Acta Informatica manuscript No.
(will be inserted by the editor)

Joost Engelfriet

The time complexity of typechecking tree-walking tree transducers

the date of receipt and acceptance should be inserted later

Abstract Tree-walking tree transducers can be typechecked in double exponential time. More generally, compositions of k tree-walking tree transducers can be typechecked in $(k+1)$-fold exponential time. Consequently k-pebble tree transducers, which form a model of XML transformations and XML queries, can be typechecked in $(k+2)$-fold exponential time. The results hold for both ranked and unranked trees.

1 Introduction

In [15] the k-pebble tree transducer was introduced by Milo, Suciu, and Vianu as a formal model of XML document transformation languages and XML query languages. Here we consider the (slight) reformulation of the model as defined in [6]: the reading head (at a node of the input tree) is not viewed as a pebble. In particular, the 0-pebble tree transducer will be called *tree-walking tree transducer* (as the former name is rather strange for a tree transducer that does not use any pebbles). As explained in [6], the tree-walking tree transducer is closely related to the attribute grammar [13] and can, in fact, be viewed as a notational variation of the attributed tree transducer [7,8]; attribute grammars are a well-known compiler construction tool (see, e.g., [1]).

A central problem in the verification of XML transformation programs (and XML queries) is the *typechecking problem*: given such a program and two document types, one for its input documents and one for its output documents, check whether or not all output documents conform to the output type, assuming that only input documents are processed that conform to

Joost Engelfriet
LIACS, Leiden University
P.O.Box 9512, 2300 RA Leiden, The Netherlands
E-mail: engelfri@liacs.nl

the input type. Modelling XML documents as trees, we adopt (as in [15]) the regular tree languages as type formalism, and the regular tree grammars as document type definitions (DTDs). Modelling XML transformation programs as tree transducers from a certain class C, the typechecking problem is formalized as follows: given a transducer M from the class C and two regular tree grammars that generate tree languages (i.e., types) $L_{\rm in}$ and $L_{\rm out}$, check whether or not $\tau_M(L_{\rm in}) \subseteq L_{\rm out}$. Here τ_M denotes the tree transformation realized by M; in general τ_M is a binary relation, due to the nondeterminism of M (which means that an input document may be translated into any number of output documents). As shown in [15], the typechecking problem is closely related to the *inverse type inference problem*: given a transducer M from the class C and a regular tree grammar that generates a tree language $L'_{\rm out}$, construct a regular tree grammar generating the set $L'_{\rm in}$ of all input trees for which at least one of the output trees conforms to the output type, in other words $L'_{\rm in} = \tau_M^{-1}(L'_{\rm out})$, where τ_M^{-1} is the inverse of the relation τ_M. Assuming the inverse type inference problem to be solvable for the class C, the typechecking problem for C can be decided by taking $L'_{\rm out}$ to be the complement of $L_{\rm out}$ and then checking whether or not the intersection of $L_{\rm in}$ and $L'_{\rm in}$ is empty (which is effective for regular tree languages).

It is shown in [15] that (a variation of) the inverse type inference problem for k-pebble tree transducers is solvable, and hence their typechecking problem is decidable. Unfortunately, as also proved in [15], for varying k the time complexity of the typechecking problem for k-pebble tree transducers is extremely high: it is non-elementary, i.e., cannot be expressed by an iterated exponential function. The authors observe that "the main source of complexity is the number of pebbles of the transducer". In fact, it seems that for fixed k the time complexity of the algorithm in [15] is $(k+4)$-fold iterated exponential. Here we remove two exponentials and show that the problem can be decided in $(k+2)$-fold exponential time. This is based on the decomposition of k-pebble tree transducers into a $(k+1)$-fold composition of tree-walking tree transducers, proved in [6], and the fact that the inverse type inference problem for tree-walking tree transducers can be solved in exponential time. The latter fact was proved by Bartha for attributed tree transducers in [2], without mention of the time bound. To solve the inverse type inference problem for a composition of transducers, one can just repeat the inverse type inference algorithm for each of the transducers. Thus, for $(k+1)$-fold compositions of tree-walking tree transducers the inverse type inference problem can be solved in $(k+1)$-fold exponential time; the typechecking problem takes one more exponential, due to the complementation of the output type $L_{\rm out}$.

The results hold both for ranked trees and for unranked forests. They will first be shown for ranked trees, and in the last section (Section 7) for unranked forests by a reduction to the case of ranked trees, using an idea of Perst and Seidl in [17]. After some preliminaries in Section 2, we define the tree-walking tree transducer in Section 3. We also define the k-pebble tree transducer, of which the tree-walking tree transducer is the case $k = 0$; the details of that definition are needed in Section 6 only. In Section 4 a basic case of the inverse type inference problem for tree-walking tree transducers is considered: the case where the output type L'_{out} is unrestricted, i.e., consists of all possible output trees, and thus L'_{in} is the domain of the transformation τ_M. Section 5 contains the main results. As observed above, for the result on pebble tree transducers we need their decomposition into tree-walking tree transducers; this is sketched in Section 6, which can be skipped by the reader familiar with [6].

The results of this paper were announced in [5].

2 Preliminaries

For $m, n \in \mathbb{N}$ (with $\mathbb{N} = \{0, 1, 2, \dots\}$), $[m, n]$ denotes the interval $\{k \in \mathbb{N} \mid m \leq k \leq n\}$; note that if $m > n$, then $[m, n] = \emptyset$. For a set A, $\mathcal{P}(A)$ denotes the set of all subsets of A, and $\#(A)$ denotes its cardinality. For binary relations R_1 and R_2, their composition is defined as $R_1 \circ R_2 = \{(x, z) \mid \exists y : (x, y) \in R_1, (y, z) \in R_2\}$.

In all sections except the last, all trees considered are ranked, as is the case in [15,6] (the idea being that XML documents, which are basically unranked forests, can be coded as binary trees in a well-known way). For a ranked alphabet Σ we denote by $\mathrm{rank}_\Sigma(\sigma)$ the rank of $\sigma \in \Sigma$, and by mx_Σ the maximum of all these ranks. The set of all (ranked) trees over Σ is denoted T_Σ. For a tree $s \in T_\Sigma$, root_s denotes its root, and for every node u of s, ui denotes the ith child of u (if it has one) and $u\!\uparrow$ denotes its parent (if it has one). Every node u of s has a child number (which can be "read" by a tree transducer): it equals j (with $j \in [1, mx_\Sigma]$) if $u \neq \mathrm{root}_s$ and $u = u\!\uparrow j$ (i.e., u is the jth child of its parent), and it equals 0 if $u = \mathrm{root}_s$. A descendant of a node u of s is either u itself or a descendant of one of its children (and in the latter case it is a proper descendant of u). If v is a descendant of u, then u is an ancestor of v. A ranked alphabet Σ is binary (monadic) if all its elements have rank 2 or 0 (rank 1 or 0, respectively); the elements of T_Σ are then called binary trees (monadic trees, respectively).

We assume the reader to be familiar with context-free grammars. They will be used for two purposes: first, as a formal model of document types, and second, to define the computations of tree-walking tree transducers (and pebble tree transducers). In both cases we mainly consider a special type of context-free grammar: the regular tree grammar.

