectively, and which is nonexistent when $v = u$, if $\delta = \text{stay}$. The bead at v is on top of the bead at w in the pebble stack of $\mathcal{A}'$. Thus, the bead at h is always on the top of the stack of $\mathcal{A}'$ and hence is always observable.

The automaton $\mathcal{A}'$ can still simulate $\mathcal{A}$ because if the bead (S,δ,d) is at head position h, then the invisible pebble d is observable at h by $\mathcal{A}$ if and only if $\delta = \text{stay}$. If $\mathcal{A}$ lifts d, then $\mathcal{A}'$ lifts both (S,stay,d) and d. If $\mathcal{A}$ drops another pebble d' at h, then so does $\mathcal{A}'$ (and starts a new chain of beads on top of that pebble if d' is invisible). When pebble d' is lifted again, the beads for pebble d are still available and can be used as before.

Now, suppose that $\mathcal{A}$ uses the test T at position h. If $\mathcal{A}'$ does not see a bead at position h, then it uses T as a test on the visible configuration. If $\mathcal{A}'$ sees a bead (S,δ,d) at h, then $\mathcal{A}'$ just checks whether or not S contains a final state of $\mathcal{B}_d$, i.e., whether or not $(\text{vis}(t,\pi),u,h) \in T_d$.

It remains to show how $\mathcal{A}'$ computes the beads. The path of beads is initialized by $\mathcal{A}'$ when $\mathcal{A}$ drops invisible pebble d. Then $\mathcal{A}'$ also drops pebble d, computes the relevant set S of states of $\mathcal{B}_d$, and drops bead (S,stay,d). The set S contains all initial states of $\mathcal{B}_d$, plus all states that $\mathcal{B}_d$ can reach from an initial state by applying one relevant rule with a stay-instruction (cf. the second property in Proposition 14). To find the latter states, $\mathcal{A}'$ just simulates all those rules. Note that the MSO head tests of $\mathcal{B}_d$ on $\text{vis}(t,\pi)$ are MSO tests on the visible configuration of $\mathcal{A}'$. That is because during the simulation of $\mathcal{A}$ by $\mathcal{A}'$ the visible configuration $\text{vis}(t,\pi')$ of $\mathcal{A}'$ equals the visible configuration $\text{vis}(t,\pi)$ of $\mathcal{A}$: the pebble stack π of $\mathcal{A}$ is obtained from the corresponding pebble stack π' of $\mathcal{A}'$ by removing all (invisible) beads.

The path of beads is updated as follows. If we backtrack on the path from u to h, i.e., the current bead is (S,δ,d) with $\delta \neq \text{stay}$ and we move in the direction δ, we just lift the current bead before moving. If we move away from u, we must compute new bead information. Suppose the current bead on h is (S,up,d) and we move down to the i-th child hi of h. Then the bead at hi is (S',up,d) where S' can be computed in a similar way as the set S above: $\mathcal{A}'$ simulates all computations of $\mathcal{B}_d$ that start at h in a state of S and end at hi (and note that such a computation consists of one step, possibly followed by another step with a stay-instruction). Now suppose that the current bead is (S,down_i,d), which means that u is a descendant of h. If we move up to the parent v of h, then the new bead is (S',down_j,d) where j is the child number of h. If we move down to a child v of h with child number $\neq i$, then the new bead is (S',up,d). In each of these cases S' can be computed as before, by simulating the computations of $\mathcal{B}_d$ from h to v.

In general, $\mathcal{A}$ can of course use several regular sites $T_1,\ldots,T_n$ as tests on the observable configuration. It should be obvious how to extend the proof to handle that. The beads are then of the form $(S_1,\ldots,S_n,\delta,d)$ where S_i is a set of states of a TA with MSO head tests $\mathcal{B}_{id}$ that computes the trip T_{id}. To test T_i in the presence of such a bead, $\mathcal{A}'$ just checks whether or not S_i contains a final state of $\mathcal{B}_{id}$. □

7. The power of the I-PTT

In this section we discuss some applications of the fact that the I-PTT can perform MSO head tests (Lemma 12). We prove that it can simulate the composition of two TT's of which the first is deterministic (cf. Lemma 4), and that it can simulate the bottom-up tree transducer.

Composition of TT's. We now prove that the inclusions of Lemma 4 also hold in the other direction, provided that we start with a deterministic TT.

Theorem 17. dTT ∘ dTT ⊆ I-dPTT *and* dTT ∘ TT ⊆ I-PTT.

Proof. Consider two deterministic TT's $\mathcal{M}_1$ and $\mathcal{M}_2$. Assume that input tree t is translated into tree s by transducer $\mathcal{M}_1$. We will simulate the computation of $\mathcal{M}_2$ on s directly on t using a PTT $\mathcal{M}$ with invisible pebbles. Any action taken by $\mathcal{M}_2$ on node v of tree s will be simulated by $\mathcal{M}$ on the node u of t that was the position of $\mathcal{M}_1$ when it generated v. This means that if $\mathcal{M}_2$ moves down in the tree s to one of the children of v, the computation of $\mathcal{M}_1$ is simulated until it generates that child. On the other hand, if $\mathcal{M}_2$ moves up in the tree s to the parent of v, it is necessary to backtrack on the computation of $\mathcal{M}_1$, back to the moment that parent was generated. In this way, tree s is never fully reconstructed as a whole, but at every moment $\mathcal{M}$ has access to a single node of s. The necessary node, the current node of $\mathcal{M}_2$, is continuously updated by moving back and forth along the computation of $\mathcal{M}_1$ on t.

Moving forward on the computation of $\mathcal{M}_1$ is straightforward. To be able to retrace, $\mathcal{M}$ uses its pebbles to record the output-generating steps of the computation of $\mathcal{M}_1$ on t. Each output rule of $\mathcal{M}_1$ is represented by a pebble colour, and is put on the node u of t where it was applied. The pebble colour also codes the child number of the generated node v in s. Thus the pebble stack represents a (shortest) path in s from the root to v. For each node on that path the stack contains a pebble with the rule of $\mathcal{M}_1$ used to generate that node and with its child number, from bottom to top.

Note that the determinism of $\mathcal{M}_1$ is an essential ingredient for this construction. Simulating $\mathcal{M}_2$, walking along the virtual tree s, one has to ensure that each time a node v is revisited, the same rule of $\mathcal{M}_1$ is applied to u.

The above intuitive description assumes that the input tree t is in the domain $L(\mathcal{M}_1)$ of $\mathcal{M}_1$. In fact, it suffices to construct an I-PTT $\mathcal{M}$ such that $\tau_{\mathcal{M}}(t) = \tau_{\mathcal{M}_2}(\tau_{\mathcal{M}_1}(t))$ for every such t, because $\mathcal{M}$ can then easily be adapted to start with an MSO head test verifying that the input tree is in $L(\mathcal{M}_1)$, which is regular by Corollary 9.

Let us now give the formal definitions. Let $\mathcal{M}_1 = (\Sigma, \Delta, P, \{p_0\}, R_1)$ be a deterministic TT and let $\mathcal{M}_2 = (\Delta, \Gamma, Q, Q_0, R_2)$ be an arbitrary TT. To define the I-PTT $\mathcal{M}$ it is convenient to extend the definition of an I-PTT with a new type of instruction: we allow the right-hand side of a rule to be of the form $\langle q', \text{to-top}\rangle$, which when applied to a configuration $\langle q, u, \pi\rangle$ leads to the next configuration $\langle q', v, \pi\rangle$ where v is the node in the topmost element of π. Obviously this does not extend the expressive power of the I-PTT: it is straightforward to write a subroutine that searches for the (unique observable) pebble on the tree, by first walking to the root and then executing a depth-first search of the tree until a pebble is observed.

The I-PTT $\mathcal{M}$ has input alphabet Σ and output alphabet Γ. Its set C_{i} of pebble colours consists of all pairs (ρ, i) where ρ is an output rule of $\mathcal{M}_1$, i.e., a rule of the form $\langle p, \sigma, j\rangle \to \delta(\langle p_1, \text{stay}\rangle, \ldots, \langle p_m, \text{stay}\rangle)$ with $p, p_1, \ldots, p_m \in P$, and i is a child number of Δ, i.e., $i \in [0, mx_\Delta]$. The set of states of $\mathcal{M}$ is defined to be $Q \cup (P \times [0, mx_\Delta] \times Q)$ and the set of initial states is $\{p_0\} \times \{0\} \times Q_0$. A state $q \in Q$ is used by $\mathcal{M}$ when simulating a computation step of $\mathcal{M}_2$, and a state (p, i, q) is used by $\mathcal{M}$ when simulating the computation of $\mathcal{M}_1$ that generates the i-th child of the current node of $\mathcal{M}_2$ (keeping the state q of $\mathcal{M}_2$ in memory). Initially, $\mathcal{M}$ simulates $\mathcal{M}_1$ in order to generate the root of its output tree. The rules of $\mathcal{M}$ are defined as follows.

First we define the rules that simulate $\mathcal{M}_1$. Let $\rho : \langle p, \sigma, j\rangle \to \zeta$ be a rule in R_1. If $\zeta = \langle p', \alpha\rangle$ and α is a move instruction, then $\mathcal{M}$ has the rules $\langle (p, i, q), \sigma, j, b\rangle \to \langle (p', i, q), \alpha\rangle$ for every $i \in [0, mx_\Delta]$, $q \in Q$, and $b \subseteq C_{\mathrm{i}}$ with $\#(b) \le 1$. If ρ is an output rule with $\zeta = \delta(\langle p_1, \text{stay}\rangle, \ldots, \langle p_m, \text{stay}\rangle)$, then $\mathcal{M}$ has the rules $\langle (p, i, q), \sigma, j, b\rangle \to \langle q, \text{drop}_{(\rho,i)}\rangle$ for every i, q, b as above. Thus, $\mathcal{M}$ simulates $\mathcal{M}_1$ until $\mathcal{M}_1$ generates an output node, drops the corresponding pebble, and continues simulating $\mathcal{M}_2$.

Second we define the rules that simulate $\mathcal{M}_2$. Let $\langle q, \delta, i\rangle \to \zeta$ be a rule in R_2 and let $\rho : \langle p, \sigma, j\rangle \to \delta(\langle p_1, \text{stay}\rangle, \ldots, \langle p_m, \text{stay}\rangle)$ be an output rule in R_1 (with the same δ). Then $\mathcal{M}$ has the rule $\langle q, \sigma, j, \{(\rho, i)\}\rangle \to \zeta'$ where ζ' is defined as follows. If $\zeta = \langle q', \text{down}_\ell\rangle$, then $\zeta' = \langle (p_\ell, \ell, q'), \text{stay}\rangle$. If $\zeta = \langle q', \text{up}\rangle$, then $\zeta' = \langle q', \text{lift}_{(\rho,i)}; \text{to-top}\rangle$. Otherwise, $\zeta' = \zeta$. Thus, $\mathcal{M}$ simulates every output rule or stay rule of $\mathcal{M}_2$ without changing its current node and current pebble stack, because the current node of $\mathcal{M}_2$ stays the same. To simulate a down_ℓ-instruction of $\mathcal{M}_2$, $\mathcal{M}$ starts simulating $\mathcal{M}_1$ in state p_ℓ with the child number ℓ of the next node of $\mathcal{M}_2$. Finally, $\mathcal{M}$ simulates an up-instruction of $\mathcal{M}_2$ by lifting its topmost pebble and walking to the new topmost pebble, where it continues the simulation of $\mathcal{M}_2$. □

Taking Theorem 17 and Lemma 4 together, we obtain that dTT ∘ dTT ⊆ I-dPTT ⊆ I-PTT ⊆ TT ∘ TT. It is open whether or not the first and last inclusions are proper. A way to express I-dPTT and I-PTT in terms of tree-walking tree transducers (without pebbles) would be to allow those transducers to have infinite input and output trees. Let us denote by dTT^∞ the class of transductions realized by deterministic TT's that have finite input trees but can output infinite trees. As a particular example, the TT $\mathcal{N}$ in the proof of Lemma 4 can be turned into such a deterministic TT $\mathcal{N}^\infty$ by removing all rules $\langle f, \sigma, j\rangle \to \bot$. This $\mathcal{N}^\infty$ preprocesses every input tree t into a unique "tree of trees" t_∞ consisting of top level t and infinitely many levels of copies $\hat{t}_u$ of t. Moreover, let us denote by ${}^\infty\text{TT}$ the class of transductions realized by TT's that output finite trees but can walk on infinite input trees, and similarly for ${}^\infty\text{dTT}$. It should be clear that the TT $\mathcal{M}'$ in the proof of Lemma 4 can also be viewed as working on input tree t_∞ rather than a nondeterministically generated t' (and thus never aborts its simulation of $\mathcal{M}$). It should also be clear that the proof of Theorem 17 still works when $\mathcal{M}_1$ produces an infinite output tree as input

tree for $\mathcal{M}_2$.[18] Taking these results together, we obtain that I-dPTT $=$ dTT$^\infty \circ {}^\infty$dTT and I-PTT $=$ dTT$^\infty \circ {}^\infty$TT. The formal definitions are left to the reader. Other characterizations of I-dPTT will be shown in Section 15 (Theorem 53), where we also show that I-dPTT $\subseteq$ dTT^3 (Corollary 54).

Bottom-up tree transducers. The classical top-down and bottom-up tree transducers are compared to the V-PTT at the end of [41, Section 3.1]. Obviously, TT's generalize top-down tree transducers. In fact, the latter correspond to TT's that do not use the move instructions up and stay. Moreover, the classical top-down tree transducers with regular look-ahead can be simulated by TT's with MSO head tests, and hence by I-PTT's. In general, bottom-up tree transducers cannot be simulated by V-PTT's, because otherwise every regular tree language could be accepted by a V-PTA (see below for the details), which is false as proved in [8]. We will show that every bottom-up tree transducer can be simulated by an I-PTT. This will not be used in the following sections.

A *bottom-up tree transducer* is a tuple $\mathcal{M} = (\Sigma, \Delta, P, F, R)$ where Σ and Δ are ranked alphabets, P is a finite set of states with a subset F of final states, and R is a finite set of rules of the form $\sigma(p_1(x_1), \ldots, p_m(x_m)) \to p(\zeta)$ such that $m \in \mathbb{N}$, $\sigma \in \Sigma^{(m)}$, $p_1, \ldots, p_m, p \in P$ and $\zeta \in T_\Delta(\{x_1, \ldots, x_m\})$. For $p \in P$, the sets $\tau_p \subseteq T_\Sigma \times T_\Delta$ are defined inductively as follows: the pair $(\sigma(t_1, \ldots, t_m), s)$ is in τ_p if there is a rule as above and there are pairs $(t_i, s_i) \in \tau_{p_i}$ for all $i \in [1, m]$ such that $s = \zeta[s_1, \ldots, s_m]$, which is the result of substituting s_i for every occurrence of x_i in ζ. The transduction $\tau_\mathcal{M}$ realized by $\mathcal{M}$ is the union of all τ_p with $p \in F$. The transducer $\mathcal{M}$ is deterministic if it does not have two rules with the same left-hand side. For more information see, e.g., [29, Chapter IV].

For every regular tree language L there is a deterministic bottom-up finite-state tree automaton $\mathcal{A} = (\Sigma, P, F, \delta)$ (see the proof of Lemma 10) that recognizes L and hence there is a deterministic bottom-up tree transducer $\mathcal{M}$ that realizes the transduction $\tau_L = \{(t, 1) \mid t \in L\} \cup \{(t, 0) \mid t \notin L\}$. In fact, $\mathcal{M} = (\Sigma, \{0, 1\}, P, F, R)$ where 0 and 1 have rank 0 and R is the set of all rules $\sigma(p_1(x_1), \ldots, p_m(x_m)) \to p(i)$ such that $\delta(\sigma, p_1, \ldots, p_m) = p$ and $i = 1$ if $p \in F$, $i = 0$ otherwise. A V-PTT that computes τ_L can be turned into a V-PTA that accepts L by removing every output rule $\langle q, \sigma, j, b\rangle \to 0$ and changing every output rule $\langle q, \sigma, j, b\rangle \to 1$ into $\langle q, \sigma, j, b\rangle \to \langle q_{\text{fin}}, \text{stay}\rangle$ where q_{fin} is the final state.

Let B (dB) denote the class of transductions realized by (deterministic) bottom-up tree transducers.

Theorem 18. B $\subseteq$ I-PTT *and* dB $\subseteq$ I-dPTT.

Proof. Let $\mathcal{M} = (\Sigma, \Delta, P, F, R)$ be a bottom-up tree transducer. Intuitively, for a given input tree t, the transducer $\mathcal{M}$ visits each node u of t exactly once. It arrives at the children of u in certain states $p_1, \ldots, p_m$ with certain output trees $s_1, \ldots, s_m$, and applies a rule $\sigma(p_1(x_1), \ldots, p_m(x_m)) \to p(\zeta)$ where σ is the label of u. Thus, it arrives at u in state p with output $\zeta[s_1, \ldots, s_m]$.

We construct an I-PTT $\mathcal{M}'$ with MSO head tests such that $\tau_{\mathcal{M}'} = \tau_\mathcal{M}$ (see Lemma 12). The transducer $\mathcal{M}'$ uses the rules of $\mathcal{M}$ as pebble colours. The behaviour of $\mathcal{M}'$ on a given input tree t is divided into two phases. In the first phase $\mathcal{M}'$ walks through t and (nondeterministically) drops one pebble c on each node u of t, in post-order. The input symbol σ in the left-hand side of rule c must be the label of u. Intuitively, c is the rule $\sigma(p_1(x_1), \ldots, p_m(x_m)) \to p(\zeta)$ applied by $\mathcal{M}$ at u during a possible computation. When $\mathcal{M}$ drops c on u it uses MSO head tests to check that $\mathcal{M}$ has a computation on t that arrives at the i-th child ui of u in state p_i, for every $i \in [1, m]$. This can be done because the state behaviour of $\mathcal{M}$ on t is that of a bottom-up finite-state tree automaton. Thus, the tree language $L_p = \{t \in T_\Sigma \mid \exists s : (t, s) \in \tau_p\}$ is regular for every $p \in P$ and hence the site $T_i = \{(t, u) \mid t|_{ui} \in L_{p_i}\}$ is also regular, as can easily be seen. Note that if $\mathcal{M}$ is deterministic, then this first phase of $\mathcal{M}'$ is deterministic too, because $\mathcal{M}$ arrives at each node in a unique state (during a successful computation). In the second, deterministic phase $\mathcal{M}'$ moves top-down through t, checks that the states in the guessed rules are consistent, and computes the corresponding output. First $\mathcal{M}'$ checks for the pebble $c = \sigma(p_1(x_1), \ldots, p_m(x_m)) \to p(\zeta)$ at the root u, that the state p is in F. If so, it starts a process that is the same for every node u of t. It lifts pebble c and goes into state $[c, \zeta]$, in which it will output the Δ-labeled nodes of ζ, without leaving u. In state $q = [c, \delta(\zeta_1, \ldots, \zeta_n)]$, it uses the output rules $\langle q, \sigma, j, \varnothing\rangle \to \delta(\langle[c, \zeta_1], \text{stay}\rangle, \ldots, \langle[c, \zeta_n], \text{stay}\rangle)$. When $\mathcal{M}'$ is in a state $[c, x_i]$, it calls a subroutine S_i. Subroutine S_i walks through the subtrees $t|_{um}, \ldots, t|_{u(i+1)}$ of t, depth-first right-to-left, lifts the pebbles at all the nodes of those trees in reverse post-order (which is possible because the pebbles were dropped in post-order), and returns control to $\mathcal{M}'$, which continues by moving in state c to child ui where it observes the pebble at ui (again, because of the post-order dropping). Then $\mathcal{M}$ checks that the state in the right-hand side of that pebble is p_i, and repeats the above process for node ui instead of u. It should be clear that in this way $\mathcal{M}'$ simulates the computations of $\mathcal{M}$, and so $\tau_{\mathcal{M}'} = \tau_\mathcal{M}$. Note that the bottom-up transducer $\mathcal{M}$ can disregard computed output, because in a rule as above it may be that x_i does not occur in ζ. In such a case $\mathcal{M}'$ clearly does not compute that output either, in the second phase, whereas it has checked in the first phase that $\mathcal{M}$ indeed has a computation that arrives in state p_i at the i-th child. Note also that if x_i occurs twice in ζ, then $\mathcal{M}'$ simulates in the second phase twice the same computation of $\mathcal{M}$ on the i-th subtree (which was guessed in the first phase). □

[18] To see that $L(\mathcal{M}_1)$ is regular, construct an ordinary nondeterministic TT $\mathcal{N}$ by adding to $\mathcal{M}_1$ all rules $\langle q, \sigma, j\rangle \to \bot$ such that $\mathcal{M}_1$ has no rule with left-hand side $\langle q, \sigma, j\rangle$, and all rules $\langle q, \sigma, j\rangle \to \top$ such that $\mathcal{M}_1$ has a rule with that left-hand side (where $\bot$ and $\top$ are new output symbols of rank 0). Then $L(\mathcal{M}_1)$ is the complement of $\tau_\mathcal{N}^{-1}(R)$ where R is the set of output trees of $\mathcal{N}$ with an occurrence of $\bot$. Now use Proposition 6(1).

8. Look-ahead tests

The results on look-ahead in this section are only needed in the next section (and in a minor way in Section 11). They also hold for the PTA as navigational device, computing a trip.

We say that a family $\mathcal{F}$ of PTA's (or PTT's) can *perform look-ahead tests* if an automaton (or transducer) $\mathcal{A}$ in $\mathcal{F}$ can test whether or not a PTT $\mathcal{B}$ (not necessarily in $\mathcal{F}$) has a successful computation when started in the current situation of $\mathcal{A}$ (i.e., position of the head and stack of pebbles). We require that $\Sigma^{\mathcal{A}} = \Sigma^{\mathcal{B}}$, $C_{\mathrm{v}}^{\mathcal{A}} \subseteq C_{\mathrm{v}}^{\mathcal{B}}$, $C_{\mathrm{i}}^{\mathcal{A}} \subseteq C_{\mathrm{i}}^{\mathcal{B}}$, and $k^{\mathcal{A}} \leq k^{\mathcal{B}}$ (where $\Sigma^{\mathcal{A}}$ is the input alphabet of $\mathcal{A}$, and similarly for the other notation). Since we are only interested in the existence of a successful computation, and not in its output tree, we are actually using alternating PTA's as look-ahead device (cf. Section 3). In particular, we also allow a PTA to be used as look-ahead $\mathcal{B}$, viewing it as a PTT as in the proof of Theorem 11.

In the formal definition of a PTA or PTT *with look-ahead tests* (cf. the formal definition of MSO head tests before Lemma 12), the rules are of the form $\langle q, \sigma, j, b, \mathcal{B}\rangle \to \zeta$ or $\langle q, \sigma, j, b, \neg\mathcal{B}\rangle \to \zeta$ which are relevant to a given configuration $\langle q, h, \pi\rangle$ of $\mathcal{A}$ on tree t if the transducer $\mathcal{B}$ does or does not have a successful computation on t that starts in the situation $\langle h, \pi\rangle$, i.e., if there do or do not exist $p_0 \in Q_0^{\mathcal{B}}$ and $s \in T_{\Delta^{\mathcal{B}}}$ such that $\langle p_0, h, \pi\rangle \Rightarrow_{t,\mathcal{B}}^* s$ (where $\Delta^{\mathcal{B}}$ is the output alphabet of $\mathcal{B}$), or in the case of a PTA $\mathcal{B}$, if there do or do not exist $p_0 \in Q_0^{\mathcal{B}}$, $p_f \in F^{\mathcal{B}}$, and $\langle u, \pi\rangle \in \mathrm{Sit}^{\mathcal{B}}(t)$ such that $\langle p_0, \mathrm{root}_t, \varepsilon\rangle \Rightarrow_{t,\mathcal{B}}^* \langle p_f, u, \pi\rangle$ (where $F^{\mathcal{B}}$ is the set of final states of $\mathcal{B}$).

Theorem 19. *For each $k \geq 0$, the* v_kI-PTA *and* v_kI-dPTA *can perform look-ahead tests. The same holds for the* v_kI-PTT *and* v_kI-dPTT.

Proof. Let $\mathcal{A}$ be a v_kI-PTA that performs a look-ahead test by calling some v_mI-PTT $\mathcal{B}$ (with $k \leq m$). We wish to construct a trip-equivalent v_kI-PTA $\mathcal{A}'$ that does not perform such look-ahead tests. By Lemma 1 we may construct $\mathcal{A}'$ as a PTA with stack tests, i.e., a PTA that can test whether its pebble stack is empty and if so, what the colour of the topmost pebble is.

As usual, $\mathcal{A}'$ simulates $\mathcal{A}$. Suppose that $\mathcal{A}$ uses the look-ahead test $\mathcal{B}$ in situation $\langle h, \pi\rangle$. When no pebbles are dropped, i.e., $\pi = \varepsilon$, the test whether $\mathcal{B}$, started in that situation, has a successful computation, is an MSO head test. Indeed, the site $T = \{(t, h) \mid \exists\, p_0 \in Q_0^{\mathcal{B}}, s \in T_{\Delta^{\mathcal{B}}} : \langle p_0, h, \varepsilon\rangle \Rightarrow_{t,\mathcal{B}}^* s\}$ is regular, as $\mathrm{mark}(T)$ is the domain of the v_mI-PTT $\mathcal{B}'$ that starts in the root, looks for the marked node h, and then simulates $\mathcal{B}$. Domains are regular by Corollary 9, and $\mathcal{A}'$ can perform MSO head tests by Lemma 12.

In general, one may imagine that $\mathcal{A}'$ implements the look-ahead test by simulating $\mathcal{B}$. However, when $\mathcal{A}'$ is ready with the simulation of $\mathcal{B}$, that started with the stack π of $\mathcal{A}$, $\mathcal{A}'$ must be able to recover π to continue the simulation of $\mathcal{A}$. Note that $\mathcal{B}$ can inspect π, thereby possibly destroying part of π and adding something else. For this reason the computations of $\mathcal{B}$ starting at the position of the topmost pebble of π will be precomputed. With each pebble dropped by $\mathcal{A}$, the automaton $\mathcal{A}'$ stores the set S of states p of $\mathcal{B}$ for which $\mathcal{B}$ has a successful computation when started in state p at the position u of the topmost stack element (and with the current stack of $\mathcal{A}$). Now a successful computation of $\mathcal{B}$ can be safely simulated, consisting of a part where the pebbles of $\mathcal{B}$ are on top of the stack π inherited from $\mathcal{A}$, possibly followed by a precomputed part where $\mathcal{B}$ inspects π, starting with a visit to u. We discuss how these state sets are determined, and how they are used (by $\mathcal{A}'$) to perform the look-ahead test. Rather then simulating $\mathcal{B}$, $\mathcal{A}'$ will use MSO tests on the observable configuration, which is possible by Theorem 16. The colour sets of $\mathcal{A}'$ are $C'_{\mathrm{v}} = C_{\mathrm{v}} \times 2^{Q^{\mathcal{B}}}$ and $C'_{\mathrm{i}} = C_{\mathrm{i}} \times 2^{Q^{\mathcal{B}}}$.

If $\mathcal{A}$ drops the first pebble c (i.e., $\pi = (h, c)$), then $\mathcal{A}'$ drops the pebble (c, S) where it determines for every state p of $\mathcal{B}$ whether or not $p \in S$ using an MSO head test: construct $\mathcal{B}'$ as above except that it now drops c at the marked node h before simulating $\mathcal{B}$ in state p. Thus, this time, the domain of $\mathcal{B}'$ is $\mathrm{mark}(T)$ with $T = \{(t, h) \mid \exists\, s \in T_{\Delta^{\mathcal{B}}} : \langle p, h, c\rangle \Rightarrow_{t,\mathcal{B}}^* s\}$.

Suppose now that $\mathcal{A}$ uses the look-ahead test $\mathcal{B}$ when it is in situation $\langle h, \pi\rangle$ with $\pi \neq \varepsilon$, and suppose that the topmost pebble of π has colour d and that the set of visible pebble colours that occur in π is $C_{\mathrm{v}}(\pi) = \{c_1, \ldots, c_\ell\} \subseteq C_{\mathrm{v}}$, with $\ell \in [0, k]$. Then the colour of the topmost pebble of the stack π' of $\mathcal{A}'$ is (d, S) for some set S of states of $\mathcal{B}$, and the set of visible pebble colours that occur in π' is $C_{\mathrm{v}}(\pi') = \{(c_1, S_1), \ldots, (c_\ell, S_\ell)\}$ for some $S_1, \ldots, S_\ell$. Since $\mathcal{A}'$ can perform stack tests, it can determine (d, S). Moreover, it should be clear that $\mathcal{A}'$ can determine $C_{\mathrm{v}}(\pi')$, and hence $C_{\mathrm{v}}(\pi)$, by an MSO test on the visible configuration. With this topmost colour d, this state information S, and this set $C_{\mathrm{v}}(\pi)$ of visible pebbles, the look-ahead test can be performed by $\mathcal{A}'$ as an MSO test on the observable configuration, as follows. Consider the observable configuration tree $\mathrm{obs}(t, \pi')$ with the current node h marked, see Theorem 16. We want to show that there is a regular site T over $\Sigma \times 2^{C'}$ such that $(\mathrm{obs}(t, \pi'), h) \in T$ if and only if there exist $p_0 \in Q_0^{\mathcal{B}}$ and $s \in T_{\Delta^{\mathcal{B}}}$ such that $\langle p_0, h, \pi\rangle \Rightarrow_{t,\mathcal{B}}^* s$. Indeed, $\mathrm{mark}(T)$ is the domain of a $\mathrm{v}_{m'}$I-PTT $\mathcal{B}'$, with $m' = m - \ell$. It first searches for the position u of the topmost pebble, which is the unique node of $\mathrm{obs}(t, \pi')$ of which the label contains the colour (d, S). It drops the special invisible pebble $\odot$ on u, and then proceeds to the marked node h, starts simulating $\mathcal{B}$ and halts successfully when it observes pebble $\odot$ at position u with $\mathcal{B}$ in a state of S, or when it never has observed $\odot$ and $\mathcal{B}$ halts successfully (meaning that pebbles are still on top of $\odot$ when visiting u). Note that $\mathcal{B}'$ can simulate $\mathcal{B}$, which walks on t with pebbles rather than on $\mathrm{obs}(t, \pi')$, because the colours in the labels of the nodes of $\mathrm{obs}(t, \pi')$ contain the observable pebbles on t in the stack π. Also, $\mathcal{B}'$ does not apply rules of $\mathcal{B}$ that contain a drop_{c_i}-instruction with $c_i \in C_{\mathrm{v}}(\pi)$. The domain $\mathrm{mark}(T)$ of $\mathcal{B}'$ is regular and $\mathcal{A}'$ can perform the MSO test T on its observable configuration.

The same reasoning shows that the state set for the next pebble c dropped by $\mathcal{A}$ can be computed by MSO tests on the observable configuration: again $\mathcal{B}'$ first drops the pebble c on h before starting the simulation of $\mathcal{B}$ in any state p.

Finally it should be clear that if $\mathcal{A}$ uses the look-ahead tests $\mathcal{B}_1, \ldots, \mathcal{B}_n$, then state information for every $\mathcal{B}_i$ should be stored in the pebbles, i.e., they are of the form $(c, S_1, \ldots, S_n)$ where S_i is a set of states of $\mathcal{B}_i$. □

A natural question is now whether Theorem 19 also holds for PTA's and PTT's that are allowed to perform stack tests, MSO head tests, and MSO tests on the visible and observable configuration. The answer is yes.

Let us first consider the case of stack tests. Roughly speaking, if $\mathcal{A}$ uses look-ahead tests $\mathcal{B}_1, \ldots, \mathcal{B}_n$, then we just apply the construction of Lemma 1 to both $\mathcal{A}$ and all $\mathcal{B}_i$, $i \in [1, n]$, and then apply Theorem 19 to the resulting equivalent (ordinary) PTA $\mathcal{A}'$ that calls the (ordinary) PTT's $\mathcal{B}'_1, \ldots, \mathcal{B}'_n$. It should be noted that even if $\mathcal{A}$ does *not* use stack tests but some $\mathcal{B}_i$ *does*, the construction of Lemma 1 must be applied to $\mathcal{A}$ too, because the stack that $\mathcal{B}_i$ inherits from $\mathcal{A}$ must contain the necessary additional information concerning the colours of previously dropped pebbles. Vice versa, if $\mathcal{A}$ (or another $\mathcal{B}_j$) uses stack tests but $\mathcal{B}_i$ does not, then $\mathcal{B}_i$ can just ignore the additional information in the stack of $\mathcal{A}$, but it is also correct to apply the construction of Lemma 1 to $\mathcal{B}_i$. However, not only the additional information in the stack should be passed from $\mathcal{A}'$ to $\mathcal{B}'_1, \ldots, \mathcal{B}'_n$, but also the additional information in the finite state of $\mathcal{A}'$. Thus, to be more precise, if $\mathcal{A}$ is in state q and uses the look-ahead test $\mathcal{B}_i$, then whenever $\mathcal{A}'$ is in state (q, γ), it should use the look-ahead test $\mathcal{B}'_i(\gamma)$ that is obtained from $\mathcal{B}'_i$ by changing its set $Q_0^{\mathcal{B}_i} \times \{\varepsilon\}$ of initial states into $Q_0^{\mathcal{B}_i} \times \{\gamma\}$.

For the case of MSO head tests and MSO tests on the visible configuration the proof is easier. The constructions of Lemmas 12 and 13 can be applied to $\mathcal{A}$ and $\mathcal{B}_1, \ldots, \mathcal{B}_n$ independently, depending on whether they use such tests or not. The reason is that these tests are implemented by subroutines for which the pebble stack need not be changed. Finally, for the case of MSO tests on the observable configuration the construction of Theorem 16 is again applied simultaneously to all of $\mathcal{A}$ and $\mathcal{B}_1, \ldots, \mathcal{B}_n$, with beads that take care of all the regular sites T that are used by both $\mathcal{A}$ and $\mathcal{B}_1, \ldots, \mathcal{B}_n$ as tests. That ensures that the beads of $\mathcal{A}'$ also contain the information needed by $\mathcal{B}'_1, \ldots, \mathcal{B}'_n$. Note that in this case (as opposed to the case of stack tests above) $\mathcal{A}'$ does not carry any additional information in its finite state and thus, whenever $\mathcal{A}$ uses $\mathcal{B}_i$ as look-ahead test, $\mathcal{A}'$ can use $\mathcal{B}'_i$ as look-ahead test.

A similar natural question is whether Theorem 19 also holds for PTA's and PTT's that use look-ahead, in particular whether we can allow the look-ahead transducer to use another transducer as look-ahead test. The answer is again yes, with a similar solution. In fact it can be shown that the v_kI-PTA (and v_kI-PTT) even can perform *iterated* look-ahead tests, that is, they can use look-ahead tests that use look-ahead tests that use ... look-ahead tests.

Formally, we define for $n \geq 0$ the notion of a PTA or PTT $\mathcal{A}$ *of (look-ahead) depth* n, by induction on n. Simultaneously we define the finite sets $\mathrm{test}(\mathcal{A})$ and $\mathrm{test}^*(\mathcal{A})$ of PTT's, where $\mathrm{test}(\mathcal{A})$ contains the look-ahead tests of $\mathcal{A}$, and $\mathrm{test}^*(\mathcal{A})$ contains its iterated look-ahead tests plus $\mathcal{A}$ itself. For $n = 0$, a PTA or PTT $\mathcal{A}$ of depth 0 is just a PTA or PTT (without look-ahead tests). Moreover, $\mathrm{test}(\mathcal{A}) = \varnothing$ and $\mathrm{test}^*(\mathcal{A}) = \{\mathcal{A}\}$. For $n \geq 0$, a PTA or PTT $\mathcal{A}$ of depth $n+1$ uses as look-ahead tests arbitrary PTT's of lower depth, i.e., it has rules $\langle q, \sigma, j, b, \mathcal{B}\rangle \to \zeta$ or $\langle q, \sigma, j, b, \neg\mathcal{B}\rangle \to \zeta$ where $\mathcal{B}$ is a PTT of depth $m \leq n$. Furthermore, $\mathrm{test}(\mathcal{A})$ is the set of all PTT's of depth $m \leq n$ that $\mathcal{A}$ uses as look-ahead tests, and $\mathrm{test}^*(\mathcal{A}) = \{\mathcal{A}\} \cup \bigcup_{\mathcal{B}\in\mathrm{test}(\mathcal{A})} \mathrm{test}^*(\mathcal{B})$. A PTA or PTT *with iterated look-ahead tests* is one of depth n, for some $n \in \mathbb{N}$. Note that a PTA (or PTT) of depth 1 is the same as a PTA (or PTT) with look-ahead tests. The definition of the semantics of a PTA or PTT with iterated look-ahead tests is by induction on the depth n, and is entirely analogous to the one for the case $n = 1$ as given in the beginning of this section.

