XML Transformation by Tree-Walking Transducers with Invisible Pebbles

Joost Engelfriet Hendrik Jan Hoogeboom Bart Samwel

LIACS, Leiden University, The Netherlands

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 n-ary 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 binary patterns can often be reduced through the use of invisible pebbles.

Categories and Subject Descriptors: F.4.3 [Mathematical logic and formal languages]: Formal Languages; H.2.3 [Database Management]: Languages.

General Terms: Theory, Languages.

Keywords: XML, Tree Transducer, Pebble.

1. INTRODUCTION

Pebble tree transducers, as introduced by Milo, Suciu, and Vianu [22], are a formal model of XML transformation for which typechecking is decidable. 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 [22] (or rather, the one in [11]: 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, and TT to the one that does not use any pebbles.

The navigational part of a V-PTT, i.e., the behaviour of the transducer when no output is produced, is the pebble tree automaton (V-PTA), introduced in [9], where it was shown to recognize regular tree languages only. Recently, in [5], the important result was proved that not all regular tree languages can be recognized by the V-PTA, and thus [8, 28] 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., [24]). One of the reasons for introducing invisible pebbles is that the VI-PTA, and the I-PTA, recognize exactly the regular tree languages (Theorem 10). Thus, VI-PTA can validate arbitrary generalized DTD's, and can match arbitrary MSO definable n-ary patterns (using visible pebbles to find all candidate matches as in [22], and using invisible pebbles to check the MSO requirements). We note that the I-PTA is a straightforward generalization of the 'two-way backtracking pushdown tree automaton' of Slutzki [27].

It is easy to show that every regular tree language can be recognized by an I-PTA, just simulating a bottom-up finite 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 4), similar to the one for V-PTT in [11]. 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. Hence, the typechecking problem is decidable for VI-PTT, by the same arguments as used in [22] for V-PTT. More precisely, we prove (Theorem 8, based on Theorem 5) that a VI-PTT with k visible pebbles can be typechecked in $(k+3)$-fold exponential time. It implies a $(k+2)$-fold exponential time upper bound for V-PTT with k pebbles, because the k-th pebble, which is always on top of the stack, may as well be taken invisible. This seems to improve the bound in [22] by two exponentials. Of course, for varying k the complexity is still non-elementary, but, as observed in [23], "non-elementary algorithms on tree automata have previously been seen to be feasible in practice".

As the navigational part of VI-PTT, the VI-PTA in fact defines 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 define exactly the MSO definable binary patterns (Theorem 13). This improves the result in [10] (where binary patterns are called 'trips'), because the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODS'07, June 11–14, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-685-1/07/0006 ...$5.00.

I-PTA is a more natural automaton than the one considered in [10].

One of the research goals of Marx and his colleagues (see [20, 14, 7] and the entertaining [21]) is to combine Core XPath of [15] which models the navigational part of XPath 1.0, with regular path expressions [1] (or caterpillar expressions [6]) 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. [12, Definition 6.5]). We prove (Theorem 14) 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). Due to this feature, we can use Kleene's classical construction to 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 16). In fact, Pebble XPath is the extension of Regular XPath [20, 7] with a stack of invisible pebbles; visible pebbles can easily be added too. It is conjectured in [7] that Regular XPath is not MSO complete.

We prove that 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 17). If all the observable pebbles are visible this is obvious (drop a pebble, simulate an I-PTA that recognizes the corresponding regular tree language, and return to the pebble), but if the top pebble is invisible that does not work and a look-ahead test must be used.

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

$$\texttt{for } x_1,\ldots,x_n \texttt{ where } \varphi_1 \wedge \cdots \wedge \varphi_m \texttt{ return } t,$$

where $x_1,\ldots,x_n$ are variables, each φ_k is an MSO formula with two free variables x_i and x_j, and t is an output tree with variables at the leaves. Using an MSO test, 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 (Proposition 18). 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_k, then *no* visible pebbles are needed, i.e., the query can be evaluated by an I-PTT: we use invisible pebbles $p_1,\ldots,p_n$ 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 $\varphi_k(x_i, x_{i+1})$ can be verified by an MSO test on the observable 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 [22], for the output it *does* make a difference, as pointed out in [25] 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 [25], 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 22). In fact, PFT have all the properties mentioned before for PTT.

The document transformation languages DTL and TL were introduced in [19] and [18], 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 modelled as forests. The language DTL has no variables or parameters, and its only instruction is `apply-templates`. 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 TL and I-PFT have the same expressive power (Theorem 21). Thus, in its forest version, our new model the VI-PFT can be viewed as the natural combination of the pebble tree transducer of [22] (V-PTT) and the TL program of [18] (I-PFT). Note that V-PTT and TL have incomparable expressive power. As claimed by [18], TL certainly can "describe many real-world XML transformations" (such as, all VI-PFT transformations of linear size increase). However, the visible pebbles are still a requisite for the XQuery-like queries discussed above, and we conjecture that not all such queries can be programmed in TL. As shown in [3] (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 with *non*nested pebbles can be simulated, and typechecking is no longer decidable. In XSLT 2.0 *all* (computable) queries can be programmed [17].

2. AUTOMATA & TRANSDUCERS

Let Σ be an alphabet. Unranked trees and forests over Σ are recursively defined: if $\sigma \in \Sigma$ and $t_1,\ldots,t_k$ are unranked trees, then $(t_1,\ldots,t_k)$, or $t_1 \cdots t_k$, is a forest, and $\sigma(t_1,\ldots,t_k)$ is an unranked tree. As usual trees are viewed as directed labelled graphs. The root of the tree is labelled by σ and has child number 0; it has edges to the roots of subtrees $t_1,\ldots,t_k$, which have child number 1 to k.

A *ranked* alphabet Σ has an associated mapping $\mathrm{rank}_\Sigma : \Sigma \to \mathbb{N}$. The maximal rank of symbols in Σ is denoted m_Σ. By $\Sigma^{(k)}$ we denote the symbols of rank k in Σ. Ranked trees over Σ are recursively defined as above with the requirement that $k = \mathrm{rank}_\Sigma(\sigma)$.

AUTOMATA. A *tree-walking automaton with nested pebbles* (pebble tree automaton for short, abbreviated PTA) is a finite state device that walks from node to node over its input tree following the 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 a node if it was the last one dropped dur-

ing 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*. Each of these has a distinct colour, and at most one visible pebble of each colour can be present on the tree. Second there is a set of *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 tree. Visible pebbles can be observed by the automaton at any moment when it visits a node where they were dropped. As for invisible pebbles, it can only observe such a pebble when it was the last one dropped.

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.

The PTA is specified as a tuple $\mathcal{A} = (\Sigma, Q, Q_0, C, C_{\mathrm{v}}, C_{\mathrm{i}}, R)$, 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, C_{v} and C_{i} are the finite sets of visible and invisible colours, $C = C_{\mathrm{v}} \cup C_{\mathrm{i}}$, $C_{\mathrm{v}} \cap C_{\mathrm{i}} = \varnothing$, and R is a finite set of rules. Each rule is of the form $\langle q, \sigma, b, j\rangle \to \zeta$ with $q \in Q$, $\sigma \in \Sigma$, $b \subseteq C$, $0 \le j \le m_\Sigma$, and ζ is one of the forms: $\langle \mathrm{halt}\rangle$, $\langle q', \mathrm{stay}\rangle$, $\langle q', \mathrm{up}\rangle$ provided $j \neq 0$, $\langle q', \mathrm{down}_i\rangle$ with $1 \le i \le \mathrm{rank}_\Sigma(\sigma)$, $\langle q', \mathrm{drop}_c\rangle$, and $\langle q', \mathrm{lift}\rangle$, with $q' \in Q$ and $c \in C$.