For technical convenience we allow a context-free grammar to have a set of initial nonterminals rather than just one. As usual, a context-free grammar is specified as a tuple $G = (N, T, R, \mathcal{S})$, where N is the nonterminal alphabet, $\mathcal{S} \subseteq N$ the set of initial nonterminals, T the terminal alphabet (disjoint with N), and R the finite set of productions, where each production is of the form $X \to \zeta$ with $X \in N$ and $\zeta \in (N \cup T)^*$. The language generated by G is $L(G) = \{w \in T^* \mid S \Rightarrow^* w \text{ for some } S \in \mathcal{S}\}$, where $\Rightarrow$ is the usual derivation relation. A context-free grammar is said to be *forward deterministic* if it has exactly one initial nonterminal, and distinct productions have distinct left-hand sides. Note that such a grammar generates at most one string; it will be used to capture determinism of tree-walking tree transducers. A context-free grammar is *backward deterministic* if distinct productions have distinct right-hand sides.

As a formal model of DTD (Document Type Definition) we take the regular tree grammar. A *regular tree grammar* is a context-free grammar $G = (N, T, R, \mathcal{S})$ such that (1) $T = \Sigma \cup P$ where Σ is a ranked alphabet and P consists of the comma and the left and right parentheses, and (2) every production is either of the form $X_0 \to \sigma(X_1, \dots, X_k)$ with $X_i \in N$ and $\sigma \in \Sigma$ of rank k, or of the form $X_0 \to X_1$ with $X_i \in N$. A regular tree grammar generates trees over Σ, i.e., $L(G) \subseteq T_\Sigma$. It will be specified as $G = (N, \Sigma, R, \mathcal{S})$ rather than $G = (N, T, R, \mathcal{S})$. A regular tree grammar G without chain rules, i.e., rules of the form $X_0 \to X_1$, will also be viewed as a recognizer of the tree language $L(G)$. As such, it is called a (bottom-up or top-down) *finite-state tree automaton.* Its nonterminals are then called states, its initial nonterminals are called final or initial states in the bottom-up or top-down case, respectively, and its productions are called transitions. A bottom-up finite-state tree automaton is *deterministic* if it is backward deterministic (as a context-free grammar); it is *total* if for every $\sigma(X_1, \dots, X_k)$ it has a production with that right-hand side. A *regular tree language* is a set of trees that can be generated by a regular tree grammar, or equivalently, recognized by a (total deterministic) bottom-up finite-state tree automaton.

Whenever we discuss the time complexity of an algorithm, the underlying formal model is the multitape Turing machine (though it will never be made explicit). Thus, when the algorithm has a tree grammar or tree transducer X as input, X will be encoded as a string, in the usual way, to be written on the input tape of the Turing machine, and the size of X is the length of that string. However, we deviate in one way from the usual encoding: we assume that *for a ranked alphabet Σ, the rank of each symbol is specified in unary notation* rather than decimal notation; thus, the size of Σ is $k \log k + \sum_{\sigma \in \Sigma} \mathrm{rank}_\Sigma(\sigma)$ with $k = \#(\Sigma)$, instead of the usual $k \log k + \sum_{\sigma \in \Sigma} \log(\mathrm{rank}_\Sigma(\sigma))$. This is a reasonable assumption because, except in trivial cases, each ranked symbol used in a tree grammar/transducer occurs in a context where it has as many arguments as its rank. Since this holds in particular for regular tree grammars, i.e., for type definitions, the assumption is irrelevant in the case of typechecking. Note that the assumption is vacuous when only binary alphabets Σ are considered; this is usual when modelling XML documents (cf. [15]).

3 Tree-walking tree transducers

A tree-walking tree transducer is a finite state device that translates trees into trees. The reading head of the transducer is a pointer to a node of the input tree, which can be moved along the edges of the tree; in this way the device "walks" on the input tree. In one move, depending on its current state and on the label and child number of the current node, the transducer either does not produce output, or produces one node of the output tree. In the first case, it moves to a neighbor of the current node (or stays where it is), and changes state. In the second case, the transducer spawns k independent copies of itself, where k is the rank of the produced output node; each of the copies changes state, independently. In the remaining computation, the ith copy will produce the ith subtree of the produced output node. Thus, the output tree is generated in a top-down fashion.

Formally, a *tree-walking tree transducer* (in short, *twtt*) is a tuple $M = (\Sigma, \Delta, Q, Q_0, R)$, where Σ and Δ are ranked alphabets of input and output symbols, respectively, Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, and R is a finite set of rules. A rule is of the form $\langle q, \sigma, j\rangle \to \zeta$ with $q \in Q$, $\sigma \in \Sigma$, $j \in [0, mx_\Sigma]$, and ζ is of one of the forms

1. $\langle q', \mathrm{stay}\rangle$
2. $\langle q', \mathrm{up}\rangle$, provided $j \neq 0$
3. $\langle q', \mathrm{down}_i\rangle$ with $1 \leq i \leq \mathrm{rank}_\Sigma(\sigma)$
4. $\delta(\langle q_1, \mathrm{stay}\rangle, \ldots, \langle q_k, \mathrm{stay}\rangle)$

with $q' \in Q$, $\delta \in \Delta$, $k = \mathrm{rank}_\Delta(\delta)$, and $q_1, \ldots, q_k \in Q$.

As discussed at the end of the previous section, the *size of* M is the length of the string that encodes M in the usual way, with the (unusual) convention that the ranks of input and output symbols are encoded in unary notation. Note that, by this convention, if M has size n, then $mx_\Sigma \leq n$. Instead of adopting this convention, we could have required the left-hand side of each rule to contain the sequence $(\mathrm{down}_1, \ldots, \mathrm{down}_m)$ of possible down-instructions, where m is the rank of σ; this is usual in the case of top-down tree transducers (and macro tree transducers), with down_i denoted as a variable x_i.

The twtt M is *deterministic* if it has exactly one initial state, and distinct rules have distinct left-hand sides. It is *total* if for every $\langle q, \sigma, j\rangle$ it has a rule with that left-hand side.

For every input tree $s \in T_\Sigma$ we define a regular tree grammar $G_{M,s}$. As nonterminals it has all pairs $\langle q, u\rangle$ with $q \in Q$ and u a node of s. The initial nonterminals are all $\langle q_0, \mathrm{root}_s\rangle$ with $q_0 \in Q_0$, and the terminal ranked alphabet is Δ. If $\langle q, \sigma, j\rangle \to \zeta$ is a rule of M, then $G_{M,s}$ has a production $\langle q, u\rangle \to \zeta'$ for every node u of s with label σ and child number j, where ζ' equals

1. $\langle q', u\rangle$
2. $\langle q', u{\uparrow}\rangle$
3. $\langle q', ui\rangle$
4. $\delta(\langle q_1, u\rangle, \ldots, \langle q_k, u\rangle)$

respectively (according to the possible forms of ζ mentioned above). Intuitively, the computations of M on input s are the (maximal) derivations of the grammar $G_{M,s}$. The *translation realized by* M, denoted τ_M, is defined as $\tau_M = \{(s,t) \in T_\Sigma \times T_\Delta \mid t \in L(G_{M,s})\}$. Note that if M is deterministic then every $G_{M,s}$ is forward deterministic, and hence τ_M is a (partial) function.

For technical reasons we also need (in the proofs of Lemma 1 and Theorem 1) certain subgrammars of $G_{M,s}$. For a node u of input tree $s \in T_\Sigma$ we define the regular tree grammar $G_{M,s,u}$ to be the same as $G_{M,s}$ except that its set of productions consists of all productions of $G_{M,s}$ with left-hand side $\langle q, v\rangle$ where $q \in Q$ and v is a descendant of u. Intuitively, the derivations of $G_{M,s,u}$ that start with some $\langle q, u\rangle$, model the computations of M on the subtree of s with root u. Note that the right-hand side of a production of $G_{M,s,u}$ may contain a nonterminal $\langle q', u{\uparrow}\rangle$.