Theorem 20. *For each $k \geq 0$, the v_k*I-PTA *and v_k*I-DPTA *can perform iterated look-ahead tests. The same holds for the v_k*I-PTT *and v_k*I-DPTT.

Proof. We will show that for every v_kI-PTT $\mathcal{C}$ of depth $n \geq 1$ we can construct an equivalent v_kI-PTT $\mathcal{C}'$ of depth $n-1$. The result then follows by induction. Since the construction generalizes the one of Theorem 19 (which is the case $n = 1$), we will need all PTT's in $\mathrm{test}^*(\mathcal{C}')$ to use stack tests and MSO tests on the observable configuration. Thus, for the induction to work, we first have to prove that every v_ℓI-PTT of depth $m \geq 1$ can perform such tests. For the case $m = 1$ we have already argued this after Theorem 19, and the general case can be proved in a similar way. Let $\mathcal{D}$ be a v_ℓI-PTT of depth m such that all $\mathcal{A} \in \mathrm{test}^*(\mathcal{D})$ perform stack tests. We just apply the construction of Lemma 1 simultaneously to every PTT $\mathcal{A} \in \mathrm{test}^*(\mathcal{D})$, resulting in the PTT $\mathcal{A}'$. Moreover, for all $\mathcal{A}, \mathcal{B} \in \mathrm{test}^*(\mathcal{D})$, if $\mathcal{A}$ is in state q and uses look-ahead test $\mathcal{B}$, then whenever $\mathcal{A}'$ is in state (q, γ), it uses look-ahead test $\mathcal{B}'(\gamma)$. Obviously, every $\mathcal{B}'(\gamma)$ is of the same depth as $\mathcal{B}$, and hence the resulting v_ℓI-PTT $\mathcal{D}'$ is of the same depth m as $\mathcal{D}$. For the MSO tests the argument is completely analogous to the argument for $m = 1$ after Theorem 19, applying the appropriate constructions simultaneously to all PTT $\mathcal{A} \in \mathrm{test}^*(\mathcal{D})$.

Now let $\mathcal{C}$ be a v_kI-PTT of depth $n \geq 1$ and let us construct an equivalent v_kI-PTT $\mathcal{C}'$ of smaller depth. The argument is similar to those above. Let P_0 be the set of all $\mathcal{B} \in \mathrm{test}^*(\mathcal{C})$ of depth 0, i.e., all PTT without look-ahead tests, and let P_1 contain all $\mathcal{A} \in \mathrm{test}^*(\mathcal{C})$ of depth ≥ 1. We now apply the construction of Theorem 19 simultaneously to every PTT $\mathcal{A} \in P_1$, resulting in a PTT $\mathcal{A}'$ that stores state information of every $\mathcal{B} \in P_0$ in the pebbles. If $\mathcal{A}_1 \in P_1$ uses look-ahead test $\mathcal{A}_2 \in P_1$, then $\mathcal{A}'_1$ uses look-ahead test $\mathcal{A}'_2$. Note that if $\mathcal{A} \in P_1$ uses look-ahead test $\mathcal{B} \in P_0$, then $\mathcal{A}'$ uses an MSO test instead. Thus, clearly, the depth of every $\mathcal{A}'$ is one less than the depth of $\mathcal{A}$, and so the depth of the resulting v_kI-PTT $\mathcal{C}'$ is $n-1$. Finally, we remove the stack tests and MSO tests from $\mathcal{C}'$ and its iterated look-ahead tests as explained above for $\mathcal{D}$. □

Although this result does not seem practically useful, it will become important when we propose the query language Pebble XPath in the next section, as an extension of Regular XPath. Intuitively, Pebble XPath expressions are similar to I-PTA's

with iterated look-ahead tests. We note that TA's with iterated look-ahead tests are used in [53] to prove that Regular XPath is not MSO complete.

9. Document navigation

We define *Pebble XPath*, an extension of Regular XPath [39] with pebbles. Due to its potential application to navigation in XML documents, it works on (nonempty) forests rather than trees. We prove that the trips defined by the path expressions of Pebble XPath are exactly the MSO definable trips on forests.

Pebble XPath has path expressions (denoted α, β) and node expressions (denoted φ, ψ). These expressions concern forests over an (unranked) alphabet Σ of node labels, or tags, that can be chosen arbitrarily. Since we are mainly interested in path expressions, we view the node expressions as auxiliary. A path expression describes a walk through a given nonempty forest f over Σ during which invisible coloured pebbles can be dropped on and lifted from the nodes of f, in a nested (stack-like) manner. Such a walk steps through f from node to node following both the vertical and horizontal edges in either direction. The context in which a path expression is evaluated (i.e., the situation at the start of the walk) is a pair $\langle u, \pi \rangle$ consisting of a node u of f and a stack π of pebbles that lie on the nodes of f. Formally, a *context*, or *situation*, on a forest f is an element of the set $\mathrm{Sit}(f) = N(f) \times (N(f) \times C)^*$, where $N(f)$ is the set of nodes of f and C is the finite set of colours of the pebbles (that can be chosen arbitrarily). The walk ends in another context. Thus, the semantics of a path expression is a binary relation on $\mathrm{Sit}(f)$. Similarly, the semantics of a node expression is a subset of $\mathrm{Sit}(f)$, i.e., a test on a given context. Note that the notion of a context on a forest is entirely similar to that of a situation on a ranked tree for an I-PTA with (invisible) colour set C.

For the syntax of Pebble XPath, we start with the basic path expressions, with $c \in C$:

$$\alpha_0 ::= \mathtt{child} \mid \mathtt{parent} \mid \mathtt{right} \mid \mathtt{left} \mid \mathtt{drop}_c \mid \mathtt{lift}_c$$

The first four expressions operate on the context node only (in the usual way, moving to a child, the parent, the next sibling, and the previous sibling, respectively), whereas the last two also operate on the pebble stack (dropping/lifting a pebble of colour c on/from the context node u, which is modeled by pushing/popping the pair (u, c) on/off the stack). The syntax of path expressions is

$$\alpha ::= \alpha_0 \mid ?\varphi \mid \alpha \cup \beta \mid \alpha/\beta \mid \alpha^*$$

where β is an alias of α. The three last expressions show the usual regular operations on binary relations: union, composition, and transitive-reflexive closure. The expression $?\varphi$ denotes the identity relation on the set of contexts defined by the node expression φ, i.e., it filters the current context by requiring that φ is true.

We now turn to the node expressions and start with the basic ones, with $\sigma \in \Sigma$:

$$\varphi_0 ::= \mathtt{haslabel}_\sigma \mid \mathtt{isleaf} \mid \mathtt{isroot} \mid \mathtt{isfirst} \mid \mathtt{islast} \mid \mathtt{haspebble}_c$$

The first five expressions test whether the context node has label σ, whether it is a leaf, a root, the first among its siblings, or the last among its siblings. The last expression (which is the only one that also uses the pebble stack) tests whether the topmost pebble, i.e., the most recently dropped pebble, lies on the context node and has colour c. The syntax of node expressions is

$$\varphi ::= \varphi_0 \mid \langle \alpha \rangle \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi$$

where ψ is an alias of φ. The last three expressions show the usual boolean operations. The expression $\langle \alpha \rangle$ is like a predicate $[\alpha]$ in XPath 1.0, which filters the context by requiring the existence of at least one successful α-walk starting from this context. In terms of tree-walking automata it is a look-ahead test. We will also consider the language *Pebble CAT*, which is obtained from Pebble XPath by dropping the filter tests $\varphi ::= \langle \alpha \rangle$. The expressions of Pebble CAT are *caterpillar expressions* extended with pebbles.

The formal semantics of Pebble XPath expressions is given in Tables 3 and 4. For every nonempty forest f over Σ, the semantics $[\![\alpha]\!]_f \subseteq \mathrm{Sit}(f) \times \mathrm{Sit}(f)$ and $[\![\varphi]\!]_f \subseteq \mathrm{Sit}(f)$ of path and node expressions are defined, where u, u' vary over $N(f)$, π, π' vary over $(N(f) \times C)^*$, and p varies over $N(f) \times C$. Note that $[\![\mathtt{parent}]\!]_f = [\![\mathtt{child}]\!]_f^{-1}$, $[\![\mathtt{left}]\!]_f = [\![\mathtt{right}]\!]_f^{-1}$, and $[\![\mathtt{lift}_c]\!]_f = [\![\mathtt{drop}_c]\!]_f^{-1}$. Note also that the set $[\![\langle \alpha \rangle]\!]_f$ is the domain of the binary relation $[\![\alpha]\!]_f$.

The filtering XPath expression $\alpha[\beta]$ of XPath 1.0 can here be defined as $\alpha[\beta] = \alpha/?\langle\beta\rangle$. Also, the node expression $\mathtt{loop}(\alpha)$ from [30,52] can be defined as $\mathtt{loop}(\alpha) = \langle \mathtt{drop}_c/\alpha/\mathtt{lift}_c \rangle$ where c is a colour not occurring in α. Then $[\![\mathtt{loop}(\alpha)]\!]_f = \{\langle u, \pi \rangle \mid (\langle u, \pi \rangle, \langle u, \pi \rangle) \in [\![\alpha]\!]_f\} = \{\langle u, \pi \rangle \mid (\langle u, \varepsilon \rangle, \langle u, \varepsilon \rangle) \in [\![\alpha]\!]_f\}$, because α cannot inspect the stack π and it must return to u in order to lift pebble c.

Two path expressions α and β are *equivalent*, denoted by $\alpha \equiv \beta$, if $[\![\alpha]\!]_f = [\![\beta]\!]_f$ for every nonempty forest f over Σ, and similarly for node expressions. Note that $?(\varphi \wedge \psi) \equiv ?\varphi/?\psi$ and $?(\varphi \vee \psi) \equiv ?\varphi \cup ?\psi$. Hence, using De Morgan's laws, the syntax for node expressions can be replaced by $\varphi ::= \varphi_0 \mid \neg\varphi_0 \mid \langle\alpha\rangle \mid \neg\langle\alpha\rangle$ for Pebble XPath, and $\varphi ::= \varphi_0 \mid \neg\varphi_0$ for Pebble CAT. Thus, keeping only the basic node expressions, we can always assume that the syntax for path expressions is

Table 3
Semantics of Pebble XPath path expressions.

$$
\begin{aligned}
[\![\texttt{child}]\!]_f &= \{(\langle u,\pi\rangle,\langle u',\pi\rangle) \mid u' \text{ is a child of } u\}\\
[\![\texttt{parent}]\!]_f &= \{(\langle u,\pi\rangle,\langle u',\pi\rangle) \mid u' \text{ is the parent of } u\}\\
[\![\texttt{right}]\!]_f &= \{(\langle u,\pi\rangle,\langle u',\pi\rangle) \mid u' \text{ is the next sibling of } u\}\\
[\![\texttt{left}]\!]_f &= \{(\langle u,\pi\rangle,\langle u',\pi\rangle) \mid u' \text{ is the previous sibling of } u\}\\
[\![\texttt{drop}_c]\!]_f &= \{(\langle u,\pi\rangle,\langle u,\pi\, p\rangle) \mid p=(u,c)\}\\
[\![\texttt{lift}_c]\!]_f &= \{(\langle u,\pi\, p\rangle,\langle u,\pi\rangle) \mid p=(u,c)\}\\
[\![?\varphi]\!]_f &= \{(\langle u,\pi\rangle,\langle u,\pi\rangle) \mid \langle u,\pi\rangle \in [\![\varphi]\!]_f\}\\
[\![\alpha\cup\beta]\!]_f &= [\![\alpha]\!]_f \cup [\![\beta]\!]_f\\
[\![\alpha/\beta]\!]_f &= [\![\alpha]\!]_f \circ [\![\beta]\!]_f\\
[\![\alpha^*]\!]_f &= [\![\alpha]\!]_f^*
\end{aligned}
$$

Table 4
Semantics of Pebble XPath node expressions.

$$
\begin{aligned}
[\![\texttt{haslabel}_\sigma]\!]_f &= \{\langle u,\pi\rangle \mid u \text{ has label } \sigma\}\\
[\![\texttt{isleaf}]\!]_f &= \{\langle u,\pi\rangle \mid u \text{ is a leaf}\}\\
[\![\texttt{isroot}]\!]_f &= \{\langle u,\pi\rangle \mid u \text{ is a root}\}\\
[\![\texttt{isfirst}]\!]_f &= \{\langle u,\pi\rangle \mid u \text{ is a first sibling}\}\\
[\![\texttt{islast}]\!]_f &= \{\langle u,\pi\rangle \mid u \text{ is a last sibling}\}\\
[\![\texttt{haspebble}_c]\!]_f &= \{\langle u,\pi\, p\rangle \mid p=(u,c)\}\\
[\![\langle\alpha\rangle]\!]_f &= \{\langle u,\pi\rangle \mid \exists\langle u',\pi'\rangle\colon (\langle u,\pi\rangle,\langle u',\pi'\rangle)\in[\![\alpha]\!]_f\}\\
[\![\neg\varphi]\!]_f &= \mathrm{Sit}(f)\setminus[\![\varphi]\!]_f\\
[\![\varphi\wedge\psi]\!]_f &= [\![\varphi]\!]_f \cap [\![\psi]\!]_f\\
[\![\varphi\vee\psi]\!]_f &= [\![\varphi]\!]_f \cup [\![\psi]\!]_f
\end{aligned}
$$

$$\alpha ::= \alpha_0 \mid ?\varphi_0 \mid ?\neg\varphi_0 \mid ?\langle\beta\rangle \mid ?\neg\langle\beta\rangle \mid \alpha\cup\beta \mid \alpha/\beta \mid \alpha^*$$

for Pebble XPath, and hence

$$\alpha ::= \alpha_0 \mid ?\varphi_0 \mid ?\neg\varphi_0 \mid \alpha\cup\beta \mid \alpha/\beta \mid \alpha^*$$

for Pebble CAT. In that case we will say that we assume the syntax to be in *normal form*.

Note also that all basic node expressions except $\texttt{haslabel}_\sigma$ are redundant, because $\texttt{isleaf} \equiv \neg\langle\texttt{child}\rangle$ (a node is a leaf if and only if it has no children), $\texttt{isroot} \equiv \neg\langle\texttt{parent}\rangle$, $\texttt{isfirst} \equiv \neg\langle\texttt{left}\rangle$, $\texttt{islast} \equiv \neg\langle\texttt{right}\rangle$, and $\texttt{haspebble}_c \equiv \langle\texttt{lift}_c\rangle$. However, these basic node expressions were kept in the syntax, because we also wish to consider the subset Pebble CAT in which there are no filter tests $\langle\alpha\rangle$. Note finally that when $\texttt{drop}_c$, $\texttt{lift}_c$, and $\texttt{haspebble}_c$ are removed from Pebble XPath, the resulting formalism is exactly Regular XPath [39] (and in the semantics the stack can, of course, be disregarded).

The purpose of Pebble XPath is the same as that of XPath: to define trips, i.e., binary patterns. Recall from Section 2 that a trip T over an unranked alphabet Σ is a set $T \subseteq \{(f,u,v) \mid f \in F_\Sigma,\ u,v \in N(f)\}$ where F_Σ is the set of forests over Σ. Note that f is always a nonempty forest. For a path expression α (based on Σ and some C) we say that α *defines the trip* $T(\alpha) = \{(f,u,v) \mid \exists\pi \in (N(f)\times C)^*\colon (\langle u,\varepsilon\rangle,\langle v,\pi\rangle) \in [\![\alpha]\!]_f\}$. We now define a trip T over Σ to be *definable in Pebble XPath* if there exists a Pebble XPath path expression α such that $T = T(\alpha)$. And similarly for Pebble CAT. The next theorem states that Pebble XPath and Pebble CAT have the same expressive power as MSO logic on forests.

Theorem 21. *A trip is definable in Pebble XPath if and only if it is definable in Pebble CAT if and only if it is* MSO *definable.*

As such our expressions have the desirable property of being a Core (and even Regular) XPath extension that is complete for MSO definable binary patterns. Other such extensions were considered in [30] (TMNF caterpillar expressions) and [52] (μRegular XPath). Pebble CAT is similar to PCAT of [30] which has the same expressive power as the V-PTA (and thus less than MSO by [8]). In PCAT the nesting of pebbles is defined syntactically rather than semantically.

The proof of Theorem 21 is given in the remainder of this section. It should be clear that Pebble CAT is closely related to the I-PTA. In fact, we will show later that their relationship can be viewed as the classical equivalence of regular expressions and finite automata. The remainder of the proof is then directly based on the fact that the I-PTA has the same expressive power as MSO logic for defining trips on trees (Theorem 15), and on the fact that the I-PTA can perform iterated look-ahead tests (Theorem 20). One technical problem is that these theorems are formulated for ranked trees rather than unranked forests. Thus we start by adapting Pebble XPath to ranked trees and showing that it suffices to prove Theorem 21 for ranked trees instead of forests.

Table 5
Basic path expressions α_0 for a ranked tree t.

$[\![\texttt{down}_1]\!]_t = \{(\langle u,\pi\rangle,\langle u',\pi\rangle) \mid u' \text{ is the first child of } u\}$
$[\![\texttt{down}_2]\!]_t = \{(\langle u,\pi\rangle,\langle u',\pi\rangle) \mid u' \text{ is the second child of } u\}$
$[\![\texttt{up}]\!]_t = \{(\langle u,\pi\rangle,\langle u',\pi\rangle) \mid u' \text{ is the parent of } u\}$
$[\![\texttt{drop}_c]\!]_t = \{(\langle u,\pi\rangle,\langle u,\pi\, p\rangle) \mid p=(u,c)\}$
$[\![\texttt{lift}_c]\!]_t = \{(\langle u,\pi\, p\rangle,\langle u,\pi\rangle) \mid p=(u,c)\}$

Table 6
Basic node expressions φ_0 for a ranked tree t.

$[\![\texttt{haslabel}_\sigma]\!]_t = \{\langle u,\pi\rangle \mid u \text{ has label } \sigma\}$
$[\![\texttt{ischild}_0]\!]_t = \{\langle u,\pi\rangle \mid u \text{ is the root}\}$
$[\![\texttt{ischild}_1]\!]_t = \{\langle u,\pi\rangle \mid u \text{ is a first child}\}$
$[\![\texttt{ischild}_2]\!]_t = \{\langle u,\pi\rangle \mid u \text{ is a second child}\}$
$[\![\texttt{haspebble}_c]\!]_t = \{\langle u,\pi\, p\rangle \mid p=(u,c)\}$

Pebble XPath on ranked trees. Since ranked trees are a special case of unranked forests, we need not change Pebble XPath for its use on ranked trees. However, for its comparison to the I-PTA it is more convenient to change its basic path expressions α_0 and basic node expressions φ_0 as follows:

$$\alpha_0 ::= \texttt{down}_1 \mid \texttt{down}_2 \mid \texttt{up} \mid \texttt{drop}_c \mid \texttt{lift}_c$$
$$\varphi_0 ::= \texttt{haslabel}_\sigma \mid \texttt{ischild}_0 \mid \texttt{ischild}_1 \mid \texttt{ischild}_2 \mid \texttt{haspebble}_c$$

The semantics of these basic expressions for a tree t over Σ is given in Tables 5 and 6. Since we will only be interested in ranked trees that encode forests, we assume that Σ is a ranked alphabet and that the rank of each element of Σ is at most 2. Note that up has the same semantics as parent, and that the semantics of $\texttt{drop}_c$, $\texttt{lift}_c$, $\texttt{haslabel}_\sigma$, and $\texttt{haspebble}_c$ is unchanged. The remaining expressions of Pebble XPath, and their semantics (for t instead of f), are the same as for forests, cf. the last four lines of Tables 3 and 4.

We first show that for every path expression α on forests there is a path expression α' that computes the same trip as α on the binary encoding of the forests as ranked trees. We use the encoding enc′ defined in Section 2, which encodes forests over the alphabet Σ as ranked trees over the associated ranked alphabet Σ'. Note that for every forest f, $\text{enc}'(f)$ has the same nodes as f. For a trip T on forests, we define the *encoded trip* $\text{enc}'(T)$ on ranked trees by $\text{enc}'(T) = \{(\text{enc}'(f),u,v) \mid (f,u,v) \in T\}$.

Lemma 22. *For every Pebble XPath path expression α on forests over Σ, a Pebble XPath path expression α' on ranked trees over Σ' can be constructed in polynomial time such that $T(\alpha') = \text{enc}'(T(\alpha))$. If α is a Pebble CAT expression, then so is α'.*

Proof. The proof is an elementary coding exercise. Let us start with Pebble XPath. We will, in fact, define α' such that $[\![\alpha']\!]_{\text{enc}'(f)} = [\![\alpha]\!]_f$ for every $f \in F_\Sigma$, which implies the result. It clearly suffices to do this for basic path expressions α_0, and similarly for basic node expressions φ_0. As observed before, all basic node expressions except $\texttt{haslabel}_\sigma$ are redundant, so it suffices to define $\texttt{haslabel}'_\sigma \equiv \texttt{haslabel}_{\sigma 11} \vee \texttt{haslabel}_{\sigma 10} \vee \texttt{haslabel}_{\sigma 01} \vee \texttt{haslabel}_{\sigma 00}$. We now turn to the basic path expressions. We will use the auxiliary basic path expressions $\texttt{child}_1$ and $\texttt{parent}_1$ with the semantics $[\![\texttt{child}_1]\!]_f = \{(\langle u,\pi\rangle,\langle u',\pi\rangle) \mid u' \text{ is the first child of } u\}$ and $[\![\texttt{parent}_1]\!]_f = [\![\texttt{child}_1]\!]_f^{-1}$. Since clearly $\texttt{child} \equiv \texttt{child}_1/\texttt{right}^*$ and $\texttt{parent} \equiv \texttt{left}^*/\texttt{parent}_1$, it suffices to define $\texttt{child}'_1$ and $\texttt{parent}'_1$ instead of child′ and parent′, as follows: $\texttt{child}'_1 \equiv ?\varphi_1/\texttt{down}_1$ where φ_1 is the disjunction of $\texttt{haslabel}_{\sigma 11}$ and $\texttt{haslabel}_{\sigma 10}$ for all $\sigma \in \Sigma$, and $\texttt{parent}'_1 \equiv ?\texttt{ischild}_1/\texttt{up}/?\varphi_1$. Then we define $\texttt{right}' \equiv \texttt{down}_2 \cup ?\varphi_2/\texttt{down}_1$ where φ_2 is the disjunction of all $\texttt{haslabel}_{\sigma 01}$ for $\sigma \in \Sigma$. Since $[\![\texttt{left}]\!]_f$ is the inverse of $[\![\texttt{right}]\!]_f$, we define $\texttt{left}' \equiv ?\texttt{ischild}_2/\texttt{up} \cup ?\texttt{ischild}_1/\texttt{up}/?\varphi_2$. Finally, $\texttt{drop}'_c \equiv \texttt{drop}_c$ and $\texttt{lift}'_c \equiv \texttt{lift}_c$.

To prove the result for Pebble CAT, we also have to consider the other basic node expressions φ_0. Obviously, we define $\texttt{haspebble}'_c \equiv \texttt{haspebble}_c$. We define isleaf′ to be the disjunction of $\texttt{haslabel}_{\sigma 01}$ and $\texttt{haslabel}_{\sigma 00}$ for all $\sigma \in \Sigma$, and similarly, islast′ to be the disjunction of $\texttt{haslabel}_{\sigma 10}$ and $\texttt{haslabel}_{\sigma 00}$ for all $\sigma \in \Sigma$. It remains to consider isfirst and isroot. Since we may assume the syntax of α to be in normal form, it suffices to define $(?\varphi_0)'$ and $(?\neg\varphi_0)'$. We define $(?\texttt{isfirst})' \equiv ?\texttt{ischild}_0 \cup \texttt{ischild}_1/\texttt{up}/\texttt{child}'_1$ and $(?\neg\texttt{isfirst})' \equiv \texttt{up}/\texttt{right}'$ where $\texttt{child}'_1$ and right′ are defined above. For isroot, we first note that $?\texttt{isroot} \equiv \texttt{drop}_c/\texttt{left}^*/?\texttt{isroot}/?\texttt{isfirst}/\texttt{right}^*/\texttt{lift}_c$ where c is any element of C. Intuitively, we walk from the current node to the left until we arrive at the first root, and then walk back. Thus, since the first root of a forest f is encoded as the root of $\text{enc}'(f)$, we define $(?\texttt{isroot})' \equiv \texttt{drop}_c/(\texttt{left}')^*/?\texttt{ischild}_0/(\texttt{right}')^*/\texttt{lift}_c$. Finally, we define $(?\neg\texttt{isroot})'$ by $(?\neg\texttt{isroot})' \equiv \texttt{drop}_c/\texttt{parent}'/\texttt{child}'/\texttt{lift}_c$. □

Next we prove the reverse direction of Lemma 22, for Pebble CAT.

Lemma 23. *For every Pebble CAT path expression α on ranked trees over Σ' there is a Pebble CAT path expression α' on forests over Σ such that* $\mathrm{enc}'(T(\alpha')) = T(\alpha)$.

Proof. This is also an elementary coding exercise. We assume the syntax of α to be in normal form, whereas for α' we keep the full syntax. As in the previous lemma, we will define α' such that $[\![\alpha']\!]_f = [\![\alpha]\!]_{\mathrm{enc}'(f)}$. It suffices to do this for path expressions α_0, $?\varphi_0$, and $?\neg\varphi_0$. We start with α_0 and we define $\mathtt{down}'_1 \equiv \mathtt{child}/?\mathtt{isfirst} \cup ?\mathtt{isleaf}/\mathtt{right}$ and $\mathtt{down}'_2 \equiv ?\neg\mathtt{isleaf}/\mathtt{right}$. Moreover, $\mathrm{up}' \equiv ?\mathtt{isfirst}/\mathtt{parent} \cup \mathtt{left}$. Finally, $\mathtt{drop}'_c \equiv \mathtt{drop}_c$ and $\mathtt{lift}'_c \equiv \mathtt{lift}_c$. We now turn to the basic node expressions. For $\varphi_0 \equiv \mathtt{haslabel}_{\sigma^{10}}$ we define $(?\varphi_0)' \equiv ?\varphi'_0$ and $(?\neg\varphi_0)' \equiv ?\neg\varphi'_0$, where $\varphi'_0 \equiv \mathtt{haslabel}_\sigma \wedge \neg\mathtt{isleaf} \wedge \mathtt{islast}$, and similarly for $\mathtt{haslabel}_{\sigma^{11}}$, $\mathtt{haslabel}_{\sigma^{01}}$, and $\mathtt{haslabel}_{\sigma^{00}}$. We do this also for $\varphi_0 \equiv \mathtt{ischild}_0$ with $\varphi'_0 \equiv \mathtt{isroot} \wedge \mathtt{isfirst}$, and for $\varphi_0 \equiv \mathtt{haspebble}_c$ with $\varphi'_0 \equiv \mathtt{haspebble}_c$. It remains to consider $\mathtt{ischild}_1$ and $\mathtt{ischild}_2$. We define $(?\mathtt{ischild}_2)' \equiv \mathtt{left}/?\neg\mathtt{isleaf}/\mathtt{right}$ and hence $(?\neg\mathtt{ischild}_2)' \equiv ?\mathtt{isfirst} \cup \mathtt{left}/?\mathtt{isleaf}/\mathtt{right}$. For $\mathtt{ischild}_1$ the definitions of $(?\mathtt{ischild}_1)'$ and $(?\neg\mathtt{ischild}_1)'$ now follow from the fact that $?\mathtt{ischild}_1 \equiv ?\neg\mathtt{ischild}_0/?\neg\mathtt{ischild}_2$ and $?\neg\mathtt{ischild}_1 \equiv ?\mathtt{ischild}_0 \cup ?\mathtt{ischild}_2$. □

Lemmas 22 and 23 together show that if the first equivalence of Theorem 21 holds for ranked trees, then it also holds for forests. To show this also for the second equivalence, we need the next elementary lemma.

Lemma 24. *For every trip T on forests, T is* MSO *definable if and only if* $\mathrm{enc}'(T)$ *is* MSO *definable.*

Proof. (Only if) Since f and $\mathrm{enc}'(f)$ have the same nodes, for every forest f over Σ, it suffices to show that the atomic formulas $\mathrm{lab}_\sigma(x)$, $\mathrm{down}(x, y)$, and $\mathrm{next}(x, y)$ for forests can be expressed by an MSO formula for the ranked trees that encode the forests. Clearly, $\mathrm{lab}_\sigma(x)$ can be expressed by the disjunction of all $\mathrm{lab}_{\sigma^{k\ell}}(x)$ for $k, \ell \in \{0, 1\}$, as in the proof of Lemma 22. For $\mathrm{down}(x, y)$ we show that the trip $T = \{(\mathrm{enc}'(f), u, v) \mid f \models \mathrm{down}(u, v)\}$ is MSO definable. This follows from Proposition 14 because $T = T(\mathcal{B})$ for the TA $\mathcal{B}$ that has the rules (for all $k, \ell \in \{0, 1\}$, $j \in \{0, 1, 2\}$, and $\sigma \in \Sigma$):

$$\begin{aligned}
\langle p_0, \sigma^{1\ell}, j\rangle &\to \langle p, \mathrm{down}_1\rangle,\\
\langle p, \sigma^{11}, j\rangle &\to \langle p, \mathrm{down}_2\rangle,\\
\langle p, \sigma^{01}, j\rangle &\to \langle p, \mathrm{down}_1\rangle,\\
\langle p, \sigma^{k\ell}, j\rangle &\to \langle p_\infty, \mathrm{stay}\rangle,
\end{aligned}$$

where p_0 is the initial and p_∞ the final state of $\mathcal{B}$. Thus, there is a formula $\varphi(x, y)$ such that $\mathrm{enc}'(f) \models \varphi(u, v)$ if and only if $f \models \mathrm{down}(u, v)$, for every forest f, which means that $\varphi(x, y)$ expresses $\mathrm{down}(x, y)$ on the encoding of f.[19] The formula $\mathrm{next}(x, y)$ can be treated in the same way, where $\mathcal{B}$ now has the rules $\langle p_0, \sigma^{11}, j\rangle \to \langle p_\infty, \mathrm{down}_2\rangle$ and $\langle p_0, \sigma^{01}, j\rangle \to \langle p_\infty, \mathrm{down}_1\rangle$, and hence $T(\mathcal{B}) = \{(\mathrm{enc}'(f), u, v) \mid f \models \mathrm{next}(u, v)\}$.

(If) For the same reason as above, it suffices to show that the atomic formulas $\mathrm{down}_i(x, y)$ and $\mathrm{lab}_{\sigma^{k\ell}}(x)$ for ranked trees over Σ' can be expressed by an MSO formula for the forests they encode. For this we consider the path expressions $\mathtt{down}'_i$ and $\mathtt{haslabel}'_{\sigma^{10}}$ in the proof of Lemma 23, and we define

$$\begin{aligned}
\varphi_1(x, y) &\equiv (\mathrm{down}(x, y) \wedge \mathrm{first}(y)) \vee (\mathrm{leaf}(x) \wedge \mathrm{next}(x, y)),\\
\varphi_2(x, y) &\equiv \neg\,\mathrm{leaf}(x) \wedge \mathrm{next}(x, y),\\
\varphi_{10}(x) &\equiv \mathrm{lab}_\sigma(x) \wedge \neg\,\mathrm{leaf}(x) \wedge \mathrm{last}(x),
\end{aligned}$$

and similarly for the other $\varphi_{k\ell}(x, y)$. Then $\mathrm{enc}'(f) \models \mathrm{down}_i(u, v)$ if and only if $f \models \varphi_i(u, v)$, and $\mathrm{enc}'(f) \models \mathrm{lab}_{\sigma^{k\ell}}(u)$ if and only if $f \models \varphi_{k\ell}(u)$. □

From now on, when we refer to Pebble XPath or Pebble CAT we always mean their version on ranked trees.

Directive I-PTA's. For the purpose of the proof of Theorem 21 on ranked trees, we formulate the I-PTA in an alternative way and, for lack of a better name, call it the *directive* I-PTA. For an alphabet Σ (of which every element has rank at most 2) and a finite set C of colours, we define a *directive* over Σ and C to be a path expression τ with the syntax $\tau ::= \alpha_0 \mid ?\varphi_0 \mid ?\neg\varphi_0$ for the same Σ and C (where α_0 and φ_0 are as in Tables 5 and 6). The finite set of directives over Σ and C is denoted $D_{\Sigma,C}$.

A *directive* I-PTA is a tuple $\mathcal{A} = (\Sigma, Q, Q_0, F, C, R)$, where Σ, Q, Q_0, F, and C are as for an ordinary I-PTA (with $C = C_i$), and R is a finite set of rules of the form $\langle q, \tau, q'\rangle$ where $q, q' \in Q$ and $\tau \in D_{\Sigma,C}$. Thus, syntactically, $\mathcal{A}$ can be viewed as a finite automaton of which each state transition is labeled by a directive, i.e., either by a basic path expression of Pebble

[19] For the reader familiar with MSO logic we note that it is also easy to write down the formula $\varphi(x, y)$ using the equivalences in the proof of Lemma 22 and the fact that the transitive-reflexive closure of an MSO definable relation is MSO definable.

XPath, or by a basic node expression of Pebble XPath, or its negation, where the node expressions are turned into path expressions by the ?-operator. Intuitively, $?\varphi_0$ and $?\neg\varphi_0$ represent a basic test on the current situation, whereas α_0 is a basic instruction to be executed on the current situation. Just as for an ordinary I-PTA, a situation on a tree $t \in T_\Sigma$ is a pair $\langle u, \pi\rangle \in \mathrm{Sit}(t)$ and a configuration is a triple $\langle q, u, \pi\rangle$ with $q \in Q$ and $\langle u, \pi\rangle \in \mathrm{Sit}(t)$. We write $\langle q, u, \pi\rangle \Rightarrow_{t,\mathcal{A}} \langle q', u', \pi'\rangle$ if there is a rule $\langle q, \tau, q'\rangle$ such that $(\langle u, \pi\rangle, \langle u', \pi'\rangle) \in [\![\tau]\!]_t$, where $[\![\tau]\!]_t$ is the semantics of path expression τ on t (cf. Tables 5 and 6 for α_0 and φ_0, and Table 3 for the ?-operator). To indicate the directive τ that is executed by $\mathcal{A}$ in this computation step we also write $\langle q, u, \pi\rangle \Rightarrow^{\tau}_{t,\mathcal{A}} \langle q', u', \pi'\rangle$. Moreover, we define the semantics $[\![\mathcal{A}]\!]_t$ of $\mathcal{A}$ on tree t as $[\![\mathcal{A}]\!]_t = \{(\langle u, \pi\rangle, \langle u', \pi'\rangle) \in \mathrm{Sit}(t) \times \mathrm{Sit}(t) \mid \exists q_0 \in Q_0, q_\infty \in F : \langle q_0, u, \pi\rangle \Rightarrow^*_{t,\mathcal{A}} \langle q_\infty, u', \pi'\rangle\}$. Finally, the trip computed by $\mathcal{A}$ on T_Σ is $T(\mathcal{A}) = \{(t, u, v) \mid \exists \pi \in (N(t) \times C)^* : (\langle u, \varepsilon\rangle, \langle v, \pi\rangle) \in [\![\mathcal{A}]\!]_t\}$.

For the sake of the proofs below we also define $[\![\mathcal{A}]\!]_t$ for an ordinary I-PTA $\mathcal{A}$ on a tree t, in entirely the same way as above for a directive I-PTA.

A directive I-PTA $\mathcal{A}$ *with look-ahead tests* is defined similarly to the ordinary case in Section 8 (restricted to automata), by additionally allowing rules of the form $\langle q, ?\langle\mathcal{B}\rangle, q'\rangle$ and $\langle q, ?\neg\langle\mathcal{B}\rangle, q'\rangle$ where $\mathcal{B}$ is another directive I-PTA. The above semantics stays the same, with (as expected)

$$[\![?\langle\mathcal{B}\rangle]\!]_t = \{(\langle u, \pi\rangle, \langle u, \pi\rangle) \mid \exists\langle u', \pi'\rangle : (\langle u, \pi\rangle, \langle u', \pi'\rangle) \in [\![\mathcal{B}]\!]_t\}$$

and similarly for $[\![?\neg\langle\mathcal{B}\rangle]\!]_t$ (with $\neg\exists$). A directive I-PTA *with iterated look-ahead tests* is defined as in Section 8. We will use I-PTA$^{\mathrm{LA}}$ as an abbreviation of 'I-PTA with iterated look-ahead tests'.

We now show that the directive I-PTA has the same expressive power as the I-PTA (and similarly with iterated look-ahead tests). Hence Theorems 15 and 20 also hold for the directive I-PTA, i.e., it computes the MSO definable trips, and it can perform iterated look-ahead tests. In what follows, we only consider I-PTA's of which every input symbol has at most rank 2.