The *configuration* $\langle u, q, \pi\rangle$ of a PTA $\mathcal{A}$ on ranked tree t over Σ is given by the position u of the head on t, the state q of $\mathcal{A}$, and the stack π containing the positions and colours of the pebbles on the tree in the order they were dropped. An *initial* configuration equals $\langle \mathrm{root}, q_0, \lambda\rangle$ where 'root' is the root of t, $q_0 \in Q_0$, and λ is the empty stack. A rule $\langle q, \sigma, b, j\rangle \to \zeta$ can only be applied in a configuration $\langle u, q, \pi\rangle$ with state q, in a node u with label σ and child number j, and with the set b of the colours of the observable pebbles dropped on the node. More precisely, b consists of all $c \in C_{\mathrm{v}}$ such that (u, c) occurs in π, plus $c \in C_{\mathrm{i}}$ if (u, c) is the topmost element of π. When $\zeta = \langle \mathrm{halt}\rangle$ no new configuration is defined and the automaton halts. In the other cases we obtain a new configuration $\langle u', q', \pi'\rangle$. For the *move instructions* $\zeta = \langle q', \mathrm{stay}\rangle$, $\zeta = \langle q', \mathrm{up}\rangle$, and $\zeta = \langle q', \mathrm{down}_i\rangle$ the pebble stack does not change. The new state equals q' 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* $\zeta = \langle q', \mathrm{drop}_c\rangle$, and $\zeta = \langle q', \mathrm{lift}\rangle$ again the new state is q', but the node does not change. In the first case $\mathcal{A}$ drops a pebble on the current node, thus the node-colour pair (u, c) is pushed onto the pebble stack π, unless c is a visible colour and the stack already contains a pebble of that colour (so the rule cannot be applied). In the second case $\mathcal{A}$ picks a pebble from the current node, only allowed if the topmost element of the pebble stack is the pair (u, c) for some colour c, which is subsequently popped from the stack; otherwise this rule cannot be applied. We will also allow instructions like $\langle q', \mathrm{lift}; \mathrm{up}\rangle$ with the obvious meaning (first lift the pebble, then move up).

The *tree language accepted by* PTA $\mathcal{A}$ consists of all ranked trees t over Σ such that $\mathcal{A}$ has a halting computation on t which starts in an initial configuration. Note that pebbles may remain in the final configuration. Unlike the PTA from [22], our automata do *not* branch (i.e., are not alternating).

We use V_kI-PTA to denote PTA with k visible pebbles, i.e., $\#C_{\mathrm{v}} = k$, and an unbounded number of invisible pebbles, and $\mathsf{V}_k\mathsf{I}$-PTA to denote the tree languages they accept. For $k = 0$, automata that only use invisible pebbles, we also use the notation I-PTA, and for automata that only use k visible pebbles we use V_k-PTA. Moreover, TA is used for tree-walking automata without pebbles, i.e., V_0-PTA. The lower case d is added for languages when we only consider deterministic automata, which have a unique initial state and at most one applicable instruction in each configuration. Thus we have $\mathsf{V}_k\mathsf{I}$-dPTA and variants.

TRANSDUCERS. A *tree-walking tree transducer with nested pebbles* (abbreviated PTT) is a PTA that (recursively) produces an output tree over a ranked alphabet Δ. Instead of the halting instruction $\zeta = \langle \mathrm{halt}\rangle$, it has *output instructions* of the form $\zeta = \delta(\langle q_1, \mathrm{stay}\rangle, \ldots, \langle q_k, \mathrm{stay}\rangle)$ with $\delta \in \Delta$, and $q_1, \ldots, q_k \in Q$, where k is the rank of δ. Intuitively, in an applicable configuration (see above) a PTT $\mathcal{M}$ outputs δ, and for each child $\langle q_i, \mathrm{stay}\rangle$ spawns 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 k times. As a shortcut we may replace a $\langle q_i, \mathrm{stay}\rangle$ by a move instruction or a pebble instruction, with obvious semantics.

The set of trees computed by $\mathcal{M}$ in configuration $\gamma = \langle u, q, \pi\rangle$ is recursively defined. For each move instruction or pebble instruction that changes the configuration into $\langle u', q', \pi'\rangle$ the trees computed in the latter configuration are included in those computed in γ. Additionally, for each applicable output instruction $\zeta = \delta(\langle q_1, \mathrm{stay}\rangle, \ldots, \langle q_k, \mathrm{stay}\rangle)$, we add the trees with root δ, having as k subtrees those computed in the configurations $\langle u, q_i, \pi\rangle$. We use $\tau_{\mathcal{M}}$ to denote the transduction defined by transducer $\mathcal{M}$.

Similar to the notation $\mathsf{V}_k\mathsf{I}$-PTA for tree languages, we use the notation $\mathsf{V}_k\mathsf{I}$-PTT for the 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 family of transductions realized by tree-walking tree transducers without pebbles, i.e., V_0-PTT. Again, lower case d is added for deterministic automata.

EXAMPLE 1. 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. A deterministic I-PTT $\mathcal{M}$ 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, $\mathcal{M}$ 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 tags are ranked, there is no need for a binary encoding of the XML documents. The input alphabet Σ 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 consists of Σ, the binary tag $r =$ `<result>`, and the tag $e =$ `<endofresults>` of rank 0. The set of pebble colours is $C_{\mathrm{i}} = \{0, 1\}$. The attribute `initial` will not be used by $\mathcal{M}$, as it can recognize the root by its child number 0. 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 $\mathcal{M}$ first walks from Moscow to Vladivostok:
$\langle q_{\mathrm{start}}, \sigma_1, \varnothing, j\rangle \to \langle q_{\mathrm{start}}, \mathrm{down}_1\rangle$
$\langle q_{\mathrm{start}}, \sigma_0, \varnothing, 1\rangle \to \langle q_1, \mathrm{up}\rangle$
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 $i = 1$):
$\langle q_0, \lambda_0, \varnothing, 1\rangle \to \langle q_0, \mathrm{up}\rangle$
$\langle q_c, \lambda_i, \varnothing, 1\rangle \to \langle q_i, \mathrm{drop}_c; \mathrm{up}\rangle$
$\langle q_c, \lambda_1, \varnothing, 0\rangle \to r(\langle q_{\mathrm{out}}, \mathrm{stay}\rangle, \langle q_{\mathrm{next}}, \mathrm{down}_1\rangle)$
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:
$\langle q_{\mathrm{out}}, \sigma_1, \varnothing, 0\rangle \to \sigma_1(\langle q_{\mathrm{out}}, \mathrm{down}_1\rangle)$
$\langle q_{\mathrm{out}}, \sigma_1, \varnothing, 1\rangle \to \langle q_{\mathrm{out}}, \mathrm{down}_1\rangle$
$\langle q_{\mathrm{out}}, \sigma_1, \{c\}, 1\rangle \to \sigma_1(\langle q_{\mathrm{out}}, \mathrm{lift}; \mathrm{down}_1\rangle)$
$\langle q_{\mathrm{out}}, \sigma_0, \varnothing, 1\rangle \to \sigma_0$
$\langle q_{\mathrm{next}}, \sigma_1, \varnothing, 1\rangle \to \langle q_{\mathrm{next}}, \mathrm{down}_1\rangle$
$\langle q_{\mathrm{next}}, \sigma_1, \{c\}, 1\rangle \to \langle q_c, \mathrm{lift}; \mathrm{up}\rangle$
$\langle q_{\mathrm{next}}, \sigma_0, \varnothing, 1\rangle \to e$
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. □

3. DECOMPOSITION

In this section we decompose PTT into a sequence of TT, i.e., transducers without pebbles. This is useful as it will give us information on the domains of PTT, Theorem 10, and on the complexity of typechecking problems, see Theorem 8.