When disregarding the output of the twtt, we obtain the *alternating tree-walking automaton* (see, e.g., [19, Definition 4.1]). Thus, the domains of the twtt translations are the tree languages recognized by such automata. Formally, an alternating tree-walking automaton is a tuple $M = (\Sigma, Q, Q_0, R)$, defined in exactly the same way as a twtt, except that the fourth form of a right-hand side ζ is $\langle q_1, \mathrm{stay}\rangle \cdots \langle q_k, \mathrm{stay}\rangle$ with $k \in \mathbb{N}$ and $q_1, \dots, q_k \in Q$ (note that this makes the first form superfluous). The computations of M are defined analogously to the twtt, but here $G_{M,s}$ is a context-free grammar with empty terminal alphabet (and thus its sentential forms consist of nonterminals only). In the fourth case, the right-hand side ζ' of the production of $G_{M,s}$ is $\langle q_1, u\rangle \cdots \langle q_k, u\rangle$. The tree language recognized by M is $L(M) = \{s \in T_\Sigma \mid \varepsilon \in L(G_{M,s})\}$, where ε is the empty string.

The k-pebble tree transducer [15] works in the same way as the twtt, but additionally has k pebbles at its disposal, which it can use to temporarily mark the nodes of the input tree. The pebbles are dropped and lifted in a LIFO fashion: the pebble that was dropped last, must be lifted first (otherwise, typechecking would not be decidable). The transducer can detect how many pebbles are placed on the input tree and which of those are placed on the current node; it can drop or lift pebbles at the current node only.

For completeness sake we now present the formal definition of the pebble tree transducer (following its reformulation in [6]); the details will only be needed in Section 6. For $k \in \mathbb{N}$, a *k-pebble tree transducer* is a tuple $M = (\Sigma, \Delta, Q, Q_0, R)$, where the components are as for the twtt, but the rules in R are different. A rule is of the form $\langle q, \sigma, j, b\rangle \to \zeta$ with q, σ, and j as for the twtt, b is a mapping $b : [1, l] \to \{0, 1\}$ for some $l \in [0, k]$, and ζ is of one of the forms 1–4 as for the twtt or of one of the forms

5. $\langle q', \mathrm{drop}\rangle$, provided $l < k$
6. $\langle q', \mathrm{lift}\rangle$, provided $l \geq 1$ and $b(l) = 1$.

For $s \in T_\Sigma$, the nonterminals of the regular tree grammar $G_{M,s}$ are all triples $\langle q, u, \pi\rangle$ where q and u are the same as for the twtt and π is a mapping $\pi : [1, l] \to N_s$ for some $l \in [0, k]$, with N_s denoting the set of nodes of s. Intuitively, π is a stack of l pebbles, that are placed on nodes of the input tree, the other $k - l$ pebbles being unused; l is the top of the stack, i.e., $\pi(l)$ is the position of the pebble that was dropped last. The initial nonterminals are all $\langle q_0, \mathrm{root}_s, \emptyset\rangle$ with $q_0 \in Q_0$ and $\emptyset$ is the empty mapping (which is the unique mapping $[1, 0] \to N_s$). If $\langle q, \sigma, j, b\rangle \to \zeta$ is a rule of M with $b : [1, l] \to \{0, 1\}$, then $G_{M,s}$ has a production $\langle q, u, \pi\rangle \to \zeta'$ for every node u of s with label σ and child number j, and every $\pi : [1, l] \to N_s$ such that $b(m) = 1$ if and only if $\pi(m) = u$ (for all $m \in [1, l]$), and ζ' equals

5. $\langle q', u, \pi \cup \{(l + 1, u)\}\rangle$
6. $\langle q', u, \pi \setminus \{(l, u)\}\rangle$

respectively; in cases 1–4, ζ' is obtained from the one of the twtt by adding π to all nonterminals (because the pebbles are unchanged in these cases; note that in the fourth case the pebble stack is copied). The translation realized by M is defined as for the twtt.

Note that the 0-pebble tree transducer is just the twtt (disregarding the b's and π's, which are all the empty mapping). Note also that *by definition*, for each input tree s, the set $\tau_M(s)$ is a regular tree language; moreover, since there are $O(n^k)$ possible pebble stacks, where n is the size of s, the grammar $G_{M,s}$ can be constructed from s in polynomial time, cf. Proposition 3.8 of [15]: each pebble tree transducer has polynomial time data complexity.

The *alternating k-pebble tree automaton* (cf. [15, Definition 4.5]) is obtained from the k-pebble tree transducer in the same way as the alternating tree-walking automaton is obtained from the twtt, changing the fourth form of the right-hand side of a rule. These automata recognize exactly the domains of the k-pebble tree translations.

4 Domains

The main idea in the solution of the typechecking problem for k-pebble tree transducers in [15] is that the domain of such a transducer is (effectively) a regular tree language: a special case of the inverse type inference problem. In this section we present, for $k = 0$, i.e., for the twtt, a more efficient algorithm to construct a regular tree grammar for the domain.

As explained in Section 3.2 of [6], the deterministic twtt is closely related to the attribute grammar of [13]. An attribute grammar for which the values of the attributes are trees, i.e., unevaluated expressions, is called an attributed tree transducer (see [7,8]). The nondeterministic version of the attributed tree transducer (introduced in [7]) is, as to be expected, closely related to the nondeterministic twtt. It was shown by Bartha in [2] that the domain of a nondeterministic attributed tree transducer is a regular tree language. The algorithm to be presented in this section is essentially the one of [2], but instead of the top-down tree automaton of [2] we construct a bottom-up tree automaton which, in our opinion, is easier to understand. Also, of course, we have to translate the attribute grammar terminology into the one for twtts.

One of the main results on attribute grammars is the decidability of the noncircularity problem in exponential time [13]. In fact, the problem is complete in exponential time, and hence its time complexity is intrinsically exponential [10] (see also [9,4,3]). Roughly speaking, an attribute grammar G is noncircular if the corresponding twtt is always-halting. The algorithms for noncircularity in [13,10] essentially construct a regular tree grammar G' for the complement of the domain of G, i.e., for the set of all derivation trees (of the underlying context-free grammar of G) on which G is circular, and then test whether $L(G') = \emptyset$.

Thus, the following result is not surprising, and the proof we present is a simple variation of the algorithm of [13] (to be precise, of the correction to [13] that appeared in 1971). An alternative proof, without the time bound, was given in [11, Theorem 3.5] (taking into account that the domain of a deterministic twtt can be recognized by an alternating tree-walking automaton with universal states only).

In what follows, by 'exponential time' we will mean deterministic time $2^{p(n)}$ where $p(n)$ is a polynomial in n (and n is the size of the problem instance).

Lemma 1 *For every deterministic twtt M a total deterministic bottom-up finite-state tree automaton A can be constructed in exponential time such that A recognizes the domain of τ_M.*

Proof Let $M = (\Sigma, \Delta, Q, \{q_0\}, R)$. Without loss of generality we assume that M is total (add a rule $\langle q, \sigma, j\rangle \to \langle q, \mathrm{stay}\rangle$ when there is no rule with that left-hand side; since $j \leq mx_\Sigma \leq n$ where n is the size of M, this adds at most n^3 rules).

The states of A are all mappings

$$d : (Q \times [0, mx_\Sigma]) \to (\mathcal{P}(Q) \cup \{\perp\})$$

such that, for each $q \in Q$, $d(q, 0)$ is either $\emptyset$ or $\perp$. The idea is that for any tree $s \in T_\Sigma$, A arrives at the root of s in the unique state d such that the statement in the following paragraph holds.