Lemma 25. *For every directive* I-PTA$^{\mathrm{LA}}$ $\mathcal{A}$ *there is an* I-PTA$^{\mathrm{LA}}$ $\mathcal{A}'$ *such that* $T(\mathcal{A}') = T(\mathcal{A})$.

Proof. Let $\mathcal{A} = (\Sigma, Q, Q_0, F, C, R)$ be a directive I-PTA. We will, in fact, define the I-PTA $\mathcal{A}'$ such that $[\![\mathcal{A}']\!]_t = [\![\mathcal{A}]\!]_t$ for every $t \in T_\Sigma$, which implies the result.

We let $\mathcal{A}' = (\Sigma, Q, Q_0, F, C, \varnothing, C_{\mathrm{i}}, R', 0)$ where $C_{\mathrm{i}} = C$ and R' is defined as follows. If $\langle q, \alpha_0, q'\rangle$ is a rule of $\mathcal{A}$, where α_0 is a basic path expression, then $\mathcal{A}'$ has all rules $\langle q, \sigma, j, b\rangle \to \langle q', \alpha_0\rangle$. We now turn to the basic node expressions. A rule $\langle q, ?\mathtt{haslabel}_\sigma, q'\rangle$ is simulated by all rules $\langle q, \sigma, j, b\rangle \to \langle q', \mathrm{stay}\rangle$, and a rule $\langle q, ?\neg\mathtt{haslabel}_\sigma, q'\rangle$ by all rules $\langle q, \tau, j, b\rangle \to \langle q', \mathrm{stay}\rangle$ with $\tau \in \Sigma \setminus \{\sigma\}$. A rule $\langle q, ?\mathtt{ischild}_j, q'\rangle$ is simulated by all rules $\langle q, \sigma, j, b\rangle \to \langle q', \mathrm{stay}\rangle$, and a rule $\langle q, ?\neg\,\mathtt{ischild}_j, q'\rangle$ by the two rules $\langle q, \sigma, j', b\rangle \to \langle q', \mathrm{stay}\rangle$ with $j' \in \{0, 1, 2\} \setminus \{j\}$. A rule $\langle q, ?\mathtt{haspebble}_c, q'\rangle$ is simulated by all rules $\langle q, \sigma, j, \{c\}\rangle \to \langle q', \mathrm{stay}\rangle$, and a rule $\langle q, ?\neg\,\mathtt{haspebble}_c, q'\rangle$ by all rules $\langle q, \sigma, j, \varnothing\rangle \to \langle q', \mathrm{stay}\rangle$ and all rules $\langle q, \sigma, j, \{c'\}\rangle \to \langle q', \mathrm{stay}\rangle$ with $c' \in C \setminus \{c\}$.

Finally we consider look-ahead. If $\langle q, ?\langle\mathcal{B}\rangle, q'\rangle$ is a rule of $\mathcal{A}$, and $\mathcal{B}'$ is an I-PTA$^{\mathrm{LA}}$ such that $[\![\mathcal{B}']\!]_t = [\![\mathcal{B}]\!]_t$ for every $t \in T_\Sigma$, then $\mathcal{A}'$ has all the rules $\langle q, \sigma, j, b, \mathcal{B}'\rangle \to \langle q', \mathrm{stay}\rangle$ that use $\mathcal{B}'$ as a look-ahead test. Similarly, the rule $\langle q, ?\neg\langle\mathcal{B}\rangle, q'\rangle$ is simulated by all the rules $\langle q, \sigma, j, b, \neg\mathcal{B}'\rangle \to \langle q', \mathrm{stay}\rangle$. □

Lemma 26. *For every* I-PTA $\mathcal{A}$ *there is a directive* I-PTA $\mathcal{A}'$ *such that* $T(\mathcal{A}') = T(\mathcal{A})$.

Proof. Let $\mathcal{A} = (\Sigma, Q, Q_0, F, C, \varnothing, C_{\mathrm{i}}, R, 0)$ be an I-PTA with $C_{\mathrm{i}} = C$. To simplify the proof we extend the syntax of the directive I-PTA by allowing rules $\langle q, \tau, q'\rangle$ with $\tau ::= \alpha_0 \mid ?\varphi_0 \mid ?\neg\varphi_0 \mid \tau/\tau'$, where τ' is an alias of τ. This clearly does not extend their power, because a rule $\langle q, \tau/\tau', q'\rangle$ can be replaced by the two rules $\langle q, \tau, p\rangle$ and $\langle p, \tau', q'\rangle$ where p is a new state. We now construct $\mathcal{A}' = (\Sigma, Q, Q_0, F, C, R')$ where R' is defined as follows. If $\mathcal{A}$ has a rule $\langle q, \sigma, j, b\rangle \to \langle q', \alpha\rangle$, then $\mathcal{A}'$ has the rule $\langle q, \tau, q'\rangle$ such that $\tau = \tau_\sigma/\tau_j/\tau_b/\alpha$ if $\alpha \neq \mathrm{stay}$, and $\tau = \tau_\sigma/\tau_j/\tau_b$ if $\alpha = \mathrm{stay}$, where $\tau_\sigma = ?\mathtt{haslabel}_\sigma$, $\tau_j = ?\mathtt{ischild}_j$, $\tau_{\{c\}} = ?\mathtt{haspebble}_c$, and $\tau_\varnothing = ?\neg\mathtt{haspebble}_{c_1}/\cdots/?\neg\mathtt{haspebble}_{c_n}$, if $C = \{c_1, \ldots, c_n\}$. □

As observed before, a directive I-PTA $\mathcal{A}$ can be viewed as a finite automaton of which each state transition is labeled by a directive. Thus, viewing the set $D_{\Sigma,C}$ as an alphabet, $\mathcal{A}$ accepts a string language $L_{\mathrm{str}}(\mathcal{A}) \subseteq D^*_{\Sigma,C}$. We now show the (rather obvious) fact that the semantics $[\![\mathcal{A}]\!]_t$ of $\mathcal{A}$ (for every tree t over Σ) depends only on the language $L_{\mathrm{str}}(\mathcal{A})$, cf. [12, Theorem 3.11] and [5, Lemma 3]. We do this (as in [12, Definition 2.7] and [5, Section 4]) by associating a semantics $[\![L]\!]_t$ with every language $L \subseteq D^*_{\Sigma,C}$. Intuitively, a string $w = \tau_1 \cdots \tau_n$ of directives can be viewed as the path expression $\tau_1/\cdots/\tau_n$ and a language $L = \{w_1, w_2, \ldots\}$ of such strings can be viewed as the (possibly infinite) path expression $w_1 \cup w_2 \cup \cdots$. Thus, for a tree t over Σ we formally define $[\![\varepsilon]\!]_t$ to be the identity on $\mathrm{Sit}(t)$, $[\![\tau_1 \cdots \tau_n]\!]_t = [\![\tau_1]\!]_t \circ \cdots \circ [\![\tau_n]\!]_t$, and $[\![L]\!]_t = \bigcup_{w \in L} [\![w]\!]_t$. The next lemma is a special case of [12, Theorem 3.11]. Its proof is entirely similar to the one of [5, Lemma 3].

Lemma 27. $[\![\mathcal{A}]\!]_t = [\![L_{\mathrm{str}}(\mathcal{A})]\!]_t$.

Proof. A string w of directives induces a state transition relation $R_{\mathcal{A}}(w) \subseteq Q \times Q$ as follows. For $\tau \in D_{\Sigma,C}$, $R_{\mathcal{A}}(\tau) = \{(q,q') \mid \langle q,\tau,q'\rangle \in R\}$. For the empty string, $R_{\mathcal{A}}(\varepsilon)$ is the identity on Q, and $R_{\mathcal{A}}(\tau_1 \cdots \tau_n) = R_{\mathcal{A}}(\tau_1) \circ \cdots \circ R_{\mathcal{A}}(\tau_n)$. Then $L_{\mathrm{str}}(\mathcal{A}) = \{w \in D^*_{\Sigma,C} \mid R_{\mathcal{A}}(w) \cap (Q_0 \times F) \neq \varnothing\}$.

It is straightforward to show by induction that, for all configurations $\langle q,u,\pi\rangle$ and $\langle q',u',\pi'\rangle$ and for every $w = \tau_1 \cdots \tau_n$ over $D_{\Sigma,C}$, there is a computation

$$\langle q_1,u_1,\pi_1\rangle \Rightarrow^{\tau_1}_{t,\mathcal{A}} \langle q_2,u_2,\pi_2\rangle \Rightarrow^{\tau_2}_{t,\mathcal{A}} \cdots \Rightarrow^{\tau_n}_{t,\mathcal{A}} \langle q_{n+1},u_{n+1},\pi_{n+1}\rangle$$

with $\langle q_1,u_1,\pi_1\rangle = \langle q,u,\pi\rangle$ and $\langle q_{n+1},u_{n+1},\pi_{n+1}\rangle = \langle q',u',\pi'\rangle$ if and only if $(\langle u,\pi\rangle,\langle u',\pi'\rangle) \in [\![w]\!]_t$ and $(q,q') \in R_{\mathcal{A}}(w)$. From this equivalence it follows that $[\![\mathcal{A}]\!]_t$ consists of all $(\langle u,\pi\rangle,\langle u',\pi'\rangle)$ such that

$$\exists q_0 \in Q_0, q_\infty \in F, w \in D^*_{\Sigma,C} : (\langle u,\pi\rangle,\langle u',\pi'\rangle) \in [\![w]\!]_t,\ (q,q') \in R_{\mathcal{A}}(w)$$

i.e., such that $\exists\, w \in L_{\mathrm{str}}(\mathcal{A}) : (\langle u,\pi\rangle,\langle u',\pi'\rangle) \in [\![w]\!]_t$, which means that it equals $[\![L_{\mathrm{str}}(\mathcal{A})]\!]_t$. □

Proof of Theorem 21. We assume the syntax for path expressions α of Pebble XPath and Pebble CAT to be in normal form. We also add $\alpha ::= \varnothing$ to the syntax, with $[\![\varnothing]\!]_t = \varnothing$ for every tree t. That is possible because, e.g., $[\![?\mathtt{ischild}_0/?\neg\mathtt{ischild}_0]\!]_t = \varnothing$.

We first show that Pebble CAT has the same power as MSO. Let us recall that the set $D_{\Sigma,C}$ of directives τ of the directive I-PTA was defined by the syntax $\tau ::= \alpha_0 \mid ?\varphi_0 \mid ?\neg\varphi_0$. Thus, the path expressions of Pebble CAT are, in fact, exactly the usual regular expressions over the "alphabet" $D_{\Sigma,C}$. Accordingly, we define for such a path expression α the string language $L_{\mathrm{str}}(\alpha) \subseteq D^*_{\Sigma,C}$ in the obvious way, interpreting the operators $\cup$, $/$, and * as union, concatenation, and Kleene star of string languages, respectively. The next lemma is the analogue of Lemma 27, with a straightforward proof.

Lemma 28. $[\![\alpha]\!]_t = [\![L_{\mathrm{str}}(\alpha)]\!]_t$.

Proof. It is easy to see, for string languages $L_1, L_2 \subseteq D^*_{\Sigma,C}$, that $[\![L_1 \cup L_2]\!]_t = [\![L_1]\!]_t \cup [\![L_2]\!]_t$, $[\![L_1L_2]\!]_t = [\![L_1]\!]_t \circ [\![L_2]\!]_t$, and $[\![L_1^*]\!]_t = [\![L_1]\!]_t^*$, cf. [12, Lemma 2.9]. Then the proof is by induction on the structure of α. □

By Kleene's classical theorem, a string language can be accepted by a finite automaton if and only if it can be defined by a regular expression. Thus, by Lemmas 27 and 28, a trip is definable in Pebble CAT if and only if it can be computed by a directive I-PTA, and hence, by Theorem 15 (for $k = 0$) and Lemmas 25 and 26, if and only if it is MSO definable.

It remains to show that if a trip is definable in Pebble XPath, then it can be computed by a directive I-PTA. We will prove below that for every Pebble XPath path expression α there is a directive I-PTA$^{\mathrm{LA}}$ $\mathcal{A}$, i.e., a directive I-PTA with iterated look-ahead tests, such that $[\![\mathcal{A}]\!]_t = [\![\alpha]\!]_t$ for every t. This implies that α and $\mathcal{A}$ define the same trip, and then we obtain from Theorem 20 (and Lemmas 25 and 26) a directive I-PTA (without look-ahead) computing that same trip.

Let n_α be the nesting depth of subexpressions of α of the form $\langle\beta\rangle$. The proof is by induction on n_α, and $\mathcal{A}$ will be of look-ahead depth n_α. If $n_\alpha = 0$, i.e., there are no such subexpressions at all, then α is a Pebble CAT expression, and we are done by the first part of the proof. Now suppose that the result holds for nesting depth n, and let $n_\alpha = n+1$. For every subexpression $\langle\beta\rangle$ of α that is not nested within another such subexpression, let $\mathcal{A}_\beta$ be a directive I-PTA$^{\mathrm{LA}}$ of look-ahead depth n (or less) such that $[\![\mathcal{A}_\beta]\!]_t = [\![\beta]\!]_t$ for all t. We now define the extended "alphabet" $D^n_{\Sigma,C}$ to consist of all path expressions τ with the syntax $\tau ::= \alpha_0 \mid ?\varphi_0 \mid ?\neg\varphi_0 \mid ?\langle\beta\rangle \mid ?\neg\langle\beta\rangle$ where $\langle\beta\rangle$ ranges over the above subexpressions of α. Then α can be viewed as a regular expression over the alphabet $D^n_{\Sigma,C}$, and it should be clear that Lemma 28 is also valid in this case. Also, using $D^n_{\Sigma,C}$ instead of $D_{\Sigma,C}$ in the rules of the directive I-PTA, and identifying each "symbol" $?\langle\beta\rangle$ with the "symbol" $?\langle\mathcal{A}_\beta\rangle$ (and similarly for the negated tests), we obtain a subclass of the directive I-PTA$^{\mathrm{LA}}$ of look-ahead depth $n+1$, because the semantics of the path expression $?\langle\beta\rangle$ is exactly the same as the meaning of the look-ahead test $?\langle\mathcal{A}_\beta\rangle$. Again, it should be clear that Lemma 27 is also valid for these directive I-PTA's, which are finite automata over $D^n_{\Sigma,C}$. Hence, by the same Kleene argument as in the first part of the proof, there is a directive I-PTA$^{\mathrm{LA}}$ $\mathcal{A}$ of look-ahead depth $n+1$ such that $[\![\mathcal{A}]\!]_t = [\![\alpha]\!]_t$ for every tree t.

This ends the proof of Theorem 21, both for ranked trees and (by Lemmas 22, 23, and 24) for unranked forests.

Two remarks. (1) Although the MSO definable trips are, of course, closed under complement and intersection, we do not know whether the XPath 2.0 operations `intersect` and `except` can be added to the syntax of path expressions of Pebble XPath ($\alpha ::= \alpha \cap \beta \mid \alpha \setminus \beta$). That is because it is not clear whether for every I-PTA $\mathcal{A}$ there is an I-PTA $\mathcal{B}$ such that $[\![\mathcal{B}]\!]_t = \mathrm{Sit}(t) - [\![\mathcal{A}]\!]_t$ for every tree t.

(2) The language Pebble XPath meets the requirements as listed in [30]. It is *simple*, defined in an algebraic language using simple operators: in particular we believe that pebbles form a user friendly concept. It is *understandable*, as its expressive power can be characterized in terms of automata. It is *useful* in the sense that the query evaluation problem 'given path expression α and two nodes u, v in forest f, is $(f,u,v) \in T(\alpha)$?' is tractable. At least, the latter property holds for Pebble CAT, as α can be transformed into an I-PTA in polynomial time, and the problem '$(f,u,v) \in T(\alpha)$?' can then be translated

into the emptiness problem for push-down automata. For Pebble XPath the query evaluation problem is tractable for every fixed path expression α. This is explained in more detail in the next two paragraphs.

Query evaluation. For a directive I-PTA $\mathcal{A} = (\Sigma, Q, Q_0, C, R)$, the binary node relation T computed by $\mathcal{A}$ on an input tree t can be evaluated in polynomial time as follows. It is straightforward to construct from $\mathcal{A}$ and t an ordinary pushdown automaton $\mathcal{P}$ with state set $Q \times N(t)$ and pushdown alphabet $N(t) \times C$ in such a way that $\mathcal{P}$ (with the empty string as input) has the same computation steps as $\mathcal{A}$ on t. Note that the configurations of $\mathcal{P}$ are exactly the configurations $\langle q, u, \pi \rangle$ of $\mathcal{A}$ on t. Dropping and lifting a pebble corresponds to pushing and popping a pushdown symbol. Moving around in t corresponds to a change of state. To decide whether $(t, u, v) \in T$, with $u, v \in N(t)$, decide whether $\mathcal{P}$ has a computation from configuration $\langle q_0, u, \varepsilon \rangle$ (for some $q_0 \in Q_0$) to some final configuration $\langle q, v, \pi \rangle$. Clearly, $\mathcal{P}$ can be constructed in polynomial time from $\mathcal{A}$ and t, and the existence of such a computation can be verified in polynomial time.

By Lemma 22, path expressions on forests can be translated into path expressions on ranked trees in polynomial time. Since for a Pebble CAT path expression on ranked trees the corresponding directive I-PTA can be constructed in polynomial time, using Kleene's construction, Pebble CAT path expressions can be evaluated in polynomial time. This does not seem to hold for Pebble XPath, as the construction in the proof of Theorem 19 (which implements a look-ahead test by calling an I-PTA $\mathcal{B}$) is at least 2-fold exponential (because determining the domain of the related I-PTA $\mathcal{B}'$ takes 2-fold exponential time by Theorem 8). However, the data complexity of the problem is of course polynomial, i.e., for a fixed path expression α we obtain a fixed directive I-PTA $\mathcal{A}$ for which the binary node relation can be evaluated in polynomial time.

10. Pattern matching

One of the basic tree transformations in the context of XML is pattern matching. The transducer must find all sequences of nodes satisfying a certain description and generate the subtrees rooted at these nodes, for each match. More precisely, we consider queries of the form

`for` $\mathcal{X}$ `where` φ `return` r

in which $\mathcal{X}$ is a finite set of node variables, φ is an MSO formula with its free variables in $\mathcal{X}$, and r is a tree of which the leaves may be labeled with the variables in $\mathcal{X}$. In what follows we assume that $\mathcal{X}$ and r are fixed. Let $\mathcal{X} = \{x_1, \ldots, x_n\}$, where $x_1, \ldots, x_n$ is an arbitrary order of the elements of $\mathcal{X}$. The transducer must find all sequences of nodes $u_1, \ldots, u_n$ of the input tree t that match the pattern defined by $\varphi(x_1, \ldots, x_n)$, i.e., such that $t \models \varphi(u_1, \ldots, u_n)$, and for each match it must generate the output tree r in which each occurrence of the variable x_i is replaced by the subtree of t with root u_i. Usually the variables in $\mathcal{X}$ are indeed specified in a specific order $\lambda = (x_1, \ldots, x_n)$, and it is required that the transducer finds (and generates) the matches in the lexicographic document order induced by λ. We will, however, also consider the case where this requirement is dropped, and the most efficient order λ can be selected.

For convenience we assume that r is of the form $\mu(x_1, \ldots, x_n)$ for some symbol μ of rank n, and so the output for each match is $\mu(t|_{u_1}, \ldots, t|_{u_n})$ where $t|_u$ is the subtree of t with root u. For convenience we also assume that the input tree t is ranked. Moreover, we assume that the output alphabet is also ranked and contains the binary symbol @ that allows us to list the various output trees $\mu(t|_{u_1}, \ldots, t|_{u_n})$, and the nullary symbol e to indicate the end of the list of output trees (similar to the binary tag `<result>` and the nullary tag `<endofresults>` of Example 2). In Section 11 we will consider pattern matching in forests.

We now describe a total deterministic PTT $\mathcal{A}$ that executes the above query. In order to find all n-tuples of nodes matching the n-ary pattern defined by the MSO formula $\varphi(x_1, \ldots, x_n)$, and generate the corresponding output, the PTT $\mathcal{A}$ systematically enumerates *all* n-tuples of nodes of the input tree t. To do this, $\mathcal{A}$ uses visible pebbles $c_1, \ldots, c_n$ on the stack, representing the variables $x_1, \ldots, x_n$, respectively.[20] It drops them in this order and moves each of them through the input tree t in document order (i.e., in pre-order), in a nested fashion. Inductively speaking, $\mathcal{A}$ moves pebble c_1 in pre-order through t (alternately dropping and lifting c_1), and for each position u_1 of c_1 it uses pebbles $c_2, \ldots, c_n$ to enumerate all possible $(n-1)$-tuples $u_2, \ldots, u_n$ of nodes of t. For each enumerated n-tuple $u_1, \ldots, u_n$, with pebble c_i at position u_i, $\mathcal{A}$ performs the test φ, using an MSO test on the visible configuration (Lemma 13), and, in case of success, spawns a process that outputs the corresponding n-tuple of subtrees.

More precisely, if the ranked input alphabet is Σ, then φ is an MSO formula over Σ, and $\mathcal{A}$ has the ranked output alphabet $\Delta = \Sigma \cup \{\mu, @, e\}$ where μ has rank n, and @ and e have rank 2 and 0 respectively. For input tree t, the output tree s is of the form $s = @(r_1, @(r_2, \ldots @(r_k, e) \cdots))$ where each r_i corresponds to a match, i.e., there is a sequence of nodes $u_1, \ldots, u_n$ of t such that $t \models \varphi(u_1, \ldots, u_n)$ and $r_i = \mu(t|_{u_1}, \ldots, t|_{u_n})$. Moreover, the sequence $r_1, \ldots, r_k$ corresponds to the sequence of all matches, in lexicographic document order. As explained above, the visible colour set of the PTT $\mathcal{A}$ is $C_v = \{c_1, \ldots, c_n\}$, and $\mathcal{A}$ generates s by enumerating all sequences $u_1, \ldots, u_n$ of nodes of t using pebbles $c_1, \ldots, c_n$. To find out whether this sequence is a match, $\mathcal{A}$ performs the MSO test $\psi(x)$ on the visible configuration, defined by

[20] It is not necessary that all pebbles are visible, as we will discuss below, but it simplifies the description of $\mathcal{A}$.

$$\psi(x) \equiv \forall x_1, \ldots, x_n((\mathrm{peb}_{c_1}(x_1) \wedge \cdots \wedge \mathrm{peb}_{c_n}(x_n)) \to \varphi'(x_1, \ldots, x_n))$$

where $\mathrm{peb}_c(x)$ is the disjunction of all $\mathrm{lab}_{(\sigma,b)}(x)$ such that $c \in b$, and where φ' is obtained from φ by changing every atomic subformula $\mathrm{lab}_\sigma(y)$ into the disjunction of all $\mathrm{lab}_{(\sigma,b)}(y)$. Note that $\psi(x)$ is an MSO formula over $\Sigma \times 2^C$, where C is the colour set of $\mathcal{A}$. Note also that the variable x (for the head position) does not, and need not, occur in $\psi(x)$. If the sequence $u_1, \ldots, u_n$ is not a match, then $\mathcal{A}$ continues the enumeration of n-tuples. If the sequence *is* a match, then $\mathcal{A}$ outputs the symbol @ and branches into two subprocesses (as in the 5-th rule of Example 2). In the second (main) branch it continues the enumeration of n-tuples. In the first branch it outputs the symbol μ and branches into n subprocesses, where the i-th process searches for visible pebble c_i and then outputs $t|_{u_i}$. Note that, in this first branch, $\mathcal{A}$ could easily output an arbitrary tree r in which every occurrence of the variable x_i is replaced by $t|_{u_i}$. This ends the description of $\mathcal{A}$.

As the complexity of typechecking the transducer $\mathcal{A}$ depends critically on the number of visible pebbles used (see Theorem 8), we wish to minimize that number and use as few visible pebbles as possible for matching. It should be clear that, instead of using n visible pebbles, $\mathcal{A}$ can also use $n-2$ visible pebbles $c_1, \ldots, c_{n-2}$, one invisible pebble c_{n-1} on top (which is therefore always observable), and the head instead of the last pebble c_n. Then $\mathcal{A}$ can perform the MSO test $\chi(x)$ on the observable configuration, defined by $\chi(x) \equiv$

$$\forall x_1, \ldots, x_{n-1}((\mathrm{peb}_{c_1}(x_1) \wedge \cdots \wedge \mathrm{peb}_{c_{n-1}}(x_{n-1})) \to \varphi'(x_1, \ldots, x_{n-1}, x))$$

where x_n is renamed into x in φ'. Thus, from Theorem 16 we obtain the following result on the matching of arbitrary MSO definable patterns.

Theorem 29. *For $n \geq 2$, every* MSO *definable n-ary pattern can be matched by a total deterministic* V_{n-2}I-PTT. *Moreover, and in particular, every* MSO *definable unary or binary pattern can be matched by a total deterministic* I-PTT.

To further reduce the number of visible pebbles, we consider the more specific case of queries of the form

$$\texttt{for } \mathcal{X} \texttt{ where } \beta(\varphi_1, \ldots, \varphi_m) \texttt{ return } r$$

in which $\beta(\varphi_1, \ldots, \varphi_m)$ is a boolean combination (using $\wedge, \vee, \neg$) of the MSO formulas $\varphi_1, \ldots, \varphi_m$, $m \geq 2$, and each φ_ℓ, $\ell \in [1, m]$, has its free variables in $\mathcal{X}$. We will make use of the fact that not all variables in $\mathcal{X}$ need actually occur in each formula φ_ℓ. As discussed in the Introduction, the `for ⋯ where` construct in XQuery often induces patterns $\varphi_1 \wedge \cdots \wedge \varphi_m$ such that each φ_ℓ contains just two free variables, cf. [32].

Consider an arbitrary query as displayed above. Let $G_\varphi = (V_\varphi, E_\varphi)$ be the undirected graph induced by the pattern $\varphi \equiv \beta(\varphi_1, \ldots, \varphi_m)$, by which we mean that the set V_φ of vertices of G_φ consists of the free variables of φ, i.e., $V_\varphi = \mathcal{X}$, and that the set E_φ of edges of G_φ consists of the unordered pairs $\{x, y\}$ (with $x, y \in V_\varphi$, $x \neq y$) for which there exists $\ell \in [1, m]$ such that both x and y occur (free) in φ_ℓ. Note that G_φ does not depend on any order of the variables in $\mathcal{X}$. Note also that for every finite undirected graph G there exists $\varphi \equiv \varphi_1 \wedge \cdots \wedge \varphi_m$ such that G is isomorphic to G_φ.

Pattern matching φ, and executing the above query, can be done by a total deterministic PTT $\mathcal{A}$ as follows, similarly to the general PTT $\mathcal{A}$ above (as discussed before Theorem 29). Again, let $\lambda = (x_1, \ldots, x_n)$ be an arbitrary order of the variables in $\mathcal{X}$. Pebbles with distinct colours $c_1, \ldots, c_{n-1}$ are used to represent $x_1, \ldots, x_{n-1}$, dropping them in that order. For every $j \in [1, n]$, when pebbles $c_1, \ldots, c_{j-1}$ are dropped on the tree and the head is at a candidate position u_j for the variable x_j, all MSO tests φ_ℓ are performed of which the free variables are in $\{x_1, \ldots, x_j\}$ (and that have not been tested before). Thus, when $\mathcal{A}$ has enumerated a sequence $u_1, \ldots, u_n$, it can compute the boolean value of $\varphi(u_1, \ldots, u_n)$. For each match $u_1, \ldots, u_n$ the tree r is generated, such that for every occurrence of the variable x_i in r the subtree rooted at u_i is generated, by a separate process; that is straightforward, even when c_i is invisible: lift pebbles $c_{n-1}, \ldots, c_{i+1}$ one by one (in that order), and then access c_i and output $t|_{u_i}$. Note that, as before, the matches are generated in the lexicographic document order induced by the order λ.

It remains to determine which are the visible and invisible pebbles, keeping in mind that we wish to use as many invisible pebbles as possible for matching. To do the MSO tests at position u_j all pebbles c_i for which $\{x_i, x_j\} \in E_\varphi$ and $i < j$ should be observable. Hence all such pebbles under the topmost pebble c_{j-1} must be visible. These are the pebbles corresponding to the set

$$\mathrm{vis}(\lambda) = \{x_i \mid \text{there exists } \{x_i, x_j\} \in E_\varphi \text{ such that } i + 1 < j\}.$$

Thus, for $\mathcal{A}$ we define $C_\mathrm{v} = \{c_i \mid x_i \in \mathrm{vis}(\lambda)\}$ and $C_\mathrm{i} = \{c_i \mid x_i \notin \mathrm{vis}(\lambda)\}$. Note that $c_{n-1} \in C_\mathrm{i}$.

In the case where the order $\lambda = (x_1, \ldots, x_n)$ of the variables is irrelevant, we may want to determine an optimal order. A finite undirected graph $G = (V, E)$ will be called a *union of paths* if it is acyclic and has only vertices of degree at most 2. Intuitively this means that each connected component of G is a path. Thus, clearly, there is an order $v_1, \ldots, v_p$ of the vertices of G such that for all $i, j \in [1, p]$ with $i < j$, if $\{v_i, v_j\} \in E$ then $i + 1 = j$ (repeatedly pick a vertex of degree 0 or 1, and remove it from the graph together with all its incident edges). We will call this an *invisible order* of the vertices of G. Note that a graph is a union of paths if and only if it has an invisible order. Note also that every subgraph of G is also a union of paths.

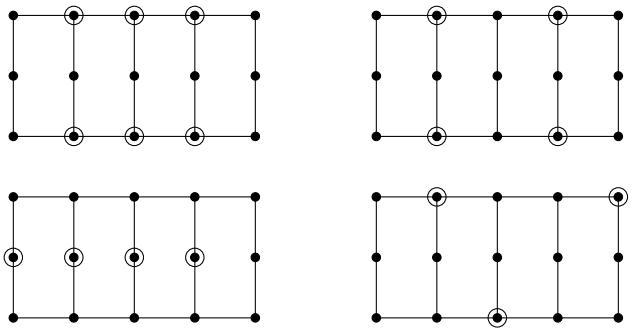

Fig. 5. Visible sets of different sizes.

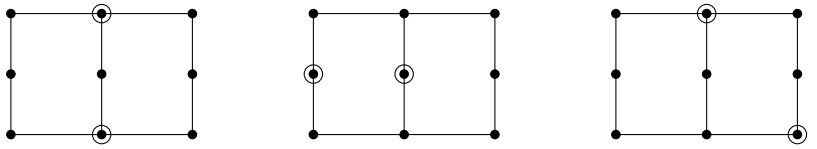

Fig. 6. Three visible sets of minimal size.

For an arbitrary finite undirected graph $G = (V, E)$, let us now say that a set $W \subseteq V$ of vertices of G is a *visible set* of G if the subgraph of G induced by $V \setminus W$, denoted by $G[V \setminus W]$, is a union of paths. By the last sentence of the previous paragraph, every superset of a visible set is also a visible set.

Lemma 30. *A set of variables $W \subseteq V_\varphi$ is a visible set of G_φ if and only if there is an order λ of V_φ such that* $\mathrm{vis}(\lambda) \subseteq W$.

Proof. (If) It is easy to verify that every $\mathrm{vis}(\lambda)$ is a visible set of G_φ. In fact, for all $i < j$, if $x_i, x_j \notin \mathrm{vis}(\lambda)$ and $\{x_i, x_j\} \in E_\varphi$, then $i + 1 = j$.

(Only if) Define the order λ on V_φ as follows. First list the vertices of W in any order. Then list the remaining vertices according to an invisible order of the vertices of $G_\varphi[V_\varphi \setminus W]$. Obviously $\mathrm{vis}(\lambda) \subseteq W$. □

Theorem 31. *Pattern $\varphi \equiv \beta(\varphi_1, \ldots, \varphi_m)$ can be matched by a total deterministic* v_k-PTT *where $k = \#(W)$ for a visible set W of G_φ. In particular, if G_φ is a union of paths, then φ can be matched by a total deterministic* 1-PTT.

Proof. By Lemma 30 there is an order λ of V_φ such that $\mathrm{vis}(\lambda) \subseteq W$. Hence at most $\#(W)$ visible pebbles suffice. If G_φ is a union of paths, then $W = \varnothing$ is a visible set. □

Lemma 30 shows that finding an order λ for which $\mathrm{vis}(\lambda)$ is of minimal size, is the same as finding a visible set W of minimal size. Unfortunately, this is an NP-complete problem. More precisely, the problem whether for a given graph $G = (V, E)$ and a given number k there is a set of vertices $V' \subseteq V$ with $\#(V') \geq k$ such that $G[V']$ is a union of paths, is NP-complete (see Problem GT21 of [28]).

We now give some examples of visible sets of a graph G. It suffices to take as visible vertices those of degree ≥ 3 in G (plus one vertex in each connected component that is a cycle). But often one can choose a smaller set.

Example 32. If G is a cycle or a star, then it has a visible set W with $\#(W) = 1$ (for a cycle any singleton is a visible set, and for a star the visible set W consists of the centre vertex).

In Figs. 5 and 6 we show graphs with the vertices of a visible set W encircled. For the graph G in Fig. 5, the upper left W consists of all vertices of degree 3. It is not minimal, in the sense that it has a proper subset that is also a visible set, as shown at the upper right. This one *is* minimal, because dropping one of the vertices from W produces a vertex of degree 3 in the complement. Another minimal visible set (of the same size) is shown at the lower left: dropping the leftmost vertex of W produces a cycle, and dropping one of the other vertices produces two vertices of degree 3. Finally, a visible set of size 3 is shown at the lower right. It is of minimal size, i.e., $\#(W) \geq 3$ for every visible set W of G. In fact, removing a vertex of degree 2 from G leaves a graph with two disjoint cycles that both must be broken, whereas removing a vertex of degree 3 from G either leaves a graph with two disjoint cycles or a graph with a cycle and a vertex of degree 3 of which the neighbourhood is disjoint with that cycle. Thus, any pattern φ such that G_φ is isomorphic to G can be matched with three visible pebbles.

Visible sets of minimal size need not be unique. For the graph in Fig. 6, three different visible sets of minimal size are shown. □

If we allow matches to occur more than once in the output, then Theorem 31 is not optimal (still assuming that the order λ is irrelevant). Using the boolean laws, the MSO formula $\varphi \equiv \beta(\varphi_1, \ldots, \varphi_m)$ can be written as a disjunction $\varphi \equiv \psi_1 \vee \cdots \vee \psi_k$ where each ψ_i is a conjunction of some of the formulas $\varphi_1, \ldots, \varphi_m$ or their negations. Now the PTT $\mathcal{A}$ can execute the queries '`for` $\mathcal{X}$ `where` ψ_i `return` r' consecutively for $i = 1, \ldots, k$. Obviously, G_{ψ_i} is a subgraph of G_φ for every $i \in [1, k]$. Hence every visible set of G_φ is also a visible set of G_{ψ_i}, and so the minimal size of the visible sets of G_{ψ_i} is at most the minimal size of the visible sets of G_φ. Thus, pattern matching formulas $\psi_1, \ldots, \psi_k$ consecutively needs at most as many visible pebbles as pattern matching φ, but it may need less. As a simple example, let $\varphi \equiv \varphi_1(x, y) \wedge (\varphi_2(y, z) \vee \varphi_3(x, z))$. Then G_φ is a triangle, which needs one visible pebble. But $\varphi \equiv \psi_1 \vee \psi_2$ where $\psi_1 \equiv \varphi_1(x, y) \wedge \varphi_2(y, z)$ and $\psi_2 \equiv \varphi_1(x, y) \wedge \varphi_3(x, z)$. Both G_{ψ_1} and G_{ψ_2} are (unions of) paths, which do not need visible pebbles. Thus, φ can be matched by an I-PTT. However, all matches for which $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ holds occur twice in the output.

We finally discuss another way to reduce the number of visible pebbles. Suppose that, for some $i \in [1, m]$, the formula φ_i has exactly two free variables $x, y \in \mathcal{X}$. Thus, the edge $\{x, y\}$ is in E_φ. Suppose moreover that the trip defined by $\varphi_i(x, y)$ is functional. Suppose finally that W is a visible set of G_φ with $x, y \in W$. Then all other edges of G_φ incident with y can be redirected to x, and y can be dropped from W. To be precise, every formula φ_j that contains the free variable y can be changed into the formula $\forall y(\varphi_i(x, y) \to \varphi_j)$ that contains the free variable x instead of y. The resulting query is obviously equivalent to the given one.