It is possible to reduce the number of visible pebbles used, by preprocessing the input tree. This was shown in [11, Lemma 9] for transducers with only visible pebbles. The basic idea of that proof can be extended to include invisible pebbles.

Let t with root r be a tree over the ranked alphabet Σ, and let x be a node of t. We construct a new tree $t^{\uparrow x}$ which equals t except that we make x the root of the tree, inverting all the edges on the path from the (old) root r to x. Thus, when $x \neq r$, node x obtains a new child, the node that used to be its parent, and becomes the root. The old root r looses a child that now will be its parent. For every other node an edge is redirected from one of the children to the parent of the node, similarly as in the tree traversal algorithm sometimes known as 'link inversion'. All nodes on this path are marked (adding information to the label) to reflect this change. In particular both the old root r and the new one x can be distinguished by their labels. Note that in this construction each node keeps the same neighbours, although the direction of some connections change. As these changes are local, and clearly marked in the tree, it is easy to see that for any tree-walking transducer $\mathcal{M}$ one can construct a new transducer $\mathcal{M}'$ that has the same behaviour as $\mathcal{M}$, when started on $t^{\uparrow x}$ rather than t, for all t over Σ.

Note that $t^{\uparrow x}$ occurs as "a complex rotation of the input tree" in [22, Example 3.7], albeit for leaves x only.

LEMMA 2. *For $k \geq 1$, $\mathsf{V}_k\mathsf{I\text{-}PTT} \subseteq \mathsf{dTT} \circ \mathsf{V}_{k-1}\mathsf{I\text{-}PTT}$, and $\mathsf{V}_k\mathsf{I\text{-}dPTT} \subseteq \mathsf{dTT} \circ \mathsf{V}_{k-1}\mathsf{I\text{-}dPTT}$.*

PROOF. The computation of a PTT $\mathcal{M}$ with k visible pebbles on tree t is simulated by a PTT $\mathcal{M}'$ with $k-1$ visible pebbles on tree t'. The new tree t' is obtained from t by adding, for each node x of t an edge to the root of a fresh copy $t^{\uparrow x}$ of input tree t. It is explained in [11] that this transformation can be computed without pebbles, by a deterministic TT.

As long as $\mathcal{M}$ does not use its visible pebbles, the simulating transducer $\mathcal{M}'$ copies $\mathcal{M}$'s moves in the original tree t, the 'top level' of t'. When the first visible pebble is dropped on a node x, this is simulated by moving down from x to copy $t^{\uparrow x}$ (and storing the colour of the pebble in the finite state). The computation is now continued as in t, but in copy $t^{\uparrow x}$, for all the pebbles now dropped and lifted, until the first visible pebble is picked up again (at node x). At that moment the simulation moves up to top level t and continues.

Moving down to $t^{\uparrow x}$ means that the pebble at x is visible to the transducer $\mathcal{M}'$, not as a pebble but as a marked node: the root of $t^{\uparrow x}$. Moving down also means that the topmost invisible pebble that was placed on t becomes unobservable: not because it is hidden by a new pebble (we do not use that pebble in the simulation) but by moving out of t. □

The tree 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 $t^{\uparrow x}$ (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 arbitrary many levels.

LEMMA 3. *$\mathsf{I\text{-}PTT} \subseteq \mathsf{TT} \circ \mathsf{TT}$, and $\mathsf{I\text{-}dPTT} \subseteq \mathsf{TT} \circ \mathsf{dTT}$.*

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'. In each node x of t the transducer $\mathcal{N}$ nondeterministically decides whether to spawn a process that, for each pebble colour c, copies t into $t_c^{\uparrow x}$, connecting x to the root of the new subtree. This is a recursive process: in each node in each copy of t it can be decided to start another new copy. In the copies $t_c^{\uparrow x}$ the

new root (the node corresponding to x) is clearly marked to represent colour c.

In this way a 'tree of trees' is constructed. The root of each copy $t_c^{\uparrow x}$ indicates an invisible pebble of colour c placed at node x in the original tree t. In each copy only one pebble is observable, the one placed at the new 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. Thus, $\tau_{\mathcal{M}} = \tau_{\mathcal{N}} \circ \tau_{\mathcal{M}'}$. □

The nondeterminism of the 'preprocessing' transducer $\mathcal{N}$ in the previous proof 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 [26].

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

THEOREM 4. *For $k \geq 0$, $\mathsf{V}_k\mathsf{I\text{-}PTT} \subseteq \mathsf{TT}^{k+2}$.*

Observe that $\mathsf{V}_k\text{-}\mathsf{PTT} \subseteq \mathsf{V}_{k-1}\mathsf{I\text{-}PTT}$ as the topmost pebble can be replaced by an invisible one, thus $\mathsf{V}_k\text{-}\mathsf{PTT} \subseteq \mathsf{TT}^{k+1}$, which was proved in [11], but also for the deterministic case.

If we start with a deterministic tree transduction, we can show that the inclusions of Lemma 3 also hold in the other direction.

4. TYPECHECKING

The domain of a tree-walking tree transducer without pebbles can be accepted by an alternating TA. Existential states in the alternation correspond to the nondeterminism of the TT, universal states correspond to the recursive way in which output trees are generated. It was shown in [27, Corollary 5.3] and [22, Theorem 4.7] that alternating TA accept the regular tree languages.

Alternatively, it is explained in Section 3.2 of [11] that (deterministic) TT are closely related to attributed tree transducers. Similarly, the nondeterministic TT correspond to the nondeterministic version of the attributed tree transducer. Hence we may transfer a result of [2] to domains of TT. An analysis of the proof gives a useful complexity bound, which improves the constructions of [27] (double exponential) and of [22] (triple exponential).

THEOREM 5. *For every TT $\mathcal{M}$ a regular tree grammar G can be constructed in exponential time such that G generates the domain of $\tau_{\mathcal{M}}$.*

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 $L(G_{\text{in}}) = \tau_{\mathcal{M}}^{-1}(L(G_{\text{out}}))$. Note that our definition differs from the one in [22], 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.

LEMMA 6. *The inverse type inference problem is solvable for TT's in exponential time.*

PROOF. Given a regular tree grammar G_{out} and a tree transducer $\mathcal{M}$ by a standard direct product construction another TT $\mathcal{M}'$ can be constructed in polynomial time such that it equals $\mathcal{M}$ except that it only generates trees in $L(G_{\text{out}})$. Thus the domain of $\tau_{\mathcal{M}'}$ equals $\tau_{\mathcal{M}}^{-1}(L(G_{\text{out}}))$ and by Theorem 5 a regular tree grammar G_{in} for it can be constructed in exponential time. □

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.

COROLLARY 7. *The inverse type inference problem is solvable for a composition of k TT's in k-fold exponential time.*

We finally turn to the *typechecking problem*. It asks, for a tree transducer $\mathcal{M}$ and two regular tree grammars G_{in} and G_{out}, whether or not $\tau_{\mathcal{M}}(L(G_{\text{in}})) \subseteq L(G_{\text{out}})$. This is solved using inverse type inference, as described in [22].

THEOREM 8. *For fixed $k \geq 0$, we can typecheck*
(1) TT^k *in $(k+1)$-fold exponential time,*
(2) $\mathsf{V}_k\text{-}\mathsf{PTT}$ *in $(k+2)$-fold exponential time,*
(3) $\mathsf{V}_k\mathsf{I\text{-}PTT}$ *in $(k+3)$-fold exponential time.*

PROOF. It suffices to prove (1) and refer for (3) to Theorem 4, of which the construction takes polynomial time (for fixed k). For (2) we use that $\mathsf{V}_k\text{-}\mathsf{PTT} \subseteq \mathsf{V}_{k-1}\mathsf{I\text{-}PTT}$.