For every $q \in Q$ and $j \in [0, mx_\Sigma]$, and for every tree $s' \in T_\Sigma$ and every node u of s' with child number j such that s is the subtree of s' with root u: if there is a derivation $\langle q, u\rangle \Rightarrow^* t$ in $G_{M,s',u}$ such that no production of that grammar is applicable to t, then $d(q, j)$ is the set of all states $q' \in Q$ such that the nonterminal $\langle q', u\!\uparrow\rangle$ occurs in t (which means that $d(q, j) = \emptyset$ in the case where $j = 0$), and if there is no such derivation, then $d(q, j) = \perp$.

More intuitively, since M is deterministic, it has a unique computation on the subtree s starting in state q at the root u (with child number j). This computation is either finite or infinite. If the computation is infinite then $d(q, j) = \perp$, whereas if it is finite then $d(q, j)$ is the set of all states that occur in the output tree t generated by the computation. Note that in the finite case, since M is total, all nonterminals occurring in t are of the form $\langle q', u\!\uparrow\rangle$. Note also that the computation does not depend on s', except for the fact that the child number of u is j.

In accordance with the above idea, the final states of A are all d such that $d(q_0, 0) = \emptyset$, and the transitions of (the total deterministic) A are all $d \to \sigma(d_1, \ldots, d_k)$, with $\mathrm{rank}_\Sigma(\sigma) = k$, where the left-hand side d is determined from the right-hand side as follows. Consider $j \in [0, mx_\Sigma]$. Now build a directed graph D_j, called *dependency graph*, with the following vertices and edges. The vertices are all $\langle q, \varphi\rangle$ with $q \in Q$ and $\varphi \in \{\mathrm{up}, \mathrm{stay}\} \cup \{\mathrm{down}_i \mid i \in [1, k]\}$. For every rule $\langle q, \sigma, j\rangle \to \zeta$ of M, if $\langle q', \varphi\rangle$ occurs in ζ then there is an edge from $\langle q, \mathrm{stay}\rangle$ to $\langle q', \varphi\rangle$. For every $i \in [1, k]$, if $d_i(q, i) \in \mathcal{P}(Q)$ and $q' \in d_i(q, i)$ then there is an edge from $\langle q, \mathrm{down}_i\rangle$ to $\langle q', \mathrm{stay}\rangle$; moreover, if $d_i(q, i) = \perp$ then there is an edge from $\langle q, \mathrm{down}_i\rangle$ to itself. From this dependency graph D_j, $d(q, j)$ is defined for every $q \in Q$ as follows: if there is an infinite directed path through D_j starting at $\langle q, \mathrm{stay}\rangle$ (i.e., if a cycle can be reached from $\langle q, \mathrm{stay}\rangle$), then $d(q, j) = \perp$; otherwise, $d(q, j)$ is the set of all q' such that there is a directed path in D_j from $\langle q, \mathrm{stay}\rangle$ to $\langle q', \mathrm{up}\rangle$.

Intuitively, a vertex $\langle q, \mathrm{stay}\rangle$ of D_j represents the twtt M in state q at some node u (with label σ and child number j) currently processed by the automaton A. Vertices $\langle q, \mathrm{up}\rangle$ and $\langle q, \mathrm{down}_i\rangle$ represent M in state q at the parent $u\!\uparrow$ of u, and at the ith child ui of u, respectively. The edges of D_j represent the computation steps of M from u to its parent, to itself, and to its children (as given by the rules of M), plus the computations from its children to itself (as given by $d_1, \ldots, d_k$, where, for the ith child, the second argument of d_i is restricted to child number i). From this it should be clear that A recognizes the domain of τ_M.

Let n be the size of M. Let $m = mx_\Sigma$ and let $k = \#(Q)$; note that $k, m \leq n$. The number of mappings d is at most $\delta = (2^k + 1)^{k(m+1)}$, which is at most $2^{c_1 n^3}$ for some constant c_1. The number of transitions of A is at most $\#(\Sigma)\delta^m$, which is at most $2^{c_2 n^4}$ for some constant c_2. The time to compute each transition is polynomial in n, because it involves reachability between vertices in a graph of polynomial size. Hence the total time to compute A is at most 2^{cn^4} for some constant c, i.e., exponential in the size of M. $\square$

We now turn to the nondeterministic case. As observed before, for nondeterministic attributed tree transducers it was shown in [2] that their domains are regular tree languages. Also, it was shown in [19, Corollary 5.3], and in [15, Theorem 4.7], that alternating tree-walking automata recognize regular tree languages (and recall that the domains of nondeterministic twtts are recognized by such automata).

From the proof of Lemma 1 one would expect that for a nondeterministic twtt M it would take *double* exponential time to compute a deterministic bottom-up tree automaton A that recognizes its domain: for each state q there are several computations on a given subtree, and thus each $d(q, j)$ would have to be a set of subsets of Q rather than just one subset of Q. The proofs mentioned above are given without time bounds. The construction in [19] indeed takes double exponential time (one exponential in the proof of [19, Theorem 5.2], and another exponential in the proof of [12, Theorems 5.4 and 6.2]). The construction in [15] even takes triple exponential time: The MSO formula constructed for the regular tree language is, roughly speaking, of the form $\forall S_1 \cdots \forall S_k\, ((\forall x\, \forall y\, \varphi) \Rightarrow \psi)$ which is equivalent to $\neg\, \exists S_1 \cdots \exists S_k\, \forall x\, \forall y\, (\varphi \Rightarrow \psi)$, where k is the number of states of the twtt and φ, ψ are quantifier-free. Since $\varphi \Rightarrow \psi$ refers to $S_1, \ldots, S_k$, the size of the corresponding bottom-up tree automaton is exponential. The $\forall x\, \forall y$ and the negation both involve the subset construction for bottom-up tree automata, which takes exponential time.

Even allowing A to be nondeterministic, there is the problem that, in a given computation, M can have several subcomputations that start in the same state q at the same node u, but use different rules with left-hand side $\langle q, \sigma, j\rangle$ (where σ is the label of u and j its child number). However, as shown in [2], this problem can be avoided, and hence exponential time suffices to compute a nondeterministic automaton A. The main idea is embodied in the next, elementary lemma.

Lemma 2 *For every context-free grammar $G = (N, T, R, \mathcal{S})$ with $L(G) \neq \emptyset$ there exists a forward deterministic context-free grammar $G' = (N, T, R', \{S\})$ with $L(G') \neq \emptyset$, $R' \subseteq R$, and $S \in \mathcal{S}$.*

Proof Consider a successful derivation tree t of G, with the root labelled S. It suffices to show the existence of a successful derivation tree t' of G with the property that for every nonterminal X of G, the *same* production is applied at each occurrence of X in t': that production can be taken as the unique production in R' with left-hand side X. Consider a fixed nonterminal X. We may assume that there are no occurrences of X at nodes u and v of t, such that v is a proper descendant of u (if so, transform t by replacing the subtree at node u by the one at node v, just as in the proof of the pumping lemma for context-free grammars). But then all occurrences of X in t are at independent nodes, and we can transform t by replacing the subtrees at those nodes by one of them (picked arbitrarily). The resulting derivation tree has the required property for nonterminal X. Now repeat this procedure for each remaining nonterminal; clearly, it preserves the required property for all previously processed nonterminals. □

Theorem 1 (cf. [2]) *For every tree-walking tree transducer M a finite-state tree automaton A can be constructed in exponential time such that A recognizes the domain of τ_M.*

Proof Let us say, informally, that a computation of M on an input tree s is *locally deterministic* if for each state q and each node u of s, it applies the *same* rule whenever it arrives in state q at node u. Formally, since such a computation is a derivation of the regular tree grammar $G_{M,s}$, we define it to be locally deterministic if for each nonterminal $\langle q, u\rangle$ the same production is applied to all its occurrences in the derivation. It now follows immediately from Lemma 2 that, to recognize the domain of τ_M, it suffices to consider locally deterministic computations of M: s is in the domain of τ_M iff $L(G_{M,s}) \neq \emptyset$, and hence, by Lemma 2, there exists a locally deterministic computation $\langle q_0, \mathrm{root}_s\rangle \Rightarrow^* t$ in $G_{M,s}$ for some $q_0 \in Q_0$ and some $t \in T_\Delta$.