11. Pebble forest transducers

The PTT transforms ranked trees, whereas XML documents are unranked forests. However, it is not difficult to use, or slightly adapt, the PTT for the transformation of forests. The most obvious, and well-known way to do this, is to encode the forests as binary trees. Let enc' be the class of all encodings enc′ (one encoding for each input alphabet Σ), and let dec be the class of all decodings dec (one decoding for each output alphabet Δ). Then we can view the class $\mathsf{enc}' \circ \mathsf{V}_k\text{I-PTT} \circ \mathsf{dec}$ as the class of forest transductions realized by V_kI-PTT's. For the input forest f this is a natural definition, because it is quite easy to visualize a PTT walking on $\mathrm{enc}'(f)$ as actually walking on f itself. For the output forest g this is also a natural definition, as it is, in fact, easy to transform a PTT that outputs $\mathrm{enc}(g)$ into a (slightly adapted type of) PTT that directly outputs g itself: change every output rule $\langle q, \sigma, j, b\rangle \to \delta(\langle q_1, \mathrm{stay}\rangle, \langle q_2, \mathrm{stay}\rangle)$ into $\langle q, \sigma, j, b\rangle \to \delta(\langle q_1, \mathrm{stay}\rangle)\langle q_2, \mathrm{stay}\rangle$, and every output rule $\langle q, \sigma, j, b\rangle \to e$ into $\langle q, \sigma, j, b\rangle \to \varepsilon$. The definition is also natural with respect to typechecking, because a forest language L is regular if and only if the tree language $\mathrm{enc}(L)$ is regular, and similarly for $\mathrm{enc}'(L)$. Since the transformation of the involved grammars can obviously be done in polynomial time, Theorem 8 in Section 5 also holds for V_kI-PTT as forest transducers.

We observe here that the class $\mathsf{enc}' \circ \mathsf{V}_k\text{I-PTT} \circ \mathsf{dec}$ does not depend on the chosen encodings and decodings, i.e., enc' can be replaced by the class enc of all encodings enc, and dec by the class dec' of all decodings dec′. In fact, a PTT that walks on $\mathrm{enc}'(f)$ can easily be simulated by one that walks on $\mathrm{enc}(f)$: the original label σ^{kl} can be determined by inspecting the children of the node with label σ. Vice versa, a PTT that walks on $\mathrm{enc}(f)$ can be simulated by one that walks on $\mathrm{enc}'(f)$: a node with label, e.g., σ^{01} represents the original node and its first child with label e; the difference between these nodes can be stored in the finite state and in the pebble colours of the simulating PTT. Moreover, a PTT that outputs $\mathrm{enc}'(g)$ can easily be simulated by one that outputs $\mathrm{enc}(g)$: change, e.g., the rule $\langle q, \sigma, j, b\rangle \to \delta^{01}(\langle q', \mathrm{stay}\rangle)$ into the two rules $\langle q, \sigma, j, b\rangle \to \delta(\langle p, \mathrm{stay}\rangle, \langle q', \mathrm{stay}\rangle)$ and $\langle p, \sigma, j, b\rangle \to e$ where p is a new state. Vice versa, a PTT $\mathcal{A}$ that outputs $\mathrm{enc}(g)$ can be simulated by a PTT $\mathcal{A}$ that outputs $\mathrm{enc}'(g)$, but that requires look-ahead (Theorem 19), as follows. If $\mathcal{A}$ has an output rule $\langle q, \sigma, j, b\rangle \to \delta(\langle q_1, \mathrm{stay}\rangle, \langle q_2, \mathrm{stay}\rangle)$, then $\mathcal{A}'$ has the rule $\langle q, \sigma, j, b, \mathcal{B}_{01}\rangle \to \delta^{01}(\langle q_2, \mathrm{stay}\rangle)$ where $\mathcal{B}_{01}$ is a look-ahead test that finds out whether $\mathcal{A}$ can generate e when started in state q_1 in the current situation. To be precise, $\mathcal{B}_{01}$ is obtained from $\mathcal{A}$ by changing its set of initial states into $\{q_1\}$ and removing all output rules that do not output e. And of course, $\mathcal{A}'$ has similar rules for the other symbols δ^{ij}.

So far so good, in particular for the input forest f. There is, however, another natural possibility for the output forest g, as introduced and investigated in [47] for macro tree transducers. It is quite natural to allow a PTT that directly outputs g, as discussed above, to not only have output rules with right-hand sides $\delta(\langle q_1, \mathrm{stay}\rangle)\langle q_2, \mathrm{stay}\rangle$ and ε, but also right-hand sides $\langle q_1, \mathrm{stay}\rangle\langle q_2, \mathrm{stay}\rangle$ and $\delta(\langle q', \mathrm{stay}\rangle)$ that realize the concatenation of forests and the formation of a tree out of a forest.

Accordingly we define a *tree-walking forest transducer with nested pebbles* (abbreviated PFT) to be the same as a PTT $\mathcal{M}$, except that its output alphabet is unranked, and its output rules are of the form $\langle q, \sigma, b, j\rangle \to \zeta$ with $\zeta = \delta(\langle q', \mathrm{stay}\rangle)$ introducing a new node with label δ and generating a forest from state q', or $\zeta = \langle q_1, \mathrm{stay}\rangle\langle q_2, \mathrm{stay}\rangle$ concatenating two forests, or $\zeta = \varepsilon$ generating the empty forest. Note that a right-hand side $\delta(\langle q_1, \mathrm{stay}\rangle)\langle q_2, \mathrm{stay}\rangle$ is also allowed, as it can easily be simulated in two steps.

Formally, an output form of the PFT $\mathcal{M}$ on an input tree t is defined to be a forest in $F_\Delta(\mathrm{Con}(t))$. Let s be an output form and let v be a leaf of s with label $\langle q, u, \pi\rangle \in \mathrm{Con}(t)$. If the rule $\langle q, \sigma, b, j\rangle \to \zeta$ is relevant to $\langle q, u, \pi\rangle$ then we write $s \Rightarrow_{t,\mathcal{M}} s'$ where s' is obtained from s as follows. If the rule is not an output rule, then the label of v is changed in the same way as for the PTA and PTT. If $\zeta = \delta(\langle q', \mathrm{stay}\rangle)$ then node v is replaced by the subtree $\delta(\langle q', u, \pi\rangle)$. If $\zeta = \langle q_1, \mathrm{stay}\rangle\langle q_2, \mathrm{stay}\rangle$ then node v is replaced by the two-node forest $\langle q_1, u, \pi\rangle\langle q_2, u, \pi\rangle$. And if $\zeta = \varepsilon$ then the node v is removed from s. The transduction realized by $\mathcal{M}$ consists of all $(t, s) \in T_\Sigma \times F_\Delta$ such that $\langle q_0, \mathrm{root}_t\rangle \Rightarrow^*_{t,\mathcal{M}} s$ for some $q_0 \in Q_0$. Thus, we have defined the PFT as a transformer of ranked trees into unranked forests. The corresponding classes of transductions are denoted by V_kI-PFT. For forest transformations one can of course consider the classes $\mathsf{enc}' \circ \mathsf{V}_k\text{I-PFT}$.

Lemma 33. *For every $k \geq 0$,*

$$(1)\ \mathsf{V}_k\text{I-PTT} \circ \mathsf{dec} \subseteq \mathsf{V}_k\text{I-PFT} \quad \textit{and} \quad (2)\ \mathsf{V}_k\text{I-PFT} \circ \mathsf{enc} \subseteq \mathsf{V}_k\text{I-PTT} \circ \text{I-dPTT}$$

and similarly for the deterministic case.

Proof. Inclusion (1) is obvious from the discussion above: change every rule $\langle q, \sigma, j, b\rangle \to \delta(\langle q_1, \text{stay}\rangle, \langle q_2, \text{stay}\rangle)$ into $\langle q, \sigma, j, b\rangle \to \delta(\langle q_1, \text{stay}\rangle)\langle q_2, \text{stay}\rangle$, and every rule $\langle q, \sigma, j, b\rangle \to e$ into $\langle q, \sigma, j, b\rangle \to \varepsilon$.

The proof of inclusion (2) is similar to the proof in [47] that every macro forest transducer can be simulated by two macro tree transducers. Let $\mathcal{M}$ be a V_kI-PFT with (unranked) output alphabet Δ. Let Δ_1 be the ranked alphabet $\Delta \cup \{@^{(2)}, e^{(0)}\}$, where every element of Δ has rank 1. We now obtain the V_kI-PTT $\mathcal{M}'$ from $\mathcal{M}$ by changing every output rule $\langle q, \sigma, b, j\rangle \to \langle q_1, \text{stay}\rangle\langle q_2, \text{stay}\rangle$ into $\langle q, \sigma, b, j\rangle \to @(\langle q_1, \text{stay}\rangle, \langle q_2, \text{stay}\rangle)$ and every output rule $\langle q, \sigma, b, j\rangle \to \varepsilon$ into $\langle q, \sigma, b, j\rangle \to e$. Let 'flat' be the mapping from T_{Δ_1} to F_Δ defined by $\text{flat}(@(t_1, t_2) = \text{flat}(t_1)\text{flat}(t_2)$, $\text{flat}(\delta(t)) = \delta(\text{flat}(t))$ and $\text{flat}(e) = \varepsilon$. Then obviously $\tau_{\mathcal{M}} = \tau_{\mathcal{M}'} \circ \text{flat}$. Thus, it remains to show that the mapping flat $\circ$ enc is in I-dPTT. We will prove this after Theorem 37. It is, in fact, not hard to see that flat $\circ$ enc is even in dTT. □

Typechecking. The inverse type inference problem and the typechecking problem are defined for PFT's as in Section 5, except that G_{out} is a regular forest grammar rather than a regular tree grammar. It follows from Lemma 33(2), together with Lemma 4, Theorem 5, and Propositions 6 and 7 that these problems can be solved for V_kI-PFT's in $(k+4)$-fold and $(k+5)$-fold exponential time. However, it is shown in [14, Section 7] that they can be solved for V_k-PFT's in the same time as for V_k-PTT's, i.e., in $(k+1)$-fold and $(k+2)$-fold exponential time, respectively. This is due to the fact (shown in [14, Lemma 4]) that inverse type inference for the mapping flat $\circ$ enc can be solved in polynomial time, cf. the proof of Lemma 33. For exactly the same reason a similar result holds for V_kI-PFT's. In other words, Theorem 8 also holds for V_kI-PFT's.

Theorem 34. *For fixed $k \geq 0$, the inverse type inference problem and the typechecking problem are solvable for V_kI-PFT's in $(k+2)$-fold and $(k+3)$-fold exponential time, respectively.*

MSO tests. It should be clear that Theorem 16 also holds for the PFT, as MSO tests only concern the input tree.

Pattern matching. Pattern matching for forests can be defined in exactly the same way as we did for trees in Section 10. Since, obviously, Lemma 24 also holds for arbitrary n-ary patterns instead of trips, we may however assume that the input forest f over Σ of the query

$$\texttt{for } \mathcal{X} \texttt{ where } \varphi \texttt{ return } r$$

is encoded as a binary tree $t = \text{enc}'(f)$ over Σ' for which we execute the query

$$\texttt{for } \mathcal{X} \texttt{ where } \varphi' \texttt{ return } r$$

where φ' is the encoding of the formula φ according to Lemma 24. Consequently, we can use a PFT to execute this query and produce for each match of $\varphi'(x_1, \ldots, x_n)$ the required output r. We may now assume that r is a forest rather than a tree, and we may for simplicity assume that r is of the form $\mu(x_1 \cdots x_n)$ for some output symbol μ. Thus, the output for each match $\varphi'(u_1, \ldots, u_n)$ is $\mu(f|_{u_1} \cdots f|_{u_n})$, and the output forest is of the form $s = r_1 r_2 \cdots r_k e$ where $r_1, \ldots, r_k$ are the outputs corresponding to all the matches. Note that e is another output symbol, and so $\Delta = \Sigma \cup \{\mu, e\}$. It should be clear how the total deterministic PTT $\mathcal{A}$ in Section 10 can be changed into a total deterministic PFT that executes this query. The only small problem is that $\mathcal{A}$ outputs the encoded subtrees $t|_{u_i}$ rather than the required subtrees $f|_{u_i}$. However, a PFT can easily transform an encoded forest $\text{enc}'(f|_u)$ into the forest $f|_u$, using rules $\langle q, \sigma^{11}, j, b\rangle \to \sigma(\langle q, \text{down}_1\rangle)\langle q, \text{down}_2\rangle$, $\langle q, \sigma^{01}, j, b\rangle \to \sigma\langle q, \text{down}_1\rangle$, $\langle q, \sigma^{10}, j, b\rangle \to \sigma(\langle q, \text{down}_1\rangle)$, and $\langle q, \sigma^{00}, j, b\rangle \to \sigma$.

From this it should be clear that Theorems 29 and 31 also hold for forest pattern matching and PFT.

Expressive power. As in [47], the PFT is more powerful than the PTT. In particular, the I-PFT is more powerful than the I-PTT that generates encoded forests, i.e., I-PTT $\circ$ dec is a proper subclass of I-PFT. In fact, it is well known (cf. [20, Lemma 7] and [26, Lemma 5.40]), and easy to see, that the height of the output tree of a functional TT $\mathcal{M}$ (which means that $\tau_{\mathcal{M}}$ is a function) is linearly bounded by the size of the input tree: otherwise $\mathcal{M}$ would be in a loop and would generate infinitely many output trees for that input tree. Since I-PTT $\subseteq$ TT $\circ$ TT by Lemma 4, this implies that for a functional I-PTT the height of the output tree is exponentially bounded by the size of the input tree. However, the following total deterministic I-PFT $\mathcal{M}_{2\text{exp}}$ outputs, for an input tree of size n, a forest of length double exponential in n. Since the height of the encoded output forest is at least the length of that forest, this transformation cannot be realized by an I-PTT that generates encoded forests. The transducer $\mathcal{M}_{2\text{exp}}$ is similar to the I-PTT $\mathcal{M}_{\text{sib}}$ of Example 2, assuming that there are large cities only. Thus, using its pebbles, it enumerates 2^n itineraries (where n is the number of intermediate cities). However, after

marking an itinerary, it does not output the itinerary, but instead branches into two identical subprocesses that continue the enumeration. After the last itinerary, $\mathcal{M}_{2\text{exp}}$ is branched into a forest of 2^{2^n} copies of itself, each of which finally outputs one symbol. Imitating $\mathcal{M}_{\text{sib}}$, the I-PFT $\mathcal{M}_{2\text{exp}}$ first walks to the leaf:

$$\langle q_{\text{start}}, \sigma_1, j, \varnothing\rangle \to \langle q_{\text{start}}, \text{down}_1\rangle$$
$$\langle q_{\text{start}}, \sigma_0, 1, \varnothing\rangle \to \langle q_1, \text{up}\rangle$$

Then, in state q_1, it marks as many cities as possible:

$$\langle q_1, \sigma_1, 1, \varnothing\rangle \to \langle q_1, \text{drop}_c; \text{up}\rangle$$
$$\langle q_1, \sigma_1, 0, \varnothing\rangle \to \langle q_{\text{next}}, \text{down}_1\rangle\langle q_{\text{next}}, \text{down}_1\rangle$$

In state q_{next} it continues the search for itineraries by unmarking the most recently marked city; when arriving at the leaf it outputs e:

$$\langle q_{\text{next}}, \sigma_1, 1, \varnothing\rangle \to \langle q_{\text{next}}, \text{down}_1\rangle$$
$$\langle q_{\text{next}}, \sigma_1, 1, \{c\}\rangle \to \langle q_1, \text{lift}_c; \text{up}\rangle$$
$$\langle q_{\text{next}}, \sigma_0, 1, \varnothing\rangle \to e$$

This ends the description of the I-PFT $\mathcal{M}_{2\text{exp}}$.

12. Document transformation

In this section we compare the I-PTT and I-PFT to the document transformation languages DTL and TL, which transform (unranked) forests. We prove that DTL can be simulated by the I-PTT, and that TL has the same expressive power as the I-PFT.

The *Document Transformation Language* DTL was introduced and studied in [38]. A *program* in the DTL framework is a tuple $\mathcal{P} = (\Sigma, \Delta, Q, Q_0, R)$ where Σ and Δ are unranked alphabets, Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, and R is a finite set of *template rules* of the form $\langle q, \varphi(x)\rangle \to f$, where f is a forest over Δ, the leaves of which can additionally be labeled by a *selector* of the form $\langle q', \psi(x, y)\rangle$; q and q' are states in Q, and φ and ψ are MSO formulas over Σ, with one and two free variables respectively. Such a rule can be applied in state q at an input node x that matches φ, i.e., satisfies $\varphi(x)$. Then program $\mathcal{P}$ outputs forest f, where each selector $\langle q', \psi(x, y)\rangle$ is recursively computed as the result of a sequence of copies of $\mathcal{P}$, started in state q' at each of the nodes y that satisfy $\psi(x, y)$, the nodes taken in pre-order (i.e., document order). Thus, $\mathcal{P}$ "jumps" from node x to each node y, according to the trip defined by the MSO formula ψ.

Formally, a configuration of $\mathcal{P}$ on input forest t is a pair $\langle p, u\rangle$ where u is a node of t and p is either a state or a selector of $\mathcal{P}$. An output form of $\mathcal{P}$ on t is a forest in $F_\Delta(\text{Con}(t))$, where $\text{Con}(t)$ is the set of configurations of $\mathcal{P}$ on t. As usual, the computation steps of $\mathcal{P}$ on t are formalized as a binary relation $\Rightarrow_{t,\mathcal{P}}$ on $F_\Delta(\text{Con}(t))$. Let s be an output form and let v be a leaf of s with label $\langle q, u\rangle \in \text{Con}(t)$, where q is a state of $\mathcal{P}$. Moreover, let $\langle q, \varphi(x)\rangle \to f$ be a template rule of $\mathcal{P}$ such that $t \models \varphi(u)$. Let $\theta_u(f)$ be the forest obtained from f by changing every selector $\langle q', \psi(x, y)\rangle$ into $\langle\langle q', \psi(x, y)\rangle, u\rangle$. Then we write $s \Rightarrow_{t,\mathcal{P}} s'$ where s' is obtained from s by replacing the node v by the forest $\theta_u(f)$. Now let s be an output form and let v be a leaf of s with label $\langle\langle q', \psi(x, y)\rangle, u\rangle$. Then we write $s \Rightarrow_{t,\mathcal{P}} s'$ where s' is obtained from s by replacing the node v by the forest $\langle q', u'_1\rangle \cdots \langle q', u'_\ell\rangle$ where $u'_1, \ldots, u'_\ell$ is the sequence of all nodes u' of t, in document order, such that $t \models \psi(u, u')$. The transduction $\tau_{\mathcal{P}}$ realized by $\mathcal{P}$ is defined by $\tau_{\mathcal{P}} = \{(t, s) \in F_\Sigma \times F_\Delta \mid \exists\, q_0 \in Q_0 : \langle q_0, \text{root}_t\rangle \Rightarrow^*_{t,\mathcal{P}} s\}$.

The DTL program $\mathcal{P}$ is *deterministic* if for every two rules $\langle q, \varphi(x)\rangle \to f$ and $\langle q, \varphi'(x)\rangle \to f'$ with the same state q, the tests $\varphi(x)$ and $\varphi'(x)$ are exclusive, meaning that the sites they define are disjoint.

We observe here that in [38] the selectors have a more complicated form, which we will discuss after the next lemma.

We have defined the DTL program such that the input t is an unranked forest, and thus it can in particular be a ranked tree. It should be clear from Lemma 24 (which also holds for sites instead of trips) that we may in fact restrict ourselves to ranked trees and assume that input forests are encoded as binary trees. Thus, *from now on we assume that* in the above definition Σ is a ranked alphabet and $t \in T_\Sigma$ is a ranked input tree. This allows us to compare DTL programs with PFT's.

Let DTL denote the transductions realized by DTL programs and dDTL those realized by deterministic DTL programs, from ranked trees to unranked forests. Thus, the class of forest transductions realized by DTL programs is equal to $\text{enc}' \circ \text{DTL}$, and similarly for the deterministic case.

Lemma 35. DTL $\subseteq$ I-PFT *and* dDTL $\subseteq$ I-dPFT.

Proof. Let $\mathcal{P} = (\Sigma, \Delta, Q, Q_0, R)$ be a DTL program. We construct an equivalent I-PFT $\mathcal{M}$ with MSO tests, cf. Theorem 16. It has the same alphabets Σ and Δ as $\mathcal{P}$. Since $\mathcal{M}$ stepwise simulates $\mathcal{P}$, its set of states consists of the states and selectors of $\mathcal{P}$, plus the states that it needs to execute the subroutines discussed below. It has the same initial states Q_0 as $\mathcal{P}$. Moreover, it uses invisible pebbles of a single colour $\odot$, and never lifts its pebbles.

For an input tree t, the transducer $\mathcal{M}$ simulates a template rule $\langle q, \varphi(x)\rangle \to f$ in state q at node u of t by first using an MSO head test to check whether $t \models \varphi(u)$. With a positive test result, it calls a subroutine S that outputs the Δ-labeled nodes of the right-hand side f. The subroutine S is started by $\mathcal{M}$ in state $[f]$. If its state is of the form $[sf']$, for a tree s and a forest f', it uses a rule $\langle [sf'], \sigma, j, b\rangle \to \langle [s], \text{stay}\rangle \langle [f'], \text{stay}\rangle$, branching the computation. If the state is of the form $[\delta(f')]$, the rule is $\langle [\delta(f')], \sigma, j, b\rangle \to \delta(\langle [f'], \text{stay}\rangle)$, and if it is of the form $[\varepsilon]$, the rule is $\langle [\varepsilon], \sigma, j, b\rangle \to \varepsilon$. If the state is of the form $[\langle q', \psi(x, y)\rangle]$, for a selector $\langle q', \psi(x, y)\rangle$, the subroutine S returns control to (this copy of) $\mathcal{M}$ in state $\langle q', \psi(x, y)\rangle$. In that state, $\mathcal{M}$ first drops a pebble $\odot$ on the current node u and then calls a subroutine $S_{q',\psi}$ that finds all nodes u' in the input tree t for which $\psi(u, u')$ holds. The subroutine does this by performing a depth-first traversal of t, starting at the root, checking in each node u' whether $t \models \psi(u, u')$ using an MSO test on the observable configuration. If true, then $S_{q',\psi}$ branches into two concatenated processes. The left branch returns control to $\mathcal{M}$ in state q', and the right branch continues the depth-first search. When the search ends, $S_{q',\psi}$ outputs ε. Thus, $S_{q',\psi}$ transforms the configuration $\langle\langle q', \psi(x, y)\rangle, u, \pi\rangle$ of $\mathcal{M}$ into the forest of configurations $\langle q', u'_1, \pi\rangle \cdots \langle q', u'_\ell, \pi\rangle$, where $u'_1, \ldots, u'_\ell$ are all such nodes u', in document order. With this definition of $\mathcal{M}$, it should be clear that $\tau_{\mathcal{M}} = \tau_{\mathcal{P}}$. □

The selectors in [38] are more general than those defined above. They can be of the form $\langle q'_1, \psi_1(x, y), \ldots, q'_m, \psi_m(x, y)\rangle$, such that the MSO formulas $\psi_1(x, y), \ldots, \psi_m(x, y)$ are mutually exclusive, i.e., the trips they define are mutually disjoint. Let $\psi(x, y)$ be the disjunction of all $\psi_i(x, y)$, $i \in [1, m]$. The execution of the above selector at node u of the input tree results in the forest $\langle q'_{i_1}, u'_1\rangle \cdots \langle q'_{i_\ell}, u'_\ell\rangle$ where $u'_1, \ldots, u'_\ell$ is the sequence of all nodes u' of t in document order such that $t \models \psi(u, u')$, and for every $j \in [1, \ell]$, i_j is the unique number in $[1, m]$ such that $t \models \psi_{i_j}(u, u'_j)$. It should be clear that Lemma 35 is still valid with these more general selectors. To execute the above selector, the I-PFT $\mathcal{M}$ calls subroutine $S_{q'_1,\psi_1,\ldots,q'_m,\psi_m}$ which in each node u' tests each of the formulas $\psi_i(u, u')$; if $\psi_i(u, u')$ is true, then the subroutine branches in two, in the first branch returning control to $\mathcal{M}$ in state q_i.

To compare DTL to I-PTT rather than I-PFT we also consider DTL programs that transform ranked trees. A DTL program $\mathcal{P} = (\Sigma, \Delta, Q, Q_0, R)$ is *ranked* if Σ and Δ are both ranked alphabets, and every rule $\langle q, \varphi(x)\rangle \to f$ satisfies the following two restrictions:

(R1) f is a ranked tree in $T_\Delta(S)$ where S is the set of selectors, and
(R2) for every selector $\langle q', \psi(x, y)\rangle$ that occurs in f, every input tree $t \in T_\Sigma$, and every node $u \in N(t)$, if $t \models \varphi(u)$ then there is a unique node $v \in N(t)$ such that $t \models \psi(u, v)$.

In other words, the trip $T(\psi(x, y))$ is functional and, for fixed input tree $t \in T_\Sigma$, it is defined for every node of t that satisfies $\varphi(x)$. Thus, execution of the selector $\langle q', \psi(x, y)\rangle$ results in a "jump" from node x to exactly one node y. This clearly implies that all reachable output forms of $\mathcal{P}$ are ranked trees in $T_\Delta(\text{Con}(t))$. Thus $\tau_{\mathcal{P}} \subseteq T_\Sigma \times T_\Delta$ is a ranked tree transformation. The class of transductions realized by ranked TL programs will be denoted by DTL_r, and by dDTL_r in the deterministic case.

Corollary 36. $\text{DTL}_\text{r} \subseteq$ I-PTT *and* $\text{dDTL}_\text{r} \subseteq$ I-dPTT.

Proof. The proof is the same as the one of Lemma 35, except for the subroutines S and $S_{q',\psi}$. The states of S are now of the form $[s]$ where s is a subtree of a right-hand side of a rule. Instead of the rules for states $[sf']$, $[\delta(f')]$, and $[\varepsilon]$, subroutine S has rules $\langle [\delta(s_1, \ldots, s_m)], \sigma, j, b\rangle \to \delta(\langle [s_1], \text{stay}\rangle, \ldots, \langle [s_m], \text{stay}\rangle)$ for every δ of rank m and all trees $s_1, \ldots, s_m$ (restricted to subtrees of right-hand sides). When subroutine $S_{q',\psi}$ finds a node u' such that $t \models \psi(u, u')$ (and it always finds one by restriction (R2)), it returns control to $\mathcal{M}$ and does not continue the depth-first search. □

It can, in fact, be shown that when output forests are encoded as binary trees, DTL is included in I-PTT. Thus, instead of I-PFT we consider the class I-PTT ∘ dec (which equals the class I-PTT ∘ dec′), cf. Section 11. The next theorem will not be used in what follows (except in the paragraph directly after the theorem).

Theorem 37. DTL ⊆ I-PTT ∘ dec *and* dDTL ⊆ I-dPTT ∘ dec.

Proof. Let $\mathcal{P} = (\Sigma, \Delta, Q, Q_0, R)$ be a DTL program. The main difficulty in outputting the binary encoding $\text{enc}(f)$ of a forest f as opposed to the construction in the proof of Lemma 35 is that here the first symbol δ of f has to be determined before any other output can be generated. We reconsider that construction, and here essentially make a depth-first sequential search over nodes in the computation tree (implemented using a stack of pebbled nodes) instead of the recursive approach. In that way an I-PTT $\mathcal{M}$ can simulate the leftmost computations of the DTL program $\mathcal{P}$.

As unranked forests with selectors can be generated by the recursive definition $f ::= \delta(f)f' \mid \langle q, \psi\rangle f \mid \varepsilon$, where f' is an alias of f, DTL rules are of the form $\langle q, \varphi(x)\rangle \to f$, where f is $\delta(f_1)f_2$, $\langle q, \psi\rangle f'$, or ε. The set of states of the transducer $\mathcal{M}$ to be constructed consists of the states of $\mathcal{P}$ and all states $[f]$ where f is a subforest of a right-hand side of a rule of $\mathcal{P}$, plus the states of the subroutines $S'_{q',\psi}$ and $S''_{q',\psi}$ discussed below. The state $[f]$ is used to generate the binary encoding of the subforest f, similarly to its use by the subroutine S in the proof of Lemma 35. The initial states of $\mathcal{M}$ are those of $\mathcal{P}$.

The pebble colours used by $\mathcal{M}$ are $\langle q, \psi, f\rangle$ where $\langle q, \psi\rangle f$ occurs in the right-hand side of a rule of $\mathcal{P}$, and the special colour $\perp$. The state and pebble stack of $\mathcal{M}$ store a part of the output form of $\mathcal{P}$ that still has to be evaluated. The output alphabet of $\mathcal{M}$ is $\Delta \cup \{e\}$ where each $\delta \in \Delta$ has rank 2 and e has rank 0.

The transducer $\mathcal{M}$ starts by dropping $\perp$ on the root. To simulate, in state q, a rule $\langle q, \varphi(x)\rangle \to f$ of $\mathcal{P}$, it uses an MSO head test to check whether φ holds for the current node, and goes into state $[f]$. We consider the above three cases for $[f]$.

In state $[\langle q', \psi\rangle f']$, pebble $\langle q', \psi, f'\rangle$ is dropped on the current node u. As in the proof of Lemma 35, $\mathcal{M}$ then calls a subroutine $S'_{q',\psi}$ which, this time, finds the first node u' (in document order) for which $\psi(u, u')$ holds, where it returns control to $\mathcal{M}$ in state q'. If $S'_{q',\psi}$ does not find such a matching node u', then it moves to the topmost pebble $\langle q', \psi, f'\rangle$, lifts it, and returns control to $\mathcal{M}$ in state $[f']$.

In state $[f] = [\delta(f_1) f_2]$, the root δ of the first tree of the forest is explicitly given, and this is captured by the I-PTT output rule $\langle [f], \sigma, j, b\rangle \to \delta(\langle [f_1], \mathrm{drop}_\perp\rangle, \langle [f_2], \mathrm{stay}\rangle)$. The symbol $\perp$ is pushed, and never popped afterwards, making the stack of pebbles effectively empty: the first copy of the transducer evaluates f_1 as left child of δ. The second copy inherits the stack and evaluates f_2 as right child of δ, together with all postponed duties as stored in the stack of pebbles. This will generate the siblings of δ in the original forest.

In state $[\varepsilon]$, the transducer $\mathcal{M}$ determines the colour of the topmost pebble, using an MSO test on the observable configuration. If it is $\perp$, it outputs e for the empty forest. Otherwise it calls subroutine $S''_{q',\psi}$ to continue the search corresponding to the topmost pebble $\langle q', \psi, f'\rangle$. That subroutine finds the first node u' after the current node u (in document order) for which $\psi(v, u')$ holds, where v is the position of the topmost pebble. Similar to $S'_{q',\psi}$, if a matching node is found it returns control to $\mathcal{M}$ in state q', and otherwise it lifts the topmost pebble and returns control to $\mathcal{M}$ in state $[f']$.

This ends the description of $\mathcal{M}$. To understand its correctness, we show how the output forms of $\mathcal{M}$ represent output forms of $\mathcal{P}$. We disregard the output forms of $\mathcal{M}$ that contain states of the subroutines $S'_{q',\psi}$ and $S''_{q',\psi}$, and view the execution of such a subroutine as one big computation step of $\mathcal{M}$ that (deterministically) changes one configuration into another. The mapping 'rep' from such restricted output forms of $\mathcal{M}$ to output forms of $\mathcal{P}$ is defined as follows. The Δ-labeled part of the output form of $\mathcal{M}$ is decoded, i.e., $\mathrm{rep}(e) = \varepsilon$ and $\mathrm{rep}(\delta(s_1, s_2)) = \delta(\mathrm{rep}(s_1))\,\mathrm{rep}(s_2)$. It remains to define 'rep' for the configurations on an input tree t that occur in the restricted output forms of $\mathcal{M}$, i.e., for every configuration $\langle p, u, \pi\rangle$ where p is a state q of $\mathcal{P}$ or a state $[f]$. We will write $\mathrm{rep}(p, u, \pi)$ instead of $\mathrm{rep}(\langle p, u, \pi\rangle)$. The definition is by induction on the structure of π, of which the topmost pebble is of the form $(v, \perp)$ or $(v, \langle q', \psi, f'\rangle)$. For a state $[f]$, we define $\mathrm{rep}([f], u, \pi(v, \perp)) = \theta_u(f)$ and

$$\mathrm{rep}([f], u, \pi(v, \langle q', \psi, f'\rangle)) = \theta_u(f)\langle q', u'_1\rangle \cdots \langle q', u'_\ell\rangle\,\mathrm{rep}([f'], v, \pi)$$

where $u'_1, \ldots, u'_\ell$ are all nodes u' after u (in document order) such that $t \models \psi(v, u')$. Note that $\mathrm{rep}([f], u, \pi) = \theta_u(f)\,\mathrm{rep}([\varepsilon], u, \pi)$ because $\theta_u(\varepsilon) = \varepsilon$, and hence

$$\mathrm{rep}([f_1 f_2], u, \pi) = \theta_u(f_1)\,\mathrm{rep}([f_2], u, \pi).$$

For a state q of $\mathcal{P}$ we define $\mathrm{rep}(q, u, \pi) = \langle q, u\rangle\,\mathrm{rep}([\varepsilon], u, \pi)$.

It is now straightforward to prove, for every initial state q_0 of $\mathcal{P}$, every input tree t, and every output form s of $\mathcal{P}$, that $\langle q_0, \mathrm{root}_t\rangle \Rightarrow^*_{t,\mathcal{P}} s$ if and only if there exists a restricted output form s' of $\mathcal{M}$ such that $\langle q_0, \mathrm{root}_t, (\mathrm{root}_t, \perp)\rangle \Rightarrow^*_{t,\mathcal{M}} s'$ and $\mathrm{rep}(s') = s$. The proof of the if-direction of this equivalence is by induction on the length of the computation, and consists of four cases, depending on the state of the configuration of $\mathcal{M}$ that is rewritten, as discussed above, viz., q, $[\langle q', \psi\rangle f']$, $[\delta(f_1) f_2]$, or $[\varepsilon]$. From the last two cases it follows that for every restricted output form s' of $\mathcal{M}$ there exists a restricted output form s'' of $\mathcal{M}$ such that $s' \Rightarrow^*_{t,\mathcal{M}} s''$, $\mathrm{rep}(s'') = \mathrm{rep}(s')$, and the states of $\mathcal{M}$ that occur in s'' are either states q of $\mathcal{P}$ or states of the form $[\langle q', \psi\rangle f']$. In the only-if-direction we only consider leftmost computations of $\mathcal{P}$, i.e., computations in which always the first configuration of the output form (in pre-order) is rewritten. If $\mathrm{rep}(s') = \mathrm{rep}(s'') = s$, with s'' as above, then the first configuration of $\mathcal{M}$ in s'' corresponds to the first configuration of $\mathcal{P}$ in s, and the proof is similar to the first two cases of the proof of the if-direction. The details are left to the reader. Since $\mathrm{rep}(s') = \mathrm{dec}(s')$ for every output tree s' of $\mathcal{M}$, the above equivalence implies that $\tau_\mathcal{M} \circ \mathrm{dec} = \tau_\mathcal{P}$. □

We are now able to finish the proof of Lemma 33(2). Consider the mapping $\mathrm{flat} : T_{\Delta_1} \to F_\Delta$ defined in that proof. It can be realized by the one-state deterministic DTL program with rules $\langle q, \mathrm{lab}_@(x)\rangle \to \langle q, \mathrm{down}_1(x, y)\rangle\langle q, \mathrm{down}_2(x, y)\rangle$, $\langle q, \mathrm{lab}_\delta(x)\rangle \to \delta(\langle q, \mathrm{down}_1(x, y)\rangle)$ for every $\delta \in \Delta$, and $\langle q, \mathrm{lab}_e(x)\rangle \to \varepsilon$. Hence, by Theorem 37, it is in I-dPTT $\circ$ dec, which means that the mapping $\mathrm{flat} \circ \mathrm{enc}$ is in I-dPTT.

In [37] the language DTL was extended to the *Transformation Language* TL where the states have parameters that hold unevaluated forests, similar to macro tree transducers with outside-in parameter evaluation [22]. In a TL program $\mathcal{P} = (\Sigma, \Delta, Q, Q_0, R)$, the set of states Q is a ranked alphabet such that the initial states in Q_0 have rank 0. The rules of TL program $\mathcal{P}$ are of the form $\langle q, \varphi(x)\rangle(z_1, \ldots, z_n) \to f$, where $n = \mathrm{rank}_Q(q)$ and $z_1, \ldots, z_n$ are the formal parameters of q, taken from a fixed infinite parameter set $Z = \{z_1, z_2, \ldots\}$. The right-hand side f of the rule is a forest of which the nodes can be labeled by a symbol from Δ, by a selector $\langle q', \psi(x, y)\rangle$, or by a formal parameter z_i with $i \in [1, n]$. A node labeled by $\langle q', \psi(x, y)\rangle$ must have $\mathrm{rank}(q')$ children, and a node labeled by parameter z_i must be a leaf. Thus, in such a forest (called an

action in [37]), selectors can be nested. We could as well allow in TL the more general selectors discussed after Lemma 35, but we restrict ourselves to the usual selectors for simplicity (and because they are the selectors in [37]). Determinism of program $\mathcal{P}$ is defined as for DTL.