Clearly, $\tau_{\mathcal{M}}(L(G_{\text{in}})) \subseteq L(G_{\text{out}})$ if and only if $L(G_{\text{in}}) \cap \tau_{\mathcal{M}}^{-1}(R_{\text{out}}) = \varnothing$, where R_{out} is the complement of $L(G_{\text{out}})$. A regular tree grammar for $R_{\text{in}} = \tau_{\mathcal{M}}^{-1}(R_{\text{out}})$ can be constructed in $(k+1)$-fold exponential time by first constructing a regular tree grammar for R_{out} in exponential time, and then using Corollary 7. Emptiness of $L(G_{\text{in}}) \cap R_{\text{in}}$ can then be tested in polynomial time as usual. □

The main conclusion from Theorem 8 is that the complexity of typechecking 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 8.

5. TREES, TESTS AND TRIPS

For 'classical' tree-walking automata with a bounded number of visible pebbles only, it was shown in [9, Section 5] that these automata accept regular tree languages only (but not all of them [5]). One of the main reasons for introducing an unbounded number of (invisible) pebbles is that they can be used to recognize all regular tree languages, unlike V-PTA.

LEMMA 9. $\mathsf{REGT} \subseteq \mathsf{I\text{-}dPTA}$.

PROOF. As the regular tree languages are recognized by deterministic bottom-up tree automata, it suffices to explain how the computation of such an automaton $\mathcal{A}$ can be simulated by a deterministic PTA with invisible pebbles. The computation of $\mathcal{A}$ on the input tree can be reconstructed by a post-order evaluation of the tree. The state of $\mathcal{A}$ on

a subtree is stored using an invisible pebble at the root of the subtree. The colour of the pebble represents the state. Each time a node is processed the PTA picks up the pebbles from its children, from right to left. Now the PTA can compute the state assumed by $\mathcal{A}$ in the node based on the states of the children, and puts that info at the node using an invisible pebble of suitable colour. It then proceeds to the next sibling, or to the parent if the present node was the rightmost child. The post-order evaluation ensures that pebbles are used in a nested fashion. □

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, e.g., by reducing V_kI-PTA to the backtracking pushdown tree automata of [27], but here we deduce it from one of our decomposition results.

THEOREM 10. *For each $k \geq 0$, $\mathsf{V}_k\mathsf{I}\text{-}\mathsf{PTA} = \mathsf{REGT}$.*

PROOF. In the proof of Lemma 6 we argued that for $\tau \in \mathsf{TT}$ and $R \in \mathsf{REGT}$ again $\tau^{-1}(R)$ is in REGT. By iteration this holds for sequences of TT's too.

A PTA $\mathcal{A}$ is easily turned into a PTT $\mathcal{M}$ that outputs single node tree halt() for trees accepted by $\mathcal{A}$. By Theorem 4 the transduction $\tau_{\mathcal{M}}$ is a composition of TT transductions, the domain $\tau_{\mathcal{M}}^{-1}(\,\mathrm{halt}()\,)$ of which is regular. □

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 $a^n b^n$ (and similarly $a^n b^n c^n$).

PATTERNS. Let Σ be a ranked alphabet and $k \geq 0$. A k-ary *pattern* over Σ is a set $T \subseteq \{\ (t, u_1, \ldots, u_k) \mid u_i \text{ node of tree } t \text{ over } \Sigma\ \}$. For $k = 0$ this is a tree language, for $k = 1$ it is a *site* (trees with a distinguished position), for $k = 2$ it is a *trip* [10] (or a binary tree-node relation [4]).

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

An MSO formula $\varphi(x_1, \ldots, x_k)$ over Σ with k free variables defines the k-ary pattern $T(\varphi) = \{\ (t, u_1, \ldots, u_k) \mid t \models \varphi(u_1, \ldots, u_k)\ \}$. It follows from the result of [8, 28] that a pattern is MSO definable iff it is regular.

With the help of an unbounded supply of invisible pebbles tree-walking automata can recognize regular tree languages, Lemma 9. Likewise V_kI-PTA can match arbitrary MSO definable k-ary patterns φ. When k visible pebbles are dropped on a sequence of k 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 8 we will reconsider pattern matching.

Ignoring the visible pebbles, it is also possible to consider the position of the head, and test whether the configuration belongs to a given regular 'marked' tree language. We say that a family of PTA can *perform* MSO *head tests* if, for a regular site T over Σ, an automaton 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. Obviously, as V-PTA cannot recognize all regular tree languages, they cannot perform MSO head tests either: for any regular tree language T the set $\{\ (t, r) \mid t \in T \text{ and } r \text{ is the root of } t\ \}$ is a regular site.

LEMMA 11. *For each $k \geq 0$, V_kI-PTA can perform MSO head tests. The same holds for V_kI-dPTA.*

PROOF. Let $\mathcal{A}_T$ be a deterministic bottom-up 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 to test whether or not the input tree with current head position h is accepted by $\mathcal{A}_T$ using invisible pebbles, in a computation starting and ending at h.

The postorder evaluation of Lemma 9 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.

Instead, first evaluate the subtree rooted at h, and subsequently the subtrees rooted at the siblings of h (with the same pebble stack). When the root u of a subtree is marked by an invisible pebble (as the starting point of the evaluation) that pebble becomes observable exactly when the postorder evaluation reaches the starting point and has lifted all the pebbles from the children for the evaluation. In this way it is always clear when the marked node is visited.

Using these evaluations the evaluation of the parent of h can be determined. Place an invisible pebble on the parent of h, and repeat the above process to find the evaluation of *its* parent. Repeating this process, eventually we reach the root of the tree, and know the outcome of the test. Then return to the original position h picking up the pebbles left on the path from that position to the root. □

This result can easily be extended, using the same proof technique: PTA can test their *visible configuration*, the position of the head together with the positions and colours of the visible pebbles. Later we will show that PTA can even test their *observable configuration*, i.e., the visible configuration plus the topmost pebble (Theorem 17).

We again introduce a new ranked alphabet $\Sigma \times 2^C$, 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 $\mathrm{vis}(t, \pi)$ is obtained by adding to the label of each node u in t the set $b \subseteq C$ such that b contains c iff (u, c) occurs in π and $c \in C_{\mathrm{v}}$. Similarly for $\mathrm{obs}(t, \pi)$, the observable configuration tree, b contains c iff (u, c) occurs in π and c is observable (i.e., $c \in C_{\mathrm{v}}$ 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_c or lift instruction, it behaves just as a TA on $\mathrm{obs}(t, \pi)$.

We say that a family of PTA can *perform* MSO *tests on the visible configuration* if, for a regular site T over $\Sigma \times 2^C$, an automaton can test whether or not $(\mathrm{vis}(t, \pi), h) \in T$, where t is the input tree, π the pebble stack and h the position of the head. Similarly for the observable configuration. Note that for a regular site T over $\Sigma \times 2^C$, $\mathrm{mark}(T)$ is a regular tree language over $\Sigma \times 2^C \times \{0,1\}$.

LEMMA 12. *For each $k \geq 0$, V_kI-PTA and V_kI-dPTA can perform MSO tests on the visible configuration.*

We now turn to the PTA as navigational device: we say that PTA $\mathcal{A}$ *computes the trip* T, if T consists of all triples (t, u, v) such that $\mathcal{A}$, on tree t, started in node u without pebbles on the tree, walks to node v, and halts, again without pebbles on the tree. 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.

The following characterization of the MSO definable trips is more elegant than the one in [10], which uses so-called marble/pebble automata, a restricted kind of V_1I-PTA.