It should be clear that for locally deterministic computations the automaton A can use the same states as in the proof of Lemma 1, with the same interpretation (but, of course, restricting attention to subcomputations of a given locally deterministic computation). The final states are also the same. The only difference is that A now has to guess, at each node u, the unique rule that is used by M (in the locally deterministic computation) at node u for each state q. Thus, the left-hand side of a transition $d \to \sigma(d_1, \ldots, d_k)$ is not any more determined uniquely by the right-hand side. Instead, there is a left-hand side d for every possible subset R' of R that contains exactly one rule $\langle q, \sigma, j\rangle \to \zeta$ for every $q \in Q$ and $j \in [0, mx_\Sigma]$. Only the rules of R' are incorporated in the dependency graphs from which d is computed.

Obviously, the time to compute the transitions of A is now at most $\#(\Sigma)2^{\#(R)}\delta^m$. Since $\#(R) \leq n$, this is (still) at most 2^{cn^4} for some constant c. □

5 Inverse type inference and typechecking

As discussed in the Introduction, the *inverse type inference problem* is to construct, for a tree transducer M and a regular tree grammar G_{out}, a regular tree grammar G_{in} such that $L(G_{\mathrm{in}}) = \tau_M^{-1}(L(G_{\mathrm{out}}))$. Note that our definition differs from the one in [15], where it is required that $L(G_{\mathrm{in}}) = \{s \mid \tau_M(s) \subseteq L(G_{\mathrm{out}})\}$; the reason is that our definition is more convenient when considering compositions of tree transducers.

The proof of the next lemma is a well-known technique in tree transducer theory, also used in [15, Proposition 4.6].

Lemma 3 *The inverse type inference problem is solvable for twtts in exponential time.*

Proof Clearly, $\tau_M^{-1}(L(G_{\mathrm{out}}))$ equals the domain of $\tau_{M'}$ where M' is any twtt such that $\tau_{M'} = \{(s,t) \in \tau_M \mid t \in L(G_{\mathrm{out}})\}$. Thus, it suffices to construct such a twtt M' in polynomial time and use Theorem 1. This is an easy product construction. Let $M = (\Sigma, \Delta, Q, Q_0, R)$ and $G_{\mathrm{out}} = (N, \Delta, R_{\mathrm{out}}, \mathcal{S})$. The set of states of M' is $Q \times N$, and its set of initial states is $Q_0 \times \mathcal{S}$. Since the removal of chain rules from a context-free grammar takes polynomial time, we may assume that G_{out} has no rules $X_0 \to X_1$, i.e., that it is a (top-down) finite-state tree automaton. For every rule $\langle q, \sigma, j\rangle \to \langle q', \varphi\rangle$ in R and every nonterminal $X \in N$, the set of rules R' of M' contains the rule $\langle (q,X), \sigma, j\rangle \to \langle (q',X), \varphi\rangle$. Moreover, for every rule $\langle q, \sigma, j\rangle \to \delta(\langle q_1, \mathrm{stay}\rangle, \dots, \langle q_k, \mathrm{stay}\rangle)$ in R and every rule $X_0 \to \delta(X_1, \dots, X_k)$ in R_{out}, R' contains the rule $\langle (q, X_0), \sigma, j\rangle \to \delta(\langle (q_1, X_1), \mathrm{stay}\rangle, \dots, \langle (q_k, X_k), \mathrm{stay}\rangle)$.

It should be clear that this M' satisfies the requirement, and that it can be constructed from M and G_{out} in quadratic time. $\square$

Due to the close connection between twtts and attribute grammars, it is not difficult to show that the time complexity of the inverse type inference problem for twtts is in fact intrinsically exponential, in the sense of [10], even for deterministic twtts with a monadic output alphabet.

We now turn to the inverse type inference problem for compositions of twtts. A sequence $M = (M_1, \dots, M_k)$ of twtts is viewed as a tree transducer that computes the translation $\tau_M = \tau_{M_1} \circ \cdots \circ \tau_{M_k}$. Since the inverse of a composition is the composition of the inverses, in the inverse order, it suffices to iterate the inverse type inference for twtts.

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.

Theorem 2 *The inverse type inference problem is solvable for compositions of k tree-walking tree transducers in k-fold exponential time.*

We finally consider the *typechecking problem.* It asks, for a tree transducer M and two regular tree grammars G_{in} and G_{out}, whether or not $\tau_M(L(G_{\mathrm{in}})) \subseteq L(G_{\mathrm{out}})$. This is solved using inverse type inference, as described in [15] (and as discussed in the Introduction).

Theorem 3 *Compositions of k tree-walking tree transducers can be typechecked in $(k+1)$-fold exponential time.*

Proof Clearly, $\tau_M(L(G_{\mathrm{in}})) \subseteq L(G_{\mathrm{out}})$ if and only if $L(G_{\mathrm{in}}) \cap \tau_M^{-1}(L'_{\mathrm{out}}) = \emptyset$, where L'_{out} is the complement of $L(G_{\mathrm{out}})$. A regular tree grammar for $L'_{\mathrm{in}} = \tau_M^{-1}(L'_{\mathrm{out}})$ can be constructed in $(k+1)$-fold exponential time by first constructing a regular tree grammar for L'_{out} in exponential time, and then using Theorem 2. With the usual product construction a regular tree grammar can then be obtained for $L(G_{\mathrm{in}}) \cap L'_{\mathrm{in}}$ in polynomial time. Finally, emptiness can be tested in polynomial time. $\square$

Note that one exponential can be taken off this result if we assume that G_{out} is a total deterministic bottom-up finite-state tree automaton (because then complementation takes constant time).

For pebble tree transducers Theorem 3 means the following. It is shown in [6, Theorem 10] that every k-pebble tree transducer M can be simulated by a composition of $k+1$ twtts. It is easy to see from the proof that these twtts can be constructed from M in polynomial time (for fixed k). For the interested reader, and for completeness sake, we give the details of this result in the next section.

Theorem 4 *For fixed k, k-pebble tree transducers can be typechecked in $(k+2)$-fold exponential time. The inverse type inference problem is solvable for k-pebble tree transducers in $(k+1)$-fold exponential time.*

As a corollary we can state the complexity of the emptiness problem for alternating k-pebble tree automata. In fact, by adding trivial output to an alternating k-pebble tree automaton, a nondeterministic k-pebble tree transducer is obtained. Emptiness of the domain of this transducer can be checked by inverse type inference.

Corollary 1 *For fixed k, the emptiness problem for alternating k-pebble tree automata can be solved in $(k+1)$-fold exponential time.*

It is shown by Samuelides and Segoufin in [18] that, for $k \geq 1$, the emptiness problem for (nondeterministic and deterministic) k-pebble tree automata is complete in k-fold exponential time. By a straightforward variation of their proof of the lower bound, it can be shown that the emptiness problem for *alternating* k-pebble tree automata is in fact $(k+1)$-fold exponential time hard. This implies that the inverse type inference problem for k-pebble tree transducers is not solvable in k-fold exponential time. Thus, for that problem, Theorem 4 is optimal (and hence, through the polynomial simulation mentioned above, also Theorem 2 is optimal). Based on this, it is likely that the typechecking result in Theorem 4 is also optimal.