An output form of $\mathcal{P}$ on input forest t is a forest of which the nodes can be labeled either by a symbol from Δ, or by a configuration $\langle q, u\rangle$ or $\langle\langle q, \psi(x, y)\rangle, u\rangle$ of $\mathcal{P}$ in which case the node must have rank(q) children. A node of an output form, or of a right-hand side of a rule, is said to be *outermost* if all its proper ancestors are labeled by a symbol from Δ. The computation steps of $\mathcal{P}$ are formalized as a binary relation on output forms, as follows (similar to the DTL case). Let s be an output form, and let v be an outermost node of s with label $\langle q, u\rangle$, where q is a state of $\mathcal{P}$. Moreover, let $\langle q, \varphi(x)\rangle(z_1, \ldots, z_n) \to f$ be a rule of $\mathcal{P}$ such that $t \models \varphi(u)$. Let $\theta_u(f)$ be defined as in the DTL case. Then we write $s \Rightarrow_{t,\mathcal{P}} s'$ where s' is obtained from s by replacing the subtree $s|_v$ with root v by the forest $\theta_u(f)$ in which every parameter z_i is replaced by the subtree $s|_{vi}$, for $i \in [1, \text{rank}(q)]$. Intuitively, the subtree $s|_{vi}$ rooted at the i-th child vi of v is the i-th actual parameter of (this occurrence of) the state q. Now let s be an output form and let v be an outermost node of s with label $\langle\langle q', \psi(x, y)\rangle, u\rangle$ and rank$(q') = m$. Then we write $s \Rightarrow_{t,\mathcal{P}} s'$ where s' is obtained from s by replacing the subtree $s|_v$ with root v by the forest $\langle q', u'_1\rangle(s|_{v1}, \ldots, s|_{vm}) \cdots \langle q', u'_\ell\rangle(s|_{v1}, \ldots, s|_{vm})$ where $u'_1, \ldots, u'_\ell$ is the sequence of all nodes u' of t, in document order, such that $t \models \psi(u, u')$. Intuitively, the actual parameters of (this occurrence of) the selector $\langle q', \psi(x, y)\rangle$ are passed to each new occurrence of the state q'. As in the DTL case, the transduction realized by $\mathcal{P}$ is defined by $\tau_{\mathcal{P}} = \{(t, s) \in F_\Sigma \times F_\Delta \mid \exists q_0 \in Q_0 : \langle q_0, \text{root}_t\rangle \Rightarrow^*_{t,\mathcal{P}} s\}$.

In [37] the denotational semantics of a TL program is given as a least fixed point. It is straightforward to show that the semantics in [37] is equivalent to the above operational semantics.[21] Also, in [37] the syntactic formulation of TL is such that in the right-hand side of a rule the states can have forests as parameters rather than trees. Such a forest parameter $s_1 \cdots s_m$, where each s_i is a tree, can be expressed in our syntactic formulation of TL as the tree $\langle @_m, x = y\rangle(s_1, \ldots, s_m)$, where $@_m$ is a special state of rank m that has the unique rule $\langle @_m, x = x\rangle(z_1, \ldots, z_m) \to z_1 \cdots z_m$.

Example 38. The transformation from Example 2 can be computed by a deterministic TL program $\mathcal{P}_{\text{sib}}$ with the following rules, where the variables i, σ_i, c, and λ_i range over the same values as in Example 2, with $c = 1$ or $i = 1$ in rule ρ_4.

$$\rho_1 : \langle q_{\text{start}}, \text{root}(x)\rangle \to \langle q_{\text{start}}, \text{leaf}(y)\rangle$$

$$\rho_2 : \langle q_{\text{start}}, \neg\text{root}(x) \wedge \text{lab}_{\sigma_0}(x)\rangle \to \langle q_1, \text{up}(x, y)\rangle(\sigma_0, e)$$

$$\rho_3 : \langle q_0, \neg\text{root}(x) \wedge \text{lab}_{\lambda_0}(x)\rangle(z_1, z_2) \to \langle q_0, \text{up}(x, y)\rangle(z_1, z_2)$$

$$\rho_4 : \langle q_c, \neg\text{root}(x) \wedge \text{lab}_{\lambda_i}(x)\rangle(z_1, z_2) \to \langle q_i, \text{up}(x, y)\rangle(\lambda_i(z_1), \langle q_c, \text{up}(x, y)\rangle(z_1, z_2))$$

$$\rho_5 : \langle q_c, \text{root}(x) \wedge \text{lab}_{\sigma_1}(x)\rangle(z_1, z_2) \to r(\sigma_1(z_1), z_2)$$

Intuitively, z_1 represents an itinerary from some city to Vladivostok, and z_2 represents a list of itineraries from Moscow to Vladivostok (viz. all itineraries that do not have z_1 as postfix), where we only consider itineraries that do not visit a small city twice in a row.

The selectors in the right-hand sides of the rules all define functional trips, and hence select just one node. Rule ρ_1 jumps from the root to the leaf, and rules ρ_2, ρ_3, ρ_4 just move to the parent.

To show the correctness of $\mathcal{P}_{\text{sib}}$, let u be a node of an input tree t, such that u is not the leaf of t. Moreover, let ζ_1 be an output tree that is an itinerary from the child of u to the leaf, of which the first stop is large ($c = 1$) or small ($c = 0$), and let ζ_2 be an arbitrary output form. Then $\langle q_c, u\rangle(\zeta_1, \zeta_2)$ generates the output form $r(s_1(\zeta_1), r(s_2(\zeta_1), \ldots r(s_n(\zeta_1), \zeta_2) \cdots))$ where $s_1, \ldots, s_n$ are all possible itineraries from the root to u such that every $s_i(\zeta_1)$ is an itinerary from root to leaf. This can be proved by induction on the number of nodes between the root and u. The base of the induction is by rule ρ_5, which generates the root label σ_1, and the induction step is by rules ρ_3 and ρ_4. In rule ρ_3 a small city is skipped. In rule ρ_4, the outermost selector $\langle q_i, \text{up}(x, y)\rangle$ generates all itineraries s_i from the root to x that include x (or rather, its label λ_i), whereas the innermost selector $\langle q_c, \text{up}(x, y)\rangle$ generates all those that do not include x. Taking $c = 1$, u equal to the parent of the leaf, and σ_0 to the label of the leaf, shows that $\langle q_1, u\rangle(\sigma_0, e)$ generates all required itineraries. That implies the correctness of $\mathcal{P}_{\text{sib}}$ by rule ρ_2.

An XSLT 1.0 program with exactly the same structure as $\mathcal{P}_{\text{sib}}$ is given in Section 13. □

As in the case of DTL, we will assume that in the above definition of TL program, the input alphabet Σ is ranked and the input forest t is a ranked tree in T_Σ. Also, *ranked* TL programs are defined as for DTL programs. In particular, for every rule $\langle q, \varphi(x)\rangle(z_1, \ldots, z_n) \to f$, the right-hand side f is a ranked tree in $T_\Delta(S \cup Z_n)$ where S is the set of selectors and $Z_n = \{z_1, \ldots, z_n\}$. The program $\mathcal{P}_{\text{sib}}$ of Example 38 is ranked.

Let TL denote the class of transductions realized by TL programs and dTL the class of those realized by deterministic TL programs, from ranked trees to unranked forests. Moreover, TL_r and dTL_r denote the classes of transductions realized by ranked programs, from ranked trees to ranked trees.

[21] It is similar to the "alternative" fixed point characterization of the OI context-free tree languages mentioned after [21, Definition 5.19].

In what follows we will prove that TL = I-PFT, and similarly for the deterministic case and for the ranked case (Theorem 46). Note that this also implies that TL programs and I-PFT's realize the same forest transductions, i.e., $\mathrm{enc}' \circ \mathsf{TL} = \mathrm{enc}' \circ \text{I-PFT}$. These equalities are variants of the well-known fact that macro grammars are equivalent to indexed grammars [24], see also [23, Theorem 5.24].

Lemma 39. $\mathsf{TL} \subseteq \text{I-PFT}$ *and* $\mathsf{dTL} \subseteq \text{I-dPFT}$. *Moreover,* $\mathsf{TL}_r \subseteq \text{I-PTT}$ *and* $\mathsf{dTL}_r \subseteq \text{I-dPTT}$.

Proof. The construction extends the one in the proof of Lemma 35. The main idea is to use pebbles to store the actual parameters. Thus, the pebble colours are of the form $([s_1], \ldots, [s_m])$ where $m \geq 0$ and $s_1, \ldots, s_m$ are subtrees of a right-hand side of a rule (in particular, the subtrees rooted at the children of a node that is labeled by a selector).

As in the DTL case, for an input tree t, the transducer $\mathcal{M}$ simulates a rule $\langle q, \varphi(x)\rangle(z_1, \ldots, z_n) \to f$ in state q at node u of t by testing whether $t \models \varphi(u)$ and, if successful, calling subroutine S. In this (nested) case, S outputs the outermost Δ-labeled nodes of f, plus the outermost Δ-labeled nodes of the actual parameters that are the values of the formal parameters z_i that occur outermost in f. For the states $[sf']$, $[\delta(f')]$, and $[\varepsilon]$, the rules of S are as in the proof of Lemma 35 (and see the proof of Corollary 36 for the ranked case). If the state of S is of the form $[\langle q', \psi(x, y)\rangle(s_1, \ldots, s_m)]$, then it drops a pebble $([s_1], \ldots, [s_m])$ on the current node u to represent the parameters, and returns control to (this copy of) $\mathcal{M}$ in state $\langle q', \psi(x, y)\rangle$. In that state, $\mathcal{M}$ calls subroutine $S_{q',\psi}$, which works as in the DTL case. Note that $\mathcal{M}$ need not drop a pebble $\odot$, as $S_{q',\psi}$ can use the pebble $([s_1], \ldots, [s_m])$ instead. Finally, if the state of S is of the form $[z_i]$ for some formal parameter z_i, this means that the corresponding actual parameter has to be evaluated. To do this, the subroutine S searches for the topmost pebble, which has some colour $([s_1], \ldots, [s_m])$. Then S lifts that pebble and changes its state to $[s_i]$, ready to evaluate s_i.

It is easy to show, for every $i \in \mathbb{N}$, that whenever $\mathcal{M}$ is in state q or state $\langle q, \psi(x, y)\rangle$ with $i \in [1, \mathrm{rank}(q)]$, and whenever S is in state $[f]$ and z_i occurs in f, then the topmost pebble with colour $([s_1], \ldots, [s_m])$ satisfies $i \in [1, m]$. Hence the last sentence of the previous paragraph never fails.

To understand the correctness of $\mathcal{M}$, we show how the output forms of $\mathcal{M}$ represent output forms of $\mathcal{P}$, similar to the correctness proof of Theorem 37. We restrict ourselves to output forms in which all the states of $\mathcal{M}$ are states of $\mathcal{P}$ or selectors of $\mathcal{P}$ or states of the subroutine S, i.e., we disregard the states of the subroutines $S_{q',\psi}$ and view the execution of such a subroutine as one big step in the computation of $\mathcal{M}$, changing a configuration $\langle\langle q', \psi(x, y)\rangle, u, \pi\rangle$ deterministically into a forest $\langle q', u'_1, \pi\rangle \cdots \langle q', u'_\ell, \pi\rangle$ (which is just a one-node tree $\langle q', u', \pi\rangle$ in the ranked case). Thus, we define a mapping 'rep' from such restricted output forms of $\mathcal{M}$ to the output forms of $\mathcal{P}$. The Δ-labeled part of the output form is not changed by 'rep', i.e., $\mathrm{rep}(sf) = \mathrm{rep}(s)\,\mathrm{rep}(f)$, $\mathrm{rep}(\varepsilon) = \varepsilon$, and $\mathrm{rep}(\delta(f)) = \delta(\mathrm{rep}(f))$ for $\delta \in \Delta$, where s is a tree and f a forest (or $\mathrm{rep}(\delta(s_1, \ldots, s_m)) = \delta(\mathrm{rep}(s_1), \ldots, \mathrm{rep}(s_m))$ in the ranked case). It remains to define 'rep' for the configurations of $\mathcal{M}$ that occur in restricted output forms, i.e., for every configuration $\langle p, u, \pi\rangle$ where p is a state q of $\mathcal{P}$, or a selector $\langle q', \psi(x, y)\rangle$ of $\mathcal{P}$, or a state $[f]$ of S (where f is a subforest of a right-hand side of a rule of $\mathcal{P}$). As before, we will write $\mathrm{rep}(p, u, \pi)$ instead of $\mathrm{rep}(\langle p, u, \pi\rangle)$. The definition is by induction on the structure of π, of which we consider the topmost pebble: let $\pi = \pi'(v, ([s_1], \ldots, [s_m]))$. If $p = q$ or $p = \langle q', \psi(x, y)\rangle$, then $\mathrm{rep}(p, u, \pi) = \langle p, u\rangle(\mathrm{rep}([s_1], v, \pi'), \ldots, \mathrm{rep}([s_m], v, \pi'))$. For $p = [f]$ we define $\mathrm{rep}([f], u, \pi)$ to be the forest $\theta_u(f)$ in which every parameter z_i is replaced by $\mathrm{rep}([s_i], v, \pi')$. Finally, for $\pi = \varepsilon$, we define $\mathrm{rep}(p, u, \varepsilon) = \langle p, u\rangle$ in the first case, and $\mathrm{rep}([f], u, \varepsilon) = \theta_u(f)$ in the second case. If we consider only reachable output forms of $\mathcal{M}$, then 'rep' is well defined (cf. the previous paragraph).

It is now straightforward to prove, for every initial state q_0 of $\mathcal{P}$, every input tree t, and every output form s of $\mathcal{P}$, that $\langle q_0, \mathrm{root}_t\rangle \Rightarrow^*_{t,\mathcal{P}} s$ if and only if there exists a restricted output form s' of $\mathcal{M}$ such that $\langle q_0, \mathrm{root}_t, \varepsilon\rangle \Rightarrow^*_{t,\mathcal{M}} s'$ and $\mathrm{rep}(s') = s$. In the proof one should use the rather obvious fact that for every restricted output form s' of $\mathcal{M}$ there exists a restricted output form s'' of $\mathcal{M}$ such that $s' \Rightarrow^*_{t,\mathcal{M}} s''$, $\mathrm{rep}(s'') = \mathrm{rep}(s')$, and no states $[f]$ of S occur in s''. The above equivalence implies that $\tau_{\mathcal{M}} = \tau_{\mathcal{P}}$. □

Example 40. The I-PTT $\mathcal{M}$ corresponding to the (ranked) TL program $\mathcal{P}_{\mathrm{sib}}$ of Example 38, according to the proof of Lemma 39, works in essentially the same way as the I-PTT $\mathcal{M}_{\mathrm{sib}}$ of Example 2. Rules ρ_1 to ρ_5 are translated into rules for $\mathcal{M}$ that are similar to the first 5 rules of $\mathcal{M}_{\mathrm{sib}}$. Rule ρ_1 can be translated into the first rule of $\mathcal{M}_{\mathrm{sib}}$, which implements the jump to the leaf. Rule ρ_2 can be translated into the rule $\langle q_{\mathrm{start}}, \sigma_0, 1, \varnothing\rangle \to \langle q_1, \mathrm{drop}_{([\sigma_0],[e])}; \mathrm{up}\rangle$. Thus, $\mathcal{M}$ drops the special pebble $([\sigma_0], [e])$ at the leaf, where $\mathcal{M}_{\mathrm{sib}}$ does not drop a pebble. Rule ρ_3 can be translated into the rule $\langle q_0, \lambda_0, 1, \varnothing\rangle \to \langle q_0, \mathrm{drop}_{([z_1],[z_2])}; \mathrm{up}\rangle$. Thus, $\mathcal{M}$ drops the "empty" pebble $([z_1], [z_2])$ whenever $\mathcal{M}_{\mathrm{sib}}$ does not drop a pebble. Rule ρ_4 can be translated into the rule $\langle q_c, \lambda_i, 1, \varnothing\rangle \to \langle q_i, \mathrm{drop}_{c(\lambda_i)}; \mathrm{up}\rangle$, where $c(\lambda_i)$ is the pebble $([\lambda_i(z_1)], [\langle q_c, \mathrm{up}(x, y)\rangle(z_1, z_2)])$ which is dropped by $\mathcal{M}$ instead of the pebble c. Note that the pebble colours $c(\lambda_i)$ and $([\sigma_0], [e])$ include the label (λ_i or σ_0) of the node on which the pebble is dropped, which is of course superfluous information. Finally, rule ρ_5 can be translated into the rule $\langle q_c, \sigma_1, 0, \varnothing\rangle \to r(\langle[\sigma_1(z_1)], \mathrm{stay}\rangle, \langle[z_2], \mathrm{stay}\rangle)$, which calls the states $[\sigma_1(z_1)]$ and $[z_2]$ of the subroutine S. In state $[\sigma_1(z_1)]$, S outputs σ_1 and goes into state $[z_1]$. We note that at any moment of time, when $\mathcal{M}$ is at node u of the input tree, all descendants of u, possibly including u itself, carry a pebble. Thus, in state $[z_i]$, S moves down to the child of u, lifts pebble $([s_1], [s_2])$ and goes into state $[s_i]$. It is now easy to see that states $[z_1]$ and $[z_2]$ of $\mathcal{M}$ correspond to states q_{out} and q_{next} of $\mathcal{M}_{\mathrm{sib}}$, respectively. In state $[z_1]$, S moves down and outputs the labels of all nodes that are marked by some pebble $c(\lambda_i)$ or $([\sigma_0], [e])$, lifting those pebbles one by one. In state $[z_2]$,

S moves down to the first pebble $c(\lambda_i)$, replaces that pebble by the "empty" pebble $([z_1],[z_2])$, and returns control to $\mathcal{M}$, which then goes into state q_c and moves up to the parent. When, in state $[z_2]$, S reaches the leaf with pebble $([\sigma_0],[e])$, it lifts that pebble and outputs e. □

Lemma 39 and Theorem 34 (for $k=0$) together provide an alternative proof of the main result of [37]: the inverse type inference problem and the typechecking problem are solvable for TL programs. The proofs are, however, similar. In [37] every TL program is decomposed into three macro tree transducers, whereas we have decomposed every I-PTT into two TT's. In general, decomposition into TT's leads to more efficient typechecking than decomposition into macro tree transducers, because (cf. Proposition 6) inverse type inference of a macro tree transducer takes double exponential time, unless the number of parameters is bounded and the output type is fixed [47]. Let us define a TL^{DB} program to be a TL program in which the MSO formulas $\varphi(x)$ and $\psi(x,y)$ in the template rules of the program are represented by deterministic bottom-up finite-state tree automata that recognize the corresponding regular sites $\text{mark}(T(\varphi))$ and trips $\text{mark}(T(\psi))$.

Theorem 41. *The inverse type inference problem and the typechecking problem are solvable for* TL^{DB} *programs in* 3*-fold and* 4*-fold exponential time, respectively.*

Proof. By Theorem 34, these problems are solvable for I-PFT's in 2-fold and 3-fold exponential time. Let us now assume that the regular sites and trips used in MSO tests of I-PFT's are also represented by deterministic bottom-up finite-state tree automata. Then it is easy to see that the construction in the proof of Lemma 39 takes polynomial time. However, the MSO tests that are used by the resulting I-PFT have to be removed, and the construction in the proof of Theorem 16 takes exponential time, as can be checked in a straightforward way. That involves checking that the constructions in the proofs of Lemmas 10, 12, and 13 take polynomial time, and so does the construction in the proof of Proposition 14 (for the nonfunctional case), i.e., in the proof of [5, Theorem 8]. The exponential in the proof of Theorem 16 is due to the use of the sets of states S of $\mathcal{B}_d$ in the colours of the beads. Hence, solving the above problems takes one more exponential for TL^{DB} programs than for I-PFT. □

A TL program $\mathcal{P}=(\Sigma,\Delta,Q,Q_0,R)$ is a *macro tree transducer*, more precisely an OI macro tree transducer (see [22]), if it is ranked, and for every rule $\langle q,\varphi(x)\rangle(z_1,\ldots,z_n)\to f$ the following hold. First, $\varphi(x)\equiv \text{lab}_\sigma(x)$ for some $\sigma\in\Sigma$. Second, for every selector $\langle q',\psi(x,y)\rangle$ that occurs in f, we have $\psi(x,y)\equiv \text{down}_i(x,y)$ for some $i\in[1,\text{rank}_\Sigma(\sigma)]$. It follows immediately from Lemma 39 that macro tree transducers can be simulated by I-PTT. Let MT_{OI} denote the class of tree transductions realized by OI macro tree transducers, and dMT_{OI} the corresponding deterministic class.

Corollary 42. $\text{MT}_{\text{OI}}\subseteq\text{I-PTT}$ *and* $\text{dMT}_{\text{OI}}\subseteq\text{I-dPTT}$.

The inclusions are proper because for every OI macro tree transduction the height of the output tree is exponentially bounded by the height of the input tree [22, Theorem 3.24], whereas it is not difficult to construct a deterministic I-PTT $\mathcal{M}$ with input alphabet $\{\sigma,e\}$, where $\text{rank}(\sigma)=2$ and $\text{rank}(e)=0$, such that the height of the output tree is exponential in the *size* of the input tree. The transducer $\mathcal{M}$ is similar to the I-PTT $\mathcal{M}_{\text{sib}}$ of Example 2, viewing the nodes of the input tree as large cities that are ordered by document order; thus, the number of itineraries is indeed exponential in the size of the input tree. Note that by [19, Corollary 7.2] and [22, Theorem 6.18], dMT_{OI} properly contains the class DMSOT of deterministic MSO *definable tree transductions* (see also [10, Section 8]). Note also that, since dB is properly contained in dMT_{OI} by [22, Corollary 6.16], the second part of Corollary 42 strengthens the second part of Theorem 18. It is open whether or not B is contained in MT_{OI}.

We now turn to the inclusion I-PFT $\subseteq$ TL. To prove that, we need a normal form for I-PFT's. We say that a rule of an I-PFT is *initial* if the state in its left-hand side is an initial state. We define an I-PFT $\mathcal{M}=(\Sigma,\Delta,Q,Q_0,C,\varnothing,C_{\text{i}},R,0)$ with $C=C_{\text{i}}$ to be in *normal form* if its rules satisfy the following five requirements:

(1) Initial states do not appear in the right-hand side of a rule.

(2) All initial rules are of the form $\langle q_0,\sigma,0,\varnothing\rangle\to\langle q,\text{drop}_c\rangle$ for some $q_0\in Q_0$, $\sigma\in\Sigma$, $q\in Q\setminus Q_0$, and $c\in C$. Intuitively, $\mathcal{M}$ starts its computation by dropping a pebble on the root of the input tree.

(3) All non-initial rules have a left-hand side of the form $\langle q,\sigma,j,\{c\}\rangle$ with $c\in C$. Intuitively, $\mathcal{M}$ always observes the topmost pebble, i.e., that pebble is always at the position of the head.

(4) All non-initial non-output rules have a right-hand side $\langle q',\alpha\rangle$ with $q'\in Q\setminus Q_0$ and $\alpha=\text{stay}$ or $\alpha=\mu;\text{drop}_c$ or $\alpha=\text{lift}_c;\mu$ where $c\in C$ and $\mu\in\{\text{up},\text{stay}\}\cup\{\text{down}_i\mid i\in[1,mx_\Sigma]\}$. We will identify $\text{stay};\text{drop}_c$ with drop_c and $\text{lift}_c;\text{stay}$ with lift_c. Intuitively, to force that $\mathcal{M}$ always observes the topmost pebble, $\mathcal{M}$ always drops a pebble after moving, and always moves after lifting a pebble. Note that, in a successful computation, $\mathcal{M}$ never lifts the pebble that it dropped with an initial rule.

(5) There is a function δ from C to $\{\text{up},\text{stay}\}\cup\{\text{down}_i\mid i\in[1,mx_\Sigma]\}$ such that (i) if a rule of $\mathcal{M}$ has right-hand side $\langle q',\text{lift}_c;\mu\rangle$, then $\mu=\delta(c)$, and (ii) for every rule $\langle q,\sigma,j,\{d\}\rangle\to\langle q',\mu;\text{drop}_c\rangle$ of $\mathcal{M}$, if $\mu=\text{up}$ then $\delta(c)=\text{down}_j$, if $\mu=\text{stay}$ then $\delta(c)=\text{stay}$, and if $\mu=\text{down}_i$ then $\delta(c)=\text{up}$. Intuitively this means that $\mathcal{M}$, after lifting a pebble, always knows where to find the new topmost pebble.

This ends the definition of normal form. Obviously, it can also be defined for I-PTT's and for I-PTA's. The I-PTA in normal form can be viewed as a reformulation of the two-way backtracking pushdown tree automaton of [51]. The I-PTT in normal form can be viewed as a reformulation of the RT(P(S))-transducer of [13,23], where S is the storage type Tree-walk of [13].[22]

Lemma 43. *For every* I-PFT $\mathcal{M}$ *an equivalent* I-PFT $\mathcal{M}'$ *in normal form can be constructed. If* $\mathcal{M}$ *is deterministic, then so is* $\mathcal{M}'$*. The same holds for* I-PTT*.*

Proof. The idea of the construction is a simplified version of the one in the proof of Theorem 16, where "beads" are used to cover the shortest path between the head and the topmost pebble. Assuming that the I-PTA $\mathcal{A}$ in that proof starts by dropping a pebble on the root (which is never lifted), the constructed I-PTA $\mathcal{A}'$ satisfies the above requirements on the rules. To show the details, we will repeat that construction, in a simplified form. Here, the only information a bead has to carry is the position of the previous pebble or bead. Moreover, we do not have to drop a bead on the position of the topmost pebble.

Let $\mathcal{M}$ be an I-PFT with colour set C. We may obviously assume that $\mathcal{M}$ already satisfies the first two requirements above. We construct $\mathcal{M}'$ with the same states and initial states as $\mathcal{M}$, and with the colour set $C \cup B$ where $B = \{\text{up}\} \cup \{\text{down}_i \mid [1, mx_\Sigma]\}$. The function δ of requirement (5) is defined by $\delta(d) = d$ for every $d \in B$, and $\delta(c) = \text{stay}$ for every $c \in C$. The rules of $\mathcal{M}'$ are obtained from those of $\mathcal{M}$ as follows. The initial rules of $\mathcal{M}$ are also rules of $\mathcal{M}'$.

If $\langle q, \sigma, j, \varnothing\rangle \to \langle q', \text{up}\rangle$ is a rule of $\mathcal{M}$, then $\mathcal{M}'$ has the rules $\langle q, \sigma, j, \{\text{up}\}\rangle \to \langle q', \text{lift}_{\text{up}}; \text{up}\rangle$ and $\langle q, \sigma, j, \{\text{down}_i\}\rangle \to \langle q', \text{up}; \text{drop}_{\text{down}_j}\rangle$ for every i. Also, if $\langle q, \sigma, j, \{c\}\rangle \to \langle q', \text{up}\rangle$ is a rule of $\mathcal{M}$, then $\mathcal{M}'$ has the rule $\langle q, \sigma, j, \{c\}\rangle \to \langle q', \text{up}; \text{drop}_{\text{down}_j}\rangle$.

Similarly, if $\langle q, \sigma, j, \varnothing\rangle \to \langle q', \text{down}_i\rangle$ is a rule of $\mathcal{M}$, then $\mathcal{M}'$ has the rules $\langle q, \sigma, j, \{\text{down}_i\}\rangle \to \langle q', \text{lift}_{\text{down}_i}; \text{down}_i\rangle$ and $\langle q, \sigma, j, \{\mu\}\rangle \to \langle q', \text{down}_i; \text{drop}_{\text{up}}\rangle$ for every $\mu \in \{\text{up}\} \cup \{\text{down}_k \mid k \neq i\}$. Also, if $\langle q, \sigma, j, \{c\}\rangle \to \langle q', \text{down}_i\rangle$ is a rule of $\mathcal{M}$, then $\mathcal{M}'$ has the rule $\langle q, \sigma, j, \{c\}\rangle \to \langle q', \text{down}_i; \text{drop}_{\text{up}}\rangle$.

The remaining rules of $\mathcal{M}$ (viz. rules with right-hand side $\langle q', \text{stay}\rangle$, output rules, rules that lift, and non-initial rules that drop) are treated as follows. If $\langle q, \sigma, j, \varnothing\rangle \to \zeta$ is such a rule of $\mathcal{M}$, then $\mathcal{M}'$ has the rules $\langle q, \sigma, j, \{\mu\}\rangle \to \zeta$ for every bead $\mu \in B$. If $\langle q, \sigma, j, \{c\}\rangle \to \zeta$ is such a rule of $\mathcal{M}$, then it is also a rule of $\mathcal{M}'$.

It should be clear that $\mathcal{M}'$ is equivalent to $\mathcal{M}$. Whenever $\mathcal{M}$ observes the topmost pebble c, so does $\mathcal{M}'$. Whenever $\mathcal{M}$ does not observe c, $\mathcal{M}'$ observes a bead that indicates the direction of the topmost pebble. Note that if $\mathcal{M}'$ lifts pebble c of $\mathcal{M}$, the new topmost pebble/bead is always at the same position, because when c was dropped $\mathcal{M}'$ was observing the topmost pebble/bead. □

The TL program that we will construct to simulate a given I-PFT $\mathcal{M}$ will use MSO formulas $\varphi(x)$ and $\psi(x, y)$ that closely resemble the tests and instructions in the left-hand and right-hand sides of the rules of $\mathcal{M}$, respectively. Those tests and instructions are "local" in the sense that they only concern the node x, its parent, and its children. Thus, we say that a TL program $\mathcal{P}$ is *local* if in the left-hand side of a rule it only uses a formula $\varphi_{\sigma,j}(x)$ for $\sigma \in \Sigma$ and $j \in [0, mx_\Sigma]$, where $\varphi_{\sigma,0}(x) \equiv \text{lab}_\sigma(x) \wedge \text{root}(x)$ and $\varphi_{\sigma,j}(x) \equiv \text{lab}_\sigma(x) \wedge \text{child}_j(x)$ for $j \neq 0$, and in the right-hand side of that rule it only uses the formulas $\text{up}(x, y)$ (provided $j \neq 0$), $\text{stay}(x, y)$, and $\text{down}_i(x, y)$ for $i \in [1, \text{rank}_\Sigma(\sigma)]$.[23] Thus, $\mathcal{P}$ also satisfies restriction (R2) in the definition of a ranked TL program. Note that macro tree transducers, as defined before Corollary 42, are local ranked TL programs. The classes of transductions realized by local TL programs will be decorated with a subscript ℓ.

Lemma 44. I-PFT $\subseteq$ TL$_\ell$ *and* I-dPFT $\subseteq$ dTL$_\ell$*. Moreover,* I-PTT $\subseteq$ TL$_{\ell r}$ *and* I-dPTT $\subseteq$ dTL$_{\ell r}$*.*

Proof. Let $\mathcal{M} = (\Sigma, \Delta, Q, Q_0, C, \varnothing, C_\text{i}, R, 0)$ with $C = C_\text{i}$ be an I-PFT in normal form. We construct a TL program $\mathcal{P}$ that is equivalent to $\mathcal{M}$. The set of states of $\mathcal{P}$ is

$$Q_0 \cup ((Q \setminus Q_0) \times C) \cup \{q_\perp\}.$$

Each initial state has rank 0, each pair $\langle q, c\rangle$ has rank $\#(Q \setminus Q_0)$, and $q_\perp$ has rank 0. The set of initial states of $\mathcal{P}$ is Q_0. The rules of $\mathcal{P}$ are defined as follows, where we denote a state $\langle q, c\rangle$ as q^c. Let $Q \setminus Q_0 = \{q_1, \ldots, q_n\}$ where we fix the order $q_1, \ldots, q_n$.

First, if $\langle q_0, \sigma, 0, \varnothing\rangle \to \langle q, \text{drop}_c\rangle$ is an initial rule of $\mathcal{M}$, then $\mathcal{P}$ has the rule $\langle q_0, \varphi_{\sigma,0}(x)\rangle \to \langle q^c, \text{stay}(x, y)\rangle(\perp, \ldots, \perp)$, where $\perp$ abbreviates $\langle q_\perp, \text{stay}(x, y)\rangle$. There are no rules of $\mathcal{P}$ with $q_\perp$ in the left-hand side.

Second, let $\langle q, \sigma, j, \{c\}\rangle \to \zeta$ be a (non-initial) rule of $\mathcal{M}$ that does not contain a drop- or lift-instruction. Thus, ζ is of the form $\langle p, \text{stay}\rangle$, $\langle p_1, \text{stay}\rangle\langle p_2, \text{stay}\rangle$, $\delta(\langle p, \text{stay}\rangle)$, or ε, with $p, p_1, p_2 \in Q$ and $\delta \in \Delta$.[24] Then $\mathcal{P}$ has the rule $\langle q^c, \varphi_{\sigma,j}(x)\rangle(z_1, \ldots, z_n) \to \zeta'$, where ζ' is obtained from ζ by replacing every $\langle p, \text{stay}\rangle$ by $\langle p^c, \text{stay}(x, y)\rangle(z_1, \ldots, z_n)$.

[22] See also [20, Section 3.3] where the TT is related to the RT(S)-transducer for S = Tree-walk.

[23] We recall that $\text{root}(x) \equiv \neg\exists z(\text{down}(z, x))$, $\text{child}_i(x) \equiv \exists z(\text{down}_i(z, x))$, $\text{up}(x, y) \equiv \text{down}(y, x)$, and $\text{stay}(x, y) \equiv x = y$.

[24] In the case where $\mathcal{M}$ is an I-PTT, ζ is of the form $\langle p, \text{stay}\rangle$ or $\delta(\langle p_1, \text{stay}\rangle, \ldots, \langle p_m, \text{stay}\rangle)$.

Third, let $\langle q, \sigma, j, \{d\}\rangle \to \langle p, \mu; \mathrm{drop}_c\rangle$ be a rule of $\mathcal{M}$. Note that for every $\mu \in \{\mathrm{up}, \mathrm{stay}\} \cup \{\mathrm{down}_i \mid i \in [1, mx_\Sigma]\}$, there is an MSO formula $\mu(x, y)$. Then $\mathcal{P}$ has the rule

$$\langle q^d, \varphi_{\sigma,j}(x)\rangle(z_1, \ldots, z_n) \to \langle p^c, \mu(x, y)\rangle(s_1, \ldots, s_n)$$

where $s_i = \langle q_i^d, \mathrm{stay}(x, y)\rangle(z_1, \ldots, z_n)$ for every $i \in [1, n]$; thus, the rule is

$$\langle q^d, \varphi_{\sigma,j}(x)\rangle(z_1, \ldots, z_n) \to \langle p^c, \mu(x, y)\rangle(\langle q_1^d, \mathrm{stay}(x, y)\rangle(z_1, \ldots, z_n), \ldots, \langle q_n^d, \mathrm{stay}(x, y)\rangle(z_1, \ldots, z_n)).$$

Fourth and final, if $\langle q, \sigma, j, \{c\}\rangle \to \langle q_i, \mathrm{lift}_c; \mu\rangle$ is a rule of $\mathcal{M}$, then $\mathcal{P}$ has the rule

$$\langle q^c, \varphi_{\sigma,j}(x)\rangle(z_1, \ldots, z_n) \to z_i.$$

Intuitively, $\mathcal{P}$ is in state q^c when $\mathcal{M}$ is in state q and the topmost pebble of $\mathcal{M}$ is c. The parameter z_i of q^c contains the continuation of $\mathcal{M}$'s computation just after pebble c is lifted and $\mathcal{M}$ goes into state q_i. At the moment that $\mathcal{M}$ drops pebble c, $\mathcal{P}$ does not know what the state q_i of $\mathcal{M}$ will be after lifting c and thus prepares the continuation for every possible state. The correct continuation is then chosen by $\mathcal{P}$ when it simulates $\mathcal{M}$'s lifting of c. Note that due to requirement (5) of the normal form, when $\mathcal{M}$ lifts a pebble, it returns to the same node where it decided to drop the pebble (at that node, or at the parent or at one of the children of that node).