THEOREM 13. *The trips computed by* V_kI-PTA*, for any* $k \geq 0$*, are exactly the* MSO *definable trips. Similarly for* V_kI-dPTA *and functional trips.*

PROOF. Consider a trip T computed by V_kI-PTA $\mathcal{A}$: for any $(t, u, v) \in T$, starting in a 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 for the node mark of starting node u, then it simulates $\mathcal{A}$, and when halted, verifies node v is reached. By Theorem 10 this tree language is regular and hence T is MSO definable.

In [4] it is shown that the MSO definable trips (tree-node relations) can be computed by tree-walking automata with MSO head tests. By Lemma 11 they can be computed by PTA with invisible pebbles. □

6. LOOK-AHEAD TESTS

We say that a family F of VI-PTA can *perform look-ahead tests* if an automaton $\mathcal{A}$ in F can test whether or not another VI-PTA $\mathcal{B}$ (not necessarily in F) has a successful computation when started in one of its initial states in the current configuration (i.e., position of the head and stack of pebbles). We require that $C_{\mathrm{v}}^{\mathcal{A}} \subseteq C_{\mathrm{v}}^{\mathcal{B}}$ and $C_{\mathrm{i}}^{\mathcal{A}} \subseteq C_{\mathrm{i}}^{\mathcal{B}}$.

THEOREM 14. *For* $k \geq 0$*,* V_kI-PTA *can perform look-ahead tests.*

PROOF. Let $\mathcal{A}$ be a V_kI-PTA that wants to perform a look-ahead test by calling another VI-PTA $\mathcal{B}$. When no pebbles are dropped, the test whether $\mathcal{B}$, started in the current position h, successfully halts, is an MSO head test. Indeed, $\{ (t, h) \mid \mathcal{B}$ halts when started in $h \}$ is a regular site, as it is the domain of the VI-PTA $\mathcal{B}'$ that starts in the root, looks for the marked node h, then simulates $\mathcal{B}$. Domains are regular by Theorem 10; $\mathcal{A}$ can perform MSO head tests by Lemma 11.

In general, when $\mathcal{A}$ has simulated $\mathcal{B}$ on its own stack, $\mathcal{A}$ must be able to recover this stack to continue its own computation. For this reason the computations of $\mathcal{B}$ starting at the topmost pebble will be precomputed. With each pebble dropped by $\mathcal{A}$ we store the set of states S for which $\mathcal{B}$ has a successful computation when started in that state at the position u of the topmost stack element (and with the current stack)[1]. Now a successful computation of $\mathcal{B}$ can be safely simulated, consisting of a part where $\mathcal{B}$ uses its own pebbles on top of the stack inherited from $\mathcal{A}$, possibly followed by a part where $\mathcal{B}$ inspects the stack, starting by a visit to u. We discuss how that state set is determined, and how it is used (by $\mathcal{A}$) to perform the look-ahead test. For the first pebble c dropped, the set S can be determined using an MSO head test: construct $\mathcal{B}'$ as above except it now drops c at the marked node before simulating $\mathcal{B}$.

We now assume that the pebble stack is nonempty, distinguishing whether the topmost pebble is visible or invisible.

With a visible topmost pebble, containing this state information, the look-ahead test can be performed as follows. Consider the observable configuration tree $\mathrm{obs}(t, \pi) = \mathrm{vis}(t, \pi)$ with the current node h marked, see Lemma 12. The VI-PTA $\mathcal{B}'$ searches for u and drops an invisible pebble $\perp$ on it, then proceeds to h, starts simulating $\mathcal{B}$, and halts when $\mathcal{B}$ observes pebble $\perp$ at position u in a state from S, or when $\mathcal{B}$ halts never observing $\perp$ (meaning pebbles are still on top of $\perp$ when visiting u). The domain of $\mathcal{B}'$ forms a regular site T over $\Sigma \times 2^C$, and $\mathcal{A}$ can perform an MSO test for T on its observable configuration (because that is its visible configuration).

The same reasoning shows that the state set for the next pebble c dropped can be computed: again $\mathcal{B}'$ first drops the pebble c before starting the simulation.

Now we turn to the case where the topmost pebble at position u is invisible. As before it is necessary to check whether automaton $\mathcal{B}$ started in node h will reach u in a given state (or will halt before visiting u), both for performing the look-ahead test and for computing the new set S for the next pebble dropped. As before this is an MSO test on the observable configuration, but now such a test *cannot* be performed by $\mathcal{A}$: when dropping a pebble on the current node h the topmost pebble becomes unobservable!

Our solution is that $\mathcal{A}$ uses additional invisible pebbles to cover the (shortest) path from topmost pebble at u to current position h. These pebbles will be called *beads* to distinguish them from $\mathcal{A}$'s original pebbles. Again each bead carries state information on successful computations of $\mathcal{B}$ when started at the position of the bead with the current original stack.

The path of beads is updated as follows. If we backtrack on the path of u to h, i.e., we move to a position where a bead is present, we just lift the last bead. (The automaton can test from which direction the path of beads enters h by removing the bead from h, remembering its colour, and inspecting the parent and siblings of h for the presence of a bead, and finally returning to h dropping the bead again.) If we move away from u, we compute new bead information. The last bead is on one of the neighbours of h, say the parent. Consider the following automaton $\mathcal{B}'$, that works on $\mathrm{vis}(t, \pi)$ with h marked. It searches for h, and drops invisible pebble $\perp$ on the parent of h, returning to h. Then the simulation of $\mathcal{B}$ starts. The automaton $\mathcal{B}'$ halts when $\mathcal{B}$ finds $\perp$ in one of the states associated to the topmost bead, or when the simulation of $\mathcal{B}$ halts without observing $\perp$. The crucial point here is that when $\mathcal{B}$ visits the topmost pebble of the current original stack, it must once have passed the parent of h without additional pebbles on its stack (because, in the proof of Lemma 3, the shortest path from h to u in t' equals the shortest path from h to u in t).

Again, the domain of $\mathcal{B}'$ forms a regular site T over $\Sigma \times 2^C$, but now the corresponding MSO test *can* be performed by $\mathcal{A}$ on its *visible* configuration (cf. Lemma 12), because the state information can first be read from the last bead which is at the parent of h. Hence, the state information for the new bead can be computed by $\mathcal{A}$.

Finally, performing a look-ahead test when the topmost pebble is invisible, just means testing whether one of the initial states of $\mathcal{B}$ is among the states in the last bead. □

In fact it can be shown that V_kI-PTA even can perform *iterated* look-ahead tests, that is, they can simulate automata that in turn use look-ahead. This does not directly follow from the above theorem. In the proof $\mathcal{A}$ starts with an empty pebble stack and is allowed to prepare it for $\mathcal{B}$ look-ahead. When $\mathcal{B}$ calls another PTA $\mathcal{C}$, it starts on the stack inherited from $\mathcal{A}$ and has no opportunity to insert the necessary information for $\mathcal{C}$. The solution is simple. When $\mathcal{A}_1$

[1] Actually, the state information for the visible pebbles is stored in the finite control.