6 Disposing of pebbles

The aim of this section is to prove the following result from [6, Lemma 9], where it was stated without the time bound.

Theorem 5 *Let $k \geq 1$. For every k-pebble tree transducer M a twtt N and a $(k-1)$-pebble tree transducer M' can be constructed in polynomial time such that $\tau_M = \tau_N \circ \tau_{M'}$.*

Applying this theorem k times shows that, for fixed k, there is a polynomial time algorithm that constructs for every k-pebble tree transducer M a composition $M' = (M_1, \ldots, M_{k+1})$ of twtts such that $\tau_M = \tau_{M'}$, see [6, Theorem 10].

Let $M = (\Sigma, \Delta, Q, Q_0, R)$ be a k-pebble tree transducer. The construction of N and M' will be slightly different from the one in the proof of [6, Lemma 9], but the basic idea is the same. The simple idea of the proof is to preprocess the input tree $s \in T_\Sigma$ in such a way that the dropping and lifting of the first pebble can be simulated by walking into and out of specific areas of the preprocessed input tree $\mathrm{pp}(s)$. This preprocessing is independent of the given pebble tree transducer M. More precisely, $\mathrm{pp}(s)$ is obtained from s by attaching to each node u of s, as an additional (last) subtree, a fresh copy of s in which (the copy of) node u is marked; let us denote this subtree by s_u. Thus, if s has n nodes, then $\mathrm{pp}(s)$ has $n + n^2$ nodes. The subtrees s_u of $\mathrm{pp}(s)$ are the "specific areas" mentioned above. As long as there are no pebbles on s, M' simulates M on the original nodes of s in $\mathrm{pp}(s)$. When M drops its first pebble at node u, M' enters s_u and walks to the marked node. As long as the first pebble is on the tree, M' stays in s_u, simulating M; since u is marked in s_u, M' needs only $k-1$ pebbles. When M lifts the first pebble from u (and hence all pebbles are lifted), M' walks from the copy of u out of s_u, back to the original node u.

Unfortunately, this preprocessing cannot be realized by a twtt (though it can easily be realized by a 1-pebble tree transducer). For this reason we "fold" s_u at the node u, such that (the marked copy of) u becomes the root of s_u; let us denote the result by $\hat{s}_u$. Roughly, $\hat{s}_u$ is obtained from s_u by inverting the parent-child relationship between the ancestors of u (including u). Adding appropriate information to the node labels of those ancestors, allows M' to reconstruct the unfolded s_u, and to simulate M as before. Note that, with this change of $\mathrm{pp}(s)$, dropping or lifting of the first pebble can be simulated by M' in one computation step, because the marked copy of u is the last child of the original u. Now a twtt can compute $\mathrm{pp}(s)$, as follows (cf. also [15, Example 3.7]). It copies s to the output (adding primes to its labels), but when it arrives at node u it additionally outputs $\hat{s}_u$ in a side branch of the computation. That is done by walking from u to root_s and, for each node v on that path (including u and root_s), copying v to the output together with those subtrees of v that do not have nodes in common with the path; moreover, in the output, v has an additional (last) child that corresponds to its parent in s. The subtree that *has* nodes in common with the path (if $v \neq u$), is represented by the bottom symbol $\perp$ of rank 0 (and so is the "parent" of $v = \mathrm{root}_s$). Note that the nodes of s correspond one-to-one to the non-bottom nodes of $\hat{s}_u$; in particular, the path in s from u to root_s corresponds to the path in $\hat{s}_u$ from its root to the parent of its rightmost leaf. The bottom nodes of $\hat{s}_u$ will not be visited by M'.

As an example, consider $s = \sigma(\delta(a,b),c)$ where σ, δ have rank 2 and a, b, c rank 0. We will name the nodes of s by their labels. Then $\mathrm{pp}(s) = \sigma'(\delta'(a'(\hat{s}_a), b'(\hat{s}_b), \hat{s}_\delta), c'(\hat{s}_c), \hat{s}_\sigma)$ where

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

The subscripted node labels are on the rightmost paths of the $\hat{s}_u$'s; the subscripts contain "reconstruction" information, to be explained below. As another example, if s is the monadic tree $a(b^m(c(d^n(e))))$ of height $m+n+3$, and u is the c-labelled node, then $\hat{s}_u = c_{0,1}(t_1, t_2)$ with $t_1 = d^n(e)$ and t_2 is the binary tree $b_{1,1}(\perp, b_{1,1}(\perp, \ldots b_{1,1}(\perp, a_{1,0}(\perp, \perp))\cdots))$ of height $m+2$. This shows more clearly that $\hat{s}_u$ is obtained by "folding".

We now formally define the deterministic twtt N that, for given ranked alphabet Σ, realizes the preprocessing pp (called EncPeb in [6]). To simplify the definition we allow rules of the form $\langle q, \sigma, j\rangle \to \delta(\langle q_1, \varphi_1\rangle, \ldots, \langle q_m, \varphi_m\rangle)$ where φ_j is stay, up, or down$_i$ (and m is the rank of δ). Such a rule should be replaced by the rules $\langle q, \sigma, j\rangle \to \delta(\langle p_1, \text{stay}\rangle, \ldots, \langle p_m, \text{stay}\rangle)$ and $\langle p_j, \sigma, j\rangle \to \langle q_j, \varphi_j\rangle$ for all $j \in [1, m]$, where $p_1, \ldots, p_m$ are new states. Obviously this replacement can be done in quadratic time.

We introduce the rules of 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, N has a state t that always outputs $\perp$: its rules are $\langle t, \sigma, j\rangle \to \perp$. Second, N has an "identity" state d that just copies the subtree of the current node to the output, using all rules $\langle d, \sigma, j\rangle \to \sigma(\langle d, \text{down}_1\rangle, \ldots, \langle d, \text{down}_m\rangle)$. Then, N has initial state c that copies the input tree s to the output and at each node u of s "calls" the state f that computes $\hat{s}_u$ by "folding" s_u. The rules for c are

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

Note that σ' has rank $m+1$: the root of $\hat{s}_u$ is attached to u as its last child. The rules for f are

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

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

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

where ξ_j is as above. In $\sigma_{i,j}$, "reconstruction" subscript i means that the ith child of this node is $\perp$, and, as above, subscript j is the child number of the node v in s to which this node (which is in some $\hat{s}_u$) corresponds. Just as σ' and $\sigma_{0,j}$, also $\sigma_{i,j}$ has rank $m+1$: its last child corresponds to the parent of v in s.

This ends the description of the twtt N. The output alphabet Γ of N (which will also be the input alphabet of M') is the union of Σ, $\{\perp\}$, $\{\sigma' \mid \sigma \in \Sigma\}$, and $\{\sigma_{i,j} \mid \sigma \in \Sigma, i \in [0, \text{rank}_\Sigma(\sigma)], j \in [0, mx_\Sigma]\}$. Thus, N has $O(n^2)$ output symbols, where n is the size of Σ (recall that, by the assumption at the end of Section 2, $mx_\Sigma \leq n$). So, since $mx_\Gamma = mx_\Sigma + 1$, the size of Γ is polynomial in n. The set of states of N is $\{t, d, c, f\} \cup \{f_i \mid i \in [1, mx_\Sigma]\}$, with initial state c. 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 N is polynomial in the size of Σ, and it can be constructed in polynomial time.