Formally, we define a mapping 'rep' from the output forms of $\mathcal{M}$ (except the initial one) to those restricted output forms of $\mathcal{P}$ of which the outermost nodes are labeled by a symbol from Δ or by a configuration $\langle q, u\rangle$ where q is a state of $\mathcal{P}$ (thus, they are not labeled by a configuration $\langle p, u\rangle$ where p is a selector of $\mathcal{P}$). As in the proof of Lemma 39, the Δ-labeled part of the output form is not changed. Thus, it remains to define 'rep' for the configurations of $\mathcal{M}$ that contain non-initial states, which are of the form $\langle q, u, \pi(u, c)\rangle$ because the topmost pebble is always at the position of the head. We define $\mathrm{rep}(q, u, \pi(u, c)) = \langle q^c, u\rangle\, \mathrm{rep}'(\pi)$, where rep' maps the pebble stacks of $\mathcal{M}$ to sequences of output forms of $\mathcal{P}$, recursively as follows: $\mathrm{rep}'(\varepsilon) = (\bot, \ldots, \bot)$ and $\mathrm{rep}'(\pi(u, c)) = (s_1, \ldots, s_n)$ where $s_i = \langle\langle q_i^c, \mathrm{stay}(x, y)\rangle, u\rangle\, \mathrm{rep}'(\pi)$ for every $i \in [1, n]$. Note that 'rep' is injective.

It is now straightforward to prove, for every $q \in Q \setminus Q_0$, every $c \in C$, every input tree t, and every output form s of $\mathcal{P}$ (restricted as described above), that $\langle q^c, \mathrm{root}_t\rangle(\bot, \ldots, \bot) \Rightarrow^*_{t,\mathcal{P}} s$ if and only if there exists an output form s' of $\mathcal{M}$ such that $\langle q, \mathrm{root}_t, (\mathrm{root}_t, c)\rangle \Rightarrow^*_{t,\mathcal{M}} s'$ and $\mathrm{rep}(s') = s$. Since 'rep' is injective, s' is in fact unique. Note that each computation step of $\mathcal{M}$ is simulated by two (or three) computation steps of $\mathcal{P}$, where the second (and third) step executes a selector to satisfy the restriction on the output forms of $\mathcal{P}$. Due to its special form, the execution of such a selector $\psi(x, y)$ changes the label $\langle\langle q', \psi(x, y)\rangle, u\rangle$ of a node of the output form into $\langle q', u'\rangle$ where u' is the unique node of the input tree for which $\psi(u, u')$ holds.

Taking into account the initial rules of $\mathcal{M}$, it should be clear that the above equivalence proves that $\tau_{\mathcal{P}} = \tau_{\mathcal{M}}$. □

Example 45. We illustrate Lemma 44 with the deterministic I-PTT $\mathcal{M}_{\mathrm{sib}}$ of Example 2. We first construct an I-PTT $\mathcal{M}'_{\mathrm{sib}}$ in normal form that is equivalent to $\mathcal{M}_{\mathrm{sib}}$. We also allow tuples $\langle q', \mathrm{lift}_d; \mu\rangle$ in the output rules for any colour d, which can easily be handled too. The transducer $\mathcal{M}'_{\mathrm{sib}}$ has a new initial state q_{in}, in which it drops pebble $\odot$ on the root, which also serves as the pebble 'up'. The pebble 'down_1' is denoted by $\downarrow$. The normal form function δ is defined by $\delta(\odot) = \mathrm{up}$, $\delta(\downarrow) = \mathrm{down}_1$, and $\delta(c) = \mathrm{stay}$ for $c \in \{0, 1\}$. There are new states $\overline{q}_0$ and $\overline{q}_1$ in which $\mathcal{M}'_{\mathrm{sib}}$ moves up, drops pebble $\downarrow$, and goes into the corresponding unbarred state. Thus the rules for them are

$$\rho_{c,d} : \langle \overline{q}_c, \sigma, 1, \{d\}\rangle \to \langle q_c, \mathrm{up}; \mathrm{drop}_\downarrow\rangle$$

with $\sigma \in \Sigma$ and $d \in \{\odot, \downarrow, 0, 1\}$. The other rules (with $c = 1$ or $i = 0$ in rule ρ_4 as usual) are

$$\rho_0 : \langle q_{\mathrm{in}}, \sigma_1, 0, \varnothing\rangle \to \langle q_{\mathrm{start}}, \mathrm{drop}_\odot\rangle$$
$$\rho_1 : \langle q_{\mathrm{start}}, \sigma_1, j, \{\odot\}\rangle \to \langle q_{\mathrm{start}}, \mathrm{down}_1; \mathrm{drop}_\odot\rangle$$
$$\rho_2 : \langle q_{\mathrm{start}}, \sigma_0, 1, \{\odot\}\rangle \to \langle \overline{q}_1, \mathrm{stay}\rangle$$
$$\rho_3 : \langle q_0, \lambda_0, 1, \{\downarrow\}\rangle \to \langle \overline{q}_0, \mathrm{stay}\rangle$$
$$\rho_4 : \langle q_c, \lambda_i, 1, \{\downarrow\}\rangle \to \langle \overline{q}_i, \mathrm{drop}_c\rangle$$
$$\rho_5 : \langle q_c, \sigma_1, 0, \{\downarrow\}\rangle \to r(\langle q_{\mathrm{out}}, \mathrm{stay}\rangle, \langle q_{\mathrm{next}}, \mathrm{lift}_\downarrow; \mathrm{down}_1\rangle)$$
$$\rho_6 : \langle q_{\mathrm{out}}, \sigma_1, 0, \{\downarrow\}\rangle \to \sigma_1(\langle q_{\mathrm{out}}, \mathrm{lift}_\downarrow; \mathrm{down}_1\rangle)$$
$$\rho_7 : \langle q_{\mathrm{out}}, \sigma_1, 1, \{\downarrow\}\rangle \to \langle q_{\mathrm{out}}, \mathrm{lift}_\downarrow; \mathrm{down}_1\rangle$$
$$\rho_8 : \langle q_{\mathrm{out}}, \sigma_1, 1, \{c\}\rangle \to \sigma_1(\langle q_{\mathrm{out}}, \mathrm{lift}_c\rangle)$$
$$\rho_9 : \langle q_{\mathrm{out}}, \sigma_0, 1, \{\odot\}\rangle \to \sigma_0$$

$$\rho_{10} : \langle q_{\text{next}}, \sigma_1, 1, \{\downarrow\}\rangle \to \langle q_{\text{next}}, \text{lift}_{\downarrow}; \text{down}_1\rangle$$
$$\rho_{11} : \langle q_{\text{next}}, \sigma_1, 1, \{c\}\rangle \to \langle \overline{q}_c, \text{lift}_c\rangle$$
$$\rho_{12} : \langle q_{\text{next}}, \sigma_0, 1, \{\odot\}\rangle \to e$$

We now construct the deterministic TL program $\mathcal{P}$ corresponding to $\mathcal{M}'_{\text{sib}}$. The states of $\mathcal{M}'_{\text{sib}}$ after lifting $\downarrow$ are q_{out} and q_{next}. Thus, the states of $\mathcal{P}$ that are active when the topmost pebble is $\downarrow$ only need two parameters z_1, z_2 corresponding to q_{out} and q_{next}. Similarly, the states of $\mathcal{P}$ that are active when the topmost pebble is c only need two parameters z_1, z_2 corresponding to q_{out} and $\overline{q}_c$. The states of $\mathcal{P}$ that are active when the topmost pebble is $\odot$ do not need parameters, because $\odot$ is never lifted. Program $\mathcal{P}$ has the states q_{in}, $q^{\odot}_{\text{start}}$, $q^{\downarrow}_c$, $\overline{q}^{d}_c$, q^{d}_{out}, and q^{d}_{next}, where $c \in \{0,1\}$ and $d \in \{\odot, \downarrow, 0, 1\}$. Note that the state $q_{\perp}$ is superfluous. The initial state q_{in} and all states with superscript $\odot$ have rank 0, and the other states have rank 2.

Program $\mathcal{P}$ has the following rule corresponding to rule $r_{c,d}$ of $\mathcal{M}'_{\text{sib}}$, with $d \neq \odot$:

$$\rho_{c,d} : \langle \overline{q}^{d}_c, \varphi_{\sigma,1}(x)\rangle(z_1, z_2) \to \langle q^{\downarrow}_c, \text{up}(x,y)\rangle(\langle q^{d}_{\text{out}}, \text{stay}(x,y)\rangle(z_1,z_2), \langle q^{d}_{\text{next}}, \text{stay}(x,y)\rangle(z_1,z_2))$$

and for $d = \odot$ the same rule without the parameters (z_1, z_2). The other rules of $\mathcal{P}$ are

$$\rho_0 : \langle q_{\text{in}}, \varphi_{\sigma_1,0}(x)\rangle \to \langle q^{\odot}_{\text{start}}, \text{stay}(x,y)\rangle$$
$$\rho_1 : \langle q^{\odot}_{\text{start}}, \varphi_{\sigma_1,j}(x)\rangle \to \langle q^{\odot}_{\text{start}}, \text{down}_1(x,y)\rangle$$
$$\rho_2 : \langle q^{\odot}_{\text{start}}, \varphi_{\sigma_0,1}(x)\rangle \to \langle \overline{q}^{\odot}_1, \text{stay}(x,y)\rangle$$
$$\rho_3 : \langle q^{\downarrow}_0, \varphi_{\lambda_0,1}(x)\rangle(z_1,z_2) \to \langle \overline{q}^{\downarrow}_0, \text{stay}(x,y)\rangle(z_1,z_2)$$
$$\rho_4 : \langle q^{\downarrow}_c, \varphi_{\lambda_i,1}(x)\rangle(z_1,z_2) \to \langle \overline{q}^{c}_i, \text{stay}(x,y)\rangle(\langle q^{\downarrow}_{\text{out}}, \text{stay}(x,y)\rangle(z_1,z_2), \langle \overline{q}^{\downarrow}_c, \text{stay}(x,y)\rangle(z_1,z_2))$$
$$\rho_5 : \langle q^{\downarrow}_c, \varphi_{\sigma_1,0}(x)\rangle(z_1,z_2) \to r(\langle q^{\downarrow}_{\text{out}}, \text{stay}(x,y)\rangle(z_1,z_2), z_2)$$
$$\rho_6 : \langle q^{\downarrow}_{\text{out}}, \varphi_{\sigma_1,0}(x)\rangle(z_1,z_2) \to \sigma_1(z_1)$$
$$\rho_7 : \langle q^{\downarrow}_{\text{out}}, \varphi_{\sigma_1,1}(x)\rangle(z_1,z_2) \to z_1$$
$$\rho_8 : \langle q^{c}_{\text{out}}, \varphi_{\sigma_1,1}(x)\rangle(z_1,z_2) \to \sigma_1(z_1)$$
$$\rho_9 : \langle q^{\odot}_{\text{out}}, \varphi_{\sigma_0,1}(x)\rangle \to \sigma_0$$
$$\rho_{10} : \langle q^{\downarrow}_{\text{next}}, \varphi_{\sigma_1,1}(x)\rangle(z_1,z_2) \to z_2$$
$$\rho_{11} : \langle q^{c}_{\text{next}}, \varphi_{\sigma_1,1}(x)\rangle(z_1,z_2) \to z_2$$
$$\rho_{12} : \langle q^{\odot}_{\text{next}}, \varphi_{\sigma_0,1}(x)\rangle \to e$$

Applying rule ρ_6 to the right-hand side of rule ρ_5, we obtain the rule

$$\rho'_5 : \langle q^{\downarrow}_c, \varphi_{\sigma_1,0}(x)\rangle(z_1,z_2) \to r(\sigma_1(z_1), z_2)$$

which is in fact rule ρ_5 of program $\mathcal{P}_{\text{sib}}$ of Example 38, if we identify the states $q^{\downarrow}_c$ and q_c. Rules ρ_0 and ρ_1 of $\mathcal{P}$ correspond to rule ρ_1 of $\mathcal{P}_{\text{sib}}$ in an obvious way (with $q^{\odot}_{\text{start}}$ and q_{start} identified). Since program $\mathcal{P}$ is deterministic, and its states generate trees (rather than forests), we can also apply rules $\rho_7 - \rho_{12}$ to the right-hand side of rule $\rho_{c,d}$, and we obtain the rules

$$\rho'_{c,\downarrow} : \langle \overline{q}^{\downarrow}_c, \varphi_{\sigma_1,1}(x)\rangle(z_1,z_2) \to \langle q^{\downarrow}_c, \text{up}(x,y)\rangle(z_1,z_2)$$
$$\rho'_{i,c} : \langle \overline{q}^{c}_i, \varphi_{\sigma_1,1}(x)\rangle(z_1,z_2) \to \langle q^{\downarrow}_i, \text{up}(x,y)\rangle(\sigma_1(z_1), z_2)$$
$$\rho'_{c,\odot} : \langle \overline{q}^{\odot}_c, \varphi_{\sigma_0,1}(x)\rangle \to \langle q^{\downarrow}_c, \text{up}(x,y)\rangle(\sigma_0, e)$$

Applying $\rho'_{1,\odot}$ to the right-hand side of ρ_2 we obtain

$$\rho'_2 : \langle q^{\odot}_{\text{start}}, \varphi_{\sigma_0,1}(x)\rangle \to \langle q^{\downarrow}_1, \text{up}(x,y)\rangle(\sigma_0, e)$$

which is rule ρ_2 of $\mathcal{P}_{\text{sib}}$. Applying $\rho'_{0,\downarrow}$ to the right-hand side of ρ_3 we obtain

$$\rho'_3 : \langle q^{\downarrow}_0, \varphi_{\lambda_0,1}(x)\rangle(z_1,z_2) \to \langle q^{\downarrow}_0, \text{up}(x,y)\rangle(z_1,z_2)$$

which is rule ρ_3 of $\mathcal{P}_{\text{sib}}$. Finally, applying rules $\rho'_{i,c}$, ρ_7, and $\rho'_{c,\downarrow}$ to the selectors in the right-hand side of rule ρ_4, respectively, we obtain the right-hand side $\langle q_i, \text{up}(x,y)\rangle(\lambda_i(z_1), \langle q_c, \text{up}(x,y)\rangle(z_1,z_2))$ of rule ρ_4 of $\mathcal{P}_{\text{sib}}$. Thus, program $\mathcal{P}$ is essentially the same as program $\mathcal{P}_{\text{sib}}$ of Example 38. □

Lemmas 39 and 44 together prove that TL programs have the same expressive power as I-PFT's. Additionally, they prove that for every TL program there is an equivalent local one.

Theorem 46. $\mathsf{TL} = \mathsf{TL}_\ell = \text{I-PFT}$ *and* $\mathsf{dTL} = \mathsf{dTL}_\ell = \text{I-dPFT}$. *Moreover,* $\mathsf{TL}_\mathsf{r} = \mathsf{TL}_{\ell\mathsf{r}} = \text{I-PTT}$ *and* $\mathsf{dTL}_\mathsf{r} = \mathsf{dTL}_{\ell\mathsf{r}} = \text{I-dPTT}$.

Since local TL programs satisfy restriction (R2) in the definition of a ranked TL program, the equation $\mathsf{TL} = \mathsf{TL}_\ell$ shows that the pattern matching aspect that is involved in the execution of selectors, can be viewed as an extended feature. Moreover, even the "jumps" in the execution of selectors, and the arbitrary MSO head tests in the left-hand sides of rules, can be viewed as extended features of TL_ℓ.

Note that for $\mathrm{TL}_\ell^{\mathrm{DB}}$ programs the construction in the proof of Lemma 39 can easily be simplified to one that takes polynomial time and that results in an I-PFT that does not use MSO tests. That implies that the inverse type inference problem for such programs is solvable in 2-fold exponential time, and hence typechecking can be done in 3-fold exponential time (cf. Theorem 41).

The local ranked TL program is an obvious reformulation of the "macro tree-walking transducer" (2-mtt) of [37]. The inclusion $\mathsf{TL}_\mathsf{r} \subseteq \mathsf{TL}_{\ell\mathsf{r}}$ is a (slightly stronger) version of [37, Theorem 5]. Moreover, the local ranked TL program is the same as the "0-pebble macro tree transducer" of [20, Section 5.1] and it is the CFT(S)-transducer of [23] for the storage type $S =$ Tree-walk, both of which generalize the macro attributed tree transducer of [26,35] which additionally satisfies a noncircularity condition. It follows from Lemma 4 and Theorem 46 that $\mathsf{TL}_{\ell\mathsf{r}} \subseteq \mathsf{TT}^2$, which was stated as an open problem in [20, Section 8] (where $\mathsf{TL}_{\ell\mathsf{r}}$ and TT are denoted 0-PMTT and 0-PTT, respectively). In view of Lemma 43, the equality $\mathsf{TL}_{\ell\mathsf{r}} =$ I-PTT is the same as the equality CFT(S) = RT(P(S)) of [23, Theorem 5.24] for $S =$ Tree-walk, and similarly for the deterministic case.

13. A TL program in XSLT

In Tables 1 and 2 we listed a possible input document and the resulting output document for the I-PTT $\mathcal{M}_{\mathrm{sib}}$ of Example 2. In this section we present in Table 7 an XSLT 1.0 program with the same structure as the TL program $\mathcal{P}_{\mathrm{sib}}$ of Example 38. In what follows we comment on the XSLT program and its relationship to $\mathcal{P}_{\mathrm{sib}}$, abbreviated as $\mathcal{P}$.

The first rule ρ_1 of $\mathcal{P}$ corresponds to the first template of the XSLT program: this template initializes the algorithm by matching the root of the input document, jumping to the leaf by selecting the final stop, and invoking named template `start` on it.

The second rule ρ_2 of $\mathcal{P}$ corresponds to template `start`: it moves up, using the `apply-templates` instruction which selects the parent, and thus invokes the third template on that parent, which is the only template for nonroot document elements. It invokes that template with the appropriate parameters: `nextstoplarge` is 1 because `large = 1` for the final stop, `stoplist` is a list containing only the final stop, and `additionalresults` is the single element `<endofresults />`.

The remaining rules of $\mathcal{P}$ correspond to the third template, which is applied to all nonfinal stops. That template takes a partial stop list `stoplist` (from the current stop to the final stop) and generates all allowed ways to complete that stop list using the stops between the current one and the initial one. Nested below the deepest element of the output, it includes the result tree fragment passed in `additionalresults`. The third template has three parameters:

`nextstoplarge`: a boolean indicating whether or not the "next" stop (i.e., the stop at the front of `stoplist`) is a large stop; it corresponds to states q_1 and q_0 in $\mathcal{P}$, respectively,
`stoplist`: a partial list of stops (taken from the current stop to the final stop) for which this template will recursively generate all (allowed) ways in which it can be completed; it corresponds to parameter z_1 in $\mathcal{P}$,
`additionalresults`: results that are to be appended to the results that this template generates; it corresponds to parameter z_2 in $\mathcal{P}$,

where both `stoplist` and `additionalresults` are of type 'result tree fragment'.

Corresponding to rule ρ_5 of $\mathcal{P}$, the third template, when invoked on the initial stop (for which `initial = 1`), has computed a complete stop list (after adding this stop) and outputs it: it copies the initial stop and nests the remainder of the stop list (i.e., the value of its parameter `stoplist`) in it; it also includes the additional results (i.e., the value of parameter `additionalresults`).

Corresponding to rules ρ_3 and ρ_4 of $\mathcal{P}$, the third template, when invoked on an intermediate stop (for which `not(initial = 1)`), has not yet computed a complete stop list, and now calculates all allowed ways to complete it. Intuitively, it computes two result sets: one that *does not* add the current stop, and one that *does*. They are combined by passing the first result set as "additional results" to the calculation of the second one. Thus, the third template starts by computing the first result set, and, to abbreviate the remaining code, it assigns its value to a variable called `results`. In rules ρ_3 and ρ_4 of $\mathcal{P}$ this result set corresponds to the selector $\langle q_c, \mathrm{up}(x, y)\rangle(z_1, z_2)$, where $c = 0$ in ρ_3. In the case that `large = 0` and `nextstoplarge = 0`, we are not allowed to stop here because that would create two consecutive small stops. Thus the template only outputs the results that it just stored in the variable (corresponding to rule ρ_3 of $\mathcal{P}$). In the case that

Table 7
XSLT Program.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml"/>

    <xsl:template match="/">
        <xsl:for-each select="//stop[@final=1]">
            <xsl:call-template name="start" />
        </xsl:for-each>
    </xsl:template>

    <xsl:template name="start">
        <xsl:apply-templates select="parent::stop">
            <xsl:with-param name="nextstoplarge" select="@large" />
            <xsl:with-param name="stoplist">
                <xsl:copy>
                    <xsl:copy-of select="attribute::*" />
                </xsl:copy>
            </xsl:with-param>
            <xsl:with-param name="additionalresults">
                <endofresults />
            </xsl:with-param>
        </xsl:apply-templates>
    </xsl:template>

    <xsl:template match="stop">
        <xsl:param name="nextstoplarge" />
        <xsl:param name="stoplist" />
        <xsl:param name="additionalresults" />
        <xsl:if test="@initial = 1">
            <result>
                <xsl:copy>
                    <xsl:copy-of select="attribute::*" />
                    <xsl:copy-of select="$stoplist" />
                </xsl:copy>
                <xsl:copy-of select="$additionalresults" />
            </result>
        </xsl:if>
        <xsl:if test="not(@initial = 1)">
            <xsl:variable name="results">
                <xsl:apply-templates select="parent::stop">
                    <xsl:with-param name="nextstoplarge" select="$nextstoplarge" />
                    <xsl:with-param name="stoplist" select="$stoplist" />
                    <xsl:with-param name="additionalresults" select="$additionalresults" />
                </xsl:apply-templates>
            </xsl:variable>
            <xsl:if test="@large = 1 or $nextstoplarge = 1">
                <xsl:apply-templates select="parent::stop">
                    <xsl:with-param name="nextstoplarge" select="@large" />
                    <xsl:with-param name="stoplist">
                        <xsl:copy>
                            <xsl:copy-of select="attribute::*" />
                            <xsl:copy-of select="$stoplist" />
                        </xsl:copy>
                    </xsl:with-param>
                    <xsl:with-param name="additionalresults" select="$results" />
                </xsl:apply-templates>
            </xsl:if>
            <xsl:if test="@large = 0 and $nextstoplarge = 0">
                <xsl:copy-of select="$results" />
            </xsl:if>
        </xsl:if>
    </xsl:template>