$$
\begin{aligned}
[\![\mathrm{lab}_\sigma]\!]_f &= \{\ (u,\pi) \mid u \text{ has label } \sigma\ \}\\
[\![\mathrm{isleaf}]\!]_f &= \{\ (u,\pi) \mid u \text{ is a leaf}\ \}\\
[\![\mathrm{pebble}_c]\!]_f &= \{\ (u,\pi(u,c)) \mid u,\pi \text{ arbitrary}\ \}\\
[\![\langle\alpha\rangle]\!]_f &= \{\ (u,\pi) \mid \exists (v,\pi') : ((u,\pi),(v,\pi')) \in [\![\alpha]\!]_f\ \}\\
[\![\neg\varphi]\!]_f &= \{\ (u,\pi) \mid (u,\pi) \notin [\![\varphi]\!]_f\ \}\\
[\![\mathrm{child}]\!]_f &= \{\ ((u,\pi),(v,\pi)) \mid v \text{ is a child of } u\ \}\\
[\![\mathrm{right}]\!]_f &= \{\ ((u,\pi),(v,\pi)) \mid v \text{ is the next sibling of } u\ \}\\
[\![\mathrm{drop}_c]\!]_f &= \{\ ((u,\pi),(u,\pi(u,c))) \mid u,\pi \text{ arbitrary}\ \}\\
[\![\mathrm{lift}]\!]_f &= \{\ ((u,\pi(u,c)),(u,\pi)) \mid u,\pi,c \text{ arbitrary}\ \}\\
[\![?\,\varphi]\!]_f &= \{\ ((u,\pi),(u,\pi)) \mid (u,\pi) \in [\![\varphi]\!]_f\ \}\\
[\![\alpha \cup \beta]\!]_f &= [\![\alpha]\!]_f \cup [\![\beta]\!]_f\\
[\![\alpha/\beta]\!]_f &= [\![\alpha]\!]_f \circ [\![\beta]\!]_f\\
[\![\alpha^*]\!]_f &= [\![\alpha]\!]_f^*
\end{aligned}
$$

Table 1: Semantics of Pebble XPath

calls $\mathcal{A}_2$, which calls $\mathcal{A}_3$, which ... calls $\mathcal{A}_n$, then all automata $\mathcal{A}_1,\ldots,\mathcal{A}_{n-1}$ prepare their stack by adding state information for $\mathcal{A}_n$. This leads to automata $\mathcal{A}'_1,\ldots,\mathcal{A}'_{n-1}$, and the result follows by induction.

COROLLARY 15. *For $k \geq 0$, V_kI-PTA can perform iterated look-ahead tests.*

Although this result does not seem practically useful, it will become important when we propose a query language based on pebbles in the next section.

7. DOCUMENT NAVIGATION

We define *Pebble XPath*, an extension of Regular XPath [20] with pebbles. The Pebble XPath expressions are recursively defined as follows. In the definitions $c \in C$ ranges over (invisible) pebble colours, and $\sigma \in \Sigma$ ranges over node labels. We define node expressions φ and path expressions α (with elementary expressions φ_0 and α_0). The latter describe walks through a given forest.

$$
\begin{aligned}
\varphi_0 &::= \mathrm{lab}_\sigma \mid \mathrm{isleaf} \mid \mathrm{isroot} \mid \mathrm{isfirst} \mid \mathrm{islast} \mid \mathrm{pebble}_c\\
\alpha_0 &::= \mathrm{child} \mid \mathrm{parent} \mid \mathrm{right} \mid \mathrm{left} \mid \mathrm{drop}_c \mid \mathrm{lift}\\
\varphi &::= \varphi_0 \mid \langle\alpha\rangle \mid \neg\varphi \mid \varphi \wedge \varphi\\
\alpha &::= \alpha_0 \mid ?\,\varphi \mid \alpha \cup \alpha \mid \alpha/\alpha \mid \alpha^*
\end{aligned}
$$

Like we have done for PTA, the semantics of Pebble XPath is given as the transition relation from one configuration (current node and stack of pebbles) to another. For every forest f, $[\![\varphi]\!]_f$ is a subset of $\mathrm{con}(f)$ for every node expression φ while $[\![\alpha]\!]_f$ is a binary relation over $\mathrm{con}(f)$ for every path expression α. Here $\mathrm{con}(f)$ equals the set $N(f)\times(N(f)\times C)^*$, where $N(f)$ is the set of nodes of f.

The essential definitions are given in Table 1. Note that several of the elementary node expressions can be defined in terms of the elementary path expressions: $\mathrm{isleaf} \equiv \neg\langle\mathrm{child}\rangle$, $\mathrm{isroot} \equiv \neg\langle\mathrm{parent}\rangle$, $\mathrm{isfirst} \equiv \neg\langle\mathrm{left}\rangle$, $\mathrm{islast} \equiv \neg\langle\mathrm{right}\rangle$.

A binary pattern T is *Pebble XPath definable* if there is a path expression α such that $T = \{\ (f,u,v) \mid ((u,\lambda),(v,\lambda)) \in [\![\alpha]\!]_f\ \}$.

Without $\varphi ::= \langle\alpha\rangle$ we will call the language *Pebble CAT*, caterpillar expressions extended with pebbles. As every Pebble CAT expression can be seen as a program for an I-PTA (on encoded forests), the equivalence of regular expressions and finite state automata by Kleene shows that the binary relations defined by Pebble CAT expressions are those computed by I-PTA (cf. [4] for a similar result). By Theorem 13 these are the MSO definable trips. Note that $?\langle\alpha\rangle$ and $?\,\neg\langle\alpha\rangle$ correspond to look-ahead tests on the present configuration using (the I-PTA defined by) the path expression α. By Corollary 15 this does not extend the power of the I-PTA language.

THEOREM 16. *A binary pattern is Pebble XPath definable iff 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 [14] (TMNF caterpillar expressions) and [7] (μRegular XPath). Pebble CAT is similar to PCAT of [14] which has the same expressive power as V-PTA (and thus less than MSO by [5]). In PCAT the nesting of pebbles is defined syntactically rather than semantically.

We do not know whether Theorem 16 still holds when $\alpha ::= \alpha \cap \alpha$ (intersection) and/or $\alpha ::= \alpha\backslash\alpha$ (except) are added to Pebble XPath.

8. MSO TESTS AND PATTERN MATCHING

As a prelude to the discussion on pattern matching we specify, this time, the MSO tests on the observable configuration by MSO formulas over $\Sigma \times 2^C$. Rather than predicates $\mathrm{lab}_\alpha(x)$ for all $\alpha \in \Sigma \times 2^C$, we assume such formulas to have predicates $\mathrm{lab}_\sigma(x)$ for all $\sigma \in \Sigma$, and $\mathrm{peb}_c(x)$ for all $c \in C$, meaning for a node u with label (σ', b) that $\sigma' = \sigma$ and that $c \in b$, respectively. For an observable configuration tree $\mathrm{obs}(t,\pi)$ they mean, of course, that u has label σ in t and that u holds an observable c-coloured pebble, respectively. We show PTA 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. Additionally, they can *perform* MSO *jumps*: the next position of the head can be specified by a formula $\varphi(x,y)$ on $\mathrm{obs}(t,\pi)$, where the head moves from x to y.

THEOREM 17. *For $k \geq 0$, V_kI-PTA can perform* MSO *tests and* MSO *jumps on the observable configuration.*

PROOF. A PTA $\mathcal{A}$ can use a sequence of look-ahead tests (Theorem 14) to determine whether or not the pebble stack is nonempty, and, if so, what is the colour d of the topmost pebble. By Lemma 12 we may assume that $d \in C_\mathrm{i}$.

Now let $\varphi(x)$ be an MSO formula on the observable configuration tree $\mathrm{obs}(t,\pi)$, and let $\psi(x,z)$ be the MSO formula obtained from $\varphi(x)$ by changing every atomic formula $\mathrm{peb}_d(y)$ into $y = z$, and every $\mathrm{peb}_c(y)$ with $c \in C_\mathrm{i} \setminus \{d\}$ into false. Then $\varphi(x)$ is equivalent to $\exists z\,(\psi(x,z) \wedge \mathrm{peb}_d(z))$. Since $\psi(x,z)$ defines a trip over $\Sigma \times 2^{C_\mathrm{v}}$, it can be computed by an I-PTA on $\mathrm{vis}(t,\pi)$, by Theorem 13. Thus, the existence of such a trip from the position x of the reading head to the position z of the topmost pebble can be computed by a V_kI-PTA $\mathcal{B}$ on the current stack, by first simulating the I-PTA and then verifying that pebble d is on node z (note that the pebble stack of the I-PTA is empty at the end of the trip). Hence, $\mathcal{B}$ can be used as a look-ahead test, cf. Theorem 14.