We now turn to the description of M'. It has input alphabet Γ, output alphabet Δ, and the same states and initial states as M. Thus, it remains to discuss its rules. Let $\langle q, \sigma, j, b\rangle \to \zeta$ be a rule of M with $b : [1, l] \to \{0, 1\}$ for some $l \in [0, k]$; note that l is the number of pebbles that is placed on the input tree. Let $m = \mathrm{rank}_\Sigma(\sigma)$. We consider three cases.

First we consider the case where $l = 0$, and so $b = \emptyset$. Then M' has the rule $\langle q, \sigma', j, b\rangle \to \zeta'$, where ζ' is obtained from ζ by changing drop into down_{m+1} (only if drop occurs in ζ, of course; otherwise $\zeta' = \zeta$). Thus, instead of dropping the first pebble, M' enters $\hat{s}_u$.

Now let $l \geq 1$. We define b^- to be the mapping $b^- : [1, l-1] \to \{0, 1\}$ such that $b^-(m-1) = b(m)$ for every $m \in [2, l]$.

Second we consider the case where $b(1) = 1$, i.e., the first pebble is on the current node. Then M' has all rules $\langle q, \sigma_{0,j}, m+1, b^-\rangle \to \zeta'$ where ζ' is obtained from ζ by changing up into down_{m+1}, and, provided $l = 1$ (and so $b^- = \emptyset$), changing lift into up. Thus, in the latter case, instead of lifting the first pebble, M' leaves $\hat{s}_u$. Note that the child number in $\mathrm{pp}(s)$ of a node with label $\sigma_{0,j}$ is always $m+1$ (and the label of its parent is σ').

Third, the case $b(1) = 0$. Then M' has the rule $\langle q, \sigma, j, b^-\rangle \to \zeta$. Moreover, for every $i \in [1, m]$ (so $i \neq 0$) and every $j' \in [1, mx_\Gamma]$, it has the rule $\langle q, \sigma_{i,j}, j', b^-\rangle \to \zeta'$ where ζ' is obtained from ζ by changing up into down_{m+1}, and down_i into up.

This ends the description of the $(k-1)$-pebble tree transducer M'. It should now be clear that $\tau_{M'}(\mathrm{pp}(s)) = \tau_M(s)$ for every $s \in T_\Sigma$, and hence $\tau_N \circ \tau_{M'} = \tau_M$. Each rule of M is turned into at most $1 + mx_\Sigma(mx_\Sigma + 1)$ rules of M, of the same size as that rule (disregarding the logarithmic space taken by the subscript of down_{m+1}). Thus, M' can be computed from M in polynomial time, which proves Theorem 5.

7 From ranked trees to unranked forests

For an (unranked) alphabet Δ, an (unranked) *forest* over Δ is a string generated by the context-free grammar with productions $F \to \varepsilon$ (where ε is the empty string, also called empty forest) and $F \to \delta(F)F$ for every $\delta \in \Delta$. It is easy to see that the set F_Δ of forests over Δ is closed under concatenation, i.e., if f_1, f_2 are forests, then so is $f_1 f_2$. Note that strings over Δ can be viewed as forests over Δ in a natural way, by changing every δ into $\delta()$.

Any tree transducer formalism (for ranked trees) can also be used as a forest transformation device, by coding forests as binary trees, in the usual way. The encoding of the empty forest is $\mathrm{enc}(\varepsilon) = e$, where e is a special symbol of rank 0, and, recursively, the encoding $\mathrm{enc}(f)$ of a forest $f = \delta(f_1)f_2$ is $\delta(\mathrm{enc}(f_1), \mathrm{enc}(f_2))$, where each symbol $\delta \in \Delta$ is given rank 2. We will denote the corresponding ranked alphabet $\Delta \cup \{e\}$ by Δ_2^{f}. Thus, 'enc' is a bijection between F_Δ and $T_{\Delta_2^{\mathrm{f}}}$.

For tree-walking tree transducers (and, more generally, pebble tree transducers) this is perfectly fine for the input tree: walking on a forest f is basically the same as walking on its encoding $\mathrm{enc}(f)$. For the output tree it is fine too, but, since the output tree is generated in a top-down fashion, it is natural to allow the right-hand sides of the rules to use forest concatenation as a basic operation, as pointed out in [17] for macro tree transducers. This is similar to the use of string concatenation by a context-free grammar, as opposed to a right-linear grammar.

For an (unranked) alphabet Δ, let Δ_1^{f} be the ranked alphabet $\Delta \cup \{e, @\}$ where e has rank 0, @ has rank 2, and each $\delta \in \Delta$ has rank 1. Intuitively, e stands for the empty forest (as before), @ stand for forest concatenation, and each δ represents the forest operation $f \mapsto \delta(f)$. This interpretation is formalized by the "flattening" function from $T_{\Delta_1^{\mathrm{f}}}$ to F_Δ defined by: $\mathrm{flat}(e) = \varepsilon$, $\mathrm{flat}(@(t_1, t_2)) = \mathrm{flat}(t_1)\mathrm{flat}(t_2)$, and $\mathrm{flat}(\delta(t)) = \delta(\mathrm{flat}(t))$. Note that, as opposed to 'enc', the function 'flat' is not injective.

A *tree-walking forest transducer* (abbreviated *twft*) is a twtt with output alphabet Δ_1^{f} for some Δ. It outputs forests, obtained by flattening the output trees. More formally, we define the tree-to-forest translation realized by a twft M to be $\tau_M^{\mathrm{f}} = \{(s, \mathrm{flat}(t)) \mid (s, t) \in \tau_M\} = \tau_M \circ \mathrm{flat}$.

Note that the right-hand side of a rule of a twft is of one of the forms $\langle q, \varphi \rangle$, $\delta(\langle q', \mathrm{stay} \rangle)$, $@(\langle q_1, \mathrm{stay} \rangle, \langle q_2, \mathrm{stay} \rangle)$, or e. When the last two are written as $\langle q_1, \mathrm{stay} \rangle \langle q_2, \mathrm{stay} \rangle$ and ε, respectively, the transducer indeed generates forests, in the obvious way.

As suggested above, a twtt M with output alphabet Δ_2^{f} can also be viewed as a tree-to-forest translation device (and will, in this case, still be called twtt): it realizes the translation $\tau_M \circ \mathrm{enc}^{-1}$. Similar to the result in [17], the twft has more expressive power than the twtt (as tree-to-forest translation devices). This is easy to see: For a twtt, the height of the output tree is linearly bounded by the size of the input tree; this is obvious, and well known for attributed tree transducers, see [8, Lemma 5.40]. But it is straightforward to construct a twft that, by repeated copying on its way downwards, translates a monadic tree of size n into a string of length 2^n; since the encoding of such a string has height 2^n, this tree-to-forest translation cannot be realized by a twtt.

We now show that twfts can still be typechecked in 2-fold exponential time, just as twtts. It is well known that regular forest types can be defined naturally through their encoding: a set of forests is regular if and only if the set of its encodings is a regular tree language (see, e.g., [16]). Thus, in order to typecheck twfts, we will encode their output forests (if you can still follow this). For an unranked alphabet Δ, let 'app' be the "application" tree transformation from $T_{\Delta_1^{\mathrm{f}}}$ to $T_{\Delta_2^{\mathrm{f}}}$ defined by $\mathrm{app}(t) = \mathrm{enc}(\mathrm{flat}(t))$, see [17] where it is called 'eval', and [14] where it is called APP. So, typechecking the translation τ_M^{f} of a twft M amounts to typechecking the translation $\tau_M \circ \mathrm{app}$. It is not difficult to show that 'app' can be realized by a twtt, and hence, by Theorem 3, $\tau_M \circ \mathrm{app}$ can be typechecked in 3-fold exponential time. However, as in [17], we can do one exponential better, due to the following lemma (proved independently in [20, Theorem 4.5]). In the statement of this lemma, we assume app to be represented by a "tree transducer" that just consists of the alphabet Δ; thus, the lemma states that there is an algorithm that constructs in polynomial time, for an alphabet Δ and a regular tree grammar G_{out} with terminal alphabet Δ_2^{f}, a regular tree grammar G_{in} such that $L(G_{\text{in}}) = \text{app}^{-1}(L(G_{\text{out}}))$.