</xsl:stylesheet>
```

`large = 1` or `nextstoplarge = 1`, the template calculates all possible ways to complete the stop list that contain this stop, and includes as additional results those that are stored in the variable (corresponding to rule ρ_4 of $\mathcal{P}$).

14. Data complexity

In this section we show that the transduction of a deterministic PTT $\mathcal{M}$ can be realized in (1-fold) exponential time, in the sense that there is an exponential time algorithm that, for every given input tree t, computes a regular tree grammar G that generates the language $\{\tau_{\mathcal{M}}(t)\}$. If t is in the domain of $\mathcal{M}$, then G can be viewed as a DAG (directed acyclic graph) that defines the output tree $\tau_{\mathcal{M}}(t)$, in the usual sense. Thus, producing the actual output tree would take 2-fold exponential

time. If t is not in the domain of $\mathcal{M}$, then G generates the empty tree language (which can be decided in time linear in the size of G).

Theorem 47. *For every deterministic* PTT $\mathcal{M}$ *there is an exponential time algorithm that, for given input tree* t*, computes a regular tree grammar* G *such that* $L(G) = \{s \mid (t, s) \in \tau_{\mathcal{M}}\}$.

Proof. Let $\mathcal{M} = (\Sigma, \Delta, Q, \{q_0\}, C, C_{\mathrm{v}}, C_{\mathrm{i}}, R, k)$ be a deterministic v_kI-PTT. For an input tree $t \in T_\Sigma$ in the domain of $\mathcal{M}$, let us consider the computation $\langle q_0, \mathrm{root}_t, \varepsilon\rangle \Rightarrow^*_{t,\mathcal{M}} s$, where $s = \tau_{\mathcal{M}}(t)$, and let $\langle q, u, \pi\rangle$ be a configuration of $\mathcal{M}$ that occurs in that computation. We claim that the length of π is at most $N = |Q| \cdot (|C| + 1)^{k+1} \cdot n^{k+2}$, where n is the size of t.

To prove this claim we define, as an auxiliary tool, the nondeterministic v_kI-PTA $\mathcal{A}$ that is obtained from $\mathcal{M}$ by changing every output rule $\langle q, \sigma, j, b\rangle \to \delta(\langle q_1, \mathrm{stay}\rangle, \ldots, \langle q_m, \mathrm{stay}\rangle)$ of $\mathcal{M}$ into the rules $\langle q, \sigma, j, b\rangle \to \langle q_i, \mathrm{stay}\rangle$ for all $i \in [1, m]$. Intuitively, whenever $\mathcal{M}$ branches, $\mathcal{A}$ nondeterministically follows one of those branches. Thus, all computations of $\mathcal{A}$ that start with $\langle q_0, \mathrm{root}_t, \varepsilon\rangle$ are finite. Obviously, $\langle q, u, \pi\rangle$ occurs in such a computation of $\mathcal{A}$. Let $\pi = (v_1, c_1) \cdots (v_m, c_m)$ and suppose that $m > N$. For every $\ell \in [1, m]$ we define $\pi_\ell = (v_1, c_1) \cdots (v_\ell, c_\ell)$. Then there exist configurations $\langle q_\ell, u_\ell, \pi_\ell\rangle$, $\ell \in [1, m]$, such that $\langle q_0, \mathrm{root}_t, \varepsilon\rangle \Rightarrow^*_{t,\mathcal{A}} \langle q_1, u_1, \pi_1\rangle$ and $\langle q_\ell, u_\ell, \pi_\ell\rangle \Rightarrow^*_{t,\mathcal{A}} \langle q_{\ell+1}, u_{\ell+1}, \pi_{\ell+1}\rangle$ for every $\ell \in [1, m-1]$, and such that, moreover, every configuration occurring in the computation $\langle q_\ell, u_\ell, \pi_\ell\rangle \Rightarrow^*_{t,\mathcal{A}} \langle q_{\ell+1}, u_{\ell+1}, \pi_{\ell+1}\rangle$ has a pebble stack with prefix π_ℓ. Due to the choice of m, there exist $i, j \in [1, m]$ with $i < j$ such that $q_i = q_j$, $u_i = u_j$, $(v_i, c_i) = (v_j, c_j)$, and for every $v \in N(t)$ and $c \in C_{\mathrm{v}}$: (v, c) occurs in π_i if and only if (v, c) occurs in π_j. This implies that the computation $\langle q_i, u_i, \pi_i\rangle \Rightarrow^*_{t,\mathcal{A}} \langle q_j, u_j, \pi_j\rangle$ can be repeated arbitrarily many times, leading to an infinite computation of $\mathcal{A}$, which is a contradiction and proves the claim.

We now construct the regular tree grammar G. Its nonterminals are the configurations $\langle q, u, \pi\rangle$ of $\mathcal{M}$ on t such that $|\pi| \le N$. Since N is polynomial in n, the number of nonterminals of G is exponential in n. The initial nonterminal of G is $\langle q_0, \mathrm{root}_t, \varepsilon\rangle$. If $\langle q, u, \pi\rangle \Rightarrow^*_{t,\mathcal{M}} \langle q', u', \pi'\rangle \Rightarrow_{t,\mathcal{M}} \delta(\langle q_1, u', \pi'\rangle, \ldots, \langle q_m, u', \pi'\rangle)$, then $\langle q, u, \pi\rangle \to \delta(\langle q_1, u', \pi'\rangle, \ldots, \langle q_m, u', \pi'\rangle)$ is a rule of G. To decide whether $\langle q', u', \pi'\rangle \Rightarrow_{t,\mathcal{M}} \delta(\langle q_1, u', \pi'\rangle, \ldots, \langle q_m, u', \pi'\rangle)$ it suffices to inspect the output rules of $\mathcal{M}$. To decide whether $\langle q, u, \pi\rangle \Rightarrow^*_{t,\mathcal{M}} \langle q', u', \pi'\rangle$ we construct from $\mathcal{M}$ and t an ordinary pushdown automaton $\mathcal{P}$ that simulates the non-output behaviour of $\mathcal{M}$ on t, as in the query evaluation paragraph at the end of Section 9. Since, as opposed to that paragraph, $\mathcal{M}$ also has visible pebbles, $\mathcal{P}$ should keep track of those pebbles in its finite state. Let Γ be the set of all mappings $\gamma : C_{\mathrm{v}} \to N(t) \cup \{\bot\}$ such that $\#(\{c \in C_{\mathrm{v}} \mid \gamma(c) \neq \bot\}) \le k$. During $\mathcal{P}$'s computation, the mapping γ in its finite state indicates for every visible pebble whether it occurs in the current stack and, if so, on which node it is dropped. Thus, we define $\mathcal{P}$ to have state set $Q \times N(t) \times \Gamma$ and pushdown alphabet $N(t) \times C$. A configuration $\langle q, u, \pi\rangle$ of $\mathcal{M}$ is simulated by the configuration $\mathcal{P}(\langle q, u, \pi\rangle) = \langle p, \pi\rangle$ of $\mathcal{P}$ such that $p = (q, u, \gamma)$ where, for every $c \in C_{\mathrm{v}}$, if $\gamma(c) \in N(t)$ then $(\gamma(c), c)$ occurs in π, and if $\gamma(c) = \bot$ then c does not occur in π. The transitions of the automaton $\mathcal{P}$ are defined in such a way that $\mathcal{P}$ (with the empty string as input) has the same computation steps as $\mathcal{M}$ (without its output rules), i.e., such that $\langle q, u, \pi\rangle \Rightarrow_{t,\mathcal{M}} \langle q', u', \pi'\rangle$ if and only if $\mathcal{P}(\langle q, u, \pi\rangle) \Rightarrow_{\mathcal{P}} \mathcal{P}(\langle q', u', \pi'\rangle)$, where $\Rightarrow_{\mathcal{P}}$ is the computation step relation of $\mathcal{P}$. For instance, let $\mathcal{P}$ be in state (q, u, γ) and let the top element of its stack be (v, c). Let u have label σ and child number j, and let b consist of all $c' \in C_{\mathrm{v}}$ with $\gamma(c') = u$ plus c if $v = u$. If $\langle q, \sigma, j, b\rangle \to \langle q', \mathrm{drop}_d\rangle$ is a rule of $\mathcal{M}$ such that $d \in C_{\mathrm{v}}$, $\gamma(d) = \bot$, and $\#(\{c' \in C_{\mathrm{v}} \mid \gamma(c') \neq \bot\}) < k$, then $\mathcal{P}$ pushes (u, d) on its stack and goes into state (q', u, γ') where $\gamma'(d) = u$ and $\gamma'(c') = \gamma(c')$ for all $c' \neq d$. If $\langle q, \sigma, j, b\rangle \to \langle q', \mathrm{lift}_c\rangle$ is a rule of $\mathcal{M}$ such that $c \in C_{\mathrm{i}}$ and $v = u$, then $\mathcal{P}$ pops (v, c) from its stack and goes into state (q', u, γ). The transitions of $\mathcal{P}$ are defined similarly for the other non-output rules of $\mathcal{M}$. It should be clear that $\mathcal{P}$ can be constructed in time polynomial in n. Since it can be decided in polynomial time for configurations $\langle p, \pi\rangle$ and $\langle p', \pi'\rangle$ of $\mathcal{P}$ whether $\langle p, \pi\rangle \Rightarrow^*_{\mathcal{P}} \langle p', \pi'\rangle$, it can be decided whether $\langle q, u, \pi\rangle \Rightarrow^*_{t,\mathcal{M}} \langle q', u', \pi'\rangle$ in polynomial time. Hence the total time to construct G is exponential. □

Note that the first part of the above proof also shows that for every deterministic PTT the height of the output tree is exponential in the size of the input tree.

A natural question is whether Theorem 47 also holds for forest transducers, i.e., for deterministic PFT's. That is indeed the case (as the reader can easily verify), except that G is not a regular forest grammar, but a forest generating context-free grammar. To be precise, G is a context-free grammar of which every rule is of the form $X_0 \to \delta(X_1)$ or $X_0 \to X_1 X_2$ or $X \to \varepsilon$ where δ is a symbol from an unranked alphabet. If $L(G) = \{f\}$, then G can still be viewed as a DAG that defines the forest f. Thus, in this sense, by Theorem 46, deterministic TL programs can be executed in exponential time, in accordance with the result of [33] that XSLT 1.0 programs can be executed in exponential time.

Another natural question is whether there exist interesting subclasses of PTT's that can be realized in polynomial time. Here we discuss one such subclass. We define a PTT to be *bounded* if there exists $m \in \mathbb{N}$ such that output rules can only be applied when the pebble stack contains at most m pebbles. Intuitively it means that the infinitely many invisible pebbles are mainly used to check MSO properties of the observable configuration. Formally it can either be required as a dynamic property of the (successful) computations of the PTT or be incorporated statically in the semantics of the PTT. We now show that bounded PTT's can be realized in polynomial time, even in the nondeterministic case. This generalizes the result for V-PTT's [41, Proposition 3.8].

Theorem 48. *For every bounded* PTT $\mathcal{M}$ *there is a polynomial time algorithm that, for given input tree* t*, computes a regular tree grammar* G *such that* $L(G) = \{s \mid (t,s) \in \tau_{\mathcal{M}}\}$.

Proof. The construction of G is exactly the same as in the proof of Theorem 47, except that its nonterminals are now the configurations $\langle q, u, \pi\rangle$ of $\mathcal{M}$ on t such that $|\pi| \leq m$.[25] The number of nonterminals of G is therefore polynomial in the size of t, and since the pushdown automaton $\mathcal{P}$ can also be constructed (and tested) in polynomial time, the total time to construct G is polynomial. □

Again, the same result holds for PFT's, taking G to be a forest generating context-free grammar. Note that for a nondeterministic PFT $\mathcal{M}$ and an input tree t, the set $\{s \mid (t,s) \in \tau_{\mathcal{M}}\}$ is not necessarily a regular forest language.

Also, the same result holds for bounded PTT's that use MSO tests on the observable configuration. That is not immediate, because the construction in the proof of Theorem 16 does not preserve boundedness, due to the use of beads. However, it is easy to adapt the construction of the pushdown automaton $\mathcal{P}$ in the proof of Theorem 47 to incorporate the MSO tests of the v_kI-PTT $\mathcal{M}$. In fact, the observable configuration tree $\mathrm{obs}(t,\pi)$ can be constructed from t, from the mapping γ in the state of $\mathcal{P}$, and from the top element of its stack, and then $\mathrm{obs}(t,\pi)$ can be tested in linear time using a deterministic bottom-up finite-state tree automaton. An example of bounded PTT's (with MSO tests) are the pattern matching PTT's of Section 10. In that section, every PTT that matches an n-ary pattern is bounded, with bound n or even $n-1$. Hence, pattern matching PTT's can be evaluated in polynomial time. And the same is true for pattern matching PFT's, see Section 11.

15. Variations of decomposition

In this section we present a number of results the proofs of which are based on variations of the decomposition techniques used in Section 4. In the first part of the section we consider deterministic PTT's, and in the second part we consider PTT's with strong (visible) pebbles.

Deterministic PTT's. As observed at the end of Section 4 it is open whether I-dPTT $\subseteq$ dTT^2. We first show that a subclass of I-dPTT is included in dTT^2 and then we show that I-dPTT $\subseteq$ dTT^3. Hence, every deterministic PTT can be decomposed into deterministic TT's.

Recall that dTT$^{\mathrm{MSO}}$ denotes the class of transductions that are realized by deterministic TT's with MSO head tests. By Lemma 12 it is a subclass of I-dPTT. We will show that such transducers can be decomposed into two deterministic TT's of which the first never moves up. To do this we need a lemma with an alternative proof of the inclusion dTT$^{\mathrm{MSO}} \subseteq$ I-dPTT, showing that the resulting I-PTT uses its pebbles in a restricted way. The I-PTT that is constructed in the proof of Lemma 12 does not satisfy that restriction.

For the definition of normal form of an I-PTT see the paragraphs before Lemma 43. We now define an I-PTT (or I-PTA) to be *root-oriented* if it satisfies requirements (1)–(3) of the normal form, and all non-initial non-output rules have a right-hand side of one of the following five forms: $\langle q', \mathrm{down}_i; \mathrm{drop}_c\rangle$, $\langle q', \mathrm{lift}_c; \mathrm{up}\rangle$, $\langle q', \mathrm{lift}_c; \mathrm{drop}_d\rangle$, or $\langle q', \mathrm{stay}\rangle$, where $q' \in Q \setminus Q_0$, $i \in \mathbb{N}$ and $c, d \in C$. Thus, except in an initial configuration, every pebble stack is of the form $(u_1, c_1)\cdots(u_n, c_n)$ where $u_1, \ldots, u_n$ is the path from the root to the current node. The I-PTA in the proof of Lemma 10 is root-oriented.

The next lemma follows from [10, Theorem 8.12], but we provide its proof for completeness sake. Let rI-dPTT denote the class of transductions realized by root-oriented deterministic I-PTT's.[26]

Lemma 49. dTT$^{\mathrm{MSO}} \subseteq$ rI-dPTT.

Proof. Let $\mathcal{M}$ be a deterministic TT that uses a regular site T as MSO head test. For simplicity we will assume that $\mathcal{M}$ tests T in every rule. Let $\mathcal{A} = (\Sigma \times \{0,1\}, P, F, \delta)$ be a deterministic bottom-up finite-state tree automaton that recognizes $\mathrm{mark}(T)$. As usual we identify the symbols $(\sigma, 0)$ and σ. For every tree $t \in T_\Sigma$ and every node $u \in N(t)$, we define the set $\mathrm{succ}_t(u)$ of *successful states* of $\mathcal{A}$ at u to consist of all states $p \in P$ such that $\mathcal{A}$ recognizes t when started at u in state p. To be precise, $\mathrm{succ}_t(\mathrm{root}_t) = F$ and if u has label $\sigma \in \Sigma^{(m)}$ and $i \in [1,m]$, then $\mathrm{succ}_t(ui)$ is the set of all states $p \in P$ such that $\delta(\sigma, p_1, \ldots, p_{i-1}, p, p_{i+1}, \ldots, p_m) \in \mathrm{succ}_t(u)$, where p_j is the state in which $\mathcal{A}$ arrives at uj for every $j \in [1,m] \setminus \{i\}$.

We construct a root-oriented deterministic I-PTT $\mathcal{M}'$ that stepwise simulates $\mathcal{M}$ and simultaneously keeps track of $\mathrm{succ}_t(v)$ for all nodes v on the path from the root to the current node u, by storing that information in its pebble colours. It uses the I-PTA $\mathcal{A}'$ of Lemma 10 (with $\mathcal{A}$ restricted to $\Sigma \times \{0\}$) as a subroutine to compute the states in which $\mathcal{A}$ arrives at the children of u. Using these states and $\mathrm{succ}_t(u)$, it can easily test whether $(t,u) \in T$. Moreover, when moving down to a child ui of u it can use this information to compute $\mathrm{succ}_t(ui)$.

Formally, in addition to the pebble colours $p_1 \cdots p_m$ of $\mathcal{A}'$, the transducer $\mathcal{M}'$ uses pebble colours $(S, p_1 \cdots p_m)$ where $S \subseteq P$. As states it uses (apart from its initial state) the states of $\mathcal{M}$ and states of the form $(\tilde{q}, q)$ where $\tilde{q}$ is a state of $\mathcal{M}$

[25] Additionally, G has an initial nonterminal S with rules $S \to \langle q_0, \mathrm{root}_t, \varepsilon\rangle$ for every initial state q_0 of $\mathcal{M}$.

[26] In [10, Chapter 8] root-oriented I-PTT's are called tree-walking pushdown transducers, and rI-dPTT is denoted P-DTWT. They are the RT(P(TR))-transducers of [23], also called indexed tree transducers.

and q a state of $\mathcal{A}'$; in fact, q is either the main state $q_\circ$ of $\mathcal{A}'$ or it is $\bar{q}_p$ for some $p \in P$. Initially, $\mathcal{M}'$ drops pebble (F, ε) on the root and goes into state $(\tilde{q}_0, q_\circ)$ where $\tilde{q}_0$ is the initial state of $\mathcal{M}$. This incorporates rule ρ_1 of $\mathcal{A}'$. The other rules of $\mathcal{M}'$ that correspond to $\mathcal{A}'$ are as follows. First, the rule ρ_2 of $\mathcal{A}'$ together with the corresponding rule for pebble colour $(S, p_1 \cdots p_m)$, both for $m < \mathrm{rank}(\sigma)$:

$$\langle(\tilde{q}, q_\circ), \sigma, j, \{p_1 \cdots p_m\}\rangle \to \langle(\tilde{q}, q_\circ), \mathrm{down}_{m+1}; \mathrm{drop}_\varepsilon\rangle$$
$$\langle(\tilde{q}, q_\circ), \sigma, j, \{(S, p_1 \cdots p_m)\}\rangle \to \langle(\tilde{q}, q_\circ), \mathrm{down}_{m+1}; \mathrm{drop}_\varepsilon\rangle.$$

Second, the rule ρ_3 of $\mathcal{A}'$, for $m = \mathrm{rank}(\sigma)$ and $p = \delta(\sigma, p_1, \ldots, p_m)$:

$$\langle(\tilde{q}, q_\circ), \sigma, j, \{p_1 \cdots p_m\}\rangle \to \langle(\tilde{q}, \bar{q}_p), \mathrm{lift}_{p_1 \cdots p_m}; \mathrm{up}\rangle \quad \text{if } j \neq 0.$$

Third, the rule r_6 of $\mathcal{A}'$ together with the corresponding rule for pebble colour $(S, p_1 \cdots p_m)$, both for $m < \mathrm{rank}(\sigma)$:

$$\langle(\tilde{q}, \bar{q}_p), \sigma, j, \{p_1 \cdots p_m\}\rangle \to \langle(\tilde{q}, q_\circ), \mathrm{lift}_{p_1 \cdots p_m}; \mathrm{drop}_{p_1 \cdots p_m p}\rangle$$
$$\langle(\tilde{q}, \bar{q}_p), \sigma, j, \{(S, p_1 \cdots p_m)\}\rangle \to \langle(\tilde{q}, q_\circ), \mathrm{lift}_{(S, p_1 \cdots p_m)}; \mathrm{drop}_{(S, p_1 \cdots p_m p)}\rangle.$$

The subroutine $\mathcal{A}'$ is always called at a node u where $\mathcal{M}'$ observes a pebble of the form (S, ε), and when $\mathcal{A}'$ is finished $\mathcal{M}'$ is back at the same node u and observes the pebble $(S, p_1 \cdots p_m)$ where $p_1, \ldots, p_m$ are the states at which $\mathcal{A}$ arrives at the children of u.

Finally we consider the simulation of a step of $\mathcal{M}$, which either occurs when the subroutine $\mathcal{A}'$ is finished (instead of its rules ρ_4 and ρ_5), or just after the simulation of another step of $\mathcal{M}$, in which it does not move down. Suppose that $\mathcal{M}$ has a rule $\langle\tilde{q}, \sigma, j, T\rangle \to \zeta$ and that $\delta((\sigma, 1), p_1, \ldots, p_m) \in S$, or suppose that it has a rule $\langle\tilde{q}, \sigma, j, \neg T\rangle \to \zeta$ and $\delta((\sigma, 1), p_1, \ldots, p_m) \notin S$. Then $\mathcal{M}'$ has the following two rules, for $m = \mathrm{rank}(\sigma)$:

$$\langle(\tilde{q}, q_\circ), \sigma, j, \{(S, p_1 \cdots p_m)\}\rangle \to \zeta'$$
$$\langle\tilde{q}, \sigma, j, \{(S, p_1 \cdots p_m)\}\rangle \to \zeta'$$

such that

(1) if $\zeta = \langle\tilde{q}', \mathrm{up}\rangle$, then $\zeta' = \langle\tilde{q}', \mathrm{lift}_{(S, p_1 \cdots p_m)}; \mathrm{up}\rangle$,
(2) if $\zeta = \langle\tilde{q}', \mathrm{down}_i\rangle$, then $\zeta' = \langle(\tilde{q}', q_\circ), \mathrm{down}_i; \mathrm{drop}_{(S', \varepsilon)}\rangle$, where $S' = \{p \in P \mid \delta(\sigma, p_1, \ldots, p_{i-1}, p, p_{i+1}, \ldots, p_m) \in S\}$, and
(3) $\zeta' = \zeta$ otherwise.

This ends the formal description of $\mathcal{M}'$. In general, $\mathcal{M}$ uses regular sites $T_1, \ldots, T_n$ as MSO head tests, and correspondingly $\mathcal{M}'$ has pebble colours of the form $(S_1, \ldots, S_n, p_1 \cdots p_m)$ where S_i is a set of states of an automaton $\mathcal{A}_i$ recognizing $\mathrm{mark}(T_i)$. □

Let $\mathsf{dTT}_\downarrow$ denote the class of transductions realized by deterministic TT's that do not use the up-instruction. Such transducers are equivalent to classical deterministic top-down tree transducers. The next lemma is shown in [10, Theorem 8.15] but we provide its proof again, to show the connection to Lemma 4.

Lemma 50. $\mathsf{rl\text{-}dPTT} \subseteq \mathsf{dTT}_\downarrow \circ \mathsf{dTT}$.

Proof. Let $\mathcal{M}$ be a root-oriented deterministic I-PTT. Looking at the proof of Lemma 4, it should be clear that, for every input tree t, the simulating transducer $\mathcal{M}'$ only visits those nodes of t' that correspond to a sequence of instructions of $\mathcal{M}$ that starts with a drop-instruction and then consists alternatingly of a down-instruction and a drop-instruction. Consequently, the "preprocessor" $\mathcal{N}$ can be adapted so as to generate just that part of t'. The new $\mathcal{N}$ does not need the states f_i any more, but just has the initial state g and the state f. Its rules are

$$\langle g, \sigma, j\rangle \to \sigma'(\bot^m, \langle f, \mathrm{stay}\rangle^\gamma)$$
$$\langle f, \sigma, j\rangle \to \sigma'_{0,j}(\langle g, \mathrm{down}_1\rangle, \ldots, \langle g, \mathrm{down}_m\rangle, \bot^\gamma, \bot)$$

where m is the rank of σ and $\bot^n$ abbreviates the sequence $\bot, \ldots, \bot$ of length n. Note that the child number j is irrelevant. With this new, total deterministic preprocessor $\mathcal{N}$ the proof of Lemma 4 is still valid. □

The following corollary was shown in [10, Theorem 8.22], but we repeat it here for completeness sake, cf. Corollary 42.

Corollary 51. $\mathsf{rl\text{-}dPTT} = \mathsf{dTT}_\downarrow \circ \mathsf{dTT} = \mathsf{dMT}_{\mathrm{OI}}$.

Proof. The inclusion $\mathsf{dTT}_{\downarrow} \circ \mathsf{dTT} \subseteq \mathsf{dMT}_{\mathrm{OI}}$ follows from the inclusions $\mathsf{dTT} \subseteq \mathsf{dMT}_{\mathrm{OI}}$, shown in [20, Theorem 35 for $n = 0$: Lemma 34 and Theorem 31], and $\mathsf{dTT}_{\downarrow} \circ \mathsf{dMT}_{\mathrm{OI}} \subseteq \mathsf{dMT}_{\mathrm{OI}}$, shown in [22, Theorem 7.6(3)]. By Lemma 50 it now suffices to show that $\mathsf{dMT}_{\mathrm{OI}} \subseteq \mathsf{rl\text{-}dPTT}$ (which strengthens the second inclusion of Corollary 42). There are two ways of proving this, which are essentially the same. First, the proof of Lemma 39 can be adapted in a straightforward way.[27] Second, the equality $\mathsf{rl\text{-}dPTT} = \mathsf{dMT}_{\mathrm{OI}}$ is shown for total functions in [23, Theorem 5.16]. By [22, Theorem 6.18], every transduction $\tau \in \mathsf{dMT}_{\mathrm{OI}}$ is of the form $\tau_1 \circ \tau_2$ where τ_1 is the identity on a regular tree language R and $\tau_2 \in \mathsf{dMT}_{\mathrm{OI}}$ is a total function. Thus, τ_2 is in $\mathsf{rl\text{-}dPTT}$. This implies that $\tau_1 \circ \tau_2$ is in $\mathsf{rl\text{-}dPTT}$: the I-PTT just starts by checking that the input tree is in R, using the root-oriented I-PTT $\mathcal{A}'$ in the proof of Lemma 10 as a subroutine. □

We now turn to the decomposition of an arbitrary deterministic I-PTT into deterministic TT's.

Lemma 52. $\mathsf{l\text{-}dPTT} \subseteq \mathsf{tdTT}^{\mathrm{MSO}} \circ \mathsf{dTT}$.

Proof. Let $\mathcal{M} = (\Sigma, \Delta, Q, \{q_0\}, C, \varnothing, C_{\mathrm{i}}, R, 0)$ be a deterministic I-PTT with $C = C_{\mathrm{i}}$. We may assume that there is a mapping $\chi : C \to Q$ such that $\chi(c) = q'$ for every rule $\langle q, \sigma, j, b\rangle \to \langle q', \mathrm{drop}_c\rangle$ of $\mathcal{M}$. If not, then we change C into $C \times Q$ and we change every rule $\langle q, \sigma, j, b\rangle \to \langle q', \mathrm{drop}_c\rangle$ into $\langle q, \sigma, j, b\rangle \to \langle q', \mathrm{drop}_{(c,q')}\rangle$ and every rule $\langle q, \sigma, j, \{c\}\rangle \to \langle q', \mathrm{lift}_c\rangle$ into all the rules $\langle q, \sigma, j, \{(c, p)\}\rangle \to \langle q', \mathrm{lift}_{(c,p)}\rangle$. Moreover, we may assume that $C = [1, \gamma]$ for some $\gamma \in \mathbb{N}$.

As in the proof of Lemma 50 we consider the proof of Lemma 4 and adapt the preprocessor $\mathcal{N}$ to the needs of $\mathcal{M}$. Every copy of the input tree that is generated by $\mathcal{N}$ corresponds to a unique potential pebble stack π of $\mathcal{M}$. The simulating deterministic TT $\mathcal{M}'$ walks on that copy whenever $\mathcal{M}$ has pebble stack π. The idea is now to construct a variation $\mathcal{N}'$ of $\mathcal{N}$ that only generates those copies of the input tree t that correspond to reachable pebble stacks. A pebble stack π is *reachable* (on t) if $\mathcal{M}$ has a reachable output form that contains a configuration $\langle q, v, \pi\rangle$ for some $q \in Q$ and $v \in N(t)$. For a given t in the domain of $\mathcal{M}$, the number of reachable stacks is finite because $\mathcal{M}$ is deterministic and thus has a unique computation on t. Consequently $\mathcal{N}'$ can preprocess t deterministically. Then we can define a total deterministic preprocessor $\mathcal{N}''$ that starts by performing an MSO head test whether or not the input tree is in the domain of $\mathcal{M}$ (which is regular by Corollary 9). If it is, then $\mathcal{N}''$ calls $\mathcal{N}'$, and if it is not, then $\mathcal{N}''$ outputs $\perp$ and halts.

As an auxiliary tool, we define (as in the proof of Theorem 47) the nondeterministic I-PTA $\mathcal{A}$ that is obtained from $\mathcal{M}$ by changing every output rule $\langle q, \sigma, j, b\rangle \to \delta(\langle q_1, \mathrm{stay}\rangle, \ldots, \langle q_m, \mathrm{stay}\rangle)$ of $\mathcal{M}$ into the rules $\langle q, \sigma, j, b\rangle \to \langle q_i, \mathrm{stay}\rangle$ for $i \in [1, m]$. Intuitively, whenever $\mathcal{M}$ branches, $\mathcal{A}$ nondeterministically follows one of those branches. Obviously a nonempty pebble stack π with top element (u, c) is reachable if and only if $\langle \chi(c), u, \pi\rangle$ is a reachable configuration of $\mathcal{A}$ (see footnote 9). Note that $\langle \chi(c), u, \pi\rangle$ is the configuration of $\mathcal{M}$ just after dropping pebble c at node u.

For pebble colour c, we consider the site T_c consisting of all pairs (t, u) such that one-pebble stack (u, c) is reachable, i.e., such that $\mathcal{A}$ has a computation starting in the initial configuration and ending in the configuration $\langle \chi(c), u, (u, c)\rangle$. It is not difficult to see that T_c is a regular site. In fact, $\mathrm{mark}(T_c)$ is the domain of an I-PTA $\mathcal{B}$ with stack tests that simulates $\mathcal{A}$; whenever it arrives at the marked node u in state $\chi(c)$ and it observes pebble c, then it may lift the pebble, check that its stack is empty, and accept. Stack tests are allowed by Lemma 1, and the domain of $\mathcal{B}$ is regular by Corollary 9.

We now turn to reachable pebble stacks with more than one pebble, i.e., of the form $\pi(u, c)(v, d)$. Assuming that we already know that $\pi(u, c)$ is reachable, we can find out whether $\pi(u, c)(v, d)$ is reachable through a regular trip, as follows. For pebble colours c and d, we consider the trip $T_{c,d}$ consisting of all triples (t, u, v) such that $\mathcal{A}$ has a computation on t starting in configuration $\langle \chi(c), u, (u, c)\rangle$ and ending in configuration $\langle \chi(d), v, (u, c)(v, d)\rangle$; moreover, in every intermediate configuration the bottom element of the pebble stack must be (u, c). The trip $T_{c,d}$ is regular because $\mathrm{mark}(T)$ is the domain of an I-PTA $\mathcal{B}'$ with stack tests that first walks to the marked node u. Then $\mathcal{B}'$ simulates $\mathcal{A}$, starting in state $\chi(c)$, interpreting the mark of u as pebble c (which cannot be lifted). Similar to $\mathcal{B}$ above, whenever $\mathcal{B}'$ arrives at the marked node v in state $\chi(d)$ and it observes pebble d, then it may lift the pebble, check that the stack is empty, and accept. Obviously, if $\pi(u, c)$ is reachable, then $\pi(u, c)(v, d)$ is reachable if and only if $(t, u, v) \in T_{c,d}$. Let $\mathcal{B}_{c,d}$ be a (nondeterministic) TA with MSO head tests that computes $T_{c,d}$, as in Proposition 14.

The new preprocessor $\mathcal{N}'$ is a deterministic TT with MSO head tests that works in the same way as $\mathcal{N}$ but only creates the copies of the input tree t that correspond to reachable pebble stacks. Initially it uses the test T_c at node u to decide whether it has to create a copy of t corresponding to pebble stack (u, c). If the test is positive, then, just as $\mathcal{N}$, it creates a copy of t by walking from u to every other node v of t, copying v to the output. Now recall that $\mathcal{N}$ walks from u to v along the shortest (undirected) path in t. Thus, by Proposition 14, $\mathcal{N}'$ can simulate the behaviour of TA $\mathcal{B}_{c,d}$ from u to v, for every pebble colour d (using a subset construction as in the proof of Theorem 16). Thus, arriving at v it can use the trip $T_{c,d}$ to decide whether it has to create a copy of t corresponding to pebble stack $(u, c)(v, d)$. At the next level it simulates all $\mathcal{B}_{d,d'}$ for every $d' \in C$, etcetera.

More formally, $\mathcal{N}'$ has initial state g, and all other states are of the form $(q, c, S_1, \ldots, S_\gamma)$ where q is a state of $\mathcal{N}$, $c \in C$, and S_d is a set of states of $\mathcal{B}_{c,d}$ for every $d \in C = [1, \gamma]$. We will call them "extended" states in what follows. To describe the

[27] The transducer $\mathcal{M}$ uses an additional pebble $\odot$, which it drops initially on the root and whenever it moves down (instead of calling subroutine $S_{q',\psi}$). When necessary it replaces $\odot$ by a pebble $([s_1], \ldots, [s_m])$. When subroutine S is in state $[z_i]$ for some parameter z_i, it lifts $\odot$ and moves up where it finds a pebble $([s_1], \ldots, [s_m])$.

rules of $\mathcal{N}'$, we first recall the rules of the transducer $\mathcal{N}$ from the proof of Lemma 4. Apart from the rules $\langle f, \sigma, j\rangle \to \bot$, $\mathcal{N}$ has the rules

$$\begin{aligned}
\rho_g: \quad & \langle g, \sigma, j\rangle \to \sigma'(\langle g, \mathrm{down}_1\rangle, \ldots, \langle g, \mathrm{down}_m\rangle, \langle f, \mathrm{stay}\rangle^\gamma)\\
\rho_f: \quad & \langle f, \sigma, j\rangle \to \sigma'_{0,j}(\langle g, \mathrm{down}_1\rangle, \ldots, \langle g, \mathrm{down}_m\rangle, \langle f, \mathrm{stay}\rangle^\gamma, \xi_j)\\
\rho_{f_i}: \quad & \langle f_i, \sigma, j\rangle \to \sigma'_{i,j}(\langle g, \mathrm{down}_1\rangle, \ldots, \langle g, \mathrm{down}_{i-1}\rangle, \bot,\\
& \qquad \langle g, \mathrm{down}_{i+1}\rangle, \ldots, \langle g, \mathrm{down}_m\rangle, \langle f, \mathrm{stay}\rangle^\gamma, \xi_j)
\end{aligned}$$

where $\xi_j = \langle f_j, \mathrm{up}\rangle$ for $j \neq 0$, and $\xi_0 = \bot$.

The rules of $\mathcal{N}'$ for state g are obtained from rule ρ_g by adding all possible combinations of the MSO head tests T_c and their negations to the left-hand side. In the right-hand side, the sequence $\langle f, \mathrm{stay}\rangle^\gamma$ should be replaced by the sequence $\zeta_1, \ldots, \zeta_\gamma$ where $\zeta_c = \langle (f, c, I_{c,1}, \ldots, I_{c,\gamma}), \mathrm{stay}\rangle$ if T_c is true, $I_{c,d}$ being the set of initial states of $\mathcal{B}_{c,d}$, and $\zeta_c = \bot$ if T_c is false.[28] The rules of $\mathcal{N}'$ for an "extended" state $(q, c, S_1, \ldots, S_\gamma)$ are obtained from rule ρ_q as follows. In the left-hand side change q into $(q, c, S_1, \ldots, S_\gamma)$. Moreover, add all MSO head tests of $\mathcal{B}_{c,d}$ for every $d \in C$. In the right-hand side change every occurrence of a state $q' \neq f$ into the extended state $(q', c, S'_1, \ldots, S'_\gamma)$ where the set S'_d is obtained from the set S_d by simulating $\mathcal{B}_{c,d}$ appropriately, moving down to the ℓ-th child if $q' = g$ in $\langle g, \mathrm{down}_\ell\rangle$ and moving up if $q' = f_j$. Moreover, the sequence $\langle f, \mathrm{stay}\rangle^\gamma$ should be replaced by $\zeta_1, \ldots, \zeta_\gamma$ where $\zeta_d = \langle (f, d, I_{d,1}, \ldots, I_{d,\gamma}), \mathrm{stay}\rangle$ if S_d contains a final state of $\mathcal{B}_{c,d}$, and $\zeta_d = \bot$ otherwise (where $I_{d,d'}$ is defined similarly to $I_{c,d}$ above).

It should be clear that $\mathcal{N}'$ produces an output for every input tree t on which $\mathcal{M}$ has finitely many reachable pebble stacks. Thus, $\mathcal{N}'$ preprocesses t appropriately and the deterministic TT $\mathcal{M}'$ in the proof of Lemma 4 can simulate $\mathcal{M}$ on $\tau_{\mathcal{N}'}(t)$. Hence $\tau_{\mathcal{M}'}(\tau_{\mathcal{N}'}(t)) = \tau_{\mathcal{M}}(t)$ for every t in the domain of $\mathcal{M}$. □

It is easy to adapt the proof of Theorem 17 to the case where the first (deterministic) TT $\mathcal{M}_1$ uses MSO head tests; those tests can also be executed by the constructed I-PTT $\mathcal{M}$, by Lemma 12. Moreover, that proof can also easily be adapted to the case where the second transducer $\mathcal{M}_2$ is a root-oriented I-PTT. From this and Lemmas 49 and 52 we obtain the following characterizations of I-dPTT as a corollary. We do not know whether similar characterizations hold for I-PTT.

Theorem 53. $\mathsf{I\text{-}dPTT} = \mathsf{dTT}^{\mathrm{MSO}} \circ \mathsf{dTT} = \mathsf{dTT}^{\mathrm{MSO}} \circ \mathsf{dTT}^{\mathrm{MSO}} = \mathsf{dTT}^{\mathrm{MSO}} \circ \mathsf{rI\text{-}dPTT}$.

Proof. Let us show for completeness sake that $\mathsf{dTT} \circ \mathsf{rI\text{-}dPTT} \subseteq \mathsf{I\text{-}dPTT}$. The proof of Theorem 17 can easily be generalized to a root-oriented I-PTT $\mathcal{M}_2$, because the path from the root of s to the current node v of $\mathcal{M}_2$ is represented by the pebble stack of the constructed transducer $\mathcal{M}$, and so the pebbles of $\mathcal{M}_2$ can also be stored in the pebble stack of $\mathcal{M}$. For each node on that path, the stack contains a pebble with the rule of $\mathcal{M}_1$ that generates that node, with its child number, and with the pebble that $\mathcal{M}_2$ drops on that node.

Formally, the pebble colours of $\mathcal{M}$ are now triples (ρ, i, c) where c is a pebble colour of $\mathcal{M}_2$, and the states of $\mathcal{M}$ are the states of $\mathcal{M}_2$ and all 4-tuples (p, i, c, q) where c is again a pebble colour of $\mathcal{M}_2$. The initial state of $\mathcal{M}$ is now the one of $\mathcal{M}_2$, and if $\mathcal{M}_2$ has an initial rule $\langle q_0, \delta, 0, \varnothing\rangle \to \langle q, \mathrm{drop}_c\rangle$, then $\mathcal{M}$ has the rule $\langle q_0, \delta, 0, \varnothing\rangle \to \langle (p_0, 0, c, q), \mathrm{stay}\rangle$. The rules of $\mathcal{M}$ that simulate $\mathcal{M}_1$ are defined as in the proof of Theorem 17, replacing i by i, c everywhere for each c. The rules of $\mathcal{M}$ that simulate the non-initial rules of $\mathcal{M}_2$ are defined as follows. Let $\langle q, \delta, i, \{c\}\rangle \to \zeta$ be a non-initial rule of $\mathcal{M}_2$ and let $\rho : \langle p, \sigma, j\rangle \to \delta(\langle p_1, \mathrm{stay}\rangle, \ldots, \langle p_m, \mathrm{stay}\rangle)$ be an output rule of $\mathcal{M}_1$. Then $\mathcal{M}$ has the rule $\langle q, \sigma, j, \{(\rho, i, c)\}\rangle \to \zeta'$ where ζ' is defined as follows. If $\zeta = \langle q', \mathrm{down}_\ell; \mathrm{drop}_d\rangle$, then $\zeta' = \langle (p_\ell, \ell, d, q'), \mathrm{stay}\rangle$. If $\zeta = \langle q', \mathrm{lift}_c; \mathrm{up}\rangle$, then $\zeta' = \langle q', \mathrm{lift}_{(\rho,i,c)}; \text{to-top}\rangle$. If $\zeta = \langle q', \mathrm{lift}_c; \mathrm{drop}_d\rangle$, then $\zeta' = \langle q', \mathrm{lift}_{(\rho,i,c)}; \mathrm{drop}_{(\rho,i,d)}\rangle$. In the remaining cases, $\zeta' = \zeta$. □

As another corollary we obtain from the three Lemmas 49, 50, and 52 that $\mathsf{I\text{-}dPTT} \subseteq \mathsf{dTT}^3$. Moreover, $\mathsf{I\text{-}dPTT} \subseteq \mathsf{dMT}^2_{\mathrm{OI}}$ by the second equality of Corollary 51. Together with Theorem 46, that implies that $\mathsf{dTL}_{\ell\mathrm{r}} \subseteq \mathsf{dMT}^2_{\mathrm{OI}}$, which was stated as an open problem in [20, Section 8] (where $\mathsf{dTL}_{\ell\mathrm{r}}$ and $\mathsf{dMT}_{\mathrm{OI}}$ are denoted 0-DPMTT and DMTT, respectively).

Corollary 54. $\mathsf{I\text{-}dPTT} \subseteq \mathsf{dTT}_\downarrow \circ \mathsf{dTT} \circ \mathsf{dTT} \subseteq \mathsf{dMT}_{\mathrm{OI}} \circ \mathsf{dMT}_{\mathrm{OI}}$.

We are now able to prove the deterministic analogue of Theorem 5 for PTT's with at least one visible pebble.

Theorem 55. *For every $k \geq 1$, $\mathsf{V}_k\mathsf{I\text{-}dPTT} \subseteq \mathsf{dTT}^{k+2}$.*

Proof. Since it follows from Lemma 3 and Corollary 54 that $\mathsf{V}_k\mathsf{I\text{-}dPTT} \subseteq \mathsf{tdTT}^{k-1} \circ \mathsf{tdTT} \circ \mathsf{dTT}_\downarrow \circ \mathsf{dTT} \circ \mathsf{dTT}$, it suffices to show that $\mathsf{tdTT} \circ \mathsf{dTT}_\downarrow \subseteq \mathsf{dTT}$. For the sake of the proof of Lemma 61, we will show more generally that for all deterministic

[28] More precisely, $I_{c,d}$ consists of all initial states of $\mathcal{B}_{c,d}$, plus all states that $\mathcal{B}_{c,d}$ can reach from an initial state by applying a relevant rule with a stay-instruction.

TT's $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $\mathcal{M}_2$ does not use the up-instruction, a deterministic TT $\mathcal{M}$ can be constructed such that $\tau_{\mathcal{M}}(t) = \tau_{\mathcal{M}_2}(\tau_{\mathcal{M}_1}(t))$ for every input tree t in the domain of $\mathcal{M}_1$. This can be proved by a straightforward product construction, which is an easy adaptation of the construction in the proof of Theorem 17. Since transducer $\mathcal{M}_2$ never moves up, there is no need to backtrack on the computation of $\mathcal{M}_1$. Therefore, the constructed transducer $\mathcal{M}$ only considers the topmost pebble. Since that pebble is always at the position of the head, its colour can as well be stored in the finite state of $\mathcal{M}$. Hence $\mathcal{M}$ can be turned into a TT rather than an I-PTT.