An MSO jump $\varphi(x,y)$ is a regular trip on $\mathrm{obs}(t,\pi)$ from x to y. Hence by the result of [4] (cf. the proof of Theorem 13) it can be performed by a TA with MSO tests on the observable configuration. □

We note that Theorems 14 and 17 also hold for deterministic V_kI-PTA, and that analogous results hold for V_kI-PTT.

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. In order to find all n-tuples of nodes matching the n-ary pattern defined by the MSO formula $\varphi(x_1, \ldots, x_n)$, a PTT enumerates all n-tuples using $n-2$ visible pebbles, one invisible pebble on top, and the head. For each such tuple it performs the MSO test φ on the observable configuration (Theorem 17).

As discussed in the Introduction, the `for ... where` construct in XQuery often induces less general patterns. There the pattern is defined by a graph, and the test is a conjunction (or even a Boolean combination) of binary tests $\varphi(x, y)$ for each edge (x, y) of the graph, cf. [16].

Consider such a pattern Π. Let $G_\Pi = (V, E)$ be the undirected graph induced by the pattern. The set V contains the node variables involved, and E consists of the pairs (x, y) for which a test $\varphi(x, y)$ is included in Π.

Usually the variables of Π are specified in a specific order $\lambda = (x_1, \ldots, x_n)$, and the matches must be listed in the lexicographic order induced by λ. Pattern matching Π can be done by a VI-PTT as follows. Pebbles (with distinct colours) $p_1, \ldots, p_n$ are used to represent $x_1, \ldots, x_n$, dropping them in that order. When the head is at a candidate for position x_j all MSO tests $\varphi(x_i, x_j)$ for $i < j$ are performed. For each match, the subtree rooted at x_i is generated, by a separate process for each i; that is straightforward, even when p_i is invisible: lift pebbles $p_n, \ldots, p_{i+1}$ one by one, and then access p_i.

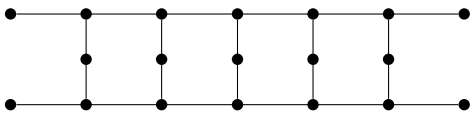
As the complexity of typechecking the transducer depends critically on the number of visible pebbles used we wish to minimize that number, and use as many invisible pebbles as possible for matching. To do the tests at node x_j all pebbles p_i for which $(x_i, x_j) \in E$ and $i < j$ should be observable. Hence all such pebbles under the topmost pebble must be visible. These are the pebbles corresponding to the set $\mathrm{vis}(\lambda) = \{\, x_i \mid \text{ there exists } (x_i, x_j) \in E \text{ with } i+1 < j \,\}$.

If the order λ is irrelevant we may want to determine an optimal order. Let us call a subset $W \subseteq V$ *visible* for Π if the subgraph of G_Π induced by $V - W$ is acyclic and has only vertices of degree at most 2. Note that every $\mathrm{vis}(\lambda)$ is visible.

PROPOSITION 18. *Pattern matching* Π *can be done by a deterministic* V_kI-PTT *where* $k = \#W$ *for a visible set* W *for* Π.

PROOF. Define the order λ on V as follows. First list the visible vertices W in any order. Then list the remaining vertices linearly ordered following the components in the subgraph induced by $V - W$. Clearly $\mathrm{vis}(\lambda) \subseteq W$. □

It suffices to take as visible nodes those of degree ≥ 3 in the pattern graph (plus one node in each connected component that is a cycle). But often one can choose a smaller set. As an example, a ladder with $3n + 4$ nodes as in the picture needs only n visible pebbles rather than $2n$.



Note that if $W = \varnothing$, then pattern matching can be done by an I-PTT. Note also that, for $x, y \in W$, if $T(\varphi(x, y))$ is functional then all other edges incident with y can be redirected to x, and y can be dropped from W.

9. DOCUMENT TRANSFORMATION

The Document Transformation Language DTL introduced in [19] transforms unranked forests. To be able to compare the power of this model to our PTT, one either can encode unranked forests as (binary) trees, or one may adapt the PTT in such a way they generate unranked forests. Here we follow the last course, as in [25].

In our framework a PTT can be used to output unranked forests replacing the output rules for ranked nodes by a facility to generate unranked nodes and unbounded sequences of trees. The new 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. Thus we obtain the pebble *forest* transducer (PFT).

A program in the DTL framework uses *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 labelled with a *selector* of the form $\langle q', \psi(x, y)\rangle$, 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 a node x that matches φ, i.e., satisfies $\varphi(x)$. Then the transducer outputs forest f, where each selector $\langle q', \psi(x, y)\rangle$ is recursively computed as the result of a sequence of copies of the transducer, started in state q' in each of the nodes y that satisfy $\psi(x, y)$, the nodes taken in depth-first order ('document order'). Note that the ψ are also used to 'jump' to another position in the tree.

Let DTL denote the forest transformations defined by DTL. For the comparison to PFT we assume the input of the DTL program to be a ranked tree.

LEMMA 19. DTL $\subseteq$ I-PFT.

PROOF. We construct an I-PFT $\mathcal{M}$ that uses invisible pebbles of a single colour, and never retrieves its pebbles (except when performing MSO tests).

The new transducer $\mathcal{M}$ simulates a DTL rule $\langle q, \varphi(x)\rangle \to f$ in state q at node u by first checking whether $\varphi(u)$ holds, cf. Theorem 17. The Δ-labelled nodes of the right-hand side f can be copied to the output. For each leaf in f labelled by the selector $\langle q', \psi(x, y)\rangle$, a copy of $\mathcal{M}$ is started that stores q' and ψ in its finite control, and executes a subroutine that finds all nodes v in the tree for which $\psi(u, v)$ holds. It first drops a pebble on the current node u and then performs a depth-first traversal of the tree, starting at the root, checking in each node v whether ψ holds (for the last pebble dropped and the position of the head, cf. again Theorem 17). If true, then the copy of $\mathcal{M}$ spawns two concatenated processes, the left branch restarts in state q' and the right branch continues the depth-first search. When the search ends, $\mathcal{M}$ outputs ε. □

It can, in fact, be shown that when output forests are encoded as binary trees, DTL $\subseteq$ I-PTT.

In [18] the language DTL was extended to TL where the states have parameters that hold unevaluated forests, similar to macro tree transducers with outside-in parameter evaluation. The rules of TL programs are of the form

$$\langle q, \varphi(x)\rangle(z_1, \ldots, z_n) \to f$$

where $z_1, \ldots, z_n$ are the parameters, the arity n depending on q. The right-hand side f is a forest of *actions* built using symbols from Δ, the parameters z_i as symbols of rank 0, and selectors $\langle q', \psi(x,y)\rangle(f_1, \ldots, f_m)$ of rank m. Thus, selectors can be nested.

EXAMPLE 20. The transformation from Example 1 can be computed by a deterministic TL program P with the following rules, where the variables i, σ_i, c, and λ_i range over the same values as in Example 1 (with, in the 4th rule, $c = 1$ or $i = 1$); also, $\mathrm{root}(x)$ and $\mathrm{leaf}(x)$ test whether node x is the root or the leaf, $\mathrm{lab}_\sigma(x)$ tests whether x has label σ, and y/x expresses that y is the parent of x.