Lemma 4 *Inverse type inference is solvable for* app *in polynomial time.*

Proof Let $G_{\text{out}} = (N, \Delta_2^{\text{f}}, R, \mathcal{S})$ be a regular tree grammar, without chain productions. Thus, its productions are of the form $X \to \delta(Y,Z)$ or $X \to e$. We construct a regular tree grammar $G_{\text{in}} = ((N \times N) \cup \{S_0\}, \Delta_1^{\text{f}}, R', \{S_0\})$ with $L(G_{\text{in}}) = \text{app}^{-1}(L(G_{\text{out}}))$, as follows. The nonterminals of G_{in} are pairs of nonterminals of G_{out}, together with a new (initial) nonterminal S_0. The rules of G_{in} in R' are

$$\begin{array}{ll}
S_0 \to \langle S, E\rangle & \text{with } S \in \mathcal{S} \text{ and } E \to e \text{ in } R \\
\langle X, Z\rangle \to @(\langle X, Y\rangle, \langle Y, Z\rangle) & \text{with } X, Y, Z \in N \\
\langle X, Z\rangle \to \delta(\langle Y, E\rangle) & \text{with } X \to \delta(Y,Z), E \to e \text{ in } R \\
\langle X, X\rangle \to e & \text{with } X \in N.
\end{array}$$

Clearly, G_{in} can be constructed from G_{out} in polynomial time. For a tree t over Δ_2^{f} and a nonterminal Y of G_{out}, let t_Y denote the tree obtained from t by changing the label e of its rightmost leaf into Y. It can be shown that, for every tree s over Δ_1^{f}, $\langle X, Y\rangle$ generates s in G_{in} if and only if X generates $\text{app}(s)_Y$ in G_{out}, which shows that $L(G_{\text{in}}) = \text{app}^{-1}(L(G_{\text{out}}))$. The proof of the statement is by structural induction on s. The cases $s = e$ and $s = \delta(s')$ are straightforward. In the case where $s = @(s_1, s_2)$ one needs the fact that $\text{app}(@(s_1, s_2))$ is obtained from $\text{app}(s_1)$ by replacing its rightmost leaf with $\text{app}(s_2)$; and hence, $\text{app}(@(s_1, s_2))_Z$ is obtained from $\text{app}(s_1)_Y$ by replacing Y with $\text{app}(s_2)_Z$. □

From this lemma, Theorem 2, and the proof of Theorem 3, we immediately obtain the following result.

Corollary 2 *A composition of $k-1$ tree-walking tree transducers and one tree-walking forest transducer can be typechecked in $(k+1)$-fold exponential time.*

Similarly, from the above lemma, the second statement of Theorem 4, and the proof of Theorem 3, we obtain that Theorem 4 also holds for k-pebble forest transducers. As for the twft, a *k-pebble forest transducer* is defined to be a k-pebble tree transducer M with output alphabet Δ_1^{f} for some Δ; typechecking its tree-to-forest translation $\tau_M^{\text{f}} = \tau_M \circ \text{flat}$ amounts to typechecking $\tau_M \circ \text{app}$.

Corollary 3 *For fixed k, k-pebble forest transducers can be typechecked in $(k+2)$-fold exponential time.*

It should now be clear that, due to the generality of Lemma 4, a similar result can be shown for any tree transducer formalism for which the complexity of typechecking is known through the inverse type inference problem: the complexity of typechecking the corresponding forest transducers differs by a polynomial from the one for the tree transducers. For macro tree transducers this is the result of [17].

Acknowledgements I thank the referees for their constructive comments. I am grateful to Hendrik Jan Hoogeboom for his technical and moral support.

References

1. A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman; *Compilers: Principles, Techniques, and Tools*, 2nd edition, Addison-Wesley, 2006
2. M. Bartha; An algebraic definition of attributed transformations, Acta Cybernetica 5 (1982), 409–421. Preliminary version in: Proc. FCT'81 (F. Gécseg, ed.), Lecture Notes in Computer Science 117, Springer-Verlag, 1981, pp.51–60
3. J. Engelfriet; The complexity of the circularity problem for attribute grammars: a note on a counterexample for a simpler construction, SIGACT News, Summer 1989, 57–59
4. J. Engelfriet, G. Filè; Passes and paths of attribute grammars, Information and Control 49 (1981), 125–169
5. J. Engelfriet, H.J. Hoogeboom, B. Samwel; XML transformation by tree-walking transducers with invisible pebbles, Proc. PODS'07 (L. Libkin, ed.), ACM Press, 2007, pp.63–72
6. J. Engelfriet, S. Maneth; A comparison of pebble tree transducers with macro tree transducers, Acta Informatica 39 (2003), 613–698
7. Z. Fülöp; On attributed tree transducers, Acta Cybernetica 5 (1981), 261–279
8. Z. Fülöp, H. Vogler; *Syntax-Directed Semantics, Formal Models Based on Tree Transducers*, Monographs in Theoretical Computer Science, An EATCS Series, Springer-Verlag, 1998
9. M. Jazayeri; A simpler construction for showing the intrinsically exponential complexity of the circularity problem for attribute grammars, Journal of the ACM 28 (1981), 715–720
10. M. Jazayeri, W. F. Ogden, W. C. Rounds; The intrinsically exponential complexity of the circularity problem for attribute grammars, Communications of the ACM 18 (1975), 697–706
11. T. Kamimura; Tree automata and attribute grammars, Information and Control 57 (1983), 1–20
12. T. Kamimura, G. Slutzki; Parallel and two-way automata on directed ordered acyclic graphs, Information and Control 49 (1981), 10–51
13. D. E. Knuth; Semantics of context-free languages, Mathematical Systems Theory 2 (1968), 127–145. Correction: Mathematical Systems Theory 5 (1971), 95–96
14. S. Maneth, A. Berlea, T. Perst, H. Seidl; XML type checking with macro tree transducers, Proc. PODS'05, ACM Press, 2005, pp.283–294
15. T. Milo, D. Suciu, V. Vianu; Typechecking for XML transformers, Journal of Computer and System Sciences 66 (2003), 66–97
16. F. Neven; Automata, Logic, and XML, Proc. CSL'02 (J.C. Bradfield, ed.), Lecture Notes in Computer Science 2471, Springer-Verlag, 2002, pp.2–26
17. T. Perst, H. Seidl; Macro forest transducers, Information Processing Letters 89 (2004), 141–149
18. M. Samuelides, L. Segoufin; Complexity of pebble tree-walking automata, Proc. FCT'07 (E. Csuhaj-Varjú, Z. Ésik, eds.), Lecture Notes in Computer Science 4639, Springer-Verlag, 2007, pp.458–469
19. G. Slutzki; Alternating tree automata, Theoretical Computer Science 41 (1985), 305–318
20. T. Yashiro; Typechecking k-pebble tree transducers: practical efficiency, Bachelor Thesis, University of Tokyo, February 2006