Formally, let $\mathcal{M}_1 = (\Sigma, \Delta, P, \{p_0\}, R_1)$ and $\mathcal{M}_2 = (\Delta, \Gamma, Q, \{q_0\}, R_2)$. The deterministic TT $\mathcal{M}$ has input alphabet Σ and output alphabet Γ. Its states are of the form (p, i, q) or (ρ, i, q), where $p \in P$, $i \in [0, mx_\Delta]$, $q \in Q$, and ρ is an output rule of $\mathcal{M}_1$, i.e., a rule of the form $\langle p, \sigma, j\rangle \to \delta(\langle p_1, \text{stay}\rangle, \ldots, \langle p_m, \text{stay}\rangle)$. Its initial state is $(p_0, 0, q_0)$. As in the proof of Theorem 17, state (p, i, q) is used by $\mathcal{M}$ when simulating the computation of $\mathcal{M}_1$ that generates the i-th child of the current node of $\mathcal{M}_2$ (keeping the state q of $\mathcal{M}_2$ in memory). A state (ρ, i, q) is used by $\mathcal{M}$ when simulating a computation step of $\mathcal{M}_2$ on the node that $\mathcal{M}_1$ has generated with rule ρ. The rules of $\mathcal{M}$ are defined as follows.

First the rules that simulate $\mathcal{M}_1$. Let $\rho : \langle p, \sigma, j\rangle \to \zeta$ be a rule in R_1. If $\zeta = \langle p', \alpha\rangle$, where α is a move instruction, then $\mathcal{M}$ has the rules $\langle (p, i, q), \sigma, j\rangle \to \langle (p', i, q), \alpha\rangle$ for every $i \in [0, mx_\Delta]$ and $q \in Q$. If ρ is an output rule, then $\mathcal{M}$ has the rules $\langle (p, i, q), \sigma, j\rangle \to \langle (\rho, i, q), \text{stay}\rangle$ for every i and q as above.

Second the rules that simulate $\mathcal{M}_2$. Let $\langle q, \delta, i\rangle \to \zeta$ be a rule in R_2 and let $\rho : \langle p, \sigma, j\rangle \to \delta(\langle p_1, \text{stay}\rangle, \ldots, \langle p_m, \text{stay}\rangle)$ be an output rule in R_1 (with the same δ). Then $\mathcal{M}$ has the rule $\langle (\rho, i, q), \sigma, j\rangle \to \zeta'$ where ζ' is obtained from ζ by changing every $\langle q', \text{stay}\rangle$ into $\langle (\rho, i, q'), \text{stay}\rangle$, and every $\langle q', \text{down}_\ell\rangle$ into $\langle (p_\ell, \ell, q'), \text{stay}\rangle$. □

Since the topmost pebble of a V-PTT can be replaced by an invisible pebble, we obtain from Theorem 55 that $\mathsf{V}_k\text{-dPTT} \subseteq \mathsf{dTT}^{k+1}$, which was proved in [20, Theorem 10].

Theorem 55 allows us to show that, in the deterministic case, $k+1$ visible pebbles are more powerful than k visible pebbles.

Theorem 56. *For every $k \geq 0$, $\mathsf{V}_k\text{I-dPTT} \subsetneq \mathsf{V}_{k+1}\text{I-dPTT}$.*

Proof. It follows from Theorem 55 and Corollary 54 (and the inclusion $\mathsf{dTT} \subseteq \mathsf{dMT}_{\text{OI}}$ in Corollary 51) that $\mathsf{V}_k\text{I-dPTT} \subseteq \mathsf{dMT}^{k+2}_{\text{OI}}$ for every $k \geq 0$. But it is proved in [20, Theorem 41] that, for every $k \geq 1$, $\mathsf{V}_k\text{-dPTT}$ is not included in $\mathsf{dMT}^k_{\text{OI}}$. Hence, since the topmost pebble of a V-PTT can be replaced by an invisible pebble, $\mathsf{V}_k\text{I-dPTT}$ is not included in $\mathsf{dMT}^{k+1}_{\text{OI}}$. □

The above proof also shows that Theorem 55 is optimal, in the sense that, for every $k \geq 1$, $\mathsf{V}_k\text{I-dPTT}$ is not included in dTT^{k+1}.

Another consequence of Theorem 55 is that, by the results of [36], all total deterministic VI-PFT transformations for which the size of the output document is linear in the size of the input document, can be programmed in TL. Let LSI be the class of all total functions φ for which there exists a constant $c \in \mathbb{N}$ such that $|\varphi(t)| \leq c \cdot |t|$ for every input tree t.

Theorem 57. *For every $k \geq 0$,*

$$\mathsf{V}_k\text{I-dPTT} \cap \mathsf{LSI} \subseteq \text{I-dPTT} = \mathsf{dTL}_{\text{r}} \quad \textit{and} \quad \mathsf{V}_k\text{I-dPFT} \cap \mathsf{LSI} \subseteq \text{I-dPFT} = \mathsf{dTL}.$$

Proof. It is shown in [36] that $\mathsf{dMT}^k_{\text{OI}} \cap \mathsf{LSI} \subseteq \mathsf{dMT}_{\text{OI}}$ for every $k \geq 1$. By Theorem 55 and Corollary 51, $\mathsf{V}_k\text{I-dPTT} \subseteq \mathsf{dMT}^{k+2}_{\text{OI}}$. And by Corollary 42 and Theorem 46, $\mathsf{dMT}_{\text{OI}} \subseteq \text{I-dPTT} = \mathsf{dTL}_{\text{r}}$. This proves the first inclusion. To prove the second inclusion, let $\varphi \in \mathsf{V}_k\text{I-dPFT} \cap \mathsf{LSI}$. Obviously, $\varphi \circ \text{enc}$ is also in LSI, and $\varphi \circ \text{enc} \in \mathsf{V}_k\text{I-dPTT} \circ \text{I-dPTT}$ by Lemma 33(2). Hence $\varphi \circ \text{enc} \in \mathsf{dMT}^{k+4}_{\text{OI}} \subseteq \text{I-dPTT}$, as above. In other words, $\varphi \in \text{I-dPTT} \circ \text{dec}$. Consequently, by Lemma 33(1) and Theorem 46, $\varphi \in \text{I-dPFT} = \mathsf{dTL}$. □

In fact, $\mathsf{V}_k\text{I-dPTT} \cap \mathsf{LSI}$ is the class of total functions in the class DMSOT of deterministic MSO definable tree transductions discussed after Corollary 42, and similarly, $\mathsf{V}_k\text{I-dPFT} \cap \mathsf{LSI}$ is the class of total functions in the class of deterministic MSO definable tree-to-forest transductions (which equals $\mathsf{DMSOT} \circ \mathsf{dec}$, because both dec and enc are MSO definable).

For the reader familiar with results about attribute grammars (which are a well-known compiler construction tool) and related formalisms, we now briefly discuss the relationship between those results and some of the above. As explained in detail in [20, Section 3.2], the total deterministic tree-walking tree transducer, i.e., the tdTT, is essentially a notational variant of the attributed tree transducer (AT) of [25,26], except that the AT is in addition required to be "noncircular", which means that no configuration can generate an output form in which that same configuration occurs. As observed at the end of Section 12, the deterministic I-PTT has the same expressive power as the deterministic TL program that is local and ranked, which corresponds to the macro attributed tree transducer (MAT) of [26,35] in the same way, i.e., the MAT is the "noncircular" $\mathsf{tdTL}_{\ell r}$ program. Since $\text{rI-dPTT} = \mathsf{dMT}_{\text{OI}}$ by Corollary 51, Lemma 49 ($\mathsf{dTT}^{\text{MSO}} \subseteq \text{rI-dPTT}$) is closely related to the well-known fact that AT (with look-ahead) can be simulated by deterministic macro tree transducers. Lemma 50 ($\text{rI-dPTT} \subseteq \mathsf{dTT}_{\downarrow} \circ \mathsf{dTT}$) is related to the fact that every total deterministic macro tree transducer can be decomposed into a deterministic top-down tree transducer followed by a YIELD mapping, which can be realized by an AT. Theorem 53

($\mathsf{I\text{-}dPTT} = \mathsf{dTT}^{\mathrm{MSO}} \circ \mathsf{dTT} = \mathsf{dTT}^{\mathrm{MSO}} \circ \mathsf{rI\text{-}dPTT}$) is closely related to the fact that every MAT can be decomposed into two AT's, and that the composition of an AT and a total deterministic macro tree transducer can be simulated by a MAT, as shown in [35, Theorem 4.8] and its proof (see also [26, Corollary 7.30]). The inclusion $\mathsf{tdTT} \circ \mathsf{dTT}_{\downarrow} \subseteq \mathsf{dTT}$ in the proof of Theorem 55 is closely related to the closure of AT under right-composition with deterministic top-down tree transducers, as shown in [25, Theorem 4.3] (see also [35, Lemma 4.11] and [26, Lemma 5.46]). We finally mention that the class DMSOT of deterministic MSO definable tree transductions is properly included in $\mathsf{dTT}^{\mathrm{MSO}}$ (see [10, Theorems 8.6 and 8.7]), as shown for attribute grammars (with look-ahead) in [6].

Strong pebbles. In the literature there are pebble automata with weak and strong pebbles. Weak pebbles (which are the pebbles considered until now) can only be lifted when the reading head is at the position where they were dropped, whereas strong pebbles can also be lifted from a distance, i.e., when the reading head is at any other position. So, strong pebbles are more like dogs that can be whistled back, or like pointers that can be erased from memory. Formally, we define a pebble colour c to be *strong* as follows. For a rule $\langle q, \sigma, j, b\rangle \to \langle q', \mathrm{lift}_c\rangle$ we do not require any more that $c \in b$. If the rule is relevant to configuration $\langle q, u, \pi\rangle$, then it is applicable whenever the topmost element of the pebble stack is (v, c) for some node v (not necessarily equal to u). That top pebble is then popped from the stack, i.e., $\pi = \pi'(v, c)$ where π' is the new stack. Strong pebbles were investigated, e.g., in [8,16,27,43,48].

It turns out that strong invisible pebbles are too strong, in the sense that they allow the recognition of nonregular tree languages, cf. the paragraph after Theorem 11. For example, the nonregular language $\{a^n\#b^n \mid n \in \mathbb{N}\}$ can be accepted by an I-PTA with strong pebbles as follows. After checking that the input string w is in $a^*\#b^*$, the automaton drops a pebble on # and walks to the left, dropping a pebble on every a. Next it walks to the end of w, and then walks to the left, lifting a pebble (from a distance) for every b it passes. It accepts w if it arrives at # and observes a pebble on #.

Thus, we will only consider the PTA and PTT with strong *visible* pebbles, abbreviated as v^+I-PTA and v^+I-PTT (and similarly for the classes of transductions they realize). Obviously, $\mathsf{V}_k\mathsf{I\text{-}PTT} \subseteq \mathsf{V}_k^+\mathsf{I\text{-}PTT}$ for every $k \geq 0$. We do not know whether the inclusion is proper.

Let us first show that the v^+I-PTA and v^+I-PTT can perform stack tests.

Lemma 58. *Let $k \geq 0$. For every v_k^+I-PTA with stack tests $\mathcal{A}$ an equivalent (ordinary) v_k^+I-PTA $\mathcal{A}'$ can be constructed in polynomial time. The construction preserves determinism and the absence of invisible pebbles. The same holds for the corresponding PTT's.*

Proof. Let $\mathcal{A} = (\Sigma, Q, Q_0, F, C, C_{\mathrm{v}}, C_{\mathrm{i}}, R, k)$. We construct $\mathcal{A}'$ in the same way as in the proof of Lemma 1, except that it additionally keeps track of the visible pebbles in its *own* stack, in the order in which they were dropped, cf. the construction of a counting PTA after Lemma 1. Thus, its states are of the form (q, γ, φ) where $q \in Q$, $\gamma \in C \cup \{\varepsilon\}$, and $\varphi \in (C'_{\mathrm{v}})^* = (C_{\mathrm{v}} \times (C \cup \{\varepsilon\}))^*$ is a string without repetitions of length $\leq k$. Its initial states are $(q, \varepsilon, \varepsilon)$ with $q \in Q_0$.

The rules of $\mathcal{A}'$ are defined as follows. Let $\langle q, \sigma, j, b, \gamma\rangle \to \langle q', \alpha\rangle$ be a rule of $\mathcal{A}$, let φ be a string over C'_{v} as above, and let b' be (the graph of) a mapping from b to $C \cup \{\varepsilon\}$. If α is a move instruction, then $\mathcal{A}'$ has the rule $\langle (q, \gamma, \varphi), \sigma, j, b'\rangle \to \langle (q', \gamma, \varphi), \alpha\rangle$ (and similarly for an output rule of a PTT). If $\alpha = \mathrm{drop}_c$, then $\mathcal{A}'$ has the rule $\langle (q, \gamma, \varphi), \sigma, j, b'\rangle \to \langle (q', c, \varphi'), \mathrm{drop}_{(c,\gamma)}\rangle$ where $\varphi' = \varphi$ if $c \in C_{\mathrm{i}}$ and $\varphi' = \varphi(c, \gamma)$ otherwise (provided $|\varphi| < k$ and (c, γ) does not occur in φ). Now let $\alpha = \mathrm{lift}_c$ and $\gamma = c$. If $c \in C_{\mathrm{i}}$ and $(c, \gamma') \in b'$, then $\mathcal{A}'$ has the rule $\langle (q, \gamma, \varphi), \sigma, j, b'\rangle \to \langle (q', \gamma', \varphi), \mathrm{lift}_{(c,\gamma')}\rangle$. If $c \in C_{\mathrm{v}}$, then $\mathcal{A}'$ has the rule $\langle (q, \gamma, \varphi(c, \gamma')), \sigma, j, b'\rangle \to \langle (q', \gamma', \varphi), \mathrm{lift}_{(c,\gamma')}\rangle$ for every $\gamma' \in C \cup \{\varepsilon\}$ such that (c, γ') does not occur in φ (with $|\varphi| < k$). □

Using this lemma, we now show that every v^+-PTT can be decomposed into TT's, as already shown in [27] in a different way.[29]

Lemma 59. *For every $k \geq 1$, $\mathsf{V}_k^+\mathsf{\text{-}PTT} \subseteq \mathsf{TT} \circ \mathsf{V}_{k-1}^+\mathsf{\text{-}PTT}$. For fixed k, the construction takes polynomial time.*

Proof. Let $\mathcal{M} = (\Sigma, \Delta, Q, Q_0, C, C_{\mathrm{v}}, C_{\mathrm{i}}, R, k)$ be a v_k^+-PTT with $C_{\mathrm{i}} = \varnothing$. The construction is similar to the one in the proof of Lemma 3, except that we use the nondeterministic "multi-level" preprocessor $\mathcal{N}$ of the proof of Lemma 4, for which we assume that $C_{\mathrm{v}} = [1, \gamma]$.

By Lemma 58 we may assume that the simulating transducer $\mathcal{M}'$ can perform stack tests. As in the proof of Lemma 3, $\mathcal{M}'$ starts by simulating $\mathcal{M}$ on the top level of the preprocessed version t' of the input tree t. When $\mathcal{M}$ drops the first pebble c on node u, $\mathcal{M}'$ enters the second level copy $\hat{t}_u$ of t corresponding to c, but it also stores c in its finite state. When $\mathcal{M}$ wants to lift pebble c and can actually do so because the pebble stack of $\mathcal{M}'$ is empty, $\mathcal{M}'$ removes c from its finite state and continues simulating $\mathcal{M}$ on $\hat{t}_u$. Note that since c can be lifted from a distance, $\mathcal{M}'$ cannot return to the top level without loosing its current position. When $\mathcal{M}$ again drops a pebble d on some second-level node that corresponds to a node v of t, $\mathcal{M}'$ enters the third level copy $\hat{t}_v$ corresponding to d, and stores d in its finite state. Thus, whenever $\mathcal{M}$ drops a bottom pebble, $\mathcal{M}'$ moves one level down in the "tree of trees" t'.

[29] In that paper the authors "think that those proofs cannot be generalized for the strong pebble case because the mapping EncPeb [···] is strongly based on weak pebble handling", where 'those proofs' mainly refers to the proof of [20, Lemma 9] in which the preprocessor is called EncPeb, see Lemma 3.

It should be noted that we could as well have taken $\gamma = 1$ for $\mathcal{N}$ and let $\mathcal{M}'$ enter the unique copy of t when it drops a pebble c, because $\mathcal{M}'$ keeps c in its finite state. However, the present construction simplifies the proof of Theorem 62.

Although the above description should be clear, let us give the formal details. As in the proof of Lemma 4, the output alphabet Γ of $\mathcal{N}$ is the union of $\{\perp\}$, $\{\sigma' \mid \sigma \in \Sigma\}$, and $\{\sigma'_{i,j} \mid \sigma \in \Sigma,\ i \in [0, \mathrm{rank}_\Sigma(\sigma)],\ j \in [0, mx_\Sigma]\}$ where, for every $\sigma \in \Sigma$ of rank m, σ' has rank $m+\gamma$ and $\sigma'_{i,j}$ has rank $m+\gamma+1$. As in the proof of Lemma 3, the v^+_{k-1}-PTT $\mathcal{M}'$ has input alphabet Γ, set of states $Q \cup (Q \times C_\mathrm{v})$, and the same initial states and pebble colours as $\mathcal{M}$. The rules of $\mathcal{M}'$ are defined as follows. Let $\langle q, \sigma, j, b\rangle \to \zeta$ be a rule of $\mathcal{M}$ with $\mathrm{rank}_\Sigma(\sigma) = m$.

First, we consider the behaviour of $\mathcal{M}'$ in state q, where we assume that $b = \varnothing$. Then $\mathcal{M}'$ has the rules $\langle q, \sigma', j, \varnothing\rangle \to \zeta_1$, $\langle q, \sigma'_{0,j}, j', \varnothing\rangle \to \zeta_2$, and $\langle q, \sigma'_{i,j}, j', \varnothing\rangle \to \zeta_{3,i}$ for every $i \in [1,m]$ and $j' \in [1, mx_\Gamma]$, where ζ_1 is obtained from ζ by changing $\langle q', \mathrm{drop}_c\rangle$ into $\langle (q', c), \mathrm{down}_{m+c}\rangle$ for every $q' \in Q$ and $c \in C_\mathrm{v}$, ζ_2 is obtained from ζ_1 by changing up into $\mathrm{down}_{m+\gamma+1}$, and $\zeta_{3,i}$ is obtained from ζ_2 by changing down_i into up. Thus, whenever the pebble stack of $\mathcal{M}$ is empty, $\mathcal{M}'$ simulates $\mathcal{M}$ on a copy of the input tree t in t', until $\mathcal{M}$ drops a pebble $c \in C_\mathrm{v}$. Then $\mathcal{M}'$ steps to the next level, and stores c in its finite state.

Second, we consider the behaviour of $\mathcal{M}'$ in state (q,c), where $c \in C_\mathrm{v}$. Rules of $\mathcal{M}'$ that have $\sigma'_{0,j}$ in their left-hand side are defined under the proviso that $c \in b$, and the other rules under the proviso that $c \notin b$. If $\zeta = \langle q', \mathrm{lift}_c\rangle$, then $\mathcal{M}'$ has the rules $\langle (q,c), \sigma', j, b, \varepsilon\rangle \to \langle q', \mathrm{stay}\rangle$, $\langle (q,c), \sigma'_{0,j}, m+c, b \setminus \{c\}, \varepsilon\rangle \to \langle q', \mathrm{stay}\rangle$, and $\langle (q,c), \sigma'_{i,j}, j', b, \varepsilon\rangle \to \langle q', \mathrm{stay}\rangle$ for every $i \in [1,m]$ and $j' \in [1, mx_\Gamma]$, where ε is the stack test that checks emptiness of the stack of $\mathcal{M}'$. Thus, when $\mathcal{M}$ lifts pebble c (at the position of c or from a distance), $\mathcal{M}'$ removes c from memory and knows that the pebble stack of $\mathcal{M}$ is empty. Otherwise, $\mathcal{M}'$ has the rules $\langle (q,c), \sigma', j, b\rangle \to \zeta_{c,1}$, $\langle (q,c), \sigma'_{0,j}, m+c, b \setminus \{c\}\rangle \to \zeta_{c,2}$, and $\langle (q,c), \sigma'_{i,j}, j', b\rangle \to \zeta_{c,3,i}$ for every $i \in [1,m]$ and $j' \in [1, mx_\Gamma]$, where $\zeta_{c,1}$ is obtained from ζ by changing every occurrence of a state q' into (q',c), $\zeta_{c,2}$ is obtained from $\zeta_{c,1}$ by changing up into $\mathrm{down}_{m+\gamma+1}$, and $\zeta_{c,3,i}$ is obtained from $\zeta_{c,2}$ by changing down_i into up. Thus, $\mathcal{M}'$ simulates $\mathcal{M}$ on a copy of the input tree in t', assuming that c is present on the node with label $\sigma'_{0,j}$ and absent on the other nodes, until c is lifted by $\mathcal{M}$. □

The next result is an immediate consequence of Lemma 59. It was proved in [27, Theorem 6.5(5)], generalizing the same result for weak pebbles in [20, Theorem 10] (cf. Theorem 55). It implies that Propositions 6(2) and 7(2) also hold for strong pebbles. Thus, for PTT's without invisible pebbles, the inverse type inference problem and the typechecking problem are solvable for strong pebbles in the same time as for weak pebbles (cf. [27, Theorem 6.7 and 6.9]). Note that it also implies that the domains of v^+-PTT's are regular (cf. Corollary 9), which was proved in [27, Corollary 6.8] and [44, Theorem 4.7].

Theorem 60. *For every $k \geq 0$, V^+_k-PTT $\subseteq \mathsf{TT}^{k+1}$. For fixed k, the construction takes polynomial time.*

To prove a similar result for deterministic PTT's with strong pebbles, we need the next small lemma.

Lemma 61. *For every $k \geq 1$, $(\mathsf{tdTT}^{\mathrm{MSO}})^k \subseteq \mathsf{dTT}_\downarrow \circ \mathsf{dTT}^k$.*

Proof. We will show by induction on k that for every $\tau \in (\mathsf{tdTT}^{\mathrm{MSO}})^k$ there exist $\tau_0 \in \mathsf{dTT}_\downarrow$ and $\tau_1, \ldots, \tau_k \in \mathsf{dTT}$ such that $\tau = \tau_0 \circ \tau_1 \circ \cdots \circ \tau_k$. Note that since τ is a total function, every output tree of $\tau_0 \circ \tau_1 \circ \cdots \circ \tau_{i-1}$ is in the domain of τ_i, for every $i \in [1,k]$. For $k = 1$ the statement is immediate from the inclusion $\mathsf{dTT}^{\mathrm{MSO}} \subseteq \mathsf{dTT}_\downarrow \circ \mathsf{dTT}$, which follows from Lemmas 49 and 50. Now consider $\tau \in (\mathsf{tdTT}^{\mathrm{MSO}})^{k+1}$. By induction and the case $k = 1$, $\tau = \tau_0 \circ \tau_1 \circ \cdots \circ \tau_k \circ \tau'_0 \circ \tau'_1$ with $\tau_0, \tau'_0 \in \mathsf{dTT}_\downarrow$ and $\tau_1, \ldots, \tau_k, \tau'_1 \in \mathsf{dTT}$. Since every output tree of $\tau_0 \circ \tau_1 \circ \cdots \circ \tau_{k-1}$ is in the domain of τ_k, we can replace $\tau_k \circ \tau'_0$ by any transduction τ' such that $\tau'(t) = \tau'_0(\tau_k(t))$ for every t in the domain of τ_k. Since $\tau_k \in \mathsf{dTT}$ and $\tau'_0 \in \mathsf{dTT}_\downarrow$, we can take $\tau' \in \mathsf{dTT}$ by the proof of Theorem 55. □

Theorem 60 was also shown in [20, Theorem 10] for weak pebbles in the deterministic case. Here we need one more deterministic TT.

Theorem 62. *For every $k \geq 1$, V^+_k-dPTT $\subseteq \mathsf{dTT}_\downarrow \circ \mathsf{dTT}^{k+1}$.*

Proof. By Lemma 61 it suffices to show that V^+_k-dPTT $\subseteq \mathsf{tdTT}^{\mathrm{MSO}} \circ \mathsf{V}^+_{k-1}$-dPTT for every $k \geq 1$. The proof of this inclusion is obtained from the proof of Lemma 59 by changing the preprocessor $\mathcal{N}$ in a similar way as in the proof of Lemma 52.

For the given deterministic v^+_k-PTT $\mathcal{M}$ we assume that $C_\mathrm{i} = \varnothing$ and $C = C_\mathrm{v} = [1, \gamma]$. As in the proof of Lemma 52, we may assume that there is a mapping $\chi : C \to Q$ that specifies the state of $\mathcal{M}$ after dropping a pebble (because we may also assume that $\mathcal{M}$ keeps track in its finite state of the pebbles in its stack, in the order in which they were dropped, cf. the proof of Lemma 58).

The new preprocessor $\mathcal{N}'$ is constructed in the same way as in the proof of Lemma 52, with a different definition of the trips $T_{c,d}$. For $c \in C$, the site T_c is defined as in that proof, i.e., it consists of all pairs (t,u) such that the configuration $\langle \chi(c), u, (u,c)\rangle$ is reachable by the automaton $\mathcal{A}$ associated with $\mathcal{M}$, which now is a nondeterministic v^+_k-PTA. The automaton $\mathcal{B}$ recognizing $\mathrm{mark}(T_c)$ is a v^+_k-PTA with stack tests (see Lemma 58). When it arrives at the marked node u in state $\chi(c)$

and observes c, it may check that c is the top pebble, lift it, check that the stack is now empty, and accept. For $c, d \in C$, the trip $T_{c,d}$ now consists of all triples (t, u, v) such that $\mathcal{A}$ has a computation on t starting in configuration $\langle \chi(c), u, (u, c)\rangle$ and ending in configuration $\langle \chi(d), v, (v, d)\rangle$, with at least one computation step. It should be clear that there is a v_k^+-PTA $\mathcal{B}'$ with stack tests that recognizes $\mathrm{mark}(T_{c,d})$: it starts by dropping c on marked node u in state $\chi(c)$, and then behaves similarly to $\mathcal{B}$ (with respect to v and d).

For every input tree t in the domain of $\mathcal{M}$, the preprocessor $\mathcal{N}'$ produces the appropriate output. In fact, if $\mathcal{N}'$ would not produce output, then there would be an infinite sequence $(u_1, c_1), (u_2, c_2), \dots$ such that $(t, u_1) \in T_{c_1}$ and $(t, u_i, u_{i+1}) \in T_{c_i, c_{i+1}}$ for every $i \geq 1$. But that would imply the existence of an infinite computation of $\mathcal{M}$ on t that starts in the initial configuration, contradicting the determinism of $\mathcal{M}$. □

Next, we decompose arbitrary v^+I-PTT's. To do that we need two TT's at each decomposition step rather than one.

Lemma 63. *For every $k \geq 1$, $\mathsf{V}_k^+\text{I-PTT} \subseteq \mathsf{TT} \circ \mathsf{TT} \circ \mathsf{V}_{k-1}^+\text{I-PTT}$. For fixed k, the construction takes polynomial time.*

Proof. The proof of Lemma 59 is also valid for v^+I-PTT, provided every reachable pebble stack of the given transducer has a visible bottom pebble (for the definition of reachable pebble stack see the proof of Lemma 52). Thus, it suffices to construct for every v_k^+I-PTT $\mathcal{M}$ a TT $\mathcal{N}$ and a v_k^+I-PTT $\mathcal{M}'$ with that property, such that $\tau_{\mathcal{N}} \circ \tau_{\mathcal{M}'} = \tau_{\mathcal{M}}$.

Let $\mathcal{M} = (\Sigma, \Delta, Q, Q_0, C, C_\mathrm{v}, C_\mathrm{i}, R, k)$. The construction is similar to the one in the proof of Lemma 4. In particular, we assume that $C_\mathrm{i} = [1, \gamma]$ and we use the same nondeterministic "multi-level" preprocessor $\mathcal{N}$ of that proof. The simulating transducer $\mathcal{M}'$ works in the same way as the one in the proof of Lemma 4 as long as the pebble stack of $\mathcal{M}$ consists of invisible pebbles only. Thus, during that time the pebble stack of $\mathcal{M}'$ is empty. As soon as $\mathcal{M}$ drops a visible pebble c, $\mathcal{M}'$ stays in the same copy of the input tree and also drops c. After that, $\mathcal{M}'$ just simulates $\mathcal{M}$ on that copy until $\mathcal{M}$ lifts c. Then $\mathcal{M}'$ also lifts c and returns to the first mode until $\mathcal{M}$ again drops a visible pebble. Note that when $\mathcal{M}$ drops c, all invisible pebbles on the input tree become unobservable until c is lifted.

Formally, the set of states of $\mathcal{M}'$ is the union of Q (used in the first mode) and $Q \times C_\mathrm{v}$ (used in the second mode). The rules for the first mode are the same as in the proof of Lemma 4, with the empty set of pebble colours added to each left-hand side. Now let $\langle q, \sigma, j, b\rangle \to \zeta$ be a rule of $\mathcal{M}$ and $\mathrm{rank}_\Sigma(\sigma) = m$. In what follows, i ranges over $[1, m]$ and j' over $[1, mx_\Gamma]$, as usual. With the following rules $\mathcal{M}'$ switches from the first to the second mode, where we assume that $\zeta = \langle q', \mathrm{drop}_c\rangle$ with $c \in C_\mathrm{v}$: if $b = \{d\}$ with $d \in C_\mathrm{i}$, then it uses the rule $\langle q, \sigma_{0,j}, m + d, \varnothing\rangle \to \langle (q', c), \mathrm{drop}_c\rangle$, and if $b = \varnothing$, then it uses the rules $\langle q, \sigma', j, \varnothing\rangle \to \langle (q', c), \mathrm{drop}_c\rangle$ and $\langle q, \sigma_{i,j}, j', \varnothing\rangle \to \langle (q', c), \mathrm{drop}_c\rangle$. The rules for the second mode are as follows, for every $c \in C_\mathrm{v}$. We first assume that ζ does not contain the instruction lift_c. Then $\mathcal{M}'$ has the rules $\langle (q, c), \sigma', j, b\rangle \to \zeta_1$, $\langle (q, c), \sigma_{0,j}, j', b\rangle \to \zeta_2$, and $\langle (q, c), \sigma_{i,j}, j', b\rangle \to \zeta_{3,i}$, where ζ_1 is obtained from ζ by changing every state q' into (q', c), ζ_2 is obtained from ζ_1 by changing up into $\mathrm{down}_{m+\gamma+1}$, and $\zeta_{3,i}$ is obtained from ζ_2 by changing down_i into up. Finally, if $\zeta = \langle q', \mathrm{lift}_c\rangle$, then $\mathcal{M}'$ switches from the second to the first mode with the following rules: $\langle (q, c), \sigma', j, b\rangle \to \zeta$, $\langle (q, c), \sigma_{0,j}, j', b\rangle \to \zeta$, and $\langle (q, c), \sigma_{i,j}, j', b\rangle \to \zeta$. □

The next result is immediate from Lemmas 63 and 4. It implies, by Propositions 6(1) and 7(1), that the inverse type inference problem and the typechecking problem are solvable for PTT's with k strong visible pebbles, in $(2k+2)$-fold and $(2k+3)$-fold exponential time, respectively. It also implies that the domains of v^+I-PTT's are regular, cf. Corollary 9.

Theorem 64. *For every $k \geq 0$, $\mathsf{V}_k^+\text{I-PTT} \subseteq \mathsf{TT}^{2k+2}$. For fixed k, the construction takes polynomial time.*

Applying the techniques in the proofs of Lemma 52 and Theorem 62 to the proof of Lemma 63, and using Lemmas 52 and 61, we obtain that every deterministic v^+I-PTT can be decomposed into deterministic TT's, cf. Theorem 55. The formal proof is straightforward.

Theorem 65. *For every $k \geq 0$, $\mathsf{V}_k^+\text{I-dPTT} \subseteq \mathsf{dTT}_\downarrow \circ \mathsf{dTT}^{2k+2}$.*

We do not know whether these results are optimal, i.e., whether the exponent $2k+2$ can be lowered.

16. Conclusion

We have shown in Theorem 5 that $\mathsf{V}_k\text{I-PTT} \subseteq \mathsf{TT}^{k+2}$, but we do not know whether this is optimal, i.e., whether or not $\mathsf{V}_k\text{I-PTT} \subseteq \mathsf{TT}^{k+1}$. Since the results on typechecking in Section 5 are based on this decomposition, we also do not know whether the time bound for typechecking v_kI-PTT's, as stated in Theorem 8, is optimal. Using the results of [48], it can be shown that the time bound for inverse type inference is optimal, cf. the discussion after [14, Corollary 1].

We have shown in Theorem 29 that all MSO definable n-ary patterns can be matched by deterministic v_{n-2}I-PTT's, but we do not know whether this is optimal, i.e., whether or not it can be done with less than $n-2$ pebbles. In particular, we do not know whether or not all MSO definable ternary patterns can be matched by I-PTT's (or, by TL programs), cf. Theorem 57.

In Section 10 we have suggested ways of reducing the number of visible pebbles in special cases. Given an MSO formula φ, can one compute the minimal number of visible pebbles that is needed to match the pattern φ?

The language TL can be extended with visible pebbles, in an obvious way. The resulting "pebble TL programs" are closely related to the *pebble macro tree transducers* that were introduced in [20]. What is the relationship between the k-pebble macro tree transducer and the V_kI-PTT? Is there an analogon of Theorem 46? It is not clear whether the proof of Theorem 46 can be generalized to the addition of visible pebbles.

We have shown in Theorem 56 that V_kI-dPTT $\subsetneq \mathsf{V}_{k+1}$I-dPTT, i.e., that $k+1$ visible pebbles are more powerful than k, in the deterministic case. We do not know whether this holds for the nondeterministic transducers, i.e., whether or not the inclusion V_kI-PTT $\subseteq \mathsf{V}_{k+1}$I-PTT is proper. We also do not know whether every functional V_kI-PTT can be simulated by a deterministic one, where a PTT $\mathcal{M}$ is functional if $\tau_{\mathcal{M}}$ is a function. If so, then the inclusion would of course be proper.

Is it decidable for a given deterministic V_{k+1}I-PTT $\mathcal{M}$ whether or not $\tau_{\mathcal{M}}$ is in V_kI-dPTT? If so, then one could compute the minimal number of visible pebbles needed to realize the transformation $\tau_{\mathcal{M}}$ by a PTT. Obviously, that would answer the above question for the pattern φ in the affirmative.

It is proved in [8] that the V_k^+-PTA has the same expressive power as the V_k-PTA, i.e., that strong pebbles are not more powerful than weak pebbles. We do not know whether or not the V_k^+-PTT is more powerful than the V_k-PTT, and neither whether or not the V_k^+I-PTT is more powerful than the V_kI-PTT.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web, Morgan Kaufmann, 2000.
[2] A.V. Aho, J.D. Ullman, Translations on a context-free grammar, Inf. Control 19 (1971) 439–475.
[3] M. Bartha, An algebraic definition of attributed transformations, Acta Cybern. 5 (1982) 409–421, preliminary version in: F. Gécseg (Ed.), Proc. FCT'81, in: Lecture Notes in Computer Science, vol. 117, Springer-Verlag, 1981, pp. 51–60.
[4] G.J. Bex, S. Maneth, F. Neven, A formal model for an expressive fragment of XSLT, Inf. Syst. 27 (2002) 21–39.
[5] R. Bloem, J. Engelfriet, Monadic second order logic and node relations on graphs and trees, in: J. Mycielski, G. Rozenberg, A. Salomaa (Eds.), Structures in Logic and Computer Science, in: Lecture Notes in Computer Science, vol. 1261, Springer-Verlag, 1997, pp. 144–161, a corrected version is available at https://www.researchgate.net/publication/221350026.
[6] R. Bloem, J. Engelfriet, A comparison of tree transductions defined by monadic second order logic and by attribute grammars, J. Comput. Syst. Sci. 61 (2000) 1–50.
[7] M. Bojanczyk, Tree-walking automata, in: C. Martín-Vide, F. Otto, H. Fernau (Eds.), Proc. LATA'08, in: Lecture Notes in Computer Science, vol. 5196, Springer-Verlag, 2008, pp. 1–2, full version available at https://www.mimuw.edu.pl/~bojan/upload/conflataBojanczyk08.pdf.
[8] M. Bojanczyk, M. Samuelides, T. Schwentick, L. Segoufin, Expressive power of pebble automata, in: M. Bugliesi, B. Preneel, V. Sassone, I. Wegener (Eds.), Proc. ICALP'06, in: Lecture Notes in Computer Science, vol. 4051, Springer-Verlag, 2006, pp. 157–168.
[9] A. Brüggemann-Klein, D. Wood, Caterpillars, context, tree automata and tree pattern matching, in: Proc. DLT'99, World Scientific, 1999, pp. 270–285.
[10] B. Courcelle, J. Engelfriet, Graph Structure and Monadic Second-Order Logic, Cambridge University Press, 2012.
[11] J. Doner, Tree acceptors and some of their applications, J. Comput. Syst. Sci. 4 (1970) 406–451.
[12] J. Engelfriet, Simple Program Schemes and Formal Languages, Lecture Notes in Computer Science, vol. 20, Springer-Verlag, 1974.
[13] J. Engelfriet, Context-free grammars with storage, Technical Report 86-11, University of Leiden, 1986, a slightly revised version is available at arXiv: 1408.0683.
[14] J. Engelfriet, The time complexity of typechecking tree-walking tree transducers, Acta Inform. 46 (2009) 139–154.
[15] J. Engelfriet, H.J. Hoogeboom, Tree-walking pebble automata, in: J. Karhumäki, H. Maurer, G. Paun, G. Rozenberg (Eds.), Jewels Are Forever, Contributions to Theoretical Computer Science in Honor of Arto Salomaa, Springer-Verlag, 1999, pp. 72–83.
[16] J. Engelfriet, H.J. Hoogeboom, Automata with nested pebbles capture first-order logic with transitive closure, Log. Methods Comput. Sci. 3 (2:3) (2007) 1–27.
[17] J. Engelfriet, H.J. Hoogeboom, J.-P. van Best, Trips on trees, Acta Cybern. 14 (1999) 51–64.
[18] J. Engelfriet, H.J. Hoogeboom, B. Samwel, XML transformation by tree-walking transducers with invisible pebbles, in: L. Libkin (Ed.), Proc. PODS'07, ACM Press, 2007, pp. 63–72.
[19] J. Engelfriet, S. Maneth, Macro tree transducers, attribute grammars, and MSO definable tree translations, Inf. Comput. 154 (1999) 34–91.
[20] J. Engelfriet, S. Maneth, A comparison of pebble tree transducers with macro tree transducers, Acta Inform. 39 (2003) 613–698.
[21] J. Engelfriet, E.M. Schmidt, IO and OI, II, J. Comput. Syst. Sci. 16 (1978) 67–99.
[22] J. Engelfriet, H. Vogler, Macro tree transducers, J. Comput. Syst. Sci. 31 (1985) 71–146.
[23] J. Engelfriet, H. Vogler, Pushdown machines for the macro tree transducer, Theor. Comput. Sci. 42 (1986) 251–368, Theor. Comput. Sci. 48 (1986) 339 (Erratum).
[24] M.J. Fischer, Grammars with macro-like productions, Ph.D. Thesis, Harvard University, 1968.
[25] Z. Fülöp, On attributed tree transducers, Acta Cybern. 5 (1981) 261–279.
[26] Z. Fülöp, H. Vogler, Syntax-Directed Semantics – Formal Models Based on Tree Transducers, Springer-Verlag, 1998.
[27] Z. Fülöp, L. Muzamel, Pebble macro tree transducers with strong pebble handling, Fundam. Inform. 89 (2008) 1–51.
[28] M.R. Garey, D.S. Johnson, Computers and Intractability – a Guide to the Theory of NP-Completeness, W.H. Freeman, San Francisco, 1979.
[29] F. Gécseg, M. Steinby, Tree Automata, Akadémiai Kiadó, Budapest, 1984, a re-edition is available at arXiv:1509.06233.
[30] E. Goris, M. Marx, Looping caterpillars, in: Proc. LICS'05, IEEE, 2005, pp. 51–60.
[31] G. Gottlob, C. Koch, R. Pichler, Efficient algorithms for processing XPath queries, in: Proc. VLDB'02, Morgan Kaufmann, 2002, pp. 95–106.
[32] G. Gottlob, C. Koch, K.U. Schulz, Conjunctive queries over trees, in: Proc. PODS'04, ACM Press, 2004, pp. 189–200.
[33] W. Janssen, A. Korlyukov, J. Van den Bussche, On the tree-transformation power of XSLT, Acta Inform. 43 (2007) 371–393.
[34] D.E. Knuth, Fundamental Algorithms, Addison-Wesley, 1968.

[35] A. Kühnemann, H. Vogler, Synthesized and inherited functions – a new computational model for syntax-directed semantics, Acta Inform. 31 (1994) 431–477.
[36] S. Maneth, The macro tree transducer hierarchy collapses for functions of linear size increase, in: P.K. Pandya, J. Radhakrishnan (Eds.), Proc. FSTTCS'03, in: Lecture Notes in Computer Science, vol. 2914, Springer-Verlag, 2003, pp. 326–337.
[37] S. Maneth, A. Berlea, T. Perst, H. Seidl, XML type checking with macro tree transducers, in: Proc. PODS'05, ACM Press, 2005, pp. 283–294, Technical Report TUM-I0407 of the Technische Universität München (2004) is available at https://www.researchgate.net/publication/221559877.
[38] S. Maneth, F. Neven, Structured document transformation based on XSL, in: Proc. DBPL'99, in: Lecture Notes in Computer Science, vol. 1949, Springer-Verlag, 2000, pp. 80–98.
[39] M. Marx, Conditional XPath, ACM Trans. Database Syst. 30 (2005) 929–959.
[40] M. Marx, Navigation in XML trees, in: The Logic in Computer Science Column, Bull. Eur. Assoc. Theor. Comput. Sci. 88 (February 2006) 126–140.
[41] T. Milo, D. Suciu, V. Vianu, Typechecking for XML transformers, J. Comput. Syst. Sci. 66 (2003) 66–97.
[42] A. Møller, M.I. Schwartzbach, The design space of type checkers for XML transformation languages, in: Proc. ICDT'05, in: Lecture Notes in Computer Science, vol. 3363, Springer-Verlag, 2005, pp. 17–36.
[43] A. Muscholl, M. Samuelides, L. Segoufin, Complementing deterministic tree-walking automata, Inf. Process. Lett. 99 (2006) 33–39.
[44] L. Muzamel, Pebble alternating tree-walking automata and their recognizing power, Acta Cybern. 18 (2008) 427–450.
[45] F. Neven, Automata, logic, and XML, in: J.C. Bradfield (Ed.), Proc. CSL'02, in: Lecture Notes in Computer Science, vol. 2471, Springer-Verlag, 2002, pp. 2–26.
[46] F. Neven, T. Schwentick, Automata- and logic-based pattern languages for tree-structured data, in: Semantics in Databases 2001, in: Lecture Notes in Computer Science, vol. 2582, Springer-Verlag, 2003, pp. 160–178.
[47] T. Perst, H. Seidl, Macro forest transducers, Inf. Process. Lett. 89 (2004) 141–149.
[48] M. Samuelides, L. Segoufin, Complexity of pebble tree-walking automata, in: E. Csuhaj-Varjú, Z. Ésik (Eds.), Proc. FCT'07, in: Lecture Notes in Computer Science, vol. 4639, Springer-Verlag, 2007, pp. 458–469.
[49] B. Samwel, Pebble scope and the power of pebble tree transducers, M.Sc. Thesis, LIACS, Leiden University, 2006.
[50] T. Schwentick, Automata for XML – a survey, J. Comput. Syst. Sci. 73 (2007) 289–315.
[51] G. Slutzki, Alternating tree automata, Theor. Comput. Sci. 41 (1985) 305–318.
[52] B. ten Cate, The expressivity of XPath with transitive closure, in: Proc. PODS'06, ACM Press, 2006, pp. 328–337.
[53] B. ten Cate, L. Segoufin, Transitive closure logic, nested tree walking automata, and XPath, J. ACM 57 (3) (2010) 18.
[54] J.W. Thatcher, J.B. Wright, Generalized finite automata theory with an application to a decision problem of second-order logic, Math. Syst. Theory 2 (1968) 57–81.