$\langle q_{\mathrm{start}}, \mathrm{root}(x)\rangle \to \langle q_{\mathrm{start}}, \mathrm{leaf}(y)\rangle$
$\langle q_{\mathrm{start}}, \mathrm{leaf}(x) \wedge \mathrm{lab}_{\sigma_0}(x)\rangle \to \langle q_1, y/x\rangle(\sigma_0, e)$
$\langle q_0, \neg\mathrm{root}(x) \wedge \mathrm{lab}_{\lambda_0}(x)\rangle(z_1, z_2) \to \langle q_0, y/x\rangle(z_1, z_2)$
$\langle q_c, \neg\mathrm{root}(x) \wedge \mathrm{lab}_{\lambda_i}(x)\rangle(z_1, z_2)$
$\quad \to \langle q_i, y/x\rangle(\langle q_{\mathrm{copy}}, y = x\rangle(z_1), \langle q_c, y/x\rangle(z_1, z_2))$
$\langle q_c, \mathrm{root}(x)\rangle(z_1, z_2) \to r(\langle q_{\mathrm{copy}}, y = x\rangle(z_1), z_2)$
$\langle q_{\mathrm{copy}}, \mathrm{lab}_{\sigma_1}(x)\rangle(z_1) \to \sigma_1(z_1)$

An XSLT 1.0 program corresponding to P can easily be written, with parameters of type 'result tree fragment'. □

THEOREM 21. $\mathsf{TL} = \mathsf{I\text{-}PFT}$.

PROOF. ($\subseteq$) The construction extends the one of the proof of Lemma 19. For convenience we ignore tests, both in TL and I-PFT. The main idea is to use states and pebble colours to store expressions (forests of actions) that are to be evaluated on the tree; the pebbles store the parameters.

To simulate, in state q, a rule $\langle q, \varphi(x)\rangle(z_1, \ldots, z_n) \to f$, go into state $[f]$. If the state is of the form $[tf]$, for a tree t and a forest f, the transducer uses rule $[tf] \to \langle [t], \mathrm{stay}\rangle\ \langle [f], \mathrm{stay}\rangle$, branching the computation. If it is of the form $[\delta(f)]$, the rule is $[\delta(f)] \to \delta(\langle [f], \mathrm{stay}\rangle)$, and if it is of the form $[\varepsilon]$, the rule is $[\varepsilon] \to \varepsilon$.

If the state is of the form $[\langle q', \psi(x,y)\rangle(f_1, \ldots, f_m)]$, then a pebble $([f_1], \ldots, [f_m])$ is dropped on the current node u to represent the parameters, and the transducer starts a copy in state q' in every node v that satisfies $\psi(u,v)$, see the construction in the proof of Lemma 19. If the state is of the form $[z_i]$ for some parameter z_i, this means the parameter has to be evaluated. The transducer searches for the top pebble $([f_1], \ldots, [f_m])$, pops it, and changes state to $[f_i]$ ready to evaluate it.

($\supseteq$) The proof is too technical to present here. We observe that the equality $\mathsf{TL} = \mathsf{I\text{-}PFT}$ is a variant of the well-known fact that macro grammars are equivalent to indexed grammars [13], see also [12, Theorem 5.24]. □

Following an idea of [25] we can transfer our typechecking results for tree transducers to forest transducers.

THEOREM 22. *For fixed $k \geq 0$, we can typecheck $\mathsf{V}_k\mathsf{I\text{-}PFT}$ in $(k+3)$-fold exponential time.*

For $k = 0$ this provides an alternative proof of the main result of [18]. In general, decomposition into TT's leads to more efficient typechecking than decomposition into macro tree transducers, because (cf. Lemma 6) inverse type inference of MTT's takes double exponential time, unless the number of parameters is bounded and the output type is fixed [25]. We can typecheck TL in 4-fold exponential time, assuming the MSO tests are represented by deterministic bottom-up tree automata.

10. REFERENCES

[1] S. Abiteboul, P. Buneman, D. Suciu. *Data on the Web.* Morgan Kaufmann, 2000.
[2] M. Bartha. An algebraic definition of attributed transformations. Acta Cybernetica 5, 409–421, 1982.
[3] G. J. Bex, S. Maneth, F. Neven. A formal model for an expressive fragment of XSLT. Information Systems 27, 21–39, 2002.
[4] R. Bloem, J. Engelfriet. Monadic second order logic and node relations on graphs and trees. In: Structures in Logic and Computer Science, LNCS 1261, 144-161, 1997.
[5] M. Bojanczyk, M. Samuelides, T. Schwentick, L. Segoufin. Expressive power of pebble automata. Proc. ICALP'06, LNCS 4051, 157-168, 2006.
[6] A. Brüggemann-Klein, D. Wood. Caterpillars, context, tree automata and tree pattern matching. Proc. DLT'99, World Scientific, 270–285, 1999.
[7] B. ten Cate. The expressivity of XPath with transitive closure. Proc. PODS'06, ACM Press, 328-337, 2006.
[8] J. Doner. Tree acceptors and some of their applications. J. Comput. and Syst. Sci. 4, 406–451, 1970.
[9] J. Engelfriet, H.J. Hoogeboom. Tree-walking pebble automata. In: Jewels are forever, Springer-Verlag, 72-83, 1999.
[10] J. Engelfriet, H.J. Hoogeboom, J.-P. Van Best. Trips on trees. Acta Cybernetica 14, 51-64, 1999.
[11] J. Engelfriet, S. Maneth. A comparison of pebble tree transducers with macro tree transducers. Acta Inform. 39, 613–698, 2003.
[12] J. Engelfriet, H. Vogler. Pushdown machines for the macro tree transducer. Theor. Comput. Sci. 42, 251–368, 1986.
[13] M.J. Fischer. Grammars with macro-like productions. Ph.D. Thesis, Harvard University, 1968.
[14] E. Goris, M. Marx. Looping caterpillars. Proc. LICS'05, IEEE, 51–60, 2005.
[15] G. Gottlob, C. Koch, R. Pichler. Efficient algorithms for processing XPath queries. Proc. VLDB'02, Morgan Kaufmann, 95–106, 2002.
[16] G. Gottlob, C. Koch, K.U. Schulz. Conjunctive queries over trees. Proc. PODS'04, ACM Press, 189–200, 2004.
[17] W. Janssen, A. Korlyukov, J. Van den Bussche. On the tree-transformation power of XSLT. Acta Inform. 43, 371–393, 2007.
[18] S. Maneth, A. Berlea, T. Perst, H. Seidl. XML type checking with macro tree transducers. Proc. PODS'05, ACM Press, 283–294, 2005.
[19] S. Maneth, F. Neven. Structured document transformation based on XSL. Proc. DBPL'99, LNCS 1949, 2000, 80–98.
[20] M. Marx. Conditional XPath. ACM T. Database Syst. 30, 929-959, 2005.
[21] M. Marx. Navigation in XML trees. In: The Logic in Computer Science Column, Bull. EATCS 88, 126-140, February 2006.
[22] T. Milo, D. Suciu, V. Vianu. Typechecking for XML transformers. J. Comput. Syst. Sci. 66, 66–97, 2003.
[23] A. Møller, M. I. Schwartzbach. The design space of type checkers for XML transformation languages. Proc. ICDT'05, LNCS 3363, 17-36, 2005.
[24] F. Neven, T. Schwentick. Automata- and logic-based pattern languages for tree-structured data. In: Semantics in Databases 2001, LNCS 2582, 160-178, 2003.
[25] T. Perst, H. Seidl. Macro forest transducers. Inform. Process. Lett. 89, 141-149, 2004.
[26] B. Samwel. Pebble scope and the power of pebble tree transducers. M.Sc. Thesis, LIACS, Leiden University, 2006.
[27] G. Slutzki. Alternating tree automata. Theor. Comput. Sci. 41, 305–318, 1985.
[28] 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, 57–81, 1